UNIVERSIDADE DO SUL DE SANTA CATARINA

ALEXANDRE PEREIRA PRAZERES

PRINCÍPIOS PARA O DESENVOLVIMENTO DE SOFTWARE SEGURO

Florianópolis

2015

ALEXANDRE PEREIRA PRAZERES

PRINCÍPIOS PARA O DESENVOLVIMENTO DE SOFTWARE SEGURO

Monografia apresentada ao Curso de Especialização em Engenharia de Projetos de *Software* da Universidade do Sul de Santa Catarina, como requisito à obtenção do título de Especialista em Engenharia de Projetos de *Software*.

Orientadora: Prof.ª Vera Rejane Niedersberg Schuhmacher, Dra.

Florianópolis

2015

ALEXANDRE PEREIRA PRAZERES

PRINCÍPIOS PARA O DESENVOLVIMENTO DE SOFTWARE SEGURO

Esta Monografia foi julgada adequada à obtenção do Título de Especialista em Engenharia de Projetos de *Software* e aprovada em sua forma final pelo Curso de Especialização em Engenharia de Projeto de *Software* da Universidade do Sul de Santa Catarina.

Florianópolis, 14 de abril de 2015.

_____

Professora e orientadora Vera Rejane Niedersberg Schuhmacher, Dra.
Universidade do Sul de Santa Catarina

_____

Prof. Aran Bey Tcholakian Morales, Dr.
Universidade do Sul de Santa Catarina

A Deus, minha família, meus amigos, minha namorada, e a tudo de bom que a vida nos permite ser e realizar.

AGRADECIMENTOS

A Deus, pois graças a Ele que tive saúde para poder realizar este sonho.

Aos meus amados pais, Sérgio e Telma, que me deram todo amor e dentre tantas outras coisas, mais essa oportunidade maravilhosa na vida.

Ao meu irmão Rodolfo por estar sempre presente.

A toda minha família que, mesmo longe fisicamente sempre está no meu coração.

A minha namorada Gabriela, que faz as coisas ruins serem minúsculas e as coisas boas serem gigantes, me dando sempre muito amor e carinho.

Aos meus amigos por dividirem minhas felicidades e frustrações, sempre estendendo a mão quando precisei.

A minha orientadora e coordenadora do curso, doutora Vera Schuhmacher, a qual admiro imensamente – desde a graduação – e agradeço por todas as horas de atenção dispensadas.

“Tente mover o mundo – o primeiro passo será mover a si mesmo” (Platão).

RESUMO

Este trabalho tem por objetivo apresentar técnicas de segurança na codificação de *softwares*, principalmente em aplicações *web*. O estudo possui bases nas normas NBR ISO/IEC 27001 e 27002, e principalmente nas técnicas gerais e praticas de dois desenvolvedores da Microsoft, Howard e LeBlanc. O estudo também contribui para elucidar a importância de eliminar, ou ao menos atenuar, as vulnerabilidades *web* mais conhecidas, descrevendo seus impactos e até soluções específicas. Apresenta um sistema *web* desenvolvido em PHP, o qual teve diversas técnicas de segurança aplicadas e o resultado dos testes de segurança efetuados por meio do software *Acunetix Vulnerability Scanner* perante o sistema implementado.

Palavras-chave: Código Seguro. Normas e Técnicas de Segurança. Vulnerabilidades *Web*.

ABSTRACT

This work aims to present security techniques in software coding, especially in web applications. The study has bases in NBR ISO/IEC 27001 and 27002 , and especially in the general techniques and practices of two Microsoft developers, Howard and LeBlanc. The study also contributes to elucidate the importance of eliminating, or at least mitigate, the most known web vulnerabilities, describing their impact and specific solutions. This work features a web system developed in PHP, which had several security techniques applied, showing the results of safety tests carried out by the Acunetix Vulnerability Scanner software in the implemented system.

Keywords: Secure Code. Safety Standards and Techniques. Web Vulnerabilities.

LISTA DE QUADROS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O uso dos sistemas de informação e das tecnologias de informação mudou drasticamente nas últimas décadas, expandindo e conectando todos os setores sociais. Esses avanços trouxeram benefícios para os cidadãos, bem como para diversos setores, tais como o financeiro, industrial, acadêmico, de negócios e de serviços. Em contrapartida, para que haja uma operação confiável e segura dessa miscelânea de tecnologias, os sistemas de informação e suas infraestruturas exigem grandes preocupações com a segurança por parte do governo, empresas, organizações e usuários que desenvolvem, compram, oferecem, gerenciam ou apenas utilizam esses sistemas de informação. (RAMSAROOP, 2003)

Há cerca de 20 anos atrás, os computadores eram como ilhas isoladas e o máximo que se podia fazer era se "auto-atacar", ou seja, o usuário podia ser afetado mas isoladamente, não disseminando uma ameaça para outros usuários. À medida que a Internet cresceu, os aplicativos passaram a estar cada vez mais interconectados e os tempos mudaram. Na era da Internet praticamente todos os dispositivos possuem acesso à Internet, possibilitando oportunidades incríveis aos desenvolvedores de *software* mas também aumentam exponencialmente as chances dos aplicativos serem atacados, já que não há limite de conectividade. (HOWARD; LEBLANC, 2005)

Com este avanço tecnológico e o crescimento da área de sistemas de informação, o controle das vulnerabilidades nos *softwares* não acompanhou tamanha evolução em tão pouco tempo. Por isso Khan e Zulkernine (2009) deixam claro que uma das fontes para os problemas de segurança é ausência de planejamento e controle dos requisitos de segurança durante o processo de desenvolvimento de *software*.

Segundo Gomes e Santos (2006), casos envolvendo vazamento de dados dos usuários, segredos industriais e tantos outros dados privados das empresas não são mais tão incomuns, causando prejuízos financeiros e até danos à marca e a imagem da empresa. Portanto a adoção de práticas de segurança atua diretamente no coração das empresas, trabalhando em paralelo com diversas atividades, muitas vezes de forma imperceptível e se tornando uma necessidade nas empresas modernas.

O problema para implantação de segurança nas empresas de desenvolvimento é que a segurança não é vista como geração de renda, levando à gerência a não investir em treinamento para produção de códigos seguros. O resultado é que a empresa só investe quando um ataque já foi bem-sucedido e a essa altura é tarde demais, além de muito mais caro, tanto

em termos financeiros quanto em termos de imagem da empresa. (HOWARD; LEBLANC, 2005)

A fim de auxiliar e compartilhar boas práticas com empresas que desenvolvem sistemas, diversos modelos, padrões e normas foram criados para atender essa demanda por *software* seguros. Elas fornecem boas práticas porém não especificam de que forma devem ser acopladas aos processos, deixando essa parte à cargo de cada organização (BRAZ, 2009).

Durante o desenvolvimento de um *software* é necessário atender os princípios de integridade e confiabilidade, visto que o índice de segurança deve aumentar enquanto que o de falhas precisa diminuir. Por este motivo, alguns padrões e normas foram estabelecidos visando a produção de *softwares* seguros, dentre elas destaca-se a ISO/IEC 15.408 e a Norma 17799 (CRUZ, 2007). Essas e outras normas importantes serão abordadas no Capítulo 3.

1.1 PROBLEMÁTICA E JUSTIFICATIVA

Segundo Howard e LeBlanc (2005), desde os primórdios a propriedade particular é protegida, até nosso mais antigo ancestral tinha leis que penalizavam pessoas que invadissem, danificassem ou roubassem uma propriedade particular. No mundo digital não é diferente, sempre que há algum motivo para proteger bens e propriedades privadas, é função dos desenvolvedores criar soluções e aplicativos que protejam o patrimônio digital.

À medida que a sociedade civil, principalmente nos países industrializados, torna-se cada vez mais dependente das tecnologias e sistemas de informação, a infraestrutura de informação e os próprios sistemas se tornaram uma área muito desejada por criminosos e organizações criminosas que aprenderam a tirar proveito do uso de sistemas de informação com propósitos ilegais (crime virtual, ou *Cybercrime*) e para a violência indiscriminada contra civis e instituições civis (terrorismo virtual, ou *Cyberterrorism*) com o propósito de tumultuar a sociedade destruindo a confiança da sociedade em seus líderes, políticas e instituições. Para proteger a sociedade e as empresas dessas ameaças, a segurança da informação precisa ser valorizada de modo que ações contra a população ou ilegais sejam combatidas e exterminadas. (RAMSAROOP, 2003)

No dia 22 de dezembro de 2014, conforme o New York Times (2014), a Coréia do Norte ficou sem internet por um dia. A causa da interrupção dos serviços foi declarada como um ataque cibernético (através de computadores utilizando a internet) feita pelos Estados Unidos ao país asiático, impossibilitando os cerca de 1.024 *IPs* do país de conectarem-se com o mundo. Esta ocorrência foi tratada como uma resposta aos milhões de

dados roubados de funcionários da gigante americana dos eletrônicos, a Sony, por lançar o filme chamado “A entrevista”, uma comédia que gira entorno da tentativa de assassinato do então líder supremo da Coréia do Norte, Kim Jong-un. Conforme a matéria, o líder supremo não gostou do filme e ameaçou a Sony e os Estados Unidos um dia antes da ocorrência do ataque à Sony. Estes fatos deixam claro que se existem países e grandes empresas tão vulneráveis a ataques de computador e Internet, o planeta Terra ainda está engatinhando em termos de segurança da informação e precisa avançar muito (e rápido) para que os dados trafegados por aplicativos e sistemas com acesso à Internet sejam seguros.

Um *software* que é alvo de especulações ou notícias que afirmem que o mesmo é inseguro perceberá que muitos potenciais clientes ou usuários, passam a fugir do produto ou da empresa, ou pior, as pessoas com algum ressentimento do produto passam a aumentar ainda mais o acumulo de publicidade negativa, provando à outras pessoas que usar o produto é perigoso. Infelizmente as pessoas tendem a acompanhar somente as notícias ruins, o que causa um dano irreparável de tal modo que chegará o momento em que as pessoas procuraram o produto do concorrente, afundando de vez o produto inseguro. (HOWARD; LEBLANC, 2005)

Conforme a Microsoft (2015), dados do FBI (*Federal Bureau of Investigation*) revelam que em 2007, 70 bilhões de dólares foram roubados através de crimes virtuais e 90% das vulnerabilidades dos sistemas são exploradas remotamente (via web).

Por isso, é preciso garantir a segurança do *software*, porém, segundo Braz (2009), apesar das normas de segurança serem bem claras, as mesmas não disponibilizam de uma forma detalhada a forma de implantação nos processos, tornando difícil para as empresas saber exatamente como atendê-las em sua plenitude. Por este motivo, este estudo foi desenvolvido e se experimentou na prática a aplicação de algumas normas e práticas sugeridas de segurança.

1.2 OBJETIVOS

Nas duas próximas seções terciárias, são descritos os objetivos deste trabalho acadêmico, também servindo como um escopo inicial do projeto.

1.2.1 Objetivo Geral

O presente trabalho tem por objetivo a aplicação de boas práticas de segurança no desenvolvimento de um sistema para manutenção de notas pessoais.

1.2.2 Objetivos Específicos

São objetivos específicos do projeto:

- conhecer as principais normas de segurança aplicáveis ao desenvolvimento de sistemas;
- conhecer ferramentas que ajudem a verificar a segurança de um *software*;
- desenvolver e implementar um sistema web em PHP que permita inserir, editar e excluir notas pessoais em um banco de dados e que utilize algumas ou todas as normas e boas práticas de segurança estudadas;
- verificar a segurança do sistema desenvolvido através de testes com ferramentas especializadas, validando os erros que as normas puderam evitar ou não.

1.3 ESTRUTURA DA MONOGRAFIA

No **Capítulo 1**, é dada uma breve introdução ao assunto, bem como os objetivos, problemática e justificativa do trabalho.

No **Capítulo 2**, a Segurança da Informação é o tema principal e os principais conceitos a seu respeito bem como as principais falhas e seus efeitos são apresentados.

No **Capítulo 3**, exibimos alguns padrões de segurança reconhecidos mundialmente para proteção do sistema, usuários e empresa desenvolvedora.

No **Capítulo 4**, a codificação segura é descrita na prática em tópicos que revelam o que pode ser feito para contornar os principais tipos de riscos encontrados nas aplicações.

No **Capítulo 5**, apresentamos *softwares* e metodologias que permitem testar se um *software* é seguro, ou que permitem – ao menos – afirmar que atende ou não ao mínimo em segurança exigido de um sistema.

No **Capítulo 6**, apresentamos um pequeno sistema que implementa normas e boas práticas de segurança destacando os trechos em que os conhecimentos em segurança da informação foram aplicados, bem como o resultado do teste de vulnerabilidades no mesmo.

2 SEGURANÇA DA INFORMAÇÃO

A informação é, segundo a norma NBR ISO/IEC 27002 (ABNT, 2005), um ativo importante para os negócios. Por atribuir valor à empresa, a informação precisa receber a proteção adequada. Para oferecer a proteção adequada, existe a segurança da informação, ou seja, uma forma de proteger informações de diversos tipos de ameaças. Essa proteção influencia no campo estratégico das empresas, contribuindo para a continuidade dos negócios, aumento das oportunidades, minimização de riscos e maximização do ROI (Retorno sobre os investimentos).

Howard e LeBlanc (2005) alertam, um sistema desenvolvido sem preocupação com a segurança da informação será alvo fácil da mídia "negativa", isto é: o *software* tem sua vulnerabilidade conhecida, então divulgada pela imprensa e as pessoas simplesmente deixam de compra-lo e passam a comprar do concorrente. Para evitar que isso ocorra, é preciso intensificar os cuidados com a codificação segura, visto que o principal produto de uma empresa de *software* é exatamente seus *softwares*.

Na definição de Campos (2007), a segurança da informação existe com o propósito de garantir a confidencialidade, integridade e disponibilidade das informações de uma empresa. Já para a norma ABNT NBR ISO/IEC 27002 (2005), é adicionado o objetivo de garantir a autenticidade. Os princípios básicos da segurança da informação conforme a norma são:

- **Autenticidade:** este princípio prega a garantia da veracidade da fonte das informações. É preciso garantir que a informação que é trafegada tenha um proprietário e que este proprietário não foi alterado;
- **Confidencialidade:** ligado à garantia de que a informação não é acessada por alguém não autorizado, tanto internamente quanto externamente à organização. A informação deve ser protegida contra cópias e distribuições não autorizadas, garantindo que a informação é confidencial e somente pessoas autorizadas possuem acesso a ela;
- **Disponibilidade:** também conhecido como continuidade do serviço, é a garantia de que as informações estão disponíveis às pessoas autorizadas, sempre que elas necessitarem;

- **Integridade:** consiste na fidedignidade da informação. A informação trafegada precisa ser original, não podendo ter nenhum dado alterado, acrescentado ou removido sem a devida autorização da pessoa responsável por ela. Basicamente a informação não pode sofrer nenhum tipo de violação.

O Tribunal de Contas da União (2007) define a segurança da informação sob os mesmos pontos citados acima e acrescenta que a informação é o principal ativo de uma organização, podendo ser considerada o recurso patrimonial mais crítico já que se não tiver a atenção necessária da empresa, pode comprometer não só a imagem da instituição como até mesmo sua continuidade.

Já Lyra (2008), além de citar os pontos já mencionados como disponibilidade, autenticidade, integridade e confidencialidade da informação, divide os principais "itens de segurança" conforme o quadro abaixo:

Quadro 1 - Envolvidos na Segurança da Informação.

Item	**Descrição**
Ativo de Informação	Um bem de grande valor para a organização, podendo ser a tecnologia, o meio que suporta a informação, o ambiente em que está inserida e as pessoas que a utilizam.
Ataque	Um incidente de segurança causado por um agente que deseja obter algum tipo de retorno de algum ativo de informação.
Vulnerabilidade	As fraquezas dos ativos de informação, muitas vezes essas vulnerabilidades podem existir mas nunca serem exploradas, porém quando isso ocorre geralmente levam a quebra de confidencialidade, integridade ou indisponibilidade da informação, mesmo que não intencionais por parte do ativo de informação.
Ameaça	Um ataque potencial a um ativo da informação. Ocorre quando um agente

	externo aproveita-se de alguma vulnerabilidade encontrada e quebra um ou mais princípios de segurança da informação.
Probabilidade	A chance de uma falha de segurança ocorrer ao somarmos o número de vulnerabilidades e ameaças a algum ativo de informação.
Impacto	É a consequência de um incidente de segurança da informação, podendo ter maiores ou menores magnitudes, já que depende do ativo de informação envolvido. Quanto maior o valor do ativo, maior o impacto de um incidente de segurança.
Controle	São todos os mecanismos utilizados para diminuir as vulnerabilidades dos ativos de informação, já que as ameaças provêm de um agente externo e não é possível agir preventivamente sobre elas.

Fonte: Lyra, 2008.

Outros fatores que não são controláveis como greves, manifestações, tempestades, vulcões, ausência de energia elétrica e etc, também estão relacionadas à segurança da informação, já que podem afetar a integridade e disponibilidade das informações. (LYRA, 2008)

2.1 FALHAS DE SEGURANÇA

As falhas de segurança são incidentes de segurança que ocorrem devido à existência de um agente que busca algum retorno nas informações da empresa, também podendo se enquadrar no item de “ataque” descrito no livro de segurança da informação de Lyra (2008). Nas seções terciárias são apresentadas as principais causas das falhas de segurança e seus efeitos.

2.1.1 Causas

Segundo a Microsoft (2014), as causas que levam a uma falha de segurança são, em sua maioria, devido à:

- ***Not a Security Bug*** **(Não é uma falha de segurança):** este primeiro item é autoexplicativo, simplesmente acredita-se que seja uma falha de segurança mas não é;
- ***Buffer Overflow / Underflow*** **(Excesso ou falta de dados na memória virtual):** é um tipo de violação de segurança de memória que ocorre quando não é verificado ou limitado o tamanho de memória que precisa ser alocada antes que as informações sejam manipuladas ou processadas por um *software*. No caso do *Overflow* um programa ultrapassa os limites do *buffer* e sobrescreve a memória adjacente, já o *Underflow* ocorre quando um buffer é lido ou esvaziado antes de ser reescrito ou preenchido;
- ***Arithmetic Error*** **(Erro aritmético):** é quando o valor de algum dado excede (*overflow*) ou não alcança (*underflow*) o limite em que foi especificado para ser manipulado;
- ***SQL / Script Injection*** **(Injeção de código):** é um tipo de falha que permite aos usuários mal intencionados modificar o comportamento de alguma ação do sistema, alterando o código fonte do programa ou de uma instrução SQL (*Structured Query Language*) ao banco de dados da aplicação;
- ***Directory Traversal*** **(Troca de diretório):** falha que permite aos usuários acessar diretórios e executar comandos fora do local especificado;
- ***Race Condition*** **(Condição de corrida):** vulnerabilidade de segurança causada por problemas na sincronização ou tempo de manipulação dos dados. Ocorre quando um *software* comete erros de sincronia ou sequenciamento de ações;
- ***Cross-Site Scripting*** **(Injeção de script no servidor):** também conhecida pela sigla XSS, esta falha envolve sistemas web e sua causa é alguma falha de segurança que permite ao "invasor" acesso indevido a recursos ou informações do site, injetando códigos maliciosos no sistema e que podem afetar todos os usuários que estejam visualizando o site;
- ***Cryptographic Weakness*** **(Fraqueza criptográfica):** uso insuficiente ou incorreto de criptografia na proteção aos dados;

- ***Weak Authentication* (Autenticação fraca):** códigos ou controles insuficientes para determinar a autenticidade do usuário que já está *logado* no sistema;
- ***Inappropriate Permission* (Permissão incorreta):** acesso a dados ou recursos que exigiriam um *login* e senha, sem estar *logado* no sistema, acessando dados confidenciais sem estar credenciado;
- ***Ineffective Secret Hiding* (Segredo mal escondido):** falha causada pela insuficiente ou incorreta proteção de chaves e senhas do sistema;
- ***Unlimited Resource Consumption* (Consumo ilimitado de recursos):** também conhecido por DoS, é um erro causado pela não verificação ou limitação dos recursos permitindo que um invasor esgote os recursos disponíveis;
- ***Incorrect / No Error Messages* (Uso incorreto ou ausência de mensagens de erro):** como o próprio nome já diz, falha causada pela ausência ou incorreta utilização das mensagens de erro;
- ***Incorrect / No Pathname Canonicalization* (Caminho canônico incorreto ou ausente):** este é um erro de caminho ou acesso a um arquivo ou recurso causado pela passagem de um usuário mal intencionado pelas restrições de localização ou de nomes;
- ***Other* (Outro):** alguma falha de segurança não reportada acima.

As falhas citadas acima são bastante comuns e citadas em diversas literaturas a respeito da segurança da informação. Os efeitos causados pelas falhas supracitadas pela Microsoft (2014) são reportados na próxima seção.

Ainda nesse contexto, segundo Ramsaroop (2003) há quatro tipos comuns de ameaças que podem ser causadas por criminosos virtuais que podem resultar em crime virtual, sabotagem, terrorismo virtual ou guerra virtual:

Quadro 2 - Ameaças Virtuais.

Denominação	**Características**
Hackers maliciosos	São aquelas pessoas que invadem sistemas de computadores sem autorização. Estas pessoas são especialmente preocupantes por não serem conhecidos seus propósitos ou identidade.

Código malicioso	São códigos que, assim como os vírus, causam sérios danos e interrupções de alguns aplicativos a redes inteiras, além de possuírem um alto custo para reparação dos danos.
Sabotagem de funcionário ou espionagem	Estes são funcionários que trabalham diretamente com os sistemas e podem causar sérios danos por terem informações e conhecimento detalhado dos sistemas, além de vulnerabilidades técnicas.
Ameaças à privacidade pessoal ou da empresa	São motivos de preocupação, pois os computadores / tecnologias, armazenam diversas informações sobre os empregados, regras de negócio, fornecedores, dados financeiros e outras informações que podem causar danos financeiros e à imagem da empresa.

Fonte: Ramsaroop, 2003.

Como apresentado no **Quadro 2** (acima), Ramsaroop (2003) salienta o quanto os computadores e a tecnologia que suportam os sistemas de informação, precisam receber os cuidados necessários em termos de segurança da informação para que não prejudiquem moralmente ou financeiramente pessoas e instituições, por isso, ainda segundo ele, "as organizações devem evoluir suas metodologias para suportar essas ameaças e prover garantia de eficácia, consistência e continuidade nas informações".

2.1.2 Efeitos

A Microsoft (2014) faz uso da sigla STRIDE para classificar efeitos decorrentes das falhas de segurança de uma aplicação. Neste acrônimo cada letra corresponde à um efeito:

- ***S – Spoofing* (Falsificação de identidade):** se trata da falsificação de identidade de um usuário;

- ***T - Tampering* (Adulteração de dados):** quando ocorre modificação de integridade da informação ou do sistema;
- ***R - Repudiation* (Repúdio):** é a repudiação ou negação de execução de ato cometido previamente por um usuário;
- ***I – Information disclosure* (Revelação de informações):** quando alguma informação é divulgada indevidamente;
- ***D – Denial of service* (Negação de serviço):** é a interrupção do serviço devido ao excesso de requisições ou informações;
- ***E – Elevation of privilege* (Elevação de privilégio):** é quando um usuário eleva suas permissões ou privilégios indevidamente.

Os efeitos citados acima - conforme a Microsoft (2014) - podem produzir impactos de baixo a alto valor, dependendo da aplicação e sistema operacional em que o *software* estiver sendo executado. Levando desde a simples reinstalação de um *software* em uma máquina de uso pessoal a incidentes que colocam a vida humana em perigo, como por exemplo, a perda de geração de energia elétrica em algum equipamento médico-hospitalar. Pelos efeitos observados, é possível perceber a importância de um controle adequado desses efeitos indesejados já que dependendo da aplicação as consequências podem ser catastróficas.

3 PADRÕES DE SEGURANÇA

Conforme Ferreira e Araújo (2008), atualmente há diversas metodologias e melhores práticas em segurança de informação. Esta necessidade surge devido às dificuldades encontradas pelas empresas de tecnologia em atualizar seus modelos e estruturas de controle, já que as tecnologias estão sempre mudando. Para auxiliar essas empresas, há padrões reconhecidos mundialmente como a NBR ISO/IEC 27001, NBR ISO/IEC 27002 e ISO/IEC 15408. Nas próximas seções secundárias são apresentados os padrões mais conhecidos.

3.1 NBR ISO/IEC 27001 e NBR ISO/IEC 27002

As normas juntas são consideradas o mais completo padrão de gerenciamento da Segurança da Informação, pois permitem criar um sistema de gestão de segurança baseado em controles definidos por normas e boas práticas internacionais. A norma já foi chamada BS

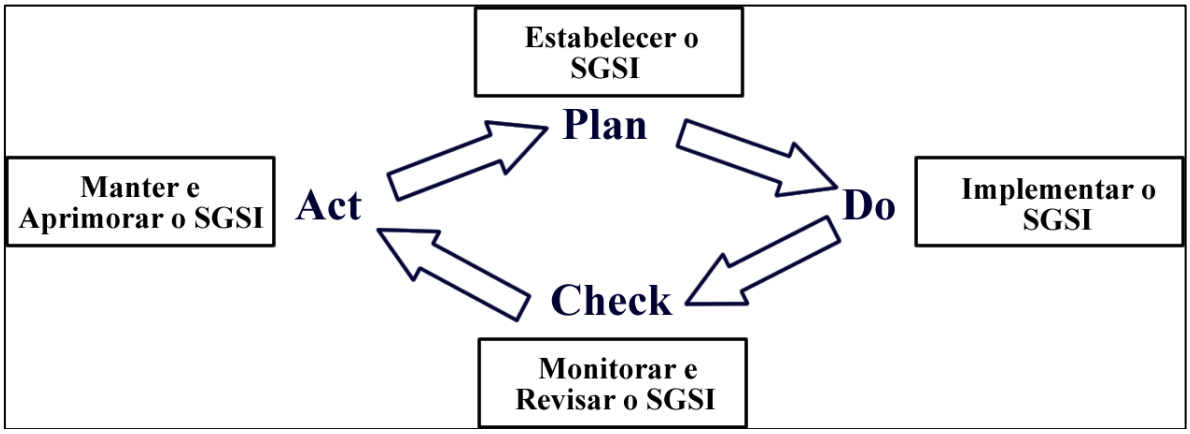
7799 (pois foi inicialmente criada pela British Standard Institute), ISO/IEC 17799 (adotada pela ISO) e posteriormente adotada pelo Brasil em 2001 com o nome NBR ISO/IEC 17799 por meio da ABNT (Associação Brasileira de Normas Técnicas), recebendo uma atualização em junho de 2005, recebendo novos capítulos e chamando-se, finalmente, NBR ISO/IEC 27002, também conhecida como BS7799 parte 1, já a parte 2 recebe outro nome: NBR ISO/IEC 27001 e abrange o ciclo PDCA (*Plan-Do-Check-Act*) (FERREIRA; ARAÚJO, 2008). Conforme a própria norma, seu objetivo é estabelecer "diretrizes e princípios gerais para iniciar, implementar, manter e melhorar a gestão da segurança da informação em uma organização" (ABNT NBR ISO/IEC 27002, 2005).

Segundo o TCU (2008), a estrutura da norma NBR ISO/IEC 27002 está distribuída da seguinte forma:

1 – Escopo;
2 – Termos e definições;
3 – Estrutura da norma;
4 – Avaliação e tratamento dos Riscos;
5 – Política de Segurança da Informação;
6 – Organizando a Segurança da Informação;
7 – Gestão de Ativos;
8 – Segurança em Recursos Humanos;
9 – Segurança física e do ambiente;
10 – Gestão das operações e comunicações;
11 – Controle de acesso;
12 – Desenvolvimento e manutenção de sistemas;
13 – Gestão de incidentes;
14 – Gestão da continuidade dos negócios;
15 – Conformidade.

Conforme Ferreira e Araújo (2008), aplicando os conceitos de cada um dos tópicos citados acima da norma em uma instituição, teremos como resultado um ISMS (*Information Security Management System*). O ISMS ou SGSI (Sistema de Gestão em Segurança da Informação) é a aplicação planejada de objetivos, diretrizes, políticas, procedimentos, modelos e outras medidas que em conjunto reduzem os riscos para Segurança da Informação. Um ISMS pode ser construído de acordo com o Ciclo PDCA, abordado na segunda parte 2 da BS 7799 ou NBR ISO/IEC 27001, conforme a Figura 1:

Figura 1 - Etapas do ISMS.



Fonte: Adaptado de Ferreira e Araújo, 2008.

A Figura 1 acima representa as etapas do Ciclo PDCA, e que segundo Ferreira e Araújo (2008) podem ser implementados conforme o quadro abaixo (Quadro 3):

Quadro 3 - Ciclo PDCA, implementação de um SGSI.

Fase I – Plan	**Fase II – Do**	**Fase III – Check**	**Fase IV – Act**
• Estruturação do SGSI; • Plano Diretor de Segurança; • Diagnóstico de Segurança; • Avaliação, Tratamento dos Riscos e Seleção dos Controles de Segurança; • Declaração de Aplicabilidade (Statement of Applicability).	• Comitê de Segurança da Informação; • Política de Segurança; • Classificação da Informação; • Plano de Continuidade dos Negócios e de TI; • Treinamento e Conscientização; • Implementação dos Controles Especificados na Declaração de	• Monitoração dos Controles de Segurança; • Gestão de Incidentes; • Revisão do nível de risco residual; • Auditoria Interna do SGSI.	• Implementação de melhorias; • Ações Corretivas e Preventivas; • Comunicação das Ações e Resultados para Alta Administração e Partes Interessadas; • Assegurar que as Melhorias foram Implementadas e Atenderam as Expectativas.

	Aplicabilidade.		

Fonte: Ferreira e Araújo, 2008.

Portanto, com a aplicação dos itens do Quadro 3 (acima), Ferreira e Araújo (2008) garantem, basicamente, que os ativos estão protegidos, que há um gerenciamento dos riscos e os objetivos de controle estão implementados, formando assim, um SGSI. Há também a certificação nas normas, que podem ser conquistadas por empresas que buscam uma espécie de "atestado" de que a empresa cumpre as boas práticas de Segurança da Informação.

Por fim, a norma NBR ISO/IEC 27001 traduzida em português custa R$ 113,00 e a NBR ISO/IEC 27002 (também traduzida) custa R$ 209,00 conforme o site oficial para compra das normas, o ABNT CATÁLOGO (2015). Esse valor é irrelevante se considerarmos tamanhas melhorias de Segurança de Informação que a aplicação das técnicas pode proporcionar as empresas, bastar saber se o valor gasto para aplicar a norma é viável e se ela é a melhor opção para determinado tipo de empresa.

3.2 ISO/IEC 15408 (*Common Criteria*)

A ISO/IEC 15408 é uma norma criada a partir de um documento padrão do mercado, o *Common Criteria for Information Technology Security Evaluation* (Critério Comum para Avaliação de Segurança de Tecnologia da Informação), por isso é também conhecida como *Common Criteria*. Esta norma fornece um conjunto de critérios fixos que permitem garantir a segurança da aplicação através de especificações de segurança características do ambiente da aplicação. (ALBUQUERQUE; RIBEIRO; 2002)

A ideia do *Common Criteria* (CC), segundo Albuquerque e Ribeiro (2002), é de que se desenvolva um sistema seguindo as normas, depois o teste em um laboratório credenciado e por fim um selo seja emitido garantindo que aquela aplicação atende os requisitos de segurança. O CC especifica que para a aplicação ser considerada segura, os requisitos de segurança, (chamado *Security Target*) precisam existir e serem alcançados. Além do *Security Target*, há também o *Protection Profile* (perfil de proteção) que se difere do anterior por não estar restrito a uma aplicação específica, podendo ser aplicado a toda uma classe de sistemas.

O CC define sete níveis de garantia de segurança (os EAL – *Evaluation Assurance Level*), porém só os quatro primeiros níveis, EAL-1 ao EAL-4, são considerados pela norma ISO/IEC 15408 já que os outros três são tão rígidos que são considerados

inviáveis, sendo essa a única diferença prática entre os dois documentos (ALBUQUERQUE; RIBEIRO; 2002). Conforme o site da própria norma, o Common Criteria Portal (2015), os 7 níveis EAL são:

1 – EAL 1: Funcionalmente testado;
2 – EAL 2: Estruturalmente testado;
3 – EAL 3: Metodicamente testado e verificado;
4 – EAL 4: Metodicamente projetado, testado e verificado;
5 – EAL 5: Semi-formalmente projetado e testado;
6 – EAL 6: Semi-formalmente projetado, testado e verificado;
7 – EAL 7: Formalmente projetado, testado e verificado.

À data deste trabalho, o site da norma (Common Criteria Portal, 2015) declarava que 1955 produtos de diversos setores diferentes já tinham sido certificados por esta norma, dentre eles sistemas e aparelhos biométricos, sistemas e dispositivos de controle de acesso, bases de dados, sistemas operacionais, produtos para certificação digital, dispositivos multifuncionais e etc. Isso mostra que apesar de ser uma norma rigorosa, é aplicável a diferentes aplicações, consolidando-se como um dos padrões de segurança mais utilizados no mercado.

4 CODIFICAÇÃO SEGURA

Este capítulo é dedicado à apresentação de técnicas que visam diminuir ou eliminar falhas de segurança comuns em aplicações web conforme Howard e LeBlanc (2005). São 9 seções que abrangem diversos tipos de ameaças e que apresentam técnicas que se implementadas ajudam a garantir um *software* mais seguro. Todos os códigos apresentados estão na linguagem de programação C++.

4.1 ESTOURO DE BUFFER

Segundo Howard e LeBlanc (2005), o estouro de *buffer* é um problema muito comum causado por práticas ruins de codificação e que pode causar danos financeiros (e morais) enormes. A primeira linha de defesa contra ele é escrever um código sólido, ou seja, bem pensado, bem codificado e bem testado, embora pareça uma dica muito superficial, impedir estouro de buffer é uma questão de escrever um aplicativo seguro. Assim deve-se

validar as entradas de dados e todas as informações que são passadas como parâmetros para as funções do sistema devem ser vistas como entradas hostis. Nada das funções além das entradas e saídas devem ser acessíveis fora dela.

Um exemplo de código que permite o estouro de *buffer* é o da Figura 2 abaixo:

Figura 2 - Estouro de *Buffer* I

```
void printLine (const char* msg){
    char buf[255];
    sprintf(buf, "Prefix %s suffix\n", msg);
    ...
    ...
}
```

Fonte: Howard e LeBlanc, 2005.

No exemplo acima, percebe-se que o programador acredita conhecer o comprimento do *buffer*, porém se algum outro programador menos cuidadoso fizer modificações na função, é possível que haja um estouro, além deste tipo de construção só trazer problemas. Como sugestão, o código abaixo seria indicado:

Figura 3 - Estouro de *Buffer* II.

```
#ifdef_DEBUG
    memset(dest, 'A', buflen); //buflen = tamanho em bytes
#endif
```

Fonte: Howard e LeBlanc, 2005.

No código mencionado, se alguém chamar a função passando um argumento "não esperado" para a função, o código irá falhar exibindo o local do erro rapidamente e facilitando a correção do bug sem depender de uma cobertura completa de testes.

Outra prática recomendada por Howard e LeBlanc (2005) é o tratamento de *strings.* Considerando que é o estouro de *buffer* mais comum, uma revisão das funções comumente utilizadas é indicada, trocando sempre que possível funções inseguras ou depreciadas pelas mais recentes e seguras. Outras dicas mais específicas são:

- Validar o tamanho das entradas de informação conferindo se o tamanho especificado persiste mesmo depois do envio dos formulários;

- Cuidar com funções que retornam o tamanho da informação, pois fazendo uso delas, permitirá que um estouro de *buffer* ocorra devido a um tamanho excessivamente grande dos dados;
- Indicar erros o mais próximo possível da origem, pois permite que a depuração seja mais fácil, além de evitar que um erro "oculto" transforme-se em uma mensagem de erro em um local inesperado, fato que sempre que possível deve ser evitado.

4.2 EXECUTAR COM O MÍNIMO DE PRIVILÉGIOS

Conforme Howard e LeBlanc (2005) no campo da segurança, quanto menos privilégios mais segurança. Toda ação deve ser executada com o mínimo de permissões possíveis, e se for dado acessos mais privilegiados que ele dure somente o tempo necessário à correta execução da funcionalidade, reduzindo a área de exploração dos possíveis invasores.

O vírus e o cavalo-de-Tróia (programa que possui um código malicioso oculto) são dois "vilões" que podem ser evitados em uma aplicação pelo simples fato de não executar um programa como administrador. Vírus conhecidos como *Back Orifice*, *SubSeven*, *FunLove* e *ILoveYou* só conseguem se proliferar ou alcançar seu risco máximo quando o usuário executa os programas infectados com nível de administrador. Abaixo vemos duas principais dicas para esta seção importante da segurança:

- Abra recursos com as permissões que são requeridas, nada mais – exemplo: se for necessário ler um dado do registro do computador, solicite acesso de apenas leitura;
- Não gravar dados do usuário em locais protegidos como, por exemplo, a pasta *C:\Arquivos de Programas* ou *C:\Windows*, no caso do Sistema Operacional Windows 8.1. Dados do usuário devem ser armazenados – sempre que possível – no diretório de perfil do usuário como, por exemplo, *C:\Users\NomeDoUsuario* (Windows 8.1).

4.3 CRIPTOGRAFIA

A criptografia, segundo Howard e LeBlanc (2005), é algo muito importante para garantir a privacidade e integridade dos dados bem como deixa mais forte a autenticação, porém, ela por si só não diminuirá os erros de programação que levam à falhas de segurança. Por isso, não basta utilizar criptografia, é preciso saber escrever códigos que permitam à criptografia alcançar seu grau máximo de segurança. Neste último quesito, há diversas dicas de desenvolvimento, dentre elas destacam-se:

- Utilizar números aleatórios ruins leva à previsibilidade das chaves, por isso, a regra para gerar bons números é gerar números uniformemente distribuídos, imprevisíveis e que possuam uma grande abrangência de números diferentes e que permita incluir todos eles, o chamado "ciclo longo e completo". A função *rand* (C++) não é indicada para gerar números aleatórios por possuir congruência linear, ou seja, um comportamento previsível;
- Criar funções próprias para criptografia não é algo recomendável. Isso pode criar o risco de alguém mais expert decifrar os códigos e destruir a segurança do sistema;
- Bastante atenção ao gerenciamento das chaves, este é o elo mais fraco de aplicativos que utilizam criptografia, por exemplo, uma chave codificada em uma imagem executável é fácil de quebrar, mesmo sem acesso ao código fonte;
- Utilize o número adequado de caracteres para chave criptográfica. Quanto mais curta mais fácil de atacar. É importante lembrar que atacar chaves simétricas como DES e RC4 exige que o invasor tente todas as chaves, no entanto chaves RSA (cifra assimétrica) exige que o invasor tente valores aleatórios criados para criar chaves públicas e privadas – o processo chamado de fatoração. A chave simétrica não é mais ou menos segura que a assimétrica, a diferença é que elas são atacadas de formas diferentes e o mais importante é seu tamanho em bits (número de caracteres), quanto maior, mais segura;
- Outra dica importante, porém não relacionada à codificação: nunca é recomendado dizer que a empresa ou o sistema possui "segurança inquebrável" ou "criptografia de qualidade militar", isso pode ser um tiro no pé caso venha a ter algum problema de segurança, levando a empresa a adquirir uma má reputação.

Para identificar quais são as principais ameaças à criptografia e como combatê-la, pode-se utilizar o Quadro 4 abaixo:

Quadro 4 - Soluções criptográficas comuns contra ameaças.

Ameaça	Técnica de atenuação	Algoritmos de exemplo
Revelação de informações	Criptografia de dados utilizando uma cifra simétrica.	RC2, RC4, DES, 3DES, AES.
Adulteração	Integridade de dados e de mensagem utilizando funções de hash, MACs ou assinaturas digitais.	SHA-1, SHA-256, SHA-384, SHA-512, MD4, MD5, HMAC, assinaturas digitais RSA, assinaturas digitais DSS, XML DSig.
Spoofing	Autenticar dados provenientes do remetente.	Certificados de chave pública e assinaturas digitais.

Fonte: Howard, LeBlanc, 2005.

No Quadro 4 acima, os algoritmos utilizados derivam da necessidade de segurança de cada aplicação, podendo um algoritmo ser ótimo para evitar a revelação de informações mas fraco contra adulteração. Tudo depende da aplicação em que a criptografia será utilizada. Por isso é preciso ter em mente quais objetivos de segurança precisam ser alcançados para cada sistema (HOWARD; LEBLANC, 2005).

4.4 PROTEGENDO DADOS CONFIDENCIAIS

Segundo Howard e LeBlanc (2005), os dados secretos possuem um vilão principal: as contas de administrador dos sistemas operacionais. Com esse privilégio as ameaças à confidencialidade (e a integridade) se tornam muito maiores. Para vencer essa batalha, os mencionados desenvolvedores da Microsoft enumeram alguns pontos principais:

- Muitas vezes um segredo não precisa ser algo codificável e descodificável. Senhas de usuários no banco de dados, por exemplo, devem ser preferencialmente um

hash armazenado (md5, sha1, sha2 e etc.), isso permite que o sistema mesmo que comprometido não permita recuperar os segredos, exceto por *brute force* (chamado de "força bruta", consiste em testar milhões de vezes as senhas com caracteres aleatórios até acertá-lo);

- Para criar códigos *hash* mais aprimorados, pode-se utilizar o *sal*. O *sal* (também chamado de s*alting*, em português "salgar") se trata de algum número aleatório ou caracteres adicionados aos dados do *hash*, criando segredos ainda mais difíceis de serem descobertos. Para linguagem C++ é recomendada a *CryptoAPI* para criar *hashes* com *sal* pois torna essa tarefa extremamente fácil;
- Em Sistemas Operacionais Windows (2000 e posteriores), por meio da linguagem de programação C++, é possível utilizar as funções *CryptProtectData* e *CryptUnprotectData* da *Data Protection API* (DAPI). A mesma permite que somente usuários com perfil de acesso administrador ou contas normais tenham certos privilégios no sistema, por exemplo, de escrita, e além disso adiciona automaticamente um código de verificação de integridade (MAC – message authentication code) aos dados criptografados, permitindo que a integridade dos dados seja verificada. Ao utilizar *CryptProtectData e CryptUnprotectData*, pode-se enviar uma senha adicional ao parâmetro *pOptionalEntropy*, como o usuário e senha do usuário do Windows, permitindo que apenas aquele usuário na rede/computador faça determinadas ações no sistema;
- Há três bases para construir aplicativos que armazenam segredos: a segurança relativa, o esforço de desenvolvimento e a facilidade de instalação. Investir mais em segurança durante o desenvolvimento diminui os custos de manutenção do *software*. O **Quadro 5** (abaixo) apresenta o esforço de desenvolvimento, segurança relativa e a facilidade de instalação das opções de segurança à dados secretos.

Quadro 5 - Relações de troca ou compensação para proteger dados secretos.

Opção	Segurança Relativa	Esforço de desenvolvimento	Facilidade de instalação
Arquivos de configuração (nenhuma criptografia, somente por comparação de informações).	Nenhuma.	Baixo.	Alta.

Segredos incorporados ao código.	Nenhuma.	Baixo.	Média.
DPAPI (máquina local).	Alta.	Médio.	Baixa.
DPAPI (dados de usuário).	Alta.	Médio.	Média.

Fonte: Adaptado de Howard, LeBlanc, 2005.

Como elucida o Quadro 5 (acima), quanto maior o esforço de desenvolvimento e menor a facilidade de instalação, maior é a segurança relativa de um sistema, portanto cabe a cada sistema adequar-se a sua necessidade de proteção à dados secretos, já que em alguns casos tamanha segurança e esforço de desenvolvimento não faz jus à necessidade do *software*.

4.5 ENTRADAS MAL-INTENCIONADAS

"Toda entrada é mal-intencionada até que se prove o contrário", essas palavras de Howard e LeBlanc (2005) deixam claro o quão importante é validar todas entradas de dados de uma aplicação, ou seja, todos os dados inseridos pelos usuários são dados não-confiáveis. Isso não significa dizer que sempre haverá pessoas mal-intencionadas utilizando o aplicativo, até mesmo usuários "honestos" podem incluir entradas que levam a uma exceção no sistema.

Para elucidar a importância da validação dos dados, um exemplo de estouro de *buffer* pode ser dado. O estouro de *buffer* ocorre por três razões: dados inseridos de fonte não confiável (possivelmente um invasor), confiança excessiva no formato em que os dados deveriam ser inseridos (tamanho do *buffer*) e em decorrência de um evento potencialmente arriscado em que o *buffer* não confiável é copiado para memória. Para exemplificar, pode-se utilizar o código C++ da Figura 4 abaixo:

Figura 4 - Confiança no *buffer*.

```
void CopyData(char *szData) {
    char cDest[32];
    strcpy(cDest,szData);
    ...
    //utilize cDest
}
```

Fonte: Howard e LeBlanc, 2005.

O código acima está perfeito, tudo depende de como *CopyData* é chamado e se *szData* é um dado confiável. Por exemplo, o seguinte código é seguro para função acima, já que é um dado confiável (criado diretamente pelo sistema):

Figura 5 - Código confiável para *CopyData*.

```
char *szNames[] = {"Michael","Cheryl","Blake"};
CopyData(szNames[1]);
```

Fonte: Howard e LeBlanc, 2005.

O código acima é seguro já que os nomes são declarados diretamente, não excedendo o limite de 32 caracteres definidos na função da Figura 4. Mas se o argumento da função *CopyData*, o *szData*, fosse criado por uma origem não confiável (o usuário, por exemplo) então *CopyData* copiaria os dados até alcançar um caractere nulo – o que causaria um estouro de *buffer* em *cDest* caso o tamanho excedesse 32 caracteres. Portanto, o código citado não é seguro em meio não confiável. Para corrigi-lo seria necessário seguir o código abaixo:

Figura 6 - Função *CopyData* mais segura.

```
void CopyData(char *szData, DWORD cbData) {
    const DWORD cbDest = 32;
    char cDest[cbDest];

    if(szData != NULL && cbDest > cbData)
        strncpy(cDest,szData,min(cbDest,cbData));

    ...
    //utilize cDest
}
```

Fonte: Howard e LeBlanc, 2005.

O código apresentado acima é muito mais seguro, pois limita a quantidade de dados copiados a *cDest* impedindo que um estouro de *buffer* ocorra naquele ponto. Portanto a lição é fazer o código mais seguro – Figura 6 – desde o início, pois mais cedo ou mais tarde o código da Figura 4 precisaria ser corrigido, além de evitar todo o transtorno causado pelo estouro do *buffer* e a corrida desenfreada para encontrar o código com problemas.

Há diversos meios de combater as entradas mal intencionadas. Dentre as principais formas pode-se mencionar:

- Definir um limite de confiança ao aplicativo – após ultrapassar uma bateria de validações, pode-se considerar que os dados dentro deste limite são confiáveis;

- Criar um ponto de estrangulamento de entrada, ou seja, os dados que não passaram nas validações ficam fora da área confiável;
- Procurar sempre por dados válidos, e não inválidos. Há duas razões para isso: pode haver mais de uma maneira válida de representar os dados e um padrão inválido pode ser aceito. Exemplo: ao invés de procurar pelas extensões de arquivo *exe*, *bat* ou *cmd* para determinar se um arquivo é perigoso, procure se o arquivo se enquadra dentro do necessário, por exemplo, *txt*, *gif* e *jpg*;
- Utilize expressões regulares para validar dados e literais mais complexos.

4.6 REPRESENTAÇÃO CANÔNICA

A representação canônica é a maneira mais direta e menos ambígua de representar algo. Segundo Howard e LeBlanc (2005), "a canonicalização é o processo por meio do qual várias formas equivalentes de um nome são resolvidas em um único nome-padrão – o nome canônico". Isto é, o arquivo "teste.php" pode estar sendo chamado em algum lugar do sistema de "../temp/teste.php" e em outro "../teste.php", porém todos fazem referência ao mesmo arquivo, o "C:/temp/teste.php". Este último é a representação canônica do arquivo, enquanto que os outros são as representações não canônicas. Quando o sistema toma decisões erradas com base em uma representação não canônica os *bugs* de segurança relacionados acontecem.

Falhas de canonicalização podem permitir acesso a arquivos não permitidos, a chamada "vulnerabilidade de revelação". Por exemplo, o Mac OS X da Apple em sua primeira versão utilizava o sistema de arquivos HFS+ que não fazia distinção entre nomes maiúsculos e minúsculos, porém o servidor Apache Web (instalado por padrão) fazia distinção e protegia diretórios baseado no nome do arquivo. Resultado: um caminho bloqueado pelo Apache chamado "/scripts/" (em minúsculo), ao ser digitado como "/SCRIPTS/" (em maiúsculo) via HFS+ o acesso ao diretório era liberado.

Se tratando de um erro de codificação que pode ser grave, Howard e LeBlanc (2005) citam as seguintes dicas de canonicalização para não deixar a aplicação falhar por este motivo:

- Um aplicativo nunca deve tomar decisões de segurança baseado no nome de um arquivo;

- Nunca utilizar palavras em sua representação “normal” de caracteres para comparar com *Strings* indesejadas. Essas palavras podem ser escritas de outra maneira e burlar a segurança. Por exemplo, comparar o valor “remover” para o parâmetro “executar” de uma URL (“pagina1.php?executar=remover”). Se um usuário mal intencionado passar o valor “%72emover” para o parâmetro “executar”, onde “%72” é a representação hexadecimal da letra “r”, o servidor *web* interpretaria como “remover” enquanto que a comparação de literais não encontraria o valor “remover” para o parâmetro “executar”, burlando a segurança do sistema e liberando um acesso proibido. Esse escape em hexadecimal é apenas uma dentre várias representações de caracteres “normais”, só para citar outros exemplos, há também a codificação de largura variável no UTF-8, codificação UCS-2 Unicode e a codificação dupla;
- Ao invés de comparar literais problemáticos, as expressões regulares devem ser utilizadas para restringir o que é permitido em um nome e rejeitar todas as outras solicitações;
- No caso de aplicações ISAPI (Internet Server Application Programming Interface) e filtros ISAPI, provavelmente as tecnologias mais vulneráveis no quesito canonicalização, recomenda-se utilizar a variável de servidor SCRIPT_TRANSLATED no caso do servidor *web* IIS (Internet Information Services), pois ela retorna nomes de arquivos corretamente canonicalizados automaticamente;
- Caminhos completos (absolutos) devem ser utilizados para localizar arquivos. Por exemplo, ao invés de referenciar o arquivo “teste.php” com “../temp/teste.php”, também conhecido como caminho *PATH*, o correto é referenciá-lo como “C:/temp/teste.php” que é o caminho completo do arquivo.

4.7 ENTRADAS NA BASE DE DADOS

Dados provenientes de entradas não confiáveis – os usuários do sistema – costumam ser inseridos na base de dados para registrar informações que agregam valor ao usuário e ao sistema, no entanto, se não tomado os devidos cuidados, grandes problemas de segurança podem ocorrer. Conforme Howard e LeBlanc (2005), scripts de Injeção SQL são os

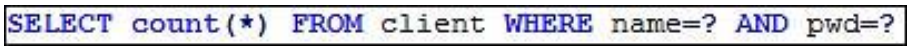
mais comuns nesta seção de estudo e a SQL enviada ao banco de dados deve possuir tratamento especial para evitar ou atenuar as ameaças aos bancos de dados.

Para evitar problemas com as entradas no banco de dados, deve-se seguir as dicas abaixo:

- Como já mencionado na seção 4.2, o perfil do usuário do banco de dados que faz as consultas da aplicação deve possuir o mínimo de privilégios – apenas o necessário à correta execução das instruções;
- Não reescrever os dados de acesso ao banco (usuário e senha) diretamente nos scripts, pois aumentam as chances de um usuário malicioso roubar esses dados;
- Utilizar funções que tratem informações passadas à SQL antes de executá-la, sobrescrevendo caracteres ou combinação de caracteres perigosos, por exemplo, as aspas simples (usadas comumente para fechar condições e acessar dados proibidos);
- *Procedures* (coleção de instruções SQL pré-compiladas) escondem a lógica do aplicativo caso o código seja comprometido, por isso, em aplicações robustas e que regras de negócio transitam pelas consultas, são geralmente bem recomendadas.

A separação das dicas acima para esta é proposital e visa dar maior destaque a utilização da dica mais importante: os comandos parametrizados, também chamados de marcadores de lugar. Conforme Howard e LeBlanc (2005), esta é uma das melhores técnicas (e de fácil utilização) para garantir a segurança das entradas no banco de dados. Comandos parametrizados podem receber os tratamentos indicados nas dicas acima e acrescentam um fator importante: a própria validação dos dados pela base de dados. Uma consulta parametrizada simples pode ser feita da seguinte forma:

Figura 7 - SQL com parametrização.



Fonte: Howard e LeBlanc, 2005.

Como pode-se ver na Figura 7, a instrução retorna o número de registros da tabela "*client*" que satisfazem a condição "name=?" e "pwd=?". Os caracteres de interrogação são os valores das entradas que serão passadas para a SQL posteriormente e que o banco de dados irá

interpretar. Elas são enviadas da seguinte maneira, em PHP com a API (*Application Programming Interface*) **PDO_MySQL**:

Figura 8 - Exemplo de utilização da SQL parametrizada em PHP com PDO_MySQL.

```php
<?php
$bd = new myDB();
$query = $bd->prepare("SELECT count(*) FROM client WHERE name=? AND pwd=?");

$query->bindValue(1, "alexandre", PDO::PARAM_STR);//name
$query->bindValue(2, "12345", PDO::PARAM_STR);//pwd

while($row = $query->fetch()) {
    print_r($row);//imprime o número de registros da tabela client
}
?>
```

Fonte: Autor, 2015.

Como se conclui pela Figura 8, não é complicado utilizar a parametrização de SQL para melhorar a segurança das consultas no banco, por isso, é extremamente recomendável sua utilização.

4.8 ENTRADAS ESPECÍFICAS DA WEB

Provavelmente o local mais perigoso de entradas de dados é também o mais utilizado atualmente, conforme Howard e LeBlanc (2005) o ambiente *web* apresenta riscos maiores que os das aplicações locais (*desktop*), porém, as dicas abaixo podem ajudar os sistemas à serem mais seguros neste ambiente hostil:

- Nunca permita que valores passados pela URL (o caso do valor “meuscript” para o argumento “nome” da URL “www.meusite.com/dados?nome=meuscript”) façam alguma inserção de informações do conteúdo HTML da página;
- Se possível, evite *cookies*, eles são apenas um cabeçalho na solicitação HTTP, portanto, invasores podem fazê-los falsificar informações e acessar dados que somente o usuário conhece. Se utilizá-los, tente ao menos criptografá-los e exigir autenticação aos dados via MAC (*Message Authentication Code*);

- Coloque entre aspas duplas todas as propriedades das *tags* HTML. Muitos ataques XSS partem da exploração de propriedades das *tags* que se inseridas envolvidas por aspas duplas, podem não funcionar ou dificultar o ataque;
- Não utilize a função *eval()* do JavaScript em entradas não confiáveis. Ela permite que praticamente qualquer código possa ser executado, portanto, evite sua utilização ao máximo;
- Por último, não armazene dados sigilosos em *cookies*, campos ocultos ou quaisquer outros dados que possam ser manipulados ou descobertos pelo usuário via exibição do código fonte da página, tráfego dos dados no site e etc.

4.9 INTERNACIONALIZAÇÃO

Esta seção focada na internacionalização, reconhecida pelos caracteres *I18N* (*I* de *internationalization*, 18 caracteres e depois a letra *N*), não trata do assunto em termos gerais restringindo-se a fornecer formas de melhorar a segurança ao utilizá-la.

Conforme Howard e LeBlanc (2005), é importante que ao projetar *softwares* para públicos internacionais siga as seguintes regras:

- Utilizar apenas o padrão de caracteres Unicode. Dentre as três principais representações binárias Unicode (UTF-8, UTF-16 e UTF-32), as duas mais recomendadas são o UTF-8, para protocolos e sistemas *web*, e o UTF-16, para aplicativos da plataforma Windows (é o padrão de codificação utilizado pelo sistema operacional). Utilizá-los deixará o código menos exposto a problemas de conversão e segurança;
- Não criar um conversor próprio de caracteres, deixando esta tarefa para os padrões reconhecidos mundialmente é uma boa prática de segurança para os casos de internacionalização;
- Não converter caracteres Unicode em outros conjuntos de caracteres/páginas de código evita que falhas de segurança e outros *bugs* ocorram com o sistema.

Conforme o site oficial do padrão internacional Unicode, o Unicode Consortium (2015), o repositório de caracteres do *Unicode Common Locale Data Repository* (CLDR), permite a conversão dos caracteres para cerca de 300 linguagens diferentes, ou seja, escrever

um caractere no padrão Unicode, permite que ele seja traduzido para cerca de 300 linguagens diferentes, o que o consolida como padrão de caracteres tanto em termos de segurança como em popularidade.

5 FERRAMENTAS E TÉCNICAS DE AVALIAÇÃO DOS RISCOS

Este capítulo apresenta informações sobre algumas das principais ferramentas do mercado para testar *softwares* web, bem como técnicas para identificação e teste dos riscos de segurança de um sistema.

5.1 Acunetix Vulnerability Scanner

Acunetix é o nome de uma empresa com sede em Malta e Reino Unido e ela é a desenvolvedora do *Acunetix Vulnerability Scanner*, seu *software* pago (possui versão de avaliação por 14 dias) que promete verificar diversas vulnerabilidades em um sistema web. Muito conhecido entre grandes empresas, presta serviços para gigantes como a Sony, NASA, Adidas, Adobe, Cisco, Samsung e a Coca-Cola. Segundo a Acunetix (2015), o *software* é pioneiro na área de testes em aplicações web e é líder em análise e detecção de vulnerabilidades neste meio.

Conforme a Acunetix (2015), o *software* possui como principais pontos:

- Encontrar ameaças de baixo risco através de um avançado scanner que combina técnicas de teste caixa preta (sem acesso ao código fonte) com *feedbacks* colocados dentro do código fonte;
- Analisador automatizado de *JavaScript* que encontra vulnerabilidades em códigos AJAX e aplicações web 2.0;
- Testes intensivos de *SQL Injection* e *Cross-Site Scripting* (XSS);
- Testes em formulários e áreas protegidas;
- Permite procurar falhas em milhares de páginas sem interrupção graças ao seu processamento multitarefa;
- O *software* compreende diversas tecnologias web complexas como REST, SOAP, XML, AJAX e JSON.

Ainda segundo a Acunetix (2015), 70% dos sites possuem vulnerabilidades que permitem acessar / roubar informações sensíveis como cartões de crédito e informações dos usuários já que hackers concentram seus esforços em sistemas baseados na web, como carrinhos de compras, formulários e conteúdos dinâmicos. Com o *software* da empresa, prometem encontrar essas falhas antes dos hackers, fazendo com que as informações e conteúdos vulneráveis sejam protegidos antes dos ataques.

O *software* em sua versão *Entreprise* custa U$ 4.995 (quatro mil e novecentos e noventa e cinco dólares), ou aproximadamente R$ 13.150 (treze mil e cento e cinquenta reais) em sua licença perpétua, já sua licença anual custa U$ 3.195 (três mil e cento e noventa e cinco dólares, ou aproximadamente R$ 8.400 (oito mil e quatrocentos reais). Nas duas versões, está incluso documentação, manuais de instrução dos diversos módulos do sistema e suporte via e-mail ou telefone (ACUNETIX, 2015). Este aplicativo em sua versão de avaliação foi o escolhido para testar o sistema desenvolvido para este trabalho já que é uma aplicação web desenvolvida em PHP com JavaScript, além de o *software* se mostrar de fácil uso. No Capítulo 6, é exibido os documentos produzidos e o resultado dos testes feitos com o *Acunetix Vulnerability Scanner*.

5.2 Testando vulnerabilidades com a ajuda do STRIDE

Esta técnica é descrita por Howard e LeBlanc (2005) e consiste em explorar com simples testes as categorias de ameaça do STRIDE. Como já explicado na seção 2.1.2, o STRIDE é uma sigla que é composta pelas classificações das ameaças de segurança. Essas classificações ajudam a determinar os tipos de testes a serem realizados e o risco das ameaças, priorizando cada tipo de teste. Abaixo segue o Quadro 6 com as técnicas de testes sugeridos:

Quadro 6 - Categorias de teste de ameaças.

Tipo de Ameaça	**Técnicas de teste**
Falsificação de Identidade	• Tentativa de forçar o aplicativo a não utilizar nenhuma autenticação; há uma opção que permita isso, que um não administrador possa configurar? • Tente forçar um protocolo de autenticação a utilizar uma versão legada menos segura? • Você pode visualizar as credenciais de um usuário válido

	na rede ou no armazenamento persistente? • "*Tokens* de segurança" (por exemplo, um *cookie*) podem ser reproduzidos para driblar uma etapa de autenticação? • Tente um ataque de força bruta contra as credenciais de um usuário; há alterações sutis na mensagem de erro que o ajudem a tentar esse ataque?
Adulteração de dados	• Tentativa de driblar a autorização ou mecanismos de controle de acesso; • É possível adulterar e depois refazer o *hash* dos dados? • Crie hashes, MACs e assinaturas digitais inválidos para ver se estão verificados corretamente; • Determine se você pode forçar o aplicativo a rever (roll-back) para um protocolo inseguro se o aplicativo utilizar um protocolo resistente a adulteração como SSL/TLS ou IPSec.
Repúdio	• Há condições que impedem o registro em *log* ou a auditoria? • É possível criar solicitações que criam dados incorretos em um log de evento? • Ações sigilosas podem ser realizadas de tal maneira que driblem as verificações de segurança?
Revelação de informações	• Tentem acessar dados que só podem ser acessados por usuários mais privilegiados. Isso inclui dados persistentes (dados baseados em arquivos, dados no registro, etc.) e dados trafegados na rede. Os *Sniffers*, ou analisadores de rede, são uma ferramenta útil para encontrar esses dados; • Mate o processo e realize uma pesquisa no disco procurando dados sigilosos que nele são gravados. Você talvez precise solicitar que os desenvolvedores marquem seus dados como sigilosos com um padrão comum em uma distribuição de depuração para localizar os dados facilmente; • Faça o aplicativo falhar de uma maneira que ele exponha

	informações úteis a um invasor. Por exemplo, mensagens de erro.
Negação de serviço (DoS)	• Inunde um processo com um número tão grande de dados que ele pare de responder a solicitações válidas; • Dados malformados travam o processo? Isso é especialmente ruim nos servidores; • Influências externas (como pouco espaço em disco, pressão por memória e limitações de recurso) podem forçar o aplicativo a falhar?
Elevação de privilégio	• Invista a maior parte do tempo em aplicativos que são executados sob contas elevadas, como serviços SYSTEM; • Você pode executar dados como código? • Um processo com privilégios elevados pode ser forçado a carregar um *shell* de comandos que, por sua vez, será executado com privilégios elevados?

Fonte: Howard e LeBlanc, 2005.

O Quadro 6 (acima) não descreve como implementar o teste, apenas enumera linhas gerais de testes para cada categoria do STRIDE. Muito útil para orientar ou descobrir quais são pontos de maior risco e que precisam ter cuidado dobrado em um sistema bem como para avaliar de modo geral a segurança de uma aplicação. Alguns pontos desta tabela também foram usados para testar o sistema deste trabalho garantindo sua presença nos resultados dos testes no Capítulo 6.

6 ESTUDO DE CASO

Neste trabalho é apresentado o sistema *web* desenvolvido utilizando a linguagem de programação PHP. O *software* é um sistema de Notas que permite criar, editar e excluir notas – tanto pessoais quanto profissionais, exigindo um usuário e senha que seriam previamente cadastrados (primeira técnica de segurança). O sistema foi testado em um servidor *web online*, efetuando testes de segurança com o uso de *softwares* ou técnicas descritas no Capítulo 5.

6.1 Tecnologias Utilizadas

O sistema web foi desenvolvido em PHP 5.5 utilizando JavaScript, CSS 3, HTML 5 e os frameworks jQuery e Bootstrap. A plataforma em que o sistema está rodando é um servidor *web* Apache 2.2 em um Sistema Operacional Linux CentOS 6, conforme a Figura 9 abaixo:

Figura 9 - Tecnologias Utilizadas.

Fonte: Adaptado pelo Autor, 2015.

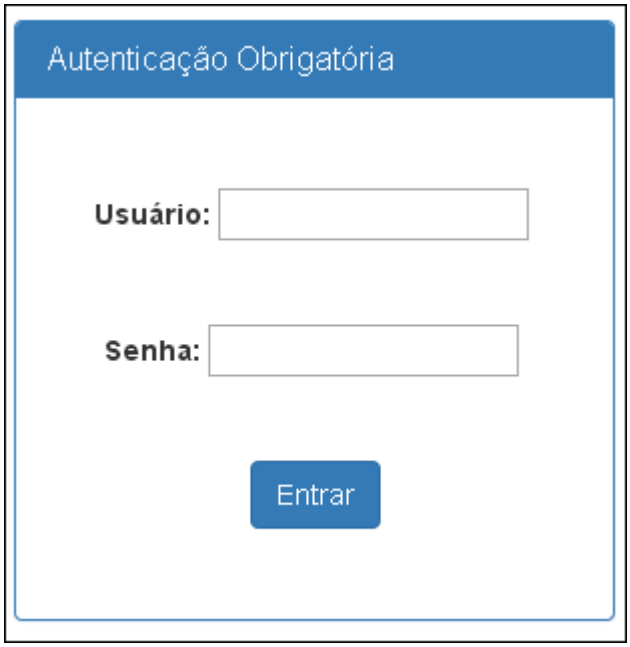
As tecnologias da imagem acima (Figura 9) foram usadas de forma livre, de maneira que o mais importante não foi seguir diretrizes de uma tecnologia específica, mas sim uma metodologia criada a partir da junção das dicas de segurança com as normas de segurança estudadas.

6.2 Interfaces

Nesta seção são apresentadas todas as telas do sistema, explicando qual ação o usuário deve fazer em cada uma delas e qual comportamento ocorre de acordo com as escolhas do usuário ou falhas no sistema.

O sistema de notas é restrito, por isso, não há um local para cadastro de usuários, eles são inseridos pelo administrador do sistema e os usuários ao digitar o endereço *web* do mesmo, encontram diretamente a tela de *login*, conforme abaixo:

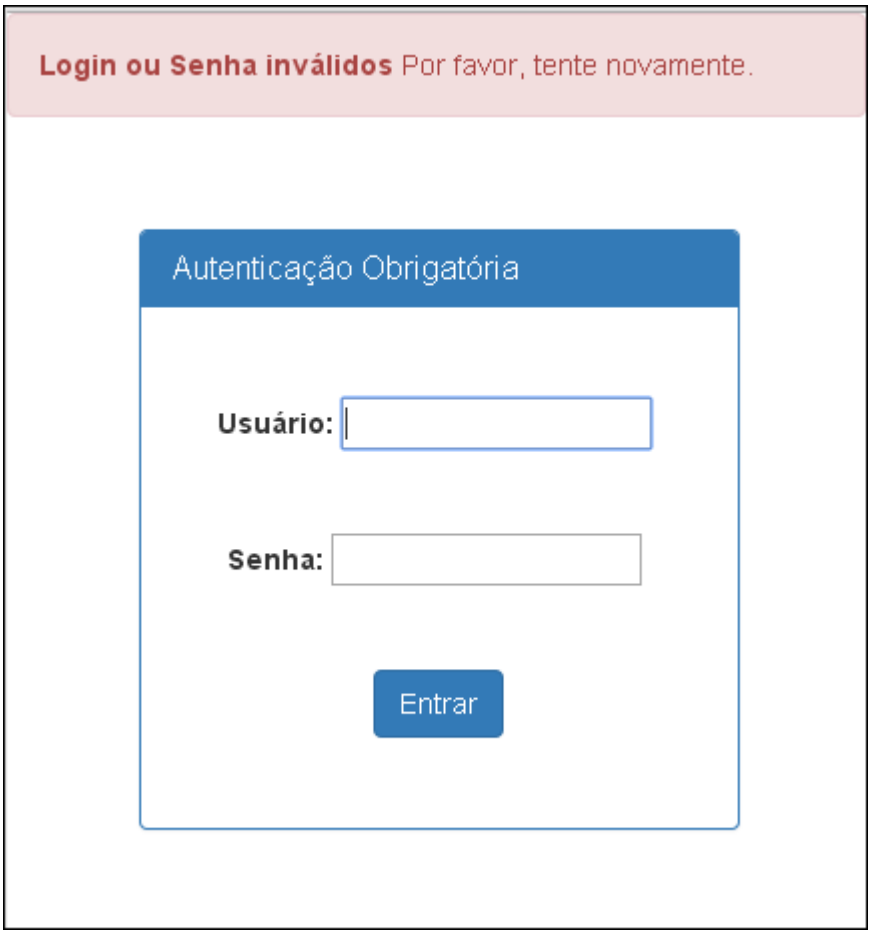
Figura 10 - Tela inicial do sistema.



Fonte: Autor, 2015.

Neste momento, o usuário deve digitar as informações de Usuário e Senha, e caso clique no botão “Entrar” sem preencher os dados, o sistema exibe uma *popup* explicando o “Você precisa preencher o campo X”, onde X é o nome do campo que não foi preenchido. Caso a combinação de *login* e senha seja inválida, é disparada a seguinte mensagem de erro de autenticação:

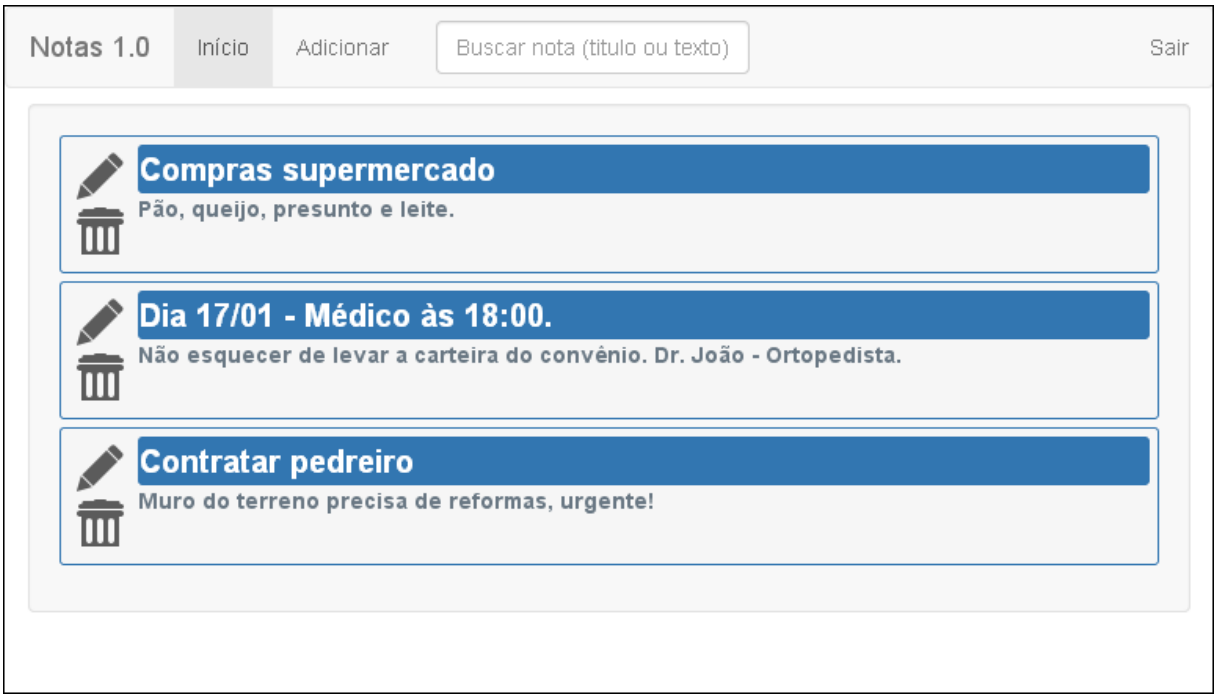
Figura 11 - Problema de autenticação no sistema.



Fonte: Autor, 2015.

Caso o usuário insira uma combinação correta de *login* e senha, o sistema é redirecionado para a tela inicial de notas:

Figura 12 - Tela inicial das notas.



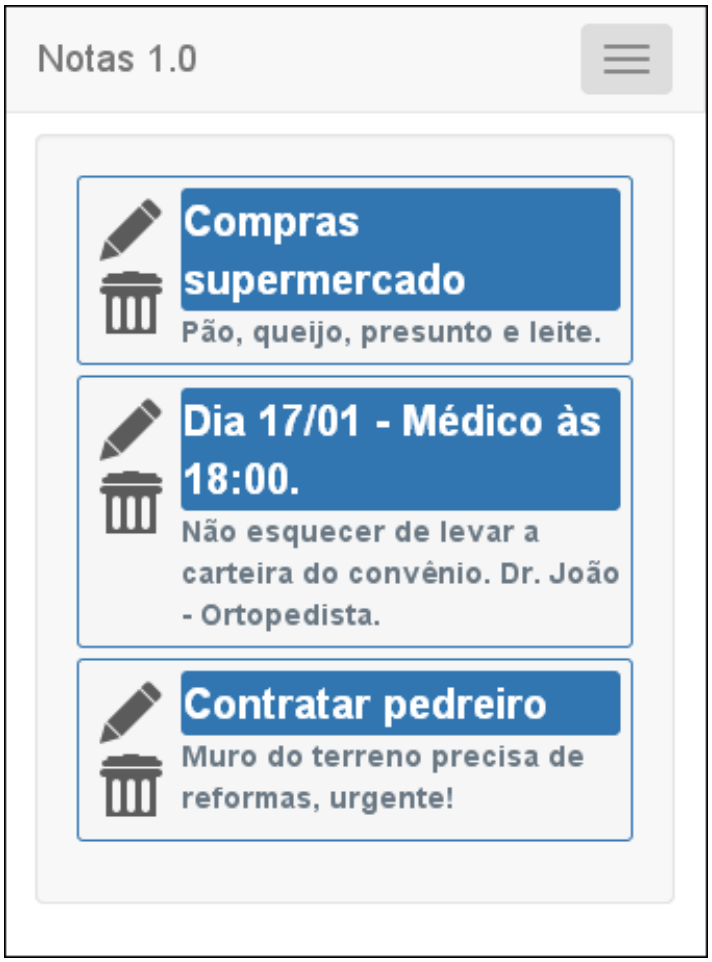
Fonte: Autor, 2015.

Como apresentado na Figura 12 (acima), a tela inicial das notas lista as últimas notas adicionadas pelo usuário em questão, exibindo os dados armazenados para o campo título (fundo azul e cor branca na fonte) e texto (fundo transparente e cor cinza na fonte).

Esta tela também deflagra o comportamento padrão do sistema, em que a página em uso que possui navegação via *menu*, neste caso a opção Início, recebe o efeito *active* do CSS, ou seja, o bloco "Início" fica com um tom de cinza mais escuro. Caso não exista nenhuma nota cadastrada para o usuário, o quadro onde as notas estão posicionadas é substituído pela mensagem "Não há notas cadastradas. Para cadastrá-las, clique em Adicionar".

No caso do sistema ser aberto em um dispositivo com menos resolução de tela disponível, graças ao seu *design* flexível com o *Bootstrap*, a tela inicial é apresentada conforme a imagem abaixo:

Figura 13 - Tela inicial das notas em forma reduzida.



Fonte: Autor, 2015.

Em sua forma reduzida de *layout*, Figura 13 acima, o sistema continua oferecendo as mesmas opções, porém elas ficam acessíveis somente após clicar no ícone do canto superior direito da tela. Ao clicar neste botão o design da tela inicial fica da seguinte maneira:

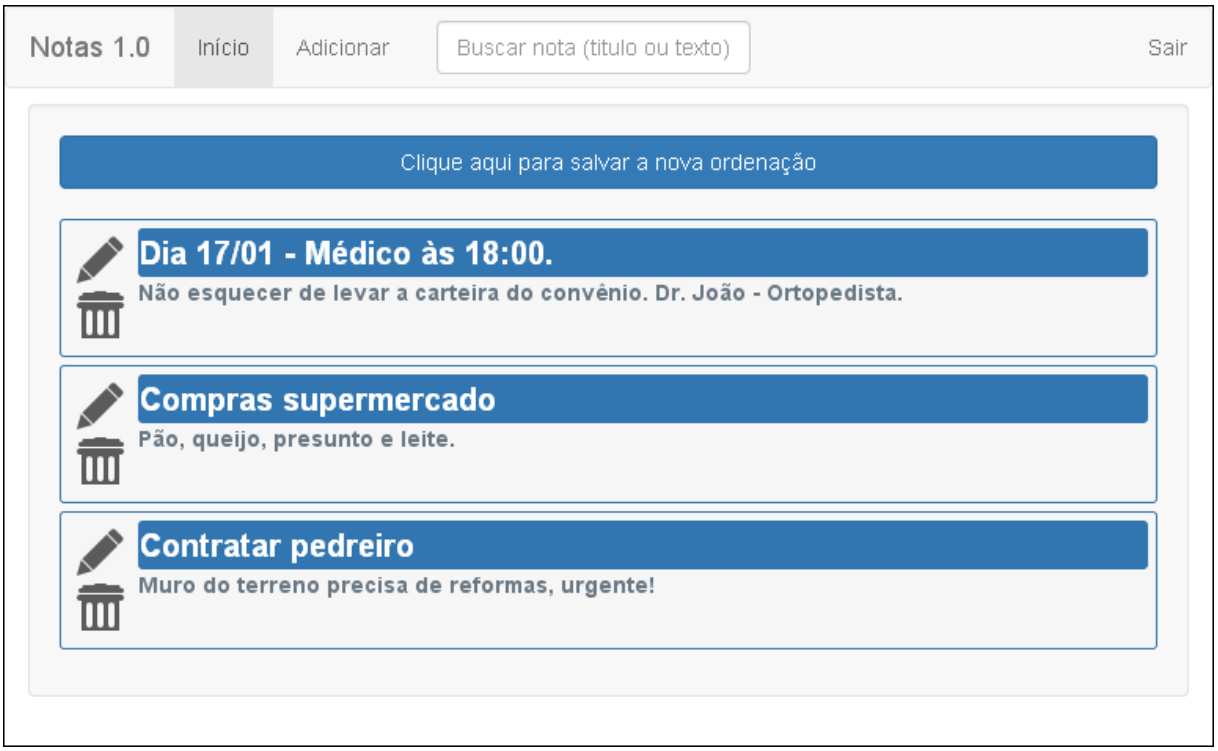
Figura 14 - Tela inicial das notas em forma reduzida com *menu* expandido.

Fonte: Autor, 2015.

Este *menu* é fixo e apresenta a mesma condição de flexibilidade em todas as páginas, só alterando sua aparência de acordo com qual função foi ativada – trocando a cor de fundo do item selecionado para um tom de cinza mais escuro.

O sistema também possui a função de reordenamento das notas, bastando o usuário clicar e arrastar as notas para qual posição na listagem desejar. Ao fazer esta ação, o sistema exibe um botão logo acima da primeira nota, solicitando ao usuário um clique para salvar a nova ordem das notas:

Figura 15 - Reordenar notas.



Fonte: Autor, 2015.

Ao clicar no botão, o mesmo some e o sistema tenta salvar a nova ordenação no banco de dados. Caso a ordenação tenha sucesso, uma mensagem exibindo o sucesso da operação surge no topo da página e após alguns segundos some:

Figura 16 - Reordenação das notas com sucesso.

Fonte: Autor, 2015.

Caso ocorra algum problema durante a execução da atualização dos dados no banco, uma mensagem de erro é exibida no topo da página:

Figura 17 - Reordenação das notas com erro.

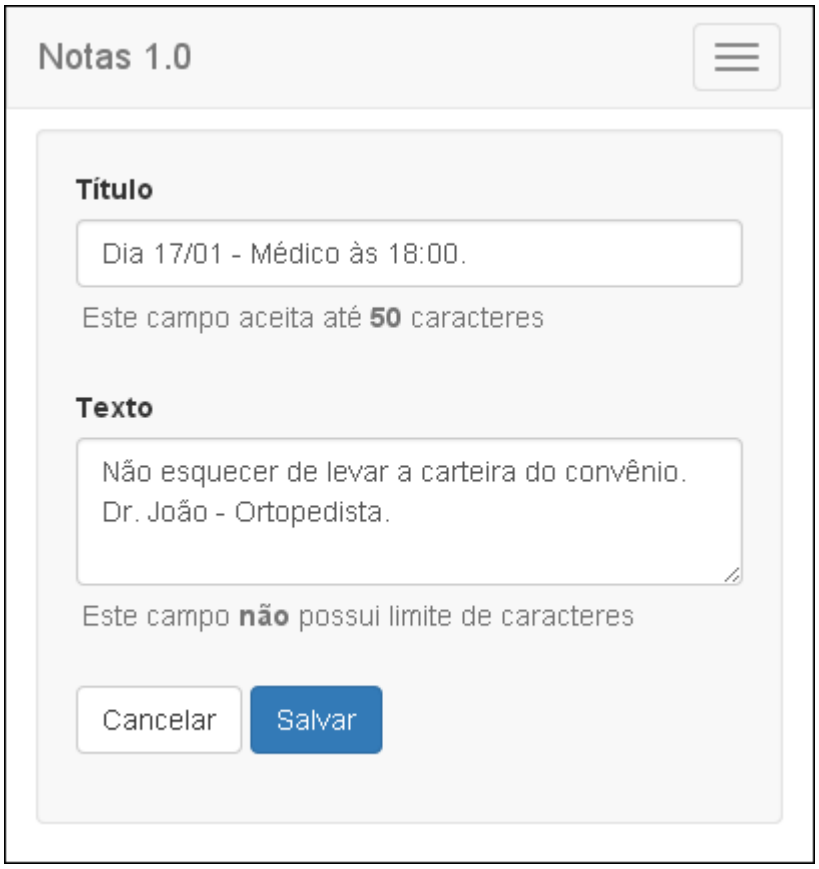
Fonte: Autor, 2015.

Neste caso, a nova ordenação não é salva no banco de dados e ao recarregar a página as notas são apresentadas na última ordem válida.

Ao lado do título e texto da nota, há dois botões correspondentes a duas funcionalidades distintas: o ícone do lápis, correspondente à função de edição da nota, permite que ao clicar o usuário seja redirecionado a página de edição da nota selecionada; e o ícone de uma lixeira, corresponde a função de exclusão da nota, que ao clicar, uma *popup* JavaScript abre questionando se o usuário deseja realmente excluir a nota.

No caso do usuário clicar no ícone do lápis (editar a nota) no primeiro registro, por exemplo, ele será redirecionado para a página de edição de notas, conforme imagem abaixo:

Figura 18 - Tela de edição de uma nota específica.



Fonte: Autor, 2015.

No caso da Figura 18 acima, antes de enviar o formulário de edição, há a validação dos campos (como em todos formulários do sistema). Caso o usuário deixe algum campo sem conteúdo uma mensagem em forma de *popup* aparece informando os campos que o usuário deve preencher. Caso passe dessa validação e o sistema consiga atualizar esta nota, o usuário é redirecionado para tela inicial de notas e a seguinte mensagem aparece no topo da página:

Figura 19 - Tela inicial de notas com mensagem de nota atualizada com sucesso.

Fonte: Autor, 2015.

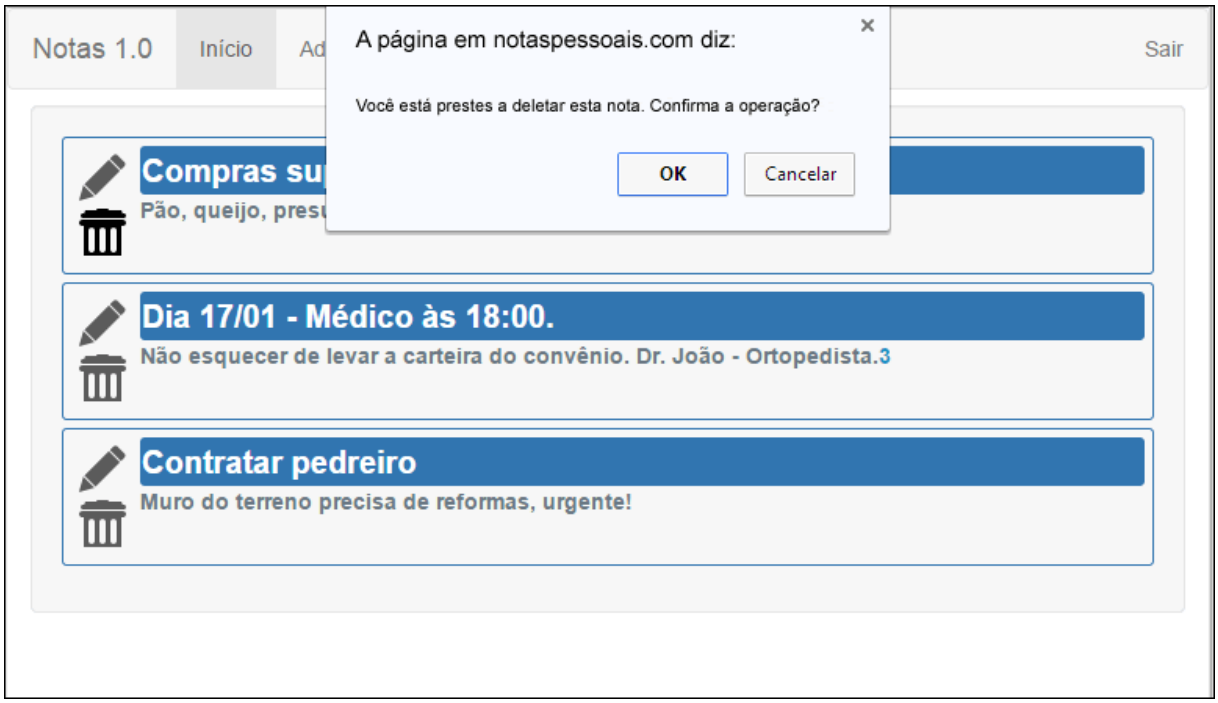
Caso não seja possível salvar as atualizações da nota, a tela fica idêntica à da Figura 19, com exceção da nota atualizada e da mensagem no topo da página, que ao invés de ser de sucesso é de erro:

Figura 20 - Mensagem de nota não atualizada (erro).

Fonte: Autor, 2015.

A outra funcionalidade disponível na tela inicial é a de exclusão de uma nota específica. Nesse caso, ao clicar no ícone da lixeira ao lado do título/texto da nota, o sistema exibe uma *popup* de confirmação exigindo que o usuário clique em "OK" para excluir a nota (clicando em "Cancelar", a janela *popup* é fechada e nada é alterado na tela de listagem):

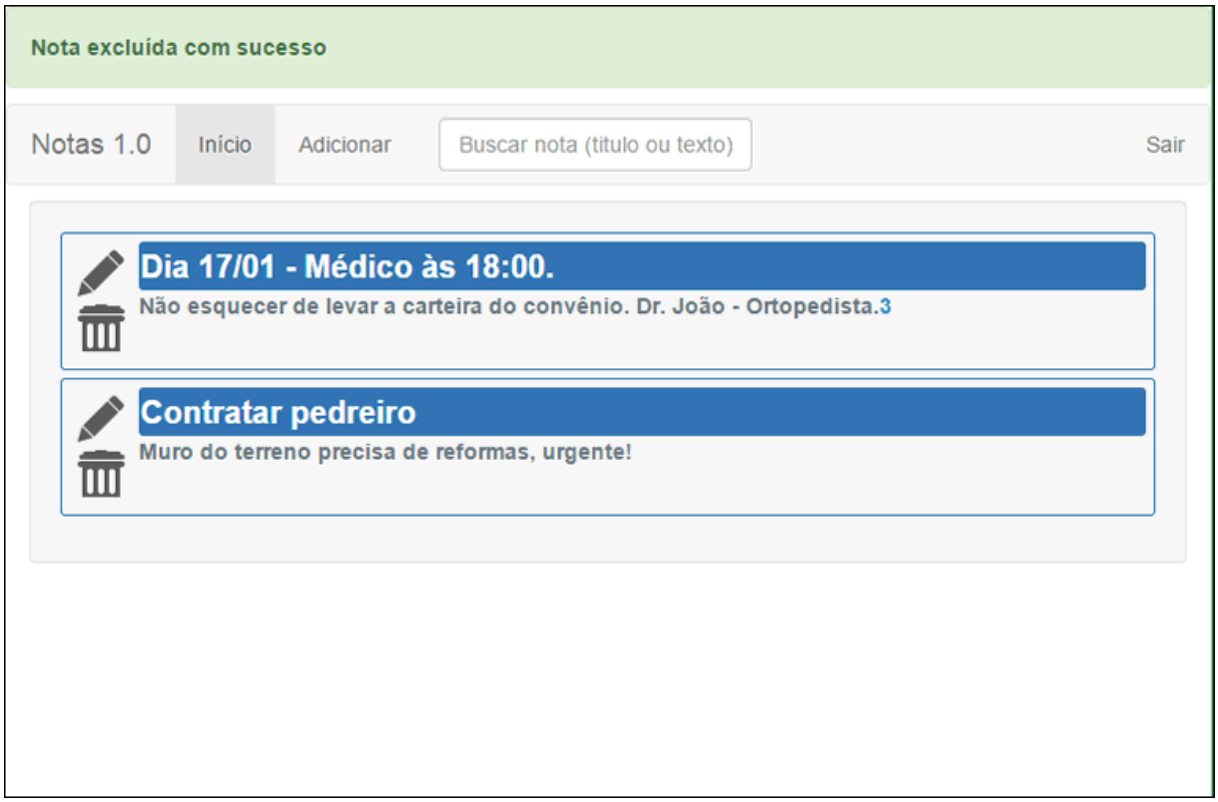
Figura 21 - Confirmação antes de excluir nota.



Fonte: Autor, 2015.

Ao clicar no botão “OK”, o sistema envia uma requisição à página “note/del.php”, onde a nota é validada e deletada caso passe para “área de segurança”, local onde todas as validações já foram feitas e os dados podem prosseguir para exclusão. Após remover a nota (em caso de sucesso), a nota excluída some da tela e a seguinte mensagem é exibida e depois escondida do usuário:

Figura 22 - Tela inicial com mensagem de sucesso após exclusão da nota.



Fonte: Autor, 2015.

No caso da Figura 22 acima, a nota some da listagem e é excluída da tabela "*note*" no banco de dados. Caso a exclusão não obtivesse sucesso, o sistema exibiria a mensagem abaixo:

Figura 23 - Mensagem de erro após tentativa de exclusão da nota (sem sucesso).

Fonte: Autor, 2015.

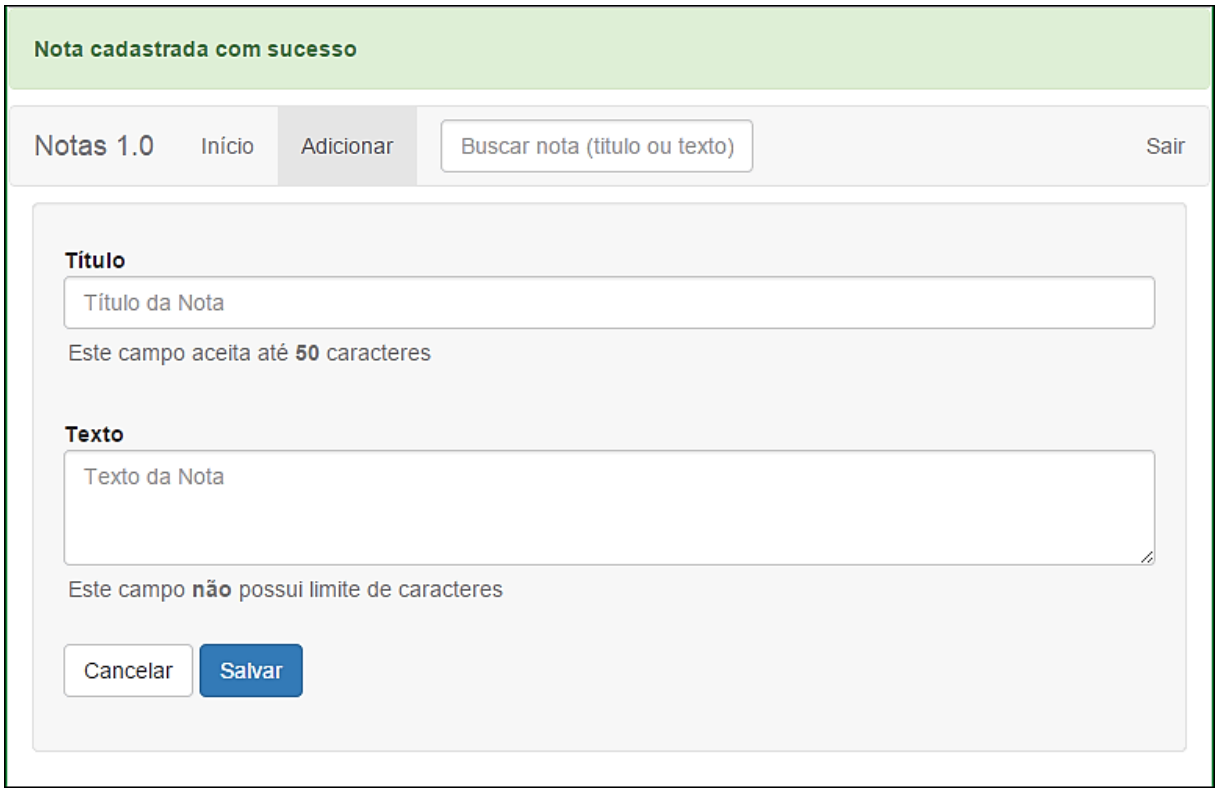
Até o momento trabalhou-se com as ações em que o usuário já possui notas cadastradas, porém, caso ele não tivesse, apenas não ficariam disponíveis as funções de reordenação, exclusão e edição de notas. Caso o usuário queira incluir uma nota, o passo a ser seguido é acessar a opção "Adicionar", na barra de *menu* superior. A tela exibida neste caso é a seguinte:

Figura 24 - Tela de adição de notas.

Fonte: Autor, 2015.

Após preencher os dados e clicar no botão salvar, os campos são validados e, em caso de sucesso, a nota é inserida e o sistema continua na tela de adição de notas. Se o usuário clicar no botão em cancelar, o sistema é redirecionado para tela de início das notas sem nenhuma mensagem no topo. No primeiro caso, o sistema tenta inserir a nota no banco de dados e em caso de sucesso o sistema recarrega a página de adição com uma mensagem de sucesso conforme a tela abaixo:

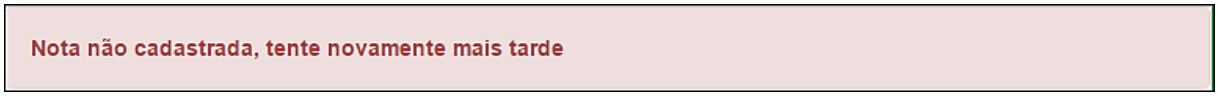
Figura 25 - Adição de nota com sucesso.



Fonte: Autor, 2015.

Caso a tentativa de adição da nota não tenha êxito na inserção no banco de dados, a mensagem abaixo é apresentada no topo da página “adicionar.php”:

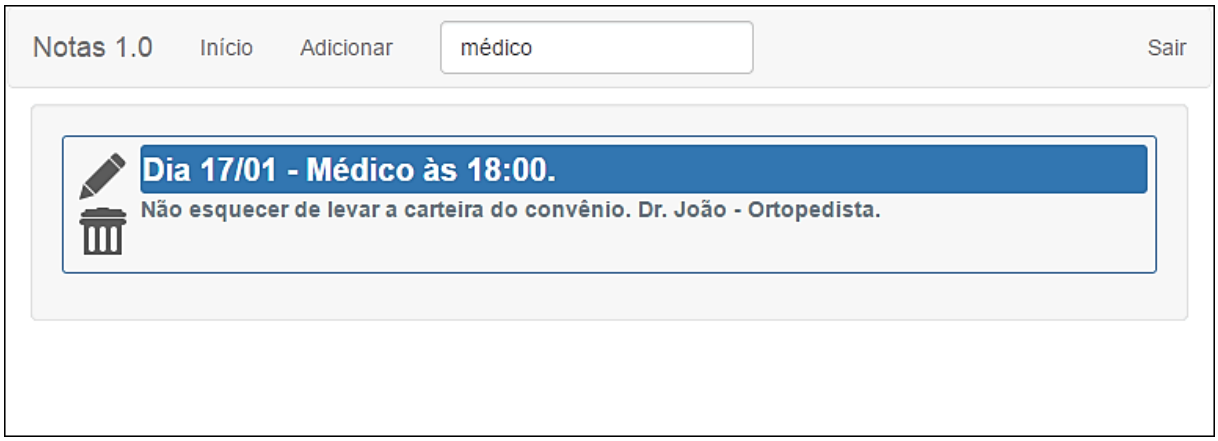
Figura 26 - Mensagem da tela de adição de nota com erro (nota não inserida).



Fonte: Autor, 2015.

A última função do sistema é a de procura de notas pelo título ou texto cadastrado. Após clicar no campo de busca, localizado no centro do *menu* superior, digitar um título ou texto de uma nota que está procurando, o usuário é enviado a página “search.php”, apresentada conforme abaixo:

Figura 27 - Tela de pesquisa de notas com resultados.

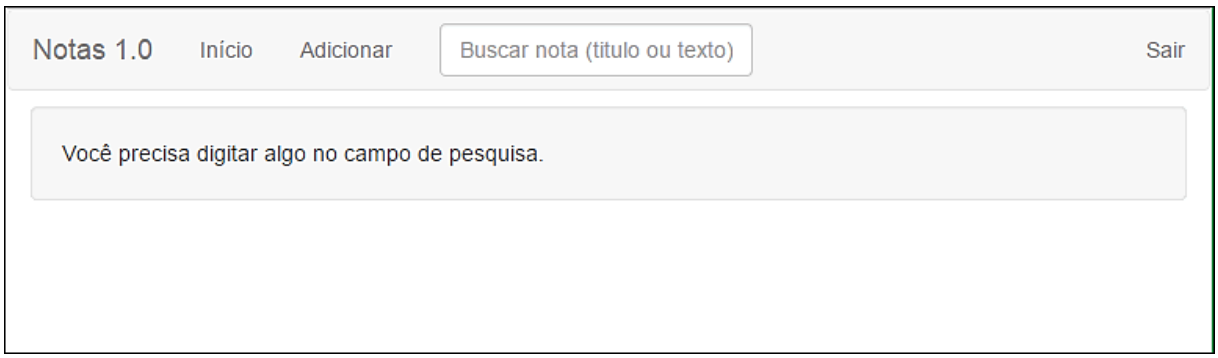


Fonte: Autor, 2015.

Caso o usuário digite uma busca sem resultados, a mesma tela é apresentada, porém com a mensagem "Nenhuma nota encontrada com o texto ou título informado". No caso acima, em que há resultados, o usuário pode executar as mesmas ações da tela inicial de notas, exceto a reordenação de notas.

Vale lembrar que se o usuário apertar a tecla *enter* do teclado sem digitar nada no campo de procura, a tela "search.php" é aberta, porém na configuração da Figura 28 abaixo:

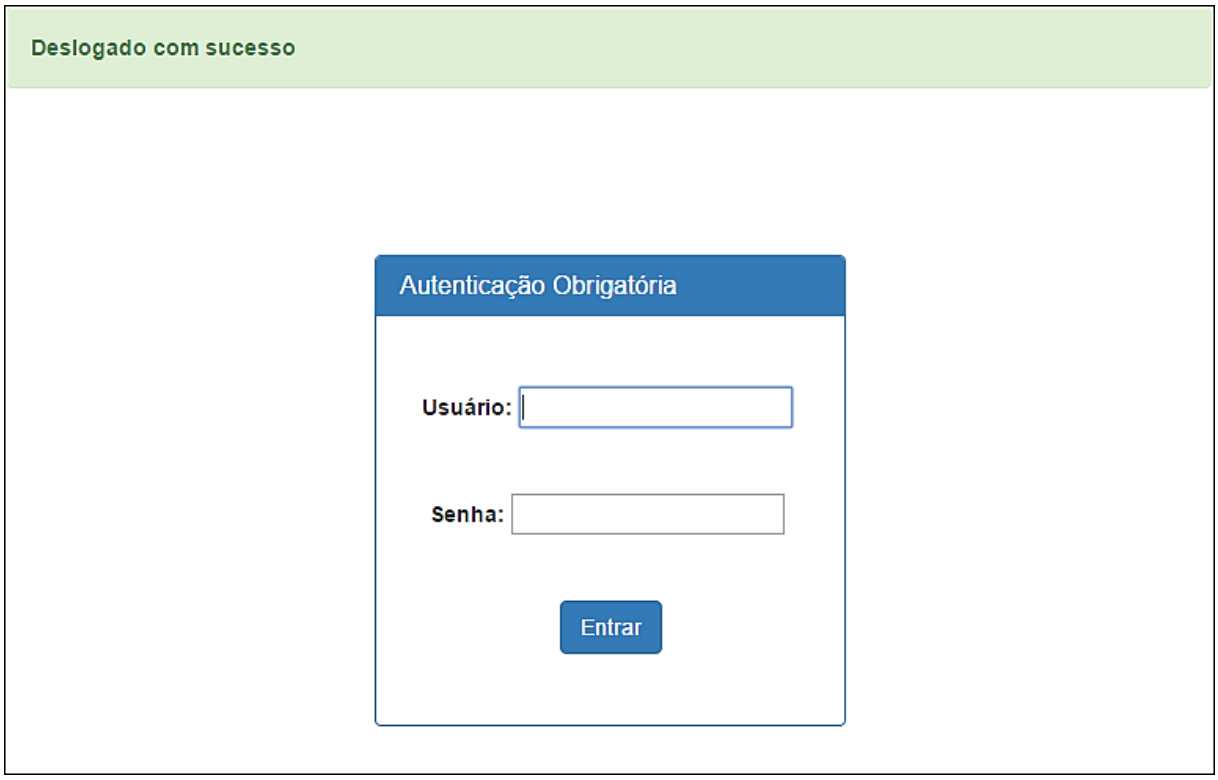
Figura 28 - Pesquisa de nota sem nenhum dado informado.



Fonte: Autor, 2015.

A última opção do *menu* superior, a opção "Sair", como o próprio nome indica, ao clicar nela, independente de onde o usuário estiver, o sistema remove os dados da sessão dele, e o usuário é redirecionado para a tela de *login* com a mensagem "*Deslogado* com sucesso", conforme a imagem abaixo:

Figura 29 - Tela de *login* após o usuário clicar na opção “Sair”.



Fonte: Autor, 2015.

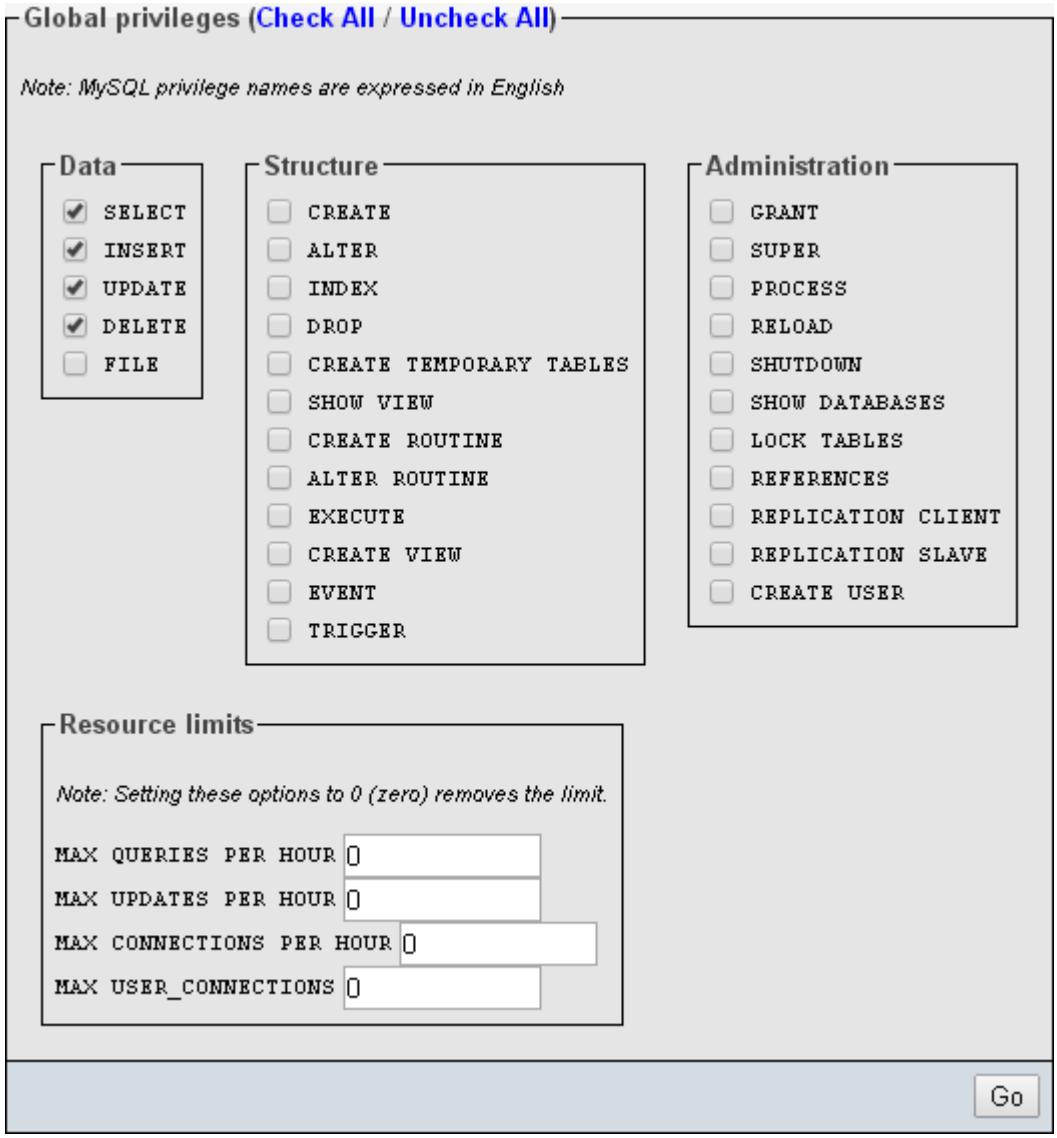
6.3 Técnicas e padrões de segurança implementados

Conforme a norma NBR ISO/IEC 27001 e 27002, especificar em que local a aplicação será utilizada e quais são seus usuários é um dos primeiros passos de segurança. Por este motivo, determinou-se que o sistema de notas funcionará em um servidor *web* dedicado e que será usado por funcionários de uma empresa fictícia, não sendo acessível ao público em geral.

Para evitar tentativas de cadastro, perda de senha e procedimentos relativos ao cadastro do usuário, os acessos dos usuários seriam cadastrados pelo gestor de informática da empresa, inserindo os novos usuários diretamente no banco de dados.

Para tratar a ameaça conhecida como *SQL Injection*, diretrizes de segurança do Site oficial do PHP – PHP.NET (2015) – e que vão de encontro ao recomendado na seção 4.2 foram utilizadas. O usuário padrão do banco de dados (MySQL) do sistema não teve permissão de acesso total, possuindo apenas as permissões necessárias à aplicação. Figura 30, abaixo:

Figura 30 - Permissões dadas ao banco de dados.



Fonte: Autor, 2015.

Outra dica valiosa contra *SQL Injection* utilizada seguindo as dicas do PHP.NET (2015) é a utilização das funções *addslashes* e *is_numeric* para validar consulta, inserção, atualização e remoção de dados no banco de dados. Essas duas funções padrão do PHP auxiliam no tratamento a literais e inteiros, respectivamente, e no sistema em questão são usadas antes de qualquer interação no banco de dados com inteiros ou literais (únicos tipos utilizados para consulta pelo sistema) não confiáveis, conforme a Figura 31 abaixo:

Figura 31 - Instruções contra a injeção de código malicioso.

```php
<?php
//função para validar LITERAIS antes de inserir na SQL
function sqlTrataString($string){
    //se a string possui algum valor
    if(isset($string)){
        $retorno = trim($string);//tira espaços no inicio e fim da string
        $retorno = addslashes($retorno);//retorna a string com barras invertidas antes de caracteres que precisam ser escapados para a query. Estes caracteres são aspas simples ('), aspas duplas ("), barra invertida (\) e NUL (o byte NULL)
        return $retorno;//retorna a string
    }
    return false;//nas páginas com interação com o banco de dados, se retornar false de sqlTrataString() a consulta não é enviada ao banco de dados
}

//função para validar INTEIROS antes de inserir na SQL
function sqlTrataInteiro($inteiro){
    //se a variável $inteiro possui algum valor
    if(isset($inteiro)){
        if(is_numeric($inteiro)){//só entra aqui se o valor for número
            return $inteiro;//retorna o inteiro
        }
    }
    return false;//nas páginas com interação com o banco de dados, se retornar false de sqlTrataInteiro() a consulta não é enviada ao banco de dados
}
?>
```

Fonte: Autor, 2015.

Para gerar as senhas dos usuários (e compará-las ao valor armazenado no banco de dados), a seguinte codificação com MD5, SHA1 e *sal* foi utilizada:

Figura 32 - Codificação utilizada para as senhas do sistema.

```php
function codifica($valor){
    return md5(sha1(md5($valor.'@-+.').'!$%*').'ç^@#');
}
```

Fonte: Autor, 2015.

O código acima não foi encontrado para decodificação nos principais bancos de dados de chaves conhecidas mesmo utilizando *sal* fixa em cada *hash* gerado. Dentre estes bancos, destaca-se o site *HashKiller* (2015) em que, conforme o criador, 43.745 bilhões de códigos MD5 e SHA-1 foram catalogados, mas a simples senha “a” gerada a partir da função acima não pôde ser decodificada. Para fins de estudo, a senha “a” codificada com MD5, SHA-1 e *sal* conforme a Figura 32 possui o valor “*27e93ad5767354dab29c9dd8f6f2f96d*”. Esta

prática protege contra a adulteração dos dados, conforme a seção 4.3, e respeita as orientações de segurança de Howard e LeBlanc (2005) já que não foi criado nenhum código de criptografia novo.

Os métodos de proteção de dados confidenciais da seção 4.4, com exceção do uso de *hashes* e *sal* (como citado no parágrafo anterior), não foram aplicados já que os estudos da mesma são voltados a sistemas em que o *software* é uma aplicação *desktop* (aplicação instalada na máquina do usuário) e que permite utilizar informações do usuário conectado à rede local ou ao computador para proteger seus dados. No caso desta aplicação *web* esta forma de proteção dos dados não foi desenvolvida, pois os usuários do sistema não utilizarão o mesmo computador para acessar o sistema e conforme o Quadro 6 da referida seção, a relação custo de desenvolvimento, facilidade de instalação e segurança relativa não compensariam a aplicação destes métodos.

As validações das entradas dos dados feitas pelos usuários seguem as diretrizes abaixo:

- Valide se é diferente de nulo;
- Valide se o tamanho da informação é menor ou igual ao definido no banco de dados;
- Valide se o tipo de informação (inteiro, literal, etc.) condiz com o esperado no banco de dados.

Após os dados inseridos passarem por estes aspectos, os dados não confiáveis entram no que Howard e LeBlanc (2005) chamam de “área segura”, onde os principais danos causados por entradas maliciosas – por exemplo, o estouro de *buffer* – tendem a não ocorrer ou serem menores. A aplicação direta destes conceitos no sistema é elucidada nas duas figuras abaixo (Figura 33 e Figura 34):

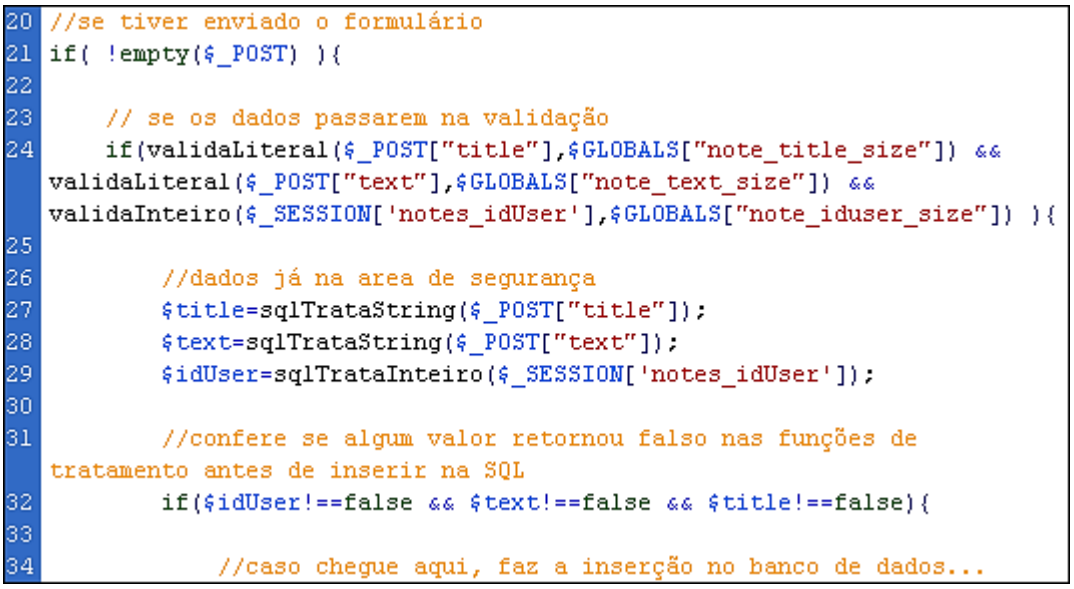
Figura 33 - Funções de validação de tipos e tamanhos.

```php
<?php
function validaInteiro($inteiro, $tamanho){
    //se as variaveis inteiro e tamanho possuem algum valor
    if(isset($inteiro) && isset($tamanho)){
        //se o inteiro for número
        if(is_numeric($inteiro)){
            //se o inteiro tiver tamanho menor ou igual ao que precisa ter ou se o argumento passado para $tamanho for zero (não tem limite de tamanho)
            if(strlen($inteiro)<=$tamanho || $tamanho == 0){
                //caso passe pelas validações acima, retorna TRUE
                return true;
            }
        }
    }
    //se chegar aqui é por que não é um inteiro válido, portanto return FALSE
    return false;
}
function validaLiteral($literal, $tamanho){
    //se as variaveis literal e tamanho possuem algum valor
    if(isset($literal) && isset($tamanho)){
        //se o literal tiver tamanho menor ou igual ao que precisa ter ou se o argumento passado para $tamanho for zero (não tem limite de tamanho)
        if(strlen($literal)<=$tamanho || $tamanho == 0){
            //caso passe pelas validações acima, retorna TRUE
            return true;
        }
    }
    //se chegar aqui é por que não é um literal válido, portanto return FALSE
    return false;
}
?>
```

Fonte: Autor, 2015.

O código da Figura 33 acima pertence ao arquivo "*functions.php*", que é incluído nos arquivos em que há interação com o banco de dados. Essas funções retornam o *boolean* "verdadeiro" caso os dados estejam no formato e tamanho correto ou "falso" quando não estão. A partir dessas validações, os dados passam a "área segura", como no trecho da página de inserção de notas abaixo:

Figura 34 - Trecho do código da página de inserção de notas.



```
20 //se tiver enviado o formulário
21 if( !empty($_POST) ){
22
23     // se os dados passarem na validação
24     if(validaLiteral($_POST["title"],$GLOBALS["note_title_size"]) &&
   validaLiteral($_POST["text"],$GLOBALS["note_text_size"]) &&
   validaInteiro($_SESSION['notes_idUser'],$GLOBALS["note_iduser_size"]) ){
25
26         //dados já na area de segurança
27         $title=sqlTrataString($_POST["title"]);
28         $text=sqlTrataString($_POST["text"]);
29         $idUser=sqlTrataInteiro($_SESSION['notes_idUser']);
30
31         //confere se algum valor retornou falso nas funções de
   tratamento antes de inserir na SQL
32         if($idUser!==false && $text!==false && $title!==false){
33
34             //caso chegue aqui, faz a inserção no banco de dados...
```

Fonte: Autor, 2015.

Na Figura 34 acima, os dados são validados quanto ao seu tipo e tamanho antes de chegarem à área de segurança onde os dados estão prestes a serem inseridos no banco de dados, porém antes de efetuar a requisição e conexão com o banco, para cada valor é chamado o tratamento específico para validar se os valores não impactarão em algum aspecto de segurança na SQL, principalmente quanto à Injeção de SQL, e só então, na linha 34, os dados passam para etapa de inserção no banco de dados.

Seguindo os conceitos de segurança abordados por Howard e LeBlanc (2005), foi utilizada a API *PDO_MySql*, mantendo-se como regra a parametrização dos argumentos para SQL. Esta API foi escolhida baseada nas comparações feitas pelo PHP.NET (2015), onde a API PDO_MySQL é relacionada como a mais recente – introduzida no PHP 5.1 –, recomendada para criação de novos projetos e com status de desenvolvimento ativo, o que significa que devem surgir melhorias nas próximas versões do PHP, além de ser a única a suportar SQL's parametrizadas tanto no lado do servidor quanto no lado do cliente. A Figura 35 abaixo demonstra como é feita a parametrização e inserção dos dados na página de adição de notas, e corresponde à continuação da Figura 34 acima:

Figura 35 - Comandos de banco de dados para inserção de notas.

```
36              //tenta inserir dados no banco
37              try {
38                  $bd = new myDB();
39                  $query = $bd->prepare("INSERT INTO note VALUES (?, ?, ?, ?,
    ?, ?, ?)");
40
41                  $query->bindValue(1, NULL, PDO::PARAM_STR);//idNote
42                  $query->bindValue(2, $idUser, PDO::PARAM_STR);
43                  $query->bindValue(3, 1, PDO::PARAM_STR);//flag
44                  $query->bindValue(4, date("Ymd"), PDO::PARAM_STR);
45                  $query->bindValue(5, $title, PDO::PARAM_STR);
46                  $query->bindValue(6, $text, PDO::PARAM_STR);
47                  $query->bindValue(7, 0, PDO::PARAM_STR);//orderNote
48
49                  $inseriu=$query->execute();
50
51              } catch (PDOException $e) {
52                  echo $e->getMessage();
53              }
54              echo "<script>";
55                  if($inseriu){
56                      echo "showMsgSucesso('Nota cadastrada com sucesso','');"
    ;
57                  }else{
58                      echo "showMsgErro('Nota não cadastrada, tente novamente
    mais tarde','');";
59                  }
60              echo "</script>";
```

Fonte: Autor, 2015.

Na Figura 35 acima, linha 38, o banco de dados é instanciado e estende a *API PDO_MySQL*, em seguida é preparada a SQL para inserção dos dados, registra-se os parâmetros com o método "*bindValue*" e em seguida é executada a *query* (consulta). Conforme a seção 4.7 deste trabalho, a adoção desta parametrização diminui o risco de Injeção de SQL, já que o próprio banco de dados fará também a validação desses dados.

Os *cookies* não foram utilizados em nenhuma parte do sistema (em PHP o *$_COOKIE*), pois conforme a seção 4.8, é um recurso interessante porém abre mais uma brecha de segurança, portanto, ao invés de *cookies* o sistema utilizou *session* (em PHP o $_SESSION), trabalhando apenas com dados no lado do servidor e para armazenar dados inerentes ao usuário: nome, *id* e *id* de privilégio.

O sistema não suporta internacionalização, ou seja, o uso por diversas linguagens diferentes, no entanto, por ser considerado o padrão mais seguro de conversão de caracteres, todas as páginas do sistema possuem o *charset* (padrão de caracteres) definido para UTF-8 (padrão recomendado para aplicações *web* conforme a seção 4.9). O uso do *charset* é demonstrado no cabeçalho da página de procura de notas, a "search.php":

Figura 36 - Charset com UTF-8.

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title><?php echo $GLOBALS["notes_title"]; ?></title>
</head>
```

Fonte: Autor, 2015.

Outros procedimentos mais simples de segurança também foram seguidos ao codificar o sistema, tais como:

- Utilização de aspas duplas em todas as propriedades de *tags* HTML (conforme a seção 4.8);
- Não utilização da função *eval()* do *JavaScript* (conforme a seção 4.8);
- Utilização de caminhos absolutos ao invés de relativos (conforme a seção 4.7);
- Validação de literais com expressão regular e somente para os retornos esperados, não os inesperados ou proibidos (seção 4.6);
- Indicação de erros sempre próxima à origem, facilitando a manutenção e descoberta de falhas do código (seção 4.1);
- Pastas de dentro do sistema sempre possuem um arquivo "index.php" para não permitir a visualização dos arquivos do diretório;

Essas foram as principais técnicas de segurança utilizadas no sistema. Nem todas foram listadas, pois não caberiam neste documento. Mesmo se tratando de um sistema simples, quase todas as linhas de código podem receber ou incrementar técnicas de segurança apresentadas neste estudo, por isso, escrever todas as técnicas seria inviável, restringindo esta seção apenas as mais importantes.

6.4 Resultados e Discussão

Nesta seção é apresentado o resultado dos testes de segurança efetuados com o *software* Acunetix Vulnerability Scanner, que varreu o sistema em busca de vulnerabilidades e encontrou 10 alertas de segurança que são categorizados, explicados e discutidos nos próximos parágrafos.

Para efetuar o teste, o *software* mencionado foi instalado e configurado para testar o sistema *online*, em uma hospedagem e domínio paga na internet (UOL Host). A verificação

aplicada com o *software* demorou 2 horas e 30 minutos, enviou cerca de 30.000 requisições para o site e apontou 10 alertas. Este número total de alertas, é separado em 4 níveis de alerta: o nível 3 (vulnerabilidade muito perigosa), nível 2 (vulnerabilidade de média periculosidade), nível 1 (vulnerabilidade de pequena periculosidade) e nível 0 (apenas informacional), sendo apresentado os resultados nessa sequencia. Como o *software* da Acunetix foi utilizado em sua versão de avaliação gratuita, os alertas não ofereceram o local exato da falha, apenas indicaram as vulnerabilidades encontradas, cabendo ao desenvolvedor encontrar o local das falhas e descobrir como corrigi-la. Além disso, nessa versão os testes são do tipo "caixa preta", ou seja, sem acesso ao código fonte da aplicação.

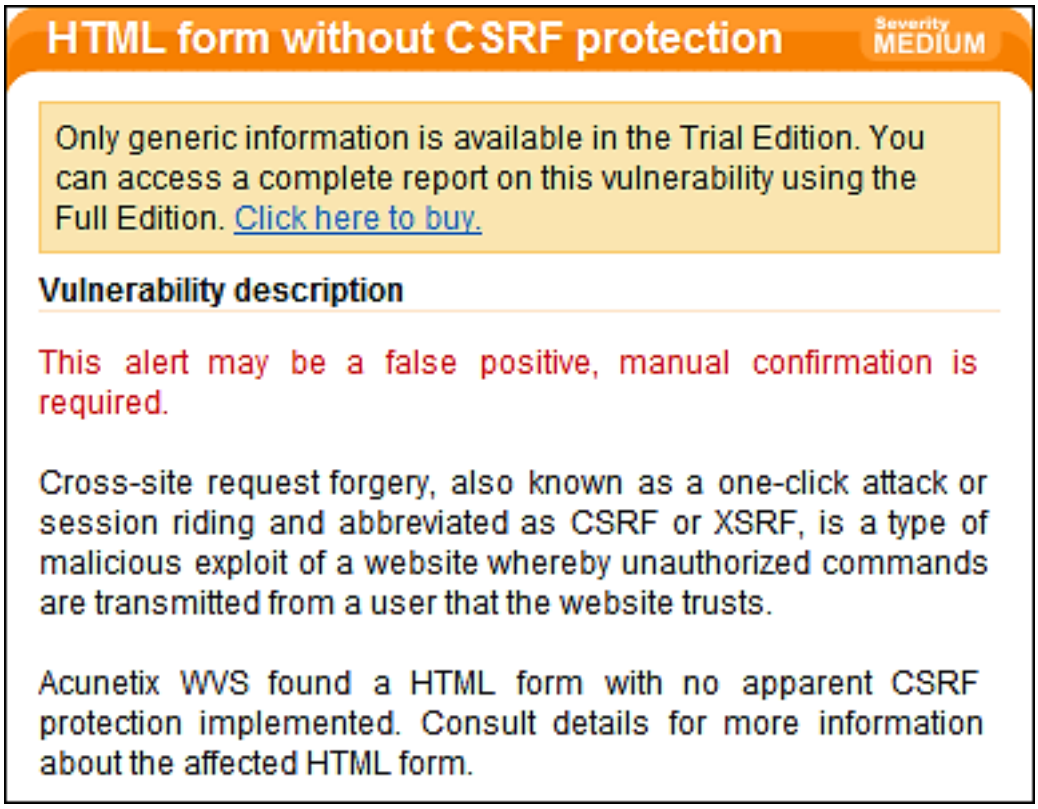
Durante a exibição das vulnerabilidades, a correção das mesmas também é apresentada, sugerindo técnicas ou apresentando o código que precisa ser implementado. Por se tratar de uma área prática de baixo referencial teórico, grande parte das referências sobre a correção dos erros é proveniente da experiência prática de usuários disponibilizadas na Internet.

A primeira categoria, o nível alto ou nível 3, em que as vulnerabilidades são as mais graves encontradas pelo aplicativo, ficou sem nenhum alerta. Portanto, o sistema desenvolvido, conforme o *software*, não possui alguma vulnerabilidade grave, podendo isso estar atrelado também as configurações de segurança do servidor da UOL que hospedou o site.

No segundo nível de vulnerabilidades, as de média periculosidade (nível 2 ou nível médio), foram encontrados três alertas de segurança.

A primeira vulnerabilidade encontrada neste nível foi a "*HTML form without CSRF protection*", em tradução livre "Formulários HTML sem proteção contra CSRF", onde CSRF é a sigla para "*Cross-Site Request Forgery*". A descrição do alerta está na Figura 37 abaixo:

Figura 37 - Vulnerabilidade: formulários HTML sem proteção contra CSRF.



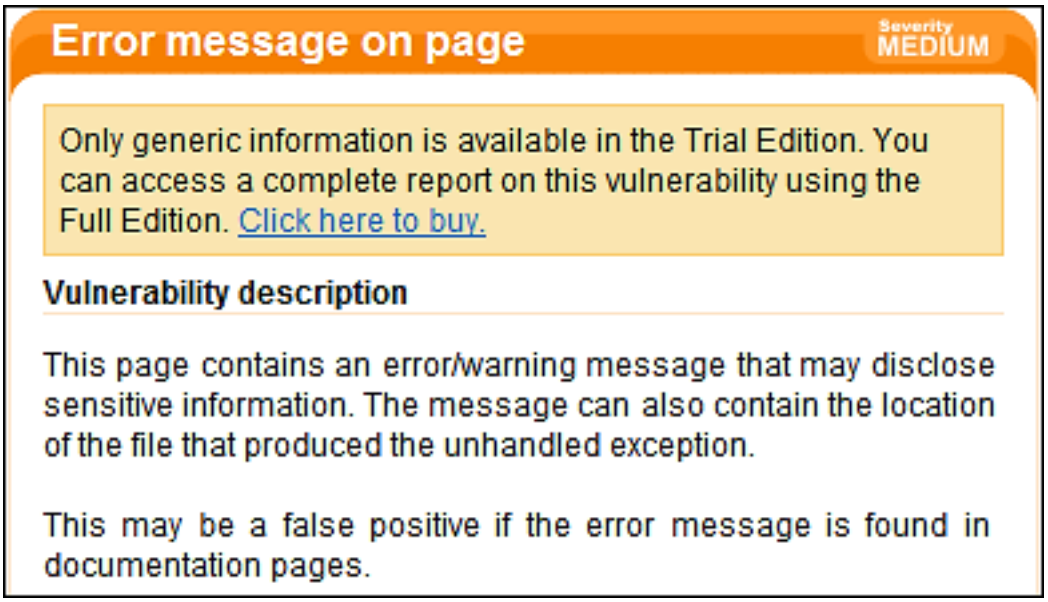
Fonte: Acunetix, 2015.

Segundo Ristic (2005), o CSRF consiste em uma exploração maliciosa pelo qual comandos não autorizados são transmitidos ao sistema *web*, a partir da confiança em que o sistema tem na identidade de um usuário. Esta falha de segurança é abordada em praticamente todas as seções secundárias do Capítulo 4, porém de forma superficial, não explicando exatamente como combater esta ameaça específica. Conforme a Acunetix (2015), quando explorada, esta ameaça pode comprometer os dados do usuário no sistema se tratando de um usuário normal, no caso de um administrador do sistema, pode comprometer a aplicação inteira.

Para evita-la, conforme o site da fundação *Open Web Application Security Project*, a OWASP (2015), deve-se utilizar *tokens* nos formulários. Usando *tokens*, os valores dos formulários são codificados e não apresentam o valor digitado pelo usuário. Essa codificação pode ser feita com *hashes* de 256 *bits* do *BASE64 encode* e, segundo a fundação, é atualmente um dos melhores métodos para evitar este tipo de ataque.

A segunda vulnerabilidade do nível 2, foi a "*Error message on page*", em tradução livre "Mensagens de erro na página". A descrição do alerta está na Figura 38 abaixo:

Figura 38 - Vulnerabilidade: mensagens de erro na página.



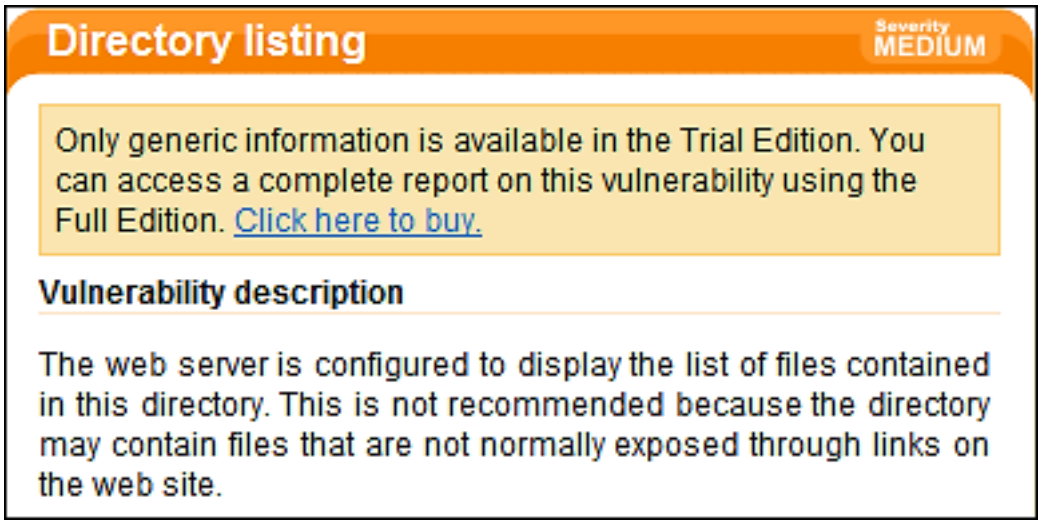
Fonte: Acunetix, 2015.

Conforme a Acunetix (2015), esta falha é causada por mensagens de erro no sistema que podem exibir informações sensíveis, como por exemplo, o local da falha. Informações como essa podem ser usadas para iniciar outros ataques portanto devem ser evitadas.

Para eliminar esta ameaça, a sugestão é revisar o código da aplicação, encontrando locais em que erros podem ocorrer e verificar as mensagens que são exibidas, modificando-as para que informações sensíveis não sejam exibidas (Acunetix, 2015).

Outra vulnerabilidade encontrada no nível de segurança “médio”, foi o “*Directory listing*”, ou em tradução livre “listagem de diretório”. O alerta apresentado pode ser conferido na integra na Figura 39 abaixo:

Figura 39 - Vulnerabilidade: listagem de arquivos do diretório.



Fonte: Acunetix, 2015.

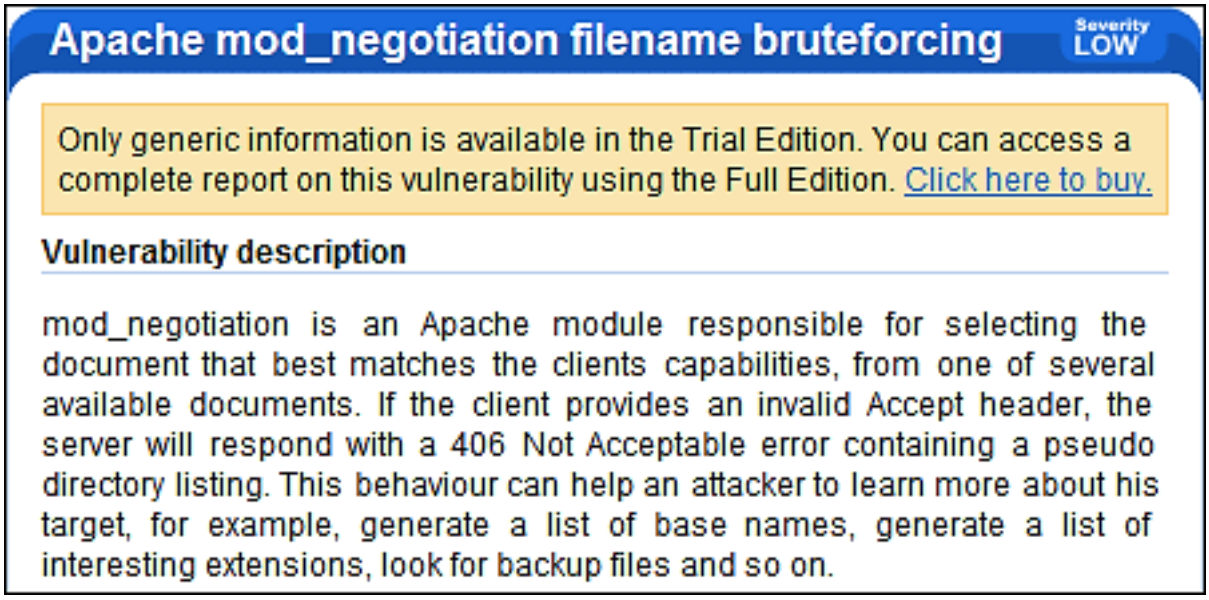
Com esta falha de segurança, usuários maliciosos podem listar todos os arquivos de um diretório, podendo acessar ou visualizar informações confidenciais do sistema ou de outros usuários (Acunetix, 2015).

No caso do sistema de notas pessoais, o problema tinha sido tratado criando arquivos *index.php* dentro de todas as pastas do sistema, mas como pôde-se perceber, não foi a melhor solução.

Conforme o HTML Staff (2015), este problema pode ser resolvido através da configuração do HTACCESS (*Hypertext access*) da página. O HTACCESS é um arquivo do tipo *.htaccess*, sem nome, localizado na raiz do sistema, executado no servidor *web* e que permite a configuração de diversos aspectos como autenticação, bloqueio de IP's, configurações da listagem de diretórios e etc. A simples instrução "*IndexIgnore* *" dentro deste tipo de arquivo, impossibilita que arquivos possam ser listados dos diretórios do sistema, atenuando ou inibindo esta ameaça.

Já no nível baixo, em que a vulnerabilidade possui baixo grau de periculosidade (nível 1), encontrou-se a vulnerabilidade "*Apache mod_negotiation filename bruteforcing*". O alerta exibido pelo Acunetix foi:

Figura 40 - Vulnerabilidade: mod_negotiation do Apache.



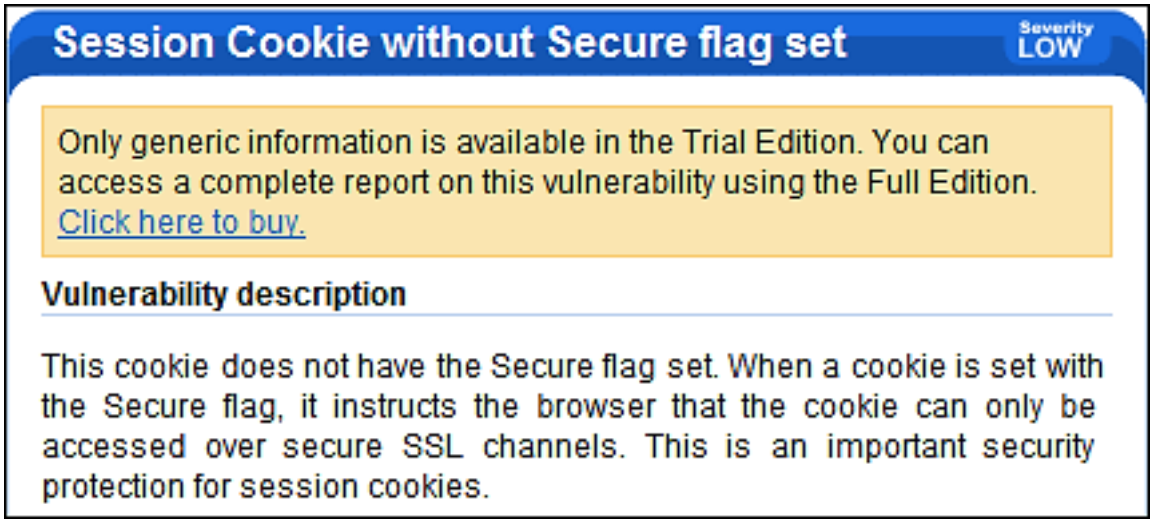
Fonte: Acunetix, 2015.

Conforme o *software*, essa falha trata-se de um módulo do *Web Service Apache*, o *mod_negotiation*, responsável por selecionar documentos correspondentes as requisições *web*, em que, ao usuário ou o sistema enviar um cabeçalho inválido (*header*) para uma página, ele emite uma mensagem de erro que lista diretórios, permitindo que usuários mal intencionados possam conhecer mais informações sobre o sistema como, por exemplo, uma listagem de extensões interessantes ou arquivos de *backup*.

Para solucionar esta falha, conforme um dos relatórios do HackerOne (2015), deve-se desabilitar a diretiva *MultiViews* das configurações do Apache. Utilizando HTACCESS isso é possível adicionando a instrução "*Options –MultiViews*".

Outra vulnerabilidade de nível 1 encontrada foi a "*Session Cookie without Secure flag set*", em tradução livre "*Cookie* de sessão sem a *flag* de segurança configurada". A Figura 41 abaixo exibe o alerta do Acunetix:

Figura 41 - Vulnerabilidade: *cookie* de sessão sem *flag* de segurança configurada.



Fonte: Acunetix, 2015.

Conforme o relatório, algum *cookie* de sessão não possui a *flag* de segurança configurada para ser acessada somente com SSL (sessão segura, com certificados de segurança). Essa *flag* é muito importante para a proteção dos cookies de sessão, por isso quando os *cookies* de sessão estão atrelados ao acesso restrito dos usuários, ela deve ser configurada.

Esta falha é similar com a próxima vulnerabilidade, por isso sua solução é apresentada em conjunto com a próxima.

A quinta vulnerabilidade apontada, terceira do nível 1, trata-se da "*Session Cookie without HttpOnly flag set*", em tradução livre "*Cookie* de sessão sem a *flag HttpOnly* configurada". A Figura 42, abaixo, exibe as informações desta vulnerabilidade conforme o *software* da Acunetix:

Figura 42 - Vulnerabilidade: *cookie* de sessão sem a *flag HttpOnly* configurada.

Fonte: Acunetix, 2015.

Como mencionado, esta vulnerabilidade é muito próxima a sua antecessora, justamente por também estar ligada com a segurança da sessão do usuário. Nesse caso, o relatório aponta que não há uma *flag* que defina que os *cookies* de sessão devam ser acessados somente pelo lado do servidor, permitindo – com essa vulnerabilidade – que o lado do cliente também possa acessar as informações da sessão. Esta vulnerabilidade permite que informações dos usuários sejam acessadas ou modificadas, trazendo possíveis danos aos dados do usuário ou do sistema como um todo.

Para corrigir esta falha, bem como a sua antecessora, um tópico do Stack Overflow (2015), uma espécie de fórum dos programadores, descreve que após adicionar as duas linhas da imagem abaixo no arquivo HTACCESS, as duas *flags* são devidamente configuradas e corrigem os dois erros:

Figura 43 - Corrigindo *flag* de *HttpOnly* e SSL dos *cookies* com HTACCESS.

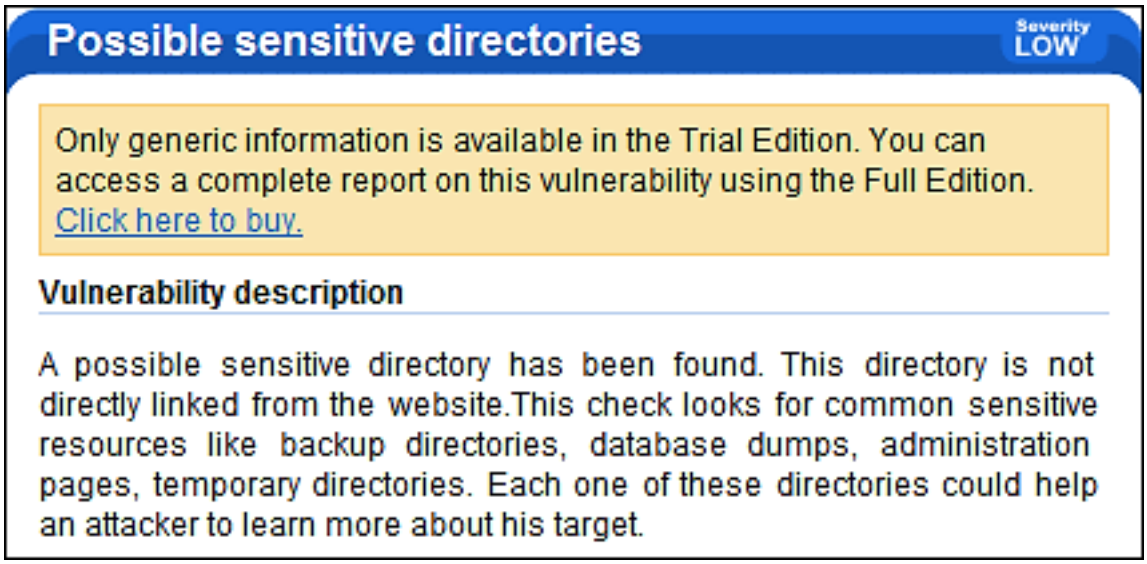
```
php_value session.cookie_httponly 1
php_value session.cookie_secure 1
```

Fonte: Stack Overflow, 2015.

Com o uso das duas linhas acima, as sessões dos usuários ficam muito mais seguras, podendo ser acessadas somente pelo lado do servidor e exigindo que o tráfego dos dados da sessão seja feito com HTTPS.

A próxima vulnerabilidade da lista, com nível de segurança baixa (nível 1), é chamada de "*Possible sensitive directories*", ou em tradução livre "Diretórios possivelmente sensíveis". Pelo relatório da Acunetix, suas informações foram descritas conforme a imagem abaixo:

Figura 44 - Vulnerabilidade: diretórios possivelmente sensíveis.



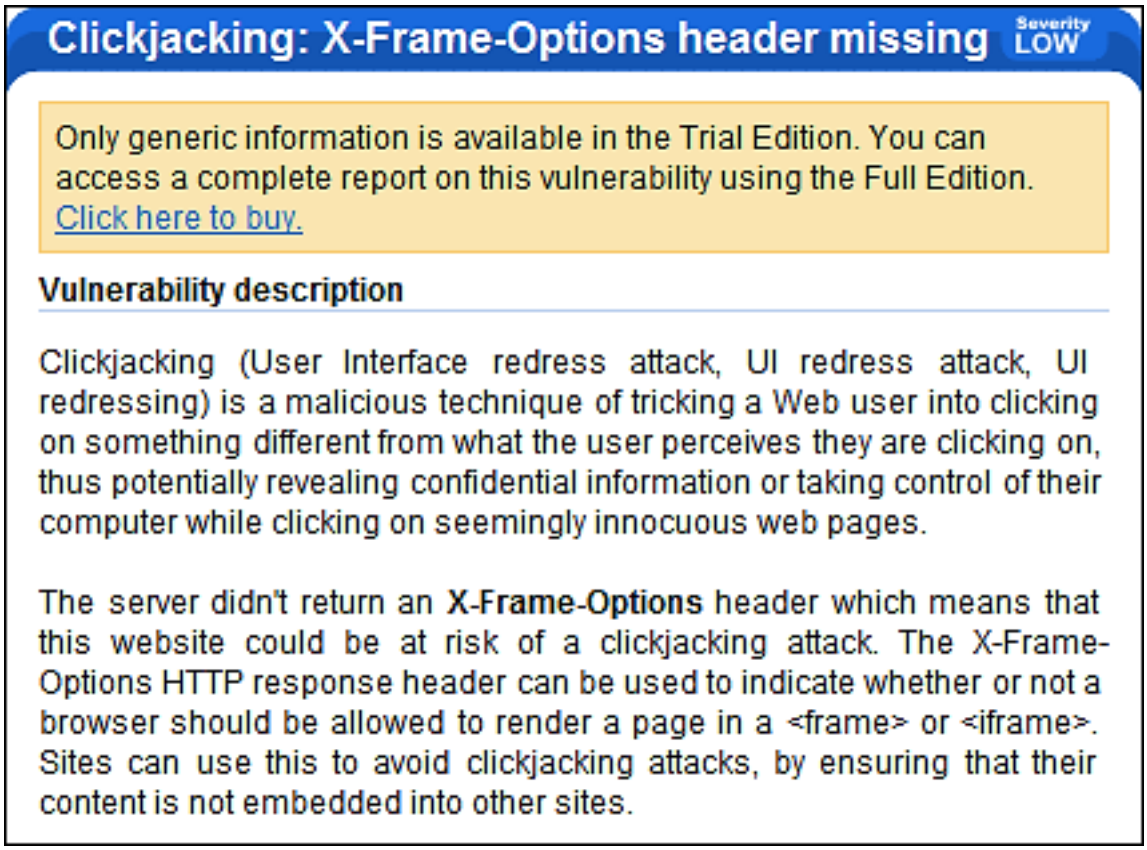
Fonte: Acunetix, 2015.

Conforme o *software*, algum diretório sensível pôde ser encontrado, mesmo sem ele ser referenciado por algum link ou arquivo dentro do sistema. As informações contidas nesses diretórios podem ajudar os invasores a ter informações sobre o alvo.

Esta falha, é corrigida automaticamente após solucionar a vulnerabilidade de nível médio, a "*Directory Listing*", já que ela instrui o uso de configurações no HTACCESS para bloquear a listagem de diretórios do sistema (veja a Figura 39).

Outra vulnerabilidade apontada no relatório foi a "*Clickjacking: X-Frame-Options header missing*", em tradução livre "*Clickjacking*: cabeçalho *X-Frame-Options* ausente". O relatório apresentou as seguintes informações:

Figura 45 - Vulnerabilidade: *clickjacking* – cabeçalho *X-Frame-Options* ausente.



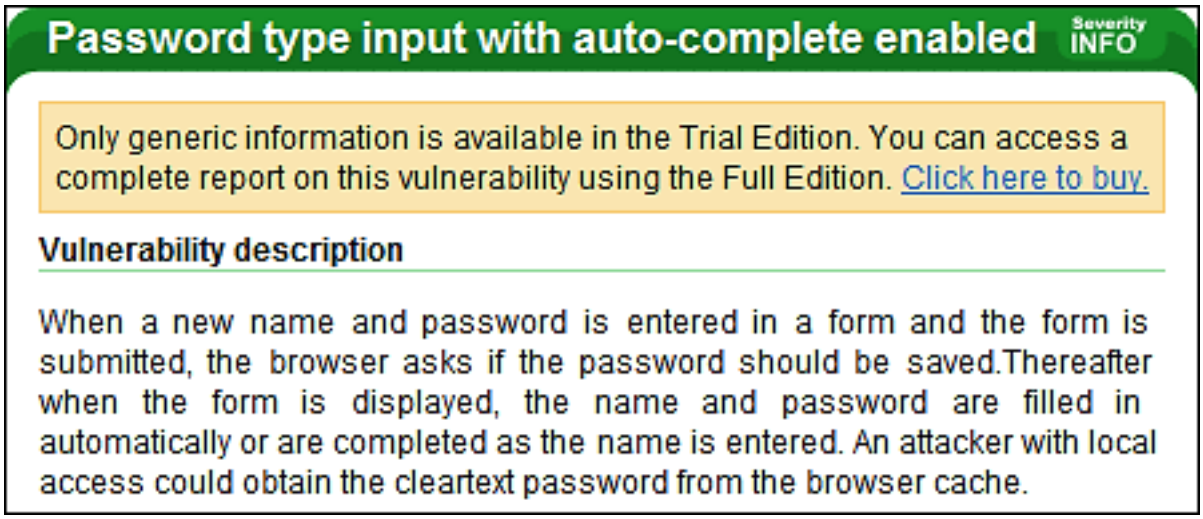
Fonte: Acunetix, 2015.

O *Clickjacking* trata-se de uma técnica maliciosa que consiste em enganar o usuário de um site ou sistema *web*, fazendo-o clicar em lugares que pareçam seguros, porém, sem perceber, o usuário está permitindo ao invasor acessar informações confidenciais ou efetuar operações maliciosas. Esta vulnerabilidade ocorre pela falta de configuração do cabeçalho *X-Frame-Options*. Sem essa configuração, sites legítimos podem ser incluídos em sites maliciosos através da *tag* <*frame*> ou <*iframe*> do HTML, levando o usuário a acreditar que está no site original (ACUNETIX, 2015).

Segundo o site *Stop Malvertising* (2015), para configurar o cabeçalho *X-Frame-Options* de forma a combater o *clickjacking*, deve-se inserir a instrução "*Header set X-Frame-Options DENY*" no arquivo HTACCESS. Ainda segundo o site, esta prática também defende o sistema do CSRF.

No quarto e último nível, as vulnerabilidades menos perigosas e de característica apenas informativa (nível 1), encontrou-se ameaças interessantes como a da Figura 46 abaixo:

Figura 46 - Vulnerabilidade: campo de senha com recurso auto completar ativo.



Fonte: Acunetix, 2015.

Como descreve o *software*, esta falha trata-se do recurso auto completar salvo em cache, permitindo que algum invasor obtenha os dados a partir do cache do navegador. Esse acesso ao cache permite que ele descubra informações sensíveis sem a permissão do usuário.

Conforme Lucas Peperaio (2015), para solucionar esta falha, a maneira mais simples é utilizar HTACCESS. Segundo ele, o código abaixo inserido em um arquivo HTACCESS corrige esta vulnerabilidade:

Figura 47 - Solucionando problema de cache com HTACCESS.

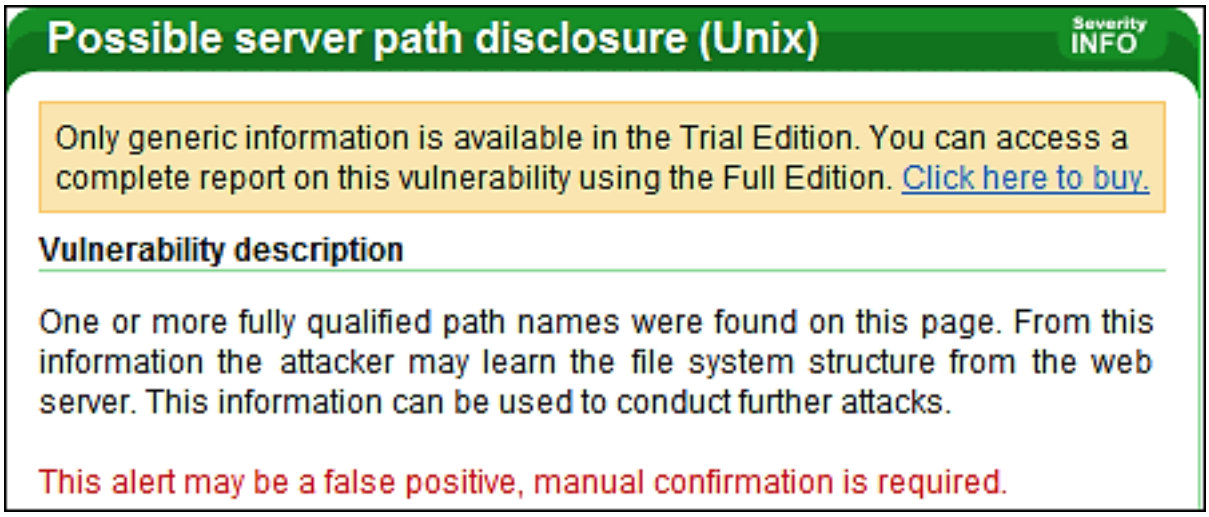
```
<IfModule mod_expires.c>
    ExpiresActive Off
  </IfModule>
  <IfModule mod_headers.c>
    Header set Cache-Control "private, no-cache, no-store, proxy-revalidate, no-transform"
  </IfModule>
</FilesMatch>
```

Fonte: Autor, 2015.

Salienta-se que esta é uma instrução que pode ajudar a combater a falha, mas não é uma garantia de sucesso, visto que atualmente os navegadores estão combatendo falhas próprias de segurança todos os dias.

A última vulnerabilidade encontrada pelo *software*, foi “Possible server path disclosure (Unix)”, ou em tradução livre “Possível divulgação do caminho do servidor (Unix)”. O *software* exibiu o seguinte alerta para esta vulnerabilidade:

Figura 48 - Vulnerabilidade: possível divulgação do caminho do servidor (Unix)



Fonte: Acunetix, 2015.

Esta vulnerabilidade permite que o invasor conheça o caminho completo do servidor, informação que pode revelar dados confidenciais como o nome do dono do site, servidor utilizado ou outras informações sensíveis (Acunetix, 2015).

Conforme o site Security Tweaks (2015), este erro já ocorreu em alguns módulos do *Wordpress*, famoso *Content Management System* (CMS), ou Sistema de Gerenciamento de Conteúdo desenvolvido em PHP, e para corrigir esta vulnerabilidade, basta não permitir a exibição de erros, já que é através deles que os invasores conhecem as informações do servidor. Para conseguir este efeito, segundo o site, basta adicionar a seguinte linha ao arquivo HTACCESS: "*php_flag display_errors off*". A instrução configura o PHP para não exibir erros impossibilitando a exibição do caminho do servidor.

Percebe-se que, apesar de aplicar diversos conceitos e técnicas para proteger o pequeno sistema, os resultados da verificação provam que não foi o suficiente para proteger o sistema de todas as vulnerabilidades. Vale salientar que o *software* Acunetix possui catalogadas 528 vulnerabilidades à data do trabalho, o que com certeza não representa o total de vulnerabilidades nas próximas semanas.

Também se percebe que a maioria dos erros apontados podem ser corrigidos através de um arquivo HTACCESS, que por sua vez, altera configurações padrão do servidor ou do PHP instalado no servidor *web*. Demonstrando que desenvolver um código bem feito e focado na segurança, pode não ser o suficiente se o servidor *web* não for bem configurado.

Para finalizar, é importante lembrar que se não fosse um sistema bem pequeno, aplicar tantas técnicas de segurança teria sido muito trabalhoso, e se houvesse um prazo muito curto para concluí-lo (fato recorrente em empresas de desenvolvimento de *software*) seria

praticamente impossível garantir o mínimo de segurança do sistema. Também vale salientar que os *softwares* são feitos por pessoas, e assim sendo, cometem erros que em alguns casos nem os próprios conseguem explicar, deixando de fazer correções ou validações que normalmente fariam.

7 CONSIDERAÇÕES FINAIS

O presente trabalho mostrou que aplicar as técnicas de codificação segura não é uma tarefa fácil, já que para cada aplicação específica, exige-se que o usuário tenha permissões e acessos específicos. No caso da pequena aplicação desenvolvida, as especificidades foram poucas pois foram definidas restrição para apenas quatro funções: inserir, editar, excluir e pesquisar notas pessoais, também não permitindo a inserção de novos usuários.

Testar o sistema com um dos *softwares* mais recomendados, o Acunetix, foi muito importante já que grandes empresas o utilizam, no entanto, todo o poder de verificação do mesmo não pode ser utilizado já que sua versão gratuita não permite testes com acesso ao código fonte, limitando-se aos testes de caixa preta, o que certamente diminuiu o número de ameaças encontradas.

O trabalho mostrou-se muito interessante em termos de aplicabilidade nas empresas. Conforme Albuquerque e Ribeiro (2002), as técnicas de segurança no desenvolvimento de *software* precisam ser difundidas por diversos aspectos, entre eles, devido ao seu impacto positivo na mitigação de riscos para as empresas, sua cooperação com as políticas de segurança interna nas empresas, fornecem *softwares* melhores e mais seguros e contribuem para a urgente necessidade de segurança digital.

Como trabalhos futuros, pretende-se:

- a aquisição do *software* Acunetix, para que seja possível testar o sistema em todos seus aspectos, corrigindo todas as ameaças de forma ainda mais simples, já que o *software* apresentaria a correção indicada e local da vulnerabilidade;
- inserir as funções de gerenciamento de usuários, gerenciamento de permissões e o uso de *tokens* com SSL na área de acesso com senha;
- e expandir o sistema de maneira que permita o gerenciamento da vida pessoal do usuário, adicionado, por exemplo, um módulo financeiro que permitisse controlar as finanças pessoais dos usuários.

Entende-se que a apresentação de exemplos, abordagens e aplicação prática das técnicas estudadas tenha contribuído para a difusão de conhecimentos na área de segurança de sistemas, principalmente no meio *web*.

REFERÊNCIAS

ABNT – Associação Brasileira de Normas Técnicas. "**ABNT NBR ISO/IEC 27002 – Tecnologia da informação – Técnicas de segurança – Código de prática para a gestão de segurança da informação".** ABNT, 2005.

ABNT CATÁLOGO – Associação Brasileira de Normas Técnicas. **Site Oficial Para Compra de Normas.** Endereço: <http://www.abntcatalogo.com.br>. Acesso em: 17/01/2015.

ACUNETIX VULNERABILITY SCANNER. **Site oficial**. Endereço: <http://www.acunetix.com>. Acesso em 02/01/2015.

ALBUQUERQUE, RICARDO; RIBEIRO, BRUNO. **"Segurança no desenvolvimento de software: como garantir a segurança do sistema para seu cliente usando a ISO/IEC"**. Rio de Janeiro: Editora Campus, 2002.

BRASIL - TRIBUNAL DE CONTAS DA UNIÃO. **"Boas práticas em segurança da informação"**. Secretaria de Fiscalização de Tecnologia da Informação – 3. Ed. – Brasília: TCU, 2008.

BRAZ, F. **Instrumentação da análise e projeto de software seguro baseada em ameaças e padrões**. Tese (Doutorado em Engenharia Elétrica) - Faculdade de tecnologia, Brasília - Brasil. 2009.

CAMPOS, A. **Sistema de segurança da informação: controlando os riscos**. 2.ed. – Florianópolis : Visual Books, 2007.

COMMON CRITERIA PORTAL. **Publications: New CC Portal**. Disponível em <http://www.commoncriteriaportal.org/cc>. Acesso em 02/01/2015.

CRUZ, E. F.; *et al*. **TSDD - Teste de segurança durante o Desenvolvimento**. União Educacional Minas Gerais – Uniminas – Uberlândia, MG – Brasil – Artigo para Especialização em Segurança da Informação, 2007.
FERREIRA, FERNANDO NICOLAU FREITAS; ARAÚJO, MÁRCIO TADEU DE. **"Política de Segurança da Informação – Guia Prático para Elaboração e Implementação"**. 2. Ed. Revisada – Rio de Janeiro: Editora Ciência Moderna Ltda, 2008.

GOMES, K. DA SILVA; SANTOS, NEWTON. **"Segurança no Desenvolvimento de Aplicações Web"**, Universidade da Amazônia – UNAMA – Centro de Ciências Exatas e Tecnológicas – CCET – Trabalho de Conclusão de Curso do Curso de Bacharel em Ciência da Computação, 2006.

HACKERONE. **"#25382 Apache mod_negotiation filename bruteforcing".** Disponível em: <https://hackerone.com/reports/25382>. Acesso em: 10/01/2015.

HASH KILLER. **"MD5 Decrypter"**. Disponível em: <http://www.hashkiller.co.uk/md5-decrypter.aspx>. Acesso em: 05/01/2015.

HTML STAFF. **“Htaccess rapidinho: Impedir listagem do diretório”**. Disponível em: <http://www.htmlstaff.org/ver.php?id=1463>. Acesso em: 20/01/2015.

KHAN, M. U. A.; ZULKERNINE, M. **Activity and Artifact Views of a Secure Software Development Process**. International Conference on Computational Science and Engineering. Vancouver - Canadá. IEEE, 2009. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5283250&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F5282954%2F5282955%2F05283250.pdf%3Farnumber%3D5283250>. Acesso em 02/01/2015.

MICROSOFT. **“Appendix B: Security Definitions for Vulnerability Work Item Tracking”**. Disponível em: <http://msdn.microsoft.com/library/cc307392.aspx>. Acesso em: 20/11/2014.

MICROSOFT. **“Appendix B: Security Definitions for Vulnerability Work Item Tracking”**. Disponível em: <http://www.microsoft.com/security/sdl/default.aspx>. Acesso em: 02/01/2015.

OWASP. **“Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet”**. Disponível em: <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet>. Acesso em 20/01/2015.

PEPERAIO, LUCAS. **“Cache de navegador com .htaccess”**. Disponível em: <http://www.lucaspeperaio.com.br/blog/cache-de-navegador-com-htaccess>. Acesso em: 09/01/2015.

PHP.NET. **“PHP: Injeção de SQL - Manual”**. Disponível em: <http://php.net/manual/pt_BR/security.database.sql-injection.php> Acesso em: 12/01/2015.

HOWARD, MICHAEL; LEBLANC, DAVID. **“Escrevendo código seguro: estratégias, técnicas práticas para codificação segura de aplicativos em um mundo em rede”**. 2.ed. – Porto Alegre: Bookman, 2005.

LYRA, MAURÍCIO ROCHA. **“Segurança e auditoria em Sistemas de Informação”**. Rio de Janeiro: Editora Ciência Moderna Ltda., 2008.

RAMSAROOP, PETER. “Cybercrime, Cyberterrorism, and Cyberwarfare: Critical Issues in Data Protection for Health Services Information Systems”. Washington / Estados Unidos da América : Editora PAHO, 2003.

RISTIC, IVAN. “**Apache Security”**. O’Relly Media, 2005.

SECURITY TWEAKS. “**Security Tweak's: HOW TO FIX WORDPRESS FULL PATH DISCLOSURE ERROR**”. Disponível em <http://www.securitytweaks.com/2013/12/how-to-fix-wordpress-full-path.html>. Acesso em 17/01/2015.

STACK OVERFLOW. “**php Session cookies http & secure flag - how do you set these?**”. Disponível em <http://stackoverflow.com/questions/22221807/session-cookies-http-secure-flag-how-do-you-set-these>. Acesso em: 15/01/2015.

STOP MALVERTISING. “**.htaccess HTTP Headers | Securing your website with .htaccess | Security**”. Disponível em < http://stopmalvertising.com/security/securing-your-website-with-.htaccess/.htaccess-http-headers.html >. Acesso em: 16/01/2015.

THE NEW YORK TIMES - PERLROTH, NICOLE; SANGER, DAVID E. **“North Korea Loses Its Link to the Internet”**. Disponível em: <http://www.nytimes.com/2014/12/23/world/asia/attack-is-suspected-as-north-korean-internet-collapses.html?_r=0>. Acesso em 30/12/2014.

UNICODE CONSORTIUM. “**Acclaim for Unicode”**. Disponível em <http://www.unicode.org/announcements/quotations.html>. Acesso em: 11/01/2015.