

UNIVERSIDADE FEDERAL DO RIO GRANDE  
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE  
SISTEMAS

ANDREI ESPELOCIM GARCIA

**Guardian of the Requests: Proxy Reverso para prevenção de Cross-Site Request Forgery**

Trabalho de Conclusão apresentado como  
requisito parcial para a obtenção do grau de  
Tecnólogo em Análise e Desenvolvimento de  
Sistemas

Prof. Márcio Josué Ramos Torres  
Orientador

Rio Grande, julho de 2016

## **CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Espelocim Garcia, Andrei

Proxy reverso para prevenção de ataques CSRF/ Andrei Espelocim Garcia. – Rio Grande: Tecnologia em Análise e desenvolvimento de sistemas da FURG, 2016.

69 f.: il.

Trabalho de Conclusão de Curso (tecnólogo) – Universidade federal do Rio Grande. Tecnologia em análise e desenvolvimento de sistemas, Rio Grande, BR–RS, 2016. Orientador: Márcio Josué Ramos Torres.

1. Proxy Reverso. 2. Segurança da Informação. 3. CSRF. I. Torres Josué Ramos, Márcio. II. Título.

## **FOLHA DE APROVAÇÃO**

Monografia sob o título “Guardian of the Requests: Proxy Reverso para prevenção de Cross-Site Request Forgery”, defendida por Andrei Espelocim Garcia e aprovada em 05 de Julho de 2016, em Rio Grande, estado do Rio Grande do Sul, pela banca examinadora constituída pelos professores:

---

Prof. Márcio Josué Ramos Torres  
Orientador

---

Prof. Igor Avila Pereira  
IFRS-Campus Rio Grande

---

Prof. Luciano Vargas Gonçalves  
IFRS-Campus Rio Grande

## **AGRADECIMENTOS**

Agradeço a minha família, amigos e colegas por estarem presentes em toda a trajetória do curso dando apoio e me incentivando.

Agradeço especialmente a minha mãe Joice Ane Espelocim, pois desde o começo torceu pela minha vitória, incentivou e me ajudou com tudo que foi possível para que eu concluísse o curso.

Agradeço aos meus professores por me passarem um pouco de seus conhecimentos e experiência, o que já contribuiu muito para o meu desenvolvimento pessoal e profissional.

Um agradecimento em especial a meu orientador Márcio Torres, que me ajudou na escolha do tema deste trabalho e também esteve incentivando e ajudando com tudo que esteve ao seu alcance.

## LISTA DE ABREVIATURAS E SIGLAS

- XML *Extensible Markup Language*
- SSL *Secure Socket Layer*
- HTTP *Hypertext Transfer Protocol*
- HTTPS *Hypertext Transfer Protocol Secure*
- FTP *File Transfer Protocol*
- CSS3 *Cascading Style Sheets 3*
- HTML *HyperText Markup Language*
- PHP *Hypertext Preprocessor*
- URL *Uniform Resource Locator*

## Sumário

1	Introdução.....	12
1.1	Motivação.....	12
1.2	Objetivo.....	13
1.3	Metodologia.....	13
2	Segurança da informação.....	14
2.1	Conceito Geral de Segurança da Informação.....	14
2.2	Características para manter uma informação segura.....	14
2.2.1	Integridade.....	14
2.2.2	Disponibilidade.....	14
2.2.3	Confidencialidade.....	15
2.3	Vulnerabilidades na informação.....	15
2.3.1	Origem das vulnerabilidades.....	15
2.3.1.1	Naturais.....	15
2.3.1.2	Organizacional.....	16
2.3.1.3	Física.....	16
2.3.1.4	Hardware.....	16
2.3.1.5	Software.....	16
2.3.1.6	Meios de armazenamento.....	17
2.3.1.7	Humanas.....	17
2.3.1.8	Comunicação.....	17
2.4	Ameaças da informação.....	17
2.5	Considerações finais deste capítulo.....	18
3	Segurança na internet.....	19
3.1	Ataques na Internet.....	20
3.1.1	Exploração de vulnerabilidades.....	20
3.1.1.1	OWASP Top 10 2013.....	20
3.1.2	Varreduras em redes.....	22
3.1.3	Falsificação de <i>e-mail</i> .....	23
3.1.4	Intercepção de tráfego.....	23
3.1.5	Força Bruta.....	23
3.1.6	Desfiguração de página.....	23
3.1.7	Negação de serviço.....	24
3.1.8	Prevenção.....	24
3.2	Considerações finais desse capítulo.....	24
4	Cross-Site Request Forgery (CSRF).....	25
4.1	Introdução à CSRF.....	25
4.2	Como funciona o ataque.....	25
4.2.1	Ataque CSRF via URL.....	25
4.2.1.1	Disparando a requisição a partir de um link colocado em uma página aleatória na internet:.....	26
4.2.1.2	<i>Escondendo</i> imagem no e-mail:.....	26
4.2.2	Ataque via POST.....	27
4.2.2.1	Ataque via formulário por POST.....	27
4.2.2.2	Ataque via formulário utilizando JavaScript.....	27
4.3	Risco que CSRF oferece para o negócio.....	28
4.4	Como prevenir CSRF.....	29
4.4.1	Soluções que não funcionam.....	29
4.4.1.1	Só aceitar solicitações <i>POST</i> .....	29

4.4.1.2	Usar um Cookie secreto com um token.....	29
4.4.1.3	Validar o cabeçalho HTTP_REFERER.....	29
4.4.2	Prevenção contra CSRF.....	29
4.4.2.1	Prevenção sem um token.....	30
4.4.3	Prevenção pelo usuário.....	31
4.4.4	OWASP CSRFGuard.....	31
4.5	Considerações Finais do Capítulo.....	32
5	Evitando CSRF com um Proxy Reverso.....	33
5.1	Conceito de proxy.....	33
5.1.1	Funcionamento de um Proxy.....	33
5.1.2	Exemplos de <i>proxys</i> disponíveis.....	34
5.1.2.1	SQUID.....	34
5.1.2.2	Winconnection.....	34
5.1.2.3	Omne Wall.....	34
5.2	O que é um servidor <i>proxy</i> reverso.....	35
5.2.1	Benefícios de um Proxy Reverso.....	35
5.3	Justificativa de escolha do Proxy Reverso.....	36
5.4	Ferramentas utilizadas no desenvolvimento do <i>proxy</i> reverso.....	36
5.4.1	Linguagem de Programação Ruby.....	36
5.4.2	Rack.....	37
5.4.3	HTTPClient.....	37
5.4.4	Sistema Operacional Ubuntu.....	38
5.4.5	GitHub.....	38
5.5	Projeto do <i>proxy</i> reverso.....	38
5.5.1	Especificações do projeto.....	39
5.5.2	Licença.....	39
5.5.3	Fluxograma do projeto.....	39
5.6	Considerações finais desse capítulo.....	42
6	Desenvolvimento.....	43
6.1	Bibliotecas auxiliares no desenvolvimento.....	43
6.1.1	Sinatra.....	43
6.1.2	Nokogiri.....	44
6.1.3	SecureRandom.....	46
6.1.4	Rufus-Scheduler.....	47
6.2	Classes do projeto.....	48
6.2.1	Classes auxiliares.....	50
6.3	Diagrama de classes.....	50
7	Estudo de Caso.....	52
7.1	Sistema acadêmico.....	52
7.1.1	Login.....	52
7.1.2	Atualizando notas.....	53
7.2	O problema.....	56
7.3	Prevenindo ataque CSRF a formulário com um <i>proxy</i> reverso.....	57
7.3.1	Criando o arquivo para iniciar o <i>proxy</i> .....	57
7.3.2	Configurando o <i>proxy</i> reverso.....	58
7.3.3	Iniciando o <i>proxy</i> reverso.....	59
7.3.4	Inserindo um <i>token</i> em um formulário com o <i>proxy</i> reverso.....	61
7.3.5	Negando POST com o <i>proxy</i> reverso.....	63
8	Conclusão.....	70
8.1	Trabalhos futuros.....	71

Referências.....72



## Lista de Figuras

Figura 1: URL com parâmetros para salvar os dados no servidor.....	25
Figura 2: URL com parâmetros para explorar CSRF.....	26
Figura 3: Link para a aplicação vulnerável parecendo ser para a página do Google.....	26
Figura 4: Exemplo de imagem sem altura e largura para disparar uma requisição.....	26
Figura 5: Formulário para enviar dados para o servidor utilizando o método POST.....	27
Figura 6: Formulário sendo submetido via JavaScript ao carregar o documento.....	28
Figura 7: Descrição dos agentes de ameaça, exploração, detecção, impactos técnicos e de negocio relacionado à CSRF.....	28
Figura 8: Formulário com token oculto para prevenir CSRF.....	30
Figura 9: Arquitetura da bliblioteca CSRFGuard (OWASP).....	31
Figura 10: Imagem mostrando o funcionamento de um proxy convencional. Fonte: SILVA pg. 20.	34
Figura 11: Servidor proxy reverso. Fonte: SILVA pg. 20.....	35
Figura 12: Requisição para o servidor utilizando a interface Rack. Fonte: autoria própria.....	37
Figura 13: Fluxograma de uma requisição GET para o proxy reverso. Fonte: autoria própria.....	40
Figura 14: Requisição POST para o proxy reverso.....	41
Figura 15: Comando para instalar o Sinatra no terminal. Fonte: autoria própria.....	43
Figura 16: Requisição GET utilizando Sinatra. Fonte: autoria própria.....	43
Figura 17: Exemplo de servidor web Utilizando Sinatra. Fonte: autoria própria.....	44
Figura 18: Requisição GET pelo servidor do Sinatra. Fonte: autoria própria.....	44
Figura 19: Instalando a gem Nokogiri pelo terminal. Fonte: autoria própria.....	44
Figura 20: Código fonte de um arquivo html. Fonte: autoria própria.....	45
Figura 21: Pesquisa por formulário em um arquivo html utilizando a gem Nokogiri. Fonte: autoria própria.....	45
Figura 22: Inserindo elemento no html utilizando a gem Nokogiri. Fonte: autoria própria.....	46
Figura 23: Comando para instalar a gem SecureRandom. Fonte: autoria própria.....	46
Figura 24: Utilizando o método uuid do módulo SecureRandom. Fonte: autoria própria.....	47
Figura 25: Utilizando o método every para realizar uma tarefa. Fonte: autoria própria.....	47
Figura 26: Execução do método every no terminal. Fonte: autoria própria.....	48
Figura 27: Diagrama de classes do proxy reverso. Fonte: autoria própria.....	51
Figura 28: Tela de login do sistema acadêmico. Fonte: autoria própria.....	53
Figura 29: Tela inicial do sistema. Fonte: autoria própria.....	53
Figura 30: Página de listagem das notas dos alunos. Fonte: autoria própria.....	54
Figura 31: Formulário para atualizar as notas de um aluno. Fonte: autoria própria.....	54
Figura 32: Código fonte do formulário da Figura 31. Fonte: autoria própria.....	55
Figura 33: Código fonte mostrando como é feita a atualização de nota de um aluno no servidor. Fonte: autoria própria.....	55
Figura 34: Tela de listagem de notas do sistema mostrando as notas atualizadas conforme o formulário da Figura 31. Fonte: autoria própria.....	56
Figura 35: Formulário para realizar um POST forjado para o sistema. Fonte: autoria própria.....	56
Figura 36: Código fonte do formulário da Figura 35. Fonte: autoria própria.....	57
Figura 37: Resultado do envio dos dados do formulário da Figura 35. Fonte: autoria própria.....	57
Figura 38: Referência para o arquivo da classe GuardianOfRequests do proxy reverso. Fonte: autoria própria.....	58
Figura 39: Definindo as configurações do proxy reverso. Fonte: autoria própria.....	59
Figura 40: Iniciando o proxy reverso. Fonte: autoria própria.....	59
Figura 41: Informações no terminal sobre o servidor do proxy reverso. Fonte: autoria própria.....	60
Figura 42: Fazendo login no sistema acadêmico através do proxy reverso. Fonte: autoria própria.	60

Figura 43: Requisições GET para o servidor do proxy reverso.....	61
Figura 44: Método possuiElemento da classe Html. Fonte: autoria própria.....	61
Figura 45: Método gerarTokens da classe ListaDeTokens. Fonte: autoria própria.....	62
Figura 46: Método insereInputsNosFormularios da classe Html. Fonte: autoria própria.....	62
Figura 47: Login do sistema acadêmico através do proxy reverso. Fonte: autoria própria.....	63
Figura 48: Código fonte de um Formulário com token. Fonte: autoria própria.....	63
Figura 49: Formulário para realizar uma requisição POST para o sistema acadêmico. Fonte: autoria própria.....	64
Figura 50: Código fonte do formulário da Figura 49. Fonte: autoria própria.....	64
Figura 51: Inserção de tokens na requisição GET. Fonte: autoria própria.....	65
Figura 52: Método excluiTokensExpirados da classe ListaDeTokens. Fonte: autoria própria.....	65
Figura 53: Lógica executada pela classe ProxyBase nas requisições POST. Fonte: autoria própria. .....	66
Figura 54: Método podePostar da classe ProxyBase. Fonte: autoria própria.....	67
Figura 55: Mensagem de retorno do proxy após a tentativa de enviar o formulário da Figura 49. Fonte: autoria própria.....	67
Figura 56: Interface do aplicativo Postman. Fonte: autoria própria.....	68
Figura 57: Status de retorno do servidor proxy reverso. Fonte: autoria própria.....	68

## RESUMO

Com o passar dos anos houve uma grande evolução da plataforma *web*, com essa evolução cresceu o número de dispositivos que acessam essa plataforma nos dias de hoje, como por exemplo: televisores, *smartphones*, *notebooks*, computadores *desktops*, entre outros. Para utilizar esses dispositivos são desenvolvidas diversas aplicações. É normalmente nessas aplicações que passam nossas informações todo dia, como por exemplo: acessar uma página de compras coletivas, pagar contas *online*, acessar páginas de bancos para realizar transações bancárias, entre outras. Contudo nem sempre o quesito segurança é observado na hora de desenvolver essas aplicações, podendo ser deixadas vulnerabilidades nas mesmas, que são exploradas por pessoas mal-intencionadas para roubar ou usar de forma indevida as informações. Sendo assim, durante este trabalho foi realizado um estudo sobre Segurança da Informação e segurança para *internet* e apresentada uma proposta de prevenção, segundo a Open Web Application Security Project (OWASP), de uma das dez vulnerabilidades mais críticas para aplicações da plataforma *web*, o Cross Site Request Forgery (CSRF). Para cumprir o que foi proposto, que era prevenir a exploração de CSRF, foi implementado um servidor *proxy* reverso. A ferramenta implementada é gratuita e de livre acesso, tendo como objetivo prevenir aplicações para qualquer linguagem de programação ou servidor *web* para o qual as aplicações foram desenvolvidas. Após realizar o desenvolvimento do *proxy* reverso foi planejada e desenvolvida uma aplicação vulnerável a ataques CSRF e elaborado um estudo de caso em cima desta aplicação, demonstrando se foi possível prevenir ataques CSRF. Ao final foi apresentada uma conclusão do trabalho, demonstrando o resultado dos testes elaborados em cima do estudo de caso e as sugestões de melhorias futuras para o *proxy* reverso.

# 1 INTRODUÇÃO

Com o passar dos anos, desde a criação da *web* até hoje houve uma grande evolução das tecnologias dessa área, permitindo que o ambiente *web* se popularize cada vez mais. Atualmente existem cada vez mais usuários utilizando aplicações disponíveis para esta plataforma, sendo através de computadores *desktops*, *notebooks* ou até mesmo de dispositivos móveis, permitindo a troca e acesso de informações com usuários de todo o mundo a qualquer instante.

Com essa grande popularização cada vez mais programadores vem se especializando em soluções para o ambiente *web*. No entanto, muitas vezes os quesitos de segurança não são levados em consideração, deixando as aplicações vulneráveis a falhas que são exploradas livremente por pessoas mal-intencionadas.

Existem hoje em dia, diversas vulnerabilidades conhecidas no ambiente *web*, essas vulnerabilidades permitem que dados sigilosos dos usuários sejam roubados e com isso cometidos crimes cibernéticos<sup>1</sup>. Dito isso se vê cada vez mais a necessidade de serem desenvolvidas soluções capazes de proteger as aplicações de criminosos que exploram suas vulnerabilidades.

No trabalho a seguir será dado ênfase em uma que é considerada, segundo a Open Web Application Security Project (OWASP), umas das dez vulnerabilidades mais críticas da *internet*, o Cross-Site Request Forgery (CSRF). Essa vulnerabilidade explora a sessão de autenticação de um usuário com uma determinada aplicação para disparar requisições para a mesma através de outras aplicações, ou até mesmo através da própria, utilizando os dados de sessão do usuário autenticado.

A proposta para prevenção desta vulnerabilidade será a criação de um *proxy* reverso, conceito que será explicado ao longo do trabalho. O *proxy* será responsável por tratar as requisições de uma determinada aplicação requisitada a um servidor *web* protegido pelo *proxy*. Para fazer a proteção das aplicações o *proxy* usará técnicas estudadas ao longo do trabalho que previnam a exploração de ataques CSRF.

## 1.1 Motivação

Como dito anteriormente, a plataforma *web* é um ambiente que vem crescendo e se popularizando com o passar do tempo, com essa crescente popularização cada vez mais são desenvolvidas aplicações para a esta plataforma, tendo em base isso uma motivação para este

---

<sup>1</sup> São atividades ilegais praticadas com o uso um computador conectado a *internet*, uma rede pública ou privada de computadores ou um sistema de computador. (BullGard).

trabalho é o estudo de tecnologias de desenvolvimento de *softwares* para esta plataforma.

Também é usado como fator motivacional o fato de no decorrer do curso não ser discutido em nenhuma disciplina, assuntos relacionados com Segurança da Informação ou segurança para *internet*.

Sendo assim a grande motivação do trabalho é prover a solução de algum problema importante para a área de Segurança da Informação dando ênfase na área de desenvolvimento de *software* para a plataforma *web*.

## 1.2 Objetivo

O objetivo geral do trabalho é fazer um estudo, projetar e implementar um *proxy* reverso para prevenir CSRF. Sendo desmembrado nos seguintes objetivos específicos:

- Estudo sobre Segurança da Informação;
- Estudo detalhado sobre segurança na *internet*, possíveis vulnerabilidades e como os usuários podem ser atacados;
- Estudar detalhadamente CSRF, fazendo um estudo desde as fases onde ocorre o ataque, até as possíveis soluções para prevenir a vulnerabilidade;
- Fazer uma pesquisa das ferramentas e tecnologias que serão utilizadas para projetar e desenvolver o *proxy* reverso;
- Projetar e desenvolver o *proxy* reverso;
- Fazer um estudo de caso em uma aplicação que não ofereça proteção contra CSRF;
- Concluir o trabalho mostrando os resultados obtidos através dos estudos.

## 1.3 Metodologia

A metodologia do trabalho será:

- Leitura de artigos e publicações disponibilizados na *internet*;
- Consultas realizadas em livros;
- Levantamentos seguidos de testes das ferramentas necessárias para a implementação do *proxy* reverso e do estudo de caso;
- Implementação do *proxy* reverso e do estudo de caso;
- Avaliação dos resultados utilizando o *proxy* reverso em conjunto com o estudo de caso implementado;

## **2 SEGURANÇA DA INFORMAÇÃO**

Neste capítulo será apresentada uma introdução sobre as três características fundamentais de Segurança da Informação que, segundo o livro Segurança da informação: uma abordagem focada em risco (DANTAS, 2011), são a integridade, confidencialidade e disponibilidade. Também será apresentada uma introdução sobre ameaças da informação e vulnerabilidades da informação.

### **2.1 Conceito Geral de Segurança da Informação**

O termo Segurança da Informação é um termo muito amplo, sendo que suas definições podem variar. Segue a definição descrita na norma ISO 27002:

É a proteção da informação contra vários tipos de ameaças para garantir a continuidade do negócio, minimizar riscos, maximizar o retorno sobre os investimentos e as oportunidades de negócios (ISO 27002).

A norma ISO 27002 estabelece diretrizes e princípios gerais para iniciar, implementar, manter e melhorar a gestão de Segurança da Informação em uma organização, servindo como guia prático para desenvolver procedimentos de Segurança da Informação.

### **2.2 Características para manter uma informação segura**

Para que se possa manter com segurança uma informação é necessário que se garanta três características fundamentais: integridade, disponibilidade e confidencialidade.

No que se diz respeito as três características citadas anteriormente, o termo Segurança da Informação está relacionado com a preservação das mesmas. Se for preservado essas características em uma informação ela vai ter garantido o valor que possui para um indivíduo ou uma organização (DANTAS, 2011). A seguir será explicado o conceito de integridade, disponibilidade e confidencialidade.

#### **2.2.1 Integridade**

O conceito de integridade da informação é garantir que ela seja mantida original, que não seja modificada ou destruída sem autorização e que permaneça consistente. Ocorrerá a quebra da integridade quando a informação for corrompida, falsificada ou alterada (DANTAS, 2011).

#### **2.2.2 Disponibilidade**

A disponibilidade da informação é garantir que os usuários autorizados possuem acesso

sempre que necessário. A quebra da disponibilidade acontece quando a informação não está disponível para seus usuários no momento em que é preciso (DANTAS, 2011).

### **2.2.3 Confidencialidade**

A confidencialidade diz respeito a informação estar disponível apenas para quem tem o acesso concedido a ela, vetando qualquer outra pessoa que não esteja autorizada a acessar a informação.

A quebra de confidencialidade ocorre quando pessoas não autorizadas têm acesso à informação. Assegurar a confidencialidade da informação é assegurar o valor que ela possui (DANTAS, 2011).

## **2.3 Vulnerabilidades na informação**

Segundo a norma NBR ISO/IEC 27002:2005, uma vulnerabilidade pode ser definida como uma fragilidade de um ativo ou grupo de ativos que pode ser explorada por uma ou mais ameaças. Um outro ponto de vista é de que vulnerabilidades são fragilidades que podem provocar danos decorrentes da utilização de dados em qualquer fase do ciclo de vida das informações (DANTAS, 2011).

### **2.3.1 Origem das vulnerabilidades**

Uma vulnerabilidade pode vir de vários lugares, por exemplo: instalações físicas desprotegidas contra incêndio, inundações e desastres naturais, material inadequado empregado nas construções, ausência de políticas de segurança para RH, funcionários sem treinamento e insatisfeitos nos locais de trabalho, ausência de procedimentos de controle de acesso e de utilização de equipamentos por pessoal contratado, equipamentos obsoletos, sem manutenção e sem restrições para sua utilização, software sem pacote de atualização e sem licença de funcionamento, entre outras (DANTAS, 2011).

As origens das vulnerabilidades podem ser classificadas como: naturais, organizacionais, físicas, de hardware<sup>2</sup>, software, de meios de armazenamento, humanas e nas comunicações (DANTAS, 2011).

#### **2.3.1.1 Naturais**

Uma vulnerabilidade natural pode ser definida como algo relacionado com condições da

---

<sup>2</sup> É toda a parte física de um equipamento eletrônico, como por exemplo: memórias de computador, processadores, teclados, mouse, etc.

natureza ou do meio ambiente. Exemplos: um local que seja sujeito a incêndio, locais próximos a rio ou lagoas propensos a enchentes, áreas onde são verificadas manifestações da natureza como terremotos, maremotos, furacões, zonas de inundação, entre outras que estejam relacionadas com condições da natureza ou do meio ambiente.(DANTAS, 2011).

### 2.3.1.2 Organizacional

As vulnerabilidades de origem organizacional são as que dizem respeito aos procedimentos e planos, políticas e todo e qualquer tipo de controle interno de uma organização que não esteja em outras classificações. Por exemplo: falta de política de segurança e treinamento, falhas ou ausência de processos procedimentos e rotinas, falta de planos de contingência, recuperação de desastres e de continuidade, entre outras que se encaixe nesse conceito (DANTAS, 2011).

### 2.3.1.3 Física

Podem ser instalações inadequadas, ausência de recursos para combate a incêndio, disposição desordenada dos cabos de energia e de rede, não identificação de pessoas e locais, acessos desprotegidos as salas de computadores, sistema deficiente, entre outras que se encaixe nesse conceito (DANTAS, 2011).

### 2.3.1.4 Hardware

As vulnerabilidades de *hardware* se caracterizam por possíveis defeitos de fabricação ou configuração de equipamentos que podem permitir ataques ou alteração nos mesmos a fim de causar danos a informação. Exemplos de vulnerabilidades causadas por *hardware*: conservação inadequada de equipamentos, falta de equipamentos de contingência, *firmware*<sup>3</sup> desatualizado, sistemas mal configurados, gerenciamento de *hardware* sendo feito por uma *interface* publica, entre outras que se encaixe nesse conceito (DANTAS, 2011).

### 2.3.1.5 Software

São vulnerabilidades que se enquadram todos as aplicações que possuem algum ponto fraco, permitindo o acesso indevido de outras pessoas. Os principais pontos em que se encontram as vulnerabilidades são na instalação e configuração indevida, o uso de e-mail que permitem a execução de códigos que causam algum dano, editores de texto que permitem a execução de vírus de macro, entre outras que se enquadrem nesse conceito (DANTAS, 2011).

---

3 É um conjunto de instruções operacionais que são programadas diretamente em um *hardware*.



### 2.3.1.6 Meios de armazenamento

São todos os tipos de armazenamentos com suporte físico ou magnético que se utiliza para armazenar as informações, como por exemplo: CD-ROM, fita magnética, discos rígidos, pendrives, entre outros que se enquadrem nessas características. Normalmente as vulnerabilidades nesses dispositivos advêm da utilização incorreta, defeitos de fabricação, mofo, magnetismo ou estática, local de armazenamento em áreas insalubres ou com alto nível de umidade (DANTAS, 2011).

### 2.3.1.7 Humanas

Normalmente são vulnerabilidades causadas por falha humana de pessoas que não conhecem os procedimentos corretos quando utilizam uma aplicação. A origem dessas vulnerabilidades pode ser: a falta de capacitação específica para executar a função que lhe é proposta, não utilização de criptografia na comunicação de informações de alto nível crítico, elaboração de senhas fracas no ambiente de trabalho, falta de conhecimentos sobre segurança, entre outras que se enquadrem nessas características (DANTAS, 2011).

### 2.3.1.8 Comunicação

São as vulnerabilidades que se enquadram sobre os meios em que a informação trafega (cabo, satélite, fibra óptica, ondas de rádio, telefone, *internet*, *fax*, etc.). O que contribui para essas vulnerabilidades são: a qualidade do ambiente onde a informação trafega, o armazenamento e a leitura dessa informação, falta de criptografia nas comunicações, problemas nos protocolos de rede, falta de filtro entre os segmentos de rede, e outras que se enquadrem nessas características (DANTAS, 2011).

## 2.4 Ameaças da informação

Com a automação dos sistemas de processamento e de armazenamento de informações, a informação tornou-se mais suscetível às ameaças, uma vez que ela está mais acessível e disponível para os usuários de uma forma geral. A norma ISO/IEC 13335-1:2004 define ameaças da informação como: a causa potencial de um incidente indesejado, que pode resultar em dano para um sistema ou para a organização (DANTAS, 2011).

As ameaças da informação podem ser: naturais que são aquelas que se originam de fenômenos da natureza, tais como: terremotos, furacões, enchentes, maremotos, tsunamis. Involuntárias, que são as que resultam de ações desprovidas de intenção para causar algum dano.

Geralmente são causadas por acidentes, erros, ou por ação inconsciente de usuários, tais como, vírus eletrônicos. Intencionais, que são aquelas deliberadas, que objetivam causar danos, tais como, fraudes, vandalismo, sabotagens, espionagem, invasões e furtos de informações (DANTAS, 2011).

## **2.5 Considerações finais deste capítulo**

Neste capítulo foram apresentadas as três características fundamentais para se manter uma informação segura e explicado sobre as ameaças e vulnerabilidades que deixam uma informação em risco.

### 3 SEGURANÇA NA INTERNET

Segundo notícia lida, o Brasil em 2015 deve superar o Japão em acessos a *internet*, sendo a quarta população ligada a internet (CORREIO BRAZILIENSE, 2015). Com essa informação, pode ser notado o quanto a *internet* está presente em nossas vidas nos dias de hoje, nos trazendo uma vasta gama de possibilidades de acesso, como por exemplo:

- Acessar sites de notícias;
- Realizar transações bancárias para pagamentos de contas de água, luz, telefone entre outras;
- Verificar extratos bancários;
- Realizar compras *online*;
- Jogar jogos *online*;
- Usar redes sociais para fazer novas amizades;
- Fazer downloads de arquivos de músicas ou vídeos.

Com todas essas possibilidades e facilidades que a *internet* nos traz, nem sempre estamos seguros. As vezes estamos suscetíveis a alguns riscos, como por exemplo:

- **Acesso a conteúdos impróprios:** existem muitos sites pornográficos disponíveis na *internet*, esses sites contêm conteúdo abusivo e podem ser acessados por menores de idade (CERT, 2012);
- **Contato com pessoas mal-intencionadas:** é muito comum pessoas tentarem se passar por outras em salas de bate-papos ou redes sociais, atraindo vítimas para praticar golpes, pedofilia ou até mesmo sequestros em alguns casos (CERT, 2012);
- **Furto e perda de dados:** pessoas com grandes conhecimentos em programação podem criar códigos maliciosos que podem roubar dados dos computadores pessoais, sem que a vítima perceba (CERT, 2012);
- **Invasão de privacidade:** muitas vezes a divulgação de nossas informações podem comprometer nossa vida pessoal e até mesmo de familiares, visto que, mesmo restringindo seu acesso, não é garantido que outras pessoas não consigam acessar. (CERT, 2012).

Existem alguns outros risco, que não foram estudados mais detalhadamente, que são segundo a CERT: janelas de *pop-up*, *plug-ins*<sup>4</sup>, complementos e extensões, *links* patrocinados,

---

<sup>4</sup> São programas desenvolvidos por terceiros que podem ser instalados em um navegador *web* ou leitor de *e-mail* (CERT, pg 42).

*banners* de propagandas, programas de distribuição de arquivos, *cookies*<sup>5</sup> e compartilhamento de recursos.

### 3.1 Ataques na Internet

Ataques na internet normalmente ocorrem por diversos objetivos, sendo visados diferentes alvos e utilizadas muitas técnicas para concluir e executar. Qualquer computador conectado a internet pode ser atacado ou participar de um ataque (CERT, 2012).

Os objetivos que levam a esses ataques são bastante variados, indo de motivos pessoais até crimes financeiros, como por exemplo:

1. **Demonstração de poder:** mostrar que uma empresa tem alguma vulnerabilidade no seu sistema e invadi-lo, para assim tentar vender serviços (CERT, 2012);
2. **Prestígio:** vangloriar-se perante outros *crackers*<sup>6</sup> por ter conseguido atacar um sistema ou site considerado difícil, fazer competição de quem consegue atacar mais sites ou sistemas vulneráveis (CERT, 2012);
3. **Motivações financeiras:** aplicar golpes na internet para conseguir proveitos financeiros (CERT, 2012);
4. **Motivações ideológicas:** atacar sites que o conteúdo exponha opinião contrária a do *cracker* (CERT, 2012);
5. **Motivações comerciais:** invadir sites ou sistemas de empresas concorrentes a fim de comprometer a imagem da empresa perante os seus clientes (CERT, 2012).

#### 3.1.1 Exploração de vulnerabilidades

Uma vulnerabilidade é a exploração de alguma falha no projeto, em configurações, em equipamentos de redes ou serviços. Quando a falha é explorada gera uma quebra de segurança, gerando assim um ataque que ocorre quando um *cracker* tenta explorar a vulnerabilidade para tentar invadir um sistema, disparar ataques contra outros computadores, acessar informações confidenciais e tornar serviços inacessíveis (CERT, 2012).

##### 3.1.1.1 OWASP Top 10 2013

O OWASP Top 10 2013 é um documento que reúne uma série de informações sobre as dez

---

5 São pequenos arquivos que são gravados no computador do usuário quando o mesmo visita um site na internet. Geralmente são utilizados para guardas informações de acesso do usuário no site em que ele visitou (CERT, pg 40).

6 Uma pessoa que tem alto conhecimento com computadores e usa esse conhecimento para quebrar a segurança de um sistema (OLHARDIGITAL).

vulnerabilidades consideradas mais críticas da *web*. Este documento é atualizado de tempos em tempos, tendo sua última atualização em 2013, e sendo a penúltima em 2010. Ele é baseado em oito conjuntos de dados de sete empresas que são especializadas em segurança de aplicações. As dez vulnerabilidades são selecionadas de acordo com a prevalência, em combinação com estimativas do consenso da exploração, detecção e impacto (OWASP, 2013).

Segundo a OWASP:

O objetivo principal do OWASP Top 10 é educar desenvolvedores, projetistas, arquitetos, gestores e organizações sobre as consequências das mais importantes vulnerabilidades de segurança de aplicações web. O Top 10 fornece técnicas básicas para se proteger contra essas áreas problemáticas de alto risco e também fornece orientação sobre onde ir a partir daqui (OWASP, 2013).

A lista com as dez vulnerabilidades consideradas mais críticas atualizadas em 2013 são:

1. **Injeção<sup>7</sup>**: é a injeção de código não confiável em um interpretador por um *cracker* de modo a iludir o interpretador a realizar comandos indesejados ou manipular dados de consultas. A injeção pode ser: *sql injection* ou falhas de sistema operacional;
2. **Quebra de Autenticação e Gerenciamento de Sessão**: é a implementação incorreta com o que diz respeito as funções de autenticação e gerenciamento de sessão do usuário, permitindo explorar falhas e ter acesso às senhas, *tokens*<sup>8</sup> e dados sigilosos dos usuários;
3. **Cross-Site Scripting (XSS)**: falhas de XSS permitem que *crackers* insiram e executem *scripts* em páginas *web* a fim de executar algum código malicioso que possa prejudicar o usuário de alguma maneira;
4. **Referência Insegura e Direta a Objetos**: essa falha ocorre quando o programador deixa exposto uma referência direta a algum objeto, como um arquivo ou diretório, permitindo o acesso a conteúdos sem autorização;
5. **Configuração Incorreta de Segurança**: uma boa segurança exige a definição de uma configuração segura para a aplicação. Servidores de aplicação, servidores *web*, banco de dados e *frameworks*<sup>9</sup>, devem ter suas configurações definidas, implementadas e mantidas. Adicionalmente, o software deve ser mantido atualizado;
6. **Exposição de Dados Sensíveis**: é a exposição de dados sensíveis por parte das aplicações como por exemplo: número de cartões de crédito, identificações fiscais, dados pessoais e credenciais de autenticação. Isso abre uma brecha para *crackers* usarem esses dados para cometerem fraudes;
7. **Falta de Função para Controle do Nível de Acesso**: grande parte das aplicações

---

<sup>7</sup> Em inglês significa *Injection*.

<sup>8</sup> Um valor único, gerado automaticamente usando algum algoritmo específico (OWASP).

<sup>9</sup> Um *framework* é uma abstração de classes comuns entre vários projetos, provendo uma funcionalidade genérica.

verificam o controle de acesso em nível de função antes de tornar uma certa funcionalidade visível para um usuário. Porém essa verificação deveria ser feita no também no servidor, a fim de evitar requisições forjadas;

8. **Cross Site Request Forgery (CSRF):** é uma vulnerabilidade que explora a autenticação de um usuário em uma aplicação para realizar uma ação em nome do mesmo. Geralmente usa técnicas de Engenharia Social<sup>10</sup> para enganar a vítima e forçá-la a realizar uma requisição forjada, a fim de que a aplicação legítima aceite e valide a requisição como se a ação estivesse sido executada pelo usuário autenticado na aplicação real;
9. **Utilização de Componentes Vulneráveis Conhecidos:** é a utilização de algum componente como modulo de *software*, biblioteca ou *framework* que tenha alguma vulnerabilidade conhecida. O uso de algum desses componentes vulneráveis, se explorado, pode causar sérias perdas de dados pessoais ou até mesmo danos a um servidor;
10. **Redirecionamentos e Encaminhamentos Inválidos:** redirecionamentos são frequentes em aplicações *web*. Sem uma validação adequada destes dados podemos ser redirecionados para sites de *phishing*<sup>11</sup> ou *malware*<sup>12</sup>, ou usar encaminhamentos para acessar páginas não autorizadas.

Como pode ser visto após essa explicação das dez vulnerabilidades mais críticas da *web*, segundo a OWASP, o CSRF possui um lugar de destaque, sendo considerado umas das vulnerabilidades mais críticas dentre muitas existentes no ambiente *web*, o que justifica a sua escolha para ser estudado mais detalhadamente neste trabalho. A seguir virá uma lista de ataques na *internet* que podem ser usados para explorar as vulnerabilidades citadas anteriormente.

### 3.1.2 Varreduras em redes

Técnica utilizada para varrer uma rede de computadores e identificar os que estão ativos, a fim de coletar informações, como serviços utilizados, sistema operacional e programas instalados (CERT, 2012).

---

10 Pode ser definido como o ato de enganar ou iludir uma pessoa para obter informações ou forçar a mesma a executar uma ação (Ian Mann, 2011).

11 É um tipo de fraude que ocorre por meio de mensagens eletrônicas aplicada por uma pessoa mal-intencionada, para obter dados pessoais ou financeiros de um usuário (CERT, pg 9).

12 São programas criados para executar uma ação danosa ou realizar atividades maliciosas em um computador. (CERT, pg 23).

### 3.1.3 Falsificação de *e-mail*

É uma técnica que consiste em alterar os campos de cabeçalhos de *e-mail* a fim de simular o seu envio aparentando que ele foi enviado de uma determinada origem, quando na verdade foi enviado de outra (CERT, 2012).

Essa técnica utiliza de características no protocolo SMTP<sup>13</sup>, que permite que os campos do cabeçalho como “From” (campo de quem enviou a mensagem), “Reply-to” (endereço de resposta da mensagem) e “Return-path” (endereço para onde possíveis erros da mensagem serão reportados), sejam falsificados (CERT, 2012).

### 3.1.4 Intercepção de tráfego

Técnica que consiste por meio de programas chamados *sniffers*<sup>14</sup> capturar dados que trafegam em uma determinada rede de computador. Essa captura pode ser de forma legítima por um administrador da rede para monitorar ou de forma maliciosa por *crackers*, com intenção de roubar informações (CERT, 2012).

### 3.1.5 Força Bruta

Consiste em adivinhar por tentativa de erro, o usuário e senha de alguém para acessar sistemas ou aplicações *web* e executar processos com os seus privilégios. As tentativas de adivinhações costumam ser baseadas em (CERT, 2012):

- Ataques de dicionários que são baixados na internet.
- Listas de palavras conhecidas, como personagens de histórias ou times de futebol.
- Sequências numéricas do teclado como: “123456”, “abcde”.
- Fazer um estudo de Engenharia Social sobre a vítima.

### 3.1.6 Desfiguração de página

Técnica que consiste em alterar o conteúdo de uma determinada página *web*. As principais formas de realizar esta técnica são (CERT, 2012):

- Exploração de erros das aplicações *web*;
- Exploração de vulnerabilidades do servidor *web* onde se encontra a aplicação;
- Exploração de vulnerabilidades da linguagem de programação da aplicação;
- Invadir o servidor e alterar diretamente os arquivos;
- Furtar as senhas para acesso à administração da aplicação.

---

<sup>13</sup> É um protocolo do tipo ASCII simples utilizado pra o envio de mensagens eletrônicas na internet (Tanenbaum, Andrew. S).

<sup>14</sup> São programas utilizados para inspecionar os dados trafegados em uma rede de computadores (CERT, pg 19) .

### 3.1.7 Negação de serviço

Técnica que consiste através da utilização de um computador retirar um serviço do ar, um computador ou uma rede conectada a *internet*, o nome desta técnica é conhecida como Denial of Service (DoS). Um outro nome conhecido relacionado a esse ataque é Distributed Denial of Service (DdoS), que é quando se utiliza vários computadores de forma distribuída para aplicar os ataques (CERT, 2012).

### 3.1.8 Prevenção

Para prevenção dos ataques citados anteriormente é recomendado ao usuário: proteger seus dados, fazer o uso de programas de proteção como antivírus e *firewalls*<sup>15</sup> e manter o computador sempre atualizado. Algumas medidas podem ser adotadas como, tais como (CERT, 2012) :

1. Cuidar na hora de escolher senhas, pois senhas fáceis podem serem facilmente roubadas. Não utilizar senha do tipo: “12345” ou “abdce”;
2. Utilizar sites seguros que ofereçam sistema de criptografia na navegação;
3. Manter o computador limpo de códigos maliciosos;
4. Instalar programas antivírus confiáveis;
5. Não expor dados confidenciais;

## 3.2 Considerações finais desse capítulo

Foi explicado neste capítulo algumas possíveis vulnerabilidades presentes na *internet*, ataques utilizados para explorar essas vulnerabilidades e a lista das dez vulnerabilidades mais críticas encontradas na *web* segundo a OWASP. Também foi explicado como os usuários podem se prevenir destas vulnerabilidades presentes no ambiente *web*.

---

15 São programas de segurança instalados em um computador para controlar o acesso entre duas redes de computadores, protegendo acessos indevidos a um computador (CERT, 2016).



## 4 CROSS-SITE REQUEST FORGERY (CSRF)

Neste capítulo será explicado o que é a vulnerabilidade conhecida como CSRF, possíveis danos que um ataque direcionado para essa vulnerabilidade possa vir a causar, exemplos de ataques que exploram CSRF e também técnicas utilizadas para proteger as aplicações dessa vulnerabilidade.

### 4.1 Introdução à CSRF

CSRF é uma vulnerabilidade que explora a autenticação de um usuário com uma aplicação. Para explorar essa vulnerabilidade são geralmente utilizadas técnicas de Engenharia Social em conjunto com as falhas presentes nas aplicações. Ataques CSRF induzem um usuário a realizar uma ação involuntária para a aplicação o qual está autenticado com diversos objetivos, como por exemplo: alterar dados, obter informações indevidas, realizar ações em nome do usuário, etc (OWASP, 2015).

### 4.2 Como funciona o ataque

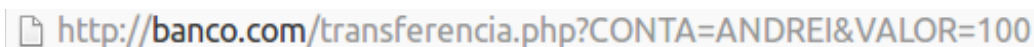
Para realizar um ataque CSRF o usuário precisa estar autenticado na aplicação vulnerável. Após o usuário estar autenticado são utilizadas técnicas de Engenharia Social para induzir o usuário a realizar a ação indevida ou realizar a ação na própria aplicação explorando as vulnerabilidades presentes.

Por meio de Engenharia Social são criadas páginas falsas contendo formulários ou *links* forjados para induzir usuário a realizar a ação indevida.

Outro meio de realizar um ataque CSRF é explorando as vulnerabilidades presentes em formulários que aceitam a inserção de *scripts*. A seguir serão demonstradas as formas mais comuns de explorar CSRF em uma aplicação.

#### 4.2.1 Ataque CSRF via URL

Este exemplo mostra a exploração dos parâmetros do método GET<sup>16</sup> presente em uma URL:



<http://banco.com/transferencia.php?CONTA=ANDREI&VALOR=100>

Figura 1: URL com parâmetros para salvar os dados no servidor.

Fonte: autoria própria.

---

16 É um método implementado pelo protocolo HTTP, utilizado para buscar um documento no servidor (Silva).

A Figura 1 mostra a URL de uma aplicação utilizando os parâmetros GET para, possivelmente, salvar dados no servidor. Caso essa aplicação só confiasse na autenticação do usuário, esta URL poderia explorar um ataque CSRF. Na figura 2 pode ser observado como poderia ser explorada essa URL para realizar o ataque.

A screenshot of a web browser address bar showing the URL: http://banco.com/transferecia.php?CONTA=HACKER&VALOR=10000. The text is in a light blue font on a white background.

Figura 2: URL com parâmetros para explorar CSRF.

Fonte: autoria própria.

Como pode ser observado na figura 2, apenas trocando os parâmetros na URL seria possível realizar um ataque CSRF nesta aplicação se a mesma não fizesse nenhuma validação no lado do servidor. A seguir serão demonstradas algumas formas de utilizar uma URL com parâmetros via GET para realizar um ataque CSRF.

#### 4.2.1.1 *Disparando a requisição a partir de um link colocado em uma página aleatória na internet:*

Ao visitar alguma página na *internet* o usuário pode se deparar com *links* chamativos para outras páginas. Esses *links* irão realizar uma requisição para a aplicação vulnerável se aproveitando da sessão do usuário. A figura 3 mostra um *link* induzindo o usuário a visitar a página do Google:

```
<a href="http://banco.com/transferecia.php CONTA=HACKER&VALOR=10000">google.com </a>
```

Figura 3: Link para a aplicação vulnerável parecendo ser para a página do Google.

Fonte: autoria própria.

Na figura 3 pode ser visto que ao visitar a página que contém o *link* o usuário não estaria visitando a página do Google, mas sim disparando uma requisição para a aplicação onde seria realizada alguma ação indevida em seu nome.

#### 4.2.1.2 *Escondendo imagem no e-mail:*

Essa técnica consiste em enviar uma imagem sem altura e largura para o *e-mail* do usuário. Assim que esse *e-mail* for aberto será disparada uma requisição para a aplicação vulnerável. Na Figura 4 pode ser observado como inserir uma imagem sem altura e largura no *e-mail*.

```
<img src = "http://banco.com/transferecia.php?CONTA=Hacker&VALOR=100000" width = "0" height = "0" border = "0">
```

Figura 4: Exemplo de imagem sem altura e largura para disparar uma requisição.

A Figura 4 mostra como poderia ser realizado um ataque CSRF enviando uma imagem escondida no *e-mail*. Durante o carregamento deste *e-mail* seria feita uma requisição para a imagem, onde o endereço da requisição seria o da aplicação vulnerável. Neste caso se não houvesse uma validação no lado do servidor poderia ocorrer um ataque CSRF.

#### 4.2.2 Ataque via POST

Os exemplos anteriores mostraram a exploração de CSRF utilizando os parâmetros de uma URL via método GET. Porém um ataque CSRF também pode ser explorado utilizando o método POST<sup>17</sup>, a seguir exemplos de ataques CSRF via POST (OWASP, 2015).

##### 4.2.2.1 Ataque via formulário por POST

A Figura 5 mostra um formulário configurado para enviar dados para o servidor utilizando o método POST.

```
<form action = "http://banco.com/transferencia.php" method = "POST">
  <input type = "hidden" name = valor "CONTA" = "HACKER" />
  <input type = "hidden" name = valor "VALOR" = "100000" />
  <input type = "submit" value = "Ver as minhas imagens" />
</form>
```

Figura 5: Formulário para enviar dados para o servidor utilizando o método POST.

Fonte: autoria própria.

Conforme mostra a Figura 5, esse formulário induz um usuário a ver algumas imagens, sendo que ao tentar vê-las, o mesmo enviaria o formulário para a aplicação vulnerável e caso não tivesse algum tipo de validação no servidor, essa postagem poderia resultar em um ataque CSRF (OWASP, 2015).

##### 4.2.2.2 Ataque via formulário utilizando JavaScript

A Figura 6 mostra o carregamento de um formulário via JavaScript:

---

17 É um método implementado no protocolo HTTP para enviar dados para um servidor (Silva).

```

<body onload="document.forms[0].submit()">
  <form action="http://banco.com/transferecia.php" method="POST">
    <input type="hidden" name=valor "CONTA"="HACKER" />
    <input type="hidden" name=valor "VALOR"="100000" />
    <input type="submit" value="Ver as minhas imagens" />
  </form>
</body>

```

Figura 6: Formulário sendo submetido via JavaScript ao carregar o documento.

Fonte: autoria própria.

O código JavaScript presente no formulário da Figura 6 submete o mesmo de forma automática ao carregar a página, desta forma esse formulário poderia ser inserido em alguma página da *internet* ou enviado por *e-mail* para realizar uma tentativa de ataque CSRF.

### 4.3 Risco que CSRF oferece para o negócio

A Figura 7 foi retirada do OWASP Top 10 2013, onde mostra uma visão dos agentes de ameaça, exploração, detecção, impactos técnicos e de negócio relaciona a CSRF.



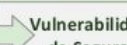
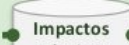

				
Agentes de Ameaça	Vetores de Ataque	Vulnerabilidades de Segurança	Impactos Técnicos	Impactos no Negócio
<b>Específico da Aplicação</b>	<b>Exploração MÉDIO</b>	<b>Prevalência COMUM</b>	<b>Detecção FÁCIL</b>	<b>Impacto MODERADO</b>
Considere qualquer pessoa que possa carregar conteúdo nos navegadores dos usuários, e assim forçá-los a fazer uma requisição para seu site. Qualquer site ou outro serviço html que usuários acessam pode fazer isso.	O atacante forja requisições HTTP falsas e engana uma vítima submetendo-a a um ataque através de tags de imagem, XSS, ou inúmeras outras técnicas. <u>Se o usuário estiver autenticado</u> , o ataque é bem sucedido.	O <u>CSRF</u> se aproveita do fato de que a maioria das aplicações web permitem que os atacantes prevejam todos os detalhes de uma ação particular da aplicação. Como os navegadores automaticamente trafegam credenciais como cookies de sessão, os atacantes podem criar páginas web maliciosas que geram requisições forjadas indistinguíveis das legítimas. Detecção de falhas de CSRF é bastante simples através de testes de penetração ou de análise de código.	Os atacantes podem enganar suas fazendo com que executem operações de mudança de estado que a vítima está autorizada a realizar, por ex., atualizando detalhes da sua conta, comprando, deslogando ou até mesmo efetuando login.	Considere o valor de negócio dos dados ou funções afetadas da aplicação. Imagine não ter a certeza se os usuários tem a intenção de realizar tais ações.  Considere o impacto na sua reputação.

Figura 7: Descrição dos agentes de ameaça, exploração, detecção, impactos técnicos e de negocio relacionado à CSRF.

Fonte: (OWASP Top 10, 2013).

A visão sobre CSRF apresentada na Figura 7, foi construída com base na metodologia OWASP Risk Methodology<sup>18</sup>, desenvolvida pela OWASP, onde são fornecidas informações genéricas sobre os agentes de aplicação, exploração, prevalência e impactos de negocio, para que possa ser identificado o risco dessa vulnerabilidade relacionado com o negócio de uma organização.

18 Mais informações disponíveis em: [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).

## 4.4 Como prevenir CSRF

Neste tópico serão apresentadas formas de prevenção de ataques CSRF conforme as recomendações da OWASP.

### 4.4.1 Soluções que não funcionam

Algumas soluções que não funcionam segundo a OWASP:

1. Só aceitar solicitações `POST`;
2. Usar um *cookie* secreto com um *token*;
3. Validar o cabeçalho `HTTP_REFERER`<sup>19</sup>.

#### 4.4.1.1 Só aceitar solicitações `POST`

Só aceitar solicitações `POST` não deixa uma aplicação protegida contra CSRF. Como já foi demonstrado no tópico 4.2.2 deste capítulo as requisições via `POST` podem ser exploradas para realizar ataques CSRF, desta forma somente aceitar o método `POST` para o envio de formulários não seria o suficiente para proteger aplicações de ataques CSRF.

#### 4.4.1.2 Usar um *Cookie* secreto com um *token*

Este método não funciona porque todos os *cookies* são enviados junto com cada requisição. Deste modo os *tokens* de autenticação que estiverem presentes em algum *cookie* seriam submetidos independentemente de haver um ou não e a requisição forjada seria aceita no servidor. Além disso não é possível identificar de qual usuário veio a sessão (OWASP, 2015).

#### 4.4.1.3 Validar o cabeçalho `HTTP_REFERER`

É uma técnica que consiste em verificar o cabeçalho `HTTP_REFERER`, que identifica de onde originou uma requisição. No entanto, este cabeçalho pode ser alterado pelo cliente e também não está presente em solicitações que utilizam o protocolo `HTTPS`. Portanto, esta técnica também não é eficaz contra CSRF.

### 4.4.2 Prevenção contra CSRF

Atualmente os desenvolvedores de aplicações *web* que queiram prevenir ataques CSRF são encorajados a utilizar um padrão de projeto denominado Sincronizador de Token (em inglês: Synchronizer Token) (OWASP, 2015).

Esse padrão requer a geração de *tokens* aleatórios que serão inseridos em *links* e formulários

---

<sup>19</sup> É um campo do cabeçalho do protocolo HTTP que especifica a URL de um documento a partir de onde partiu o pedido (W3C).

que estejam envolvidos com transações que manipulem dados sensíveis dos usuários, como por exemplo: transações bancárias, alterações de senhas, compras *online*, entre outras que se enquadrem nessas características. Sempre que for feito uma requisição e a resposta desta requisição contenha *links* e formulários que envolvam essas transações sensíveis para o usuário, o *token* deve ser validado, se tudo ocorrer bem um novo *token* deve ser gerado para que (em uma próxima solicitação) essa verificação seja feita novamente (OWASP, 2015).

É recomendável que esses *tokens* sejam introduzidos em campos ocultos de formulários, que os pedidos para o servidor utilizem o método `POST` e que o nome e valor do campo sejam gerados automaticamente utilizando algum algoritmo (OWASP, 2015). A seguir, Figura 8 mostra um exemplo de formulário com um *token* oculto.

```
<form action="http://banco.com/transfere&#223;cia.php" method="POST">
  <input type = "text" name="CONTA" value="andrei"/>
  <input type = "text" name="VALOR" value="1000"/>
  <input type = "hidden" name="csrf_protect" value="HRDAadd23jJSSDS8998ddJDDJA29dsjsSHDHD83829" />
  <input type = "submit" value = "transferir"/>
</form>
```

Figura 8: Formulário com *token* oculto para prevenir CSRF.

Para melhorar ainda mais a segurança da aplicação é recomendável que o valor e nome do *token* sejam trocados a cada solicitação feita pelo usuário. É recomendável que os *tokens* sejam protegidos da mesma forma que se protege a identificação de sessão do usuário, não deixando por exemplo, os *tokens* expostos na URL (OWASP, 2015).

#### 4.4.2.1 Prevenção sem um token

Alguns desenvolvedores utilizam técnicas sem o uso de *tokens* para tentar prevenir CSRF, por exemplo (OWASP):

- **Verificar o cabeçalho HTTP\_REFERER:** é muito comum verificar o cabeçalho `HTTP_REFERER` para validar de onde está originando uma requisição. No entanto, é considerado uma das mais fracas medidas de prevenção contra CSRF. Uma aplicação que utilize o protocolo HTTPS omite o cabeçalho `HTTP_REFERER`. Aplicações que utilizam *proxy* podem ter o cabeçalho omitido por questões de segurança, desta forma esta validação não é confiável (OWASP, 2015);
- **Verificar o ORIGIN HTTP\_HEADER:** o `ORIGIN HTTP_HEADER` foi um padrão introduzido como prevenção de CSRF e outros ataques tipos de ataques. Ao contrário do que acontece com o `HTTP_REFERER`, este cabeçalho está sempre presente em uma

requisição, mesmo em solicitações HTTPS (OWASP, 2015);

- **Challenge-Response:** o Desafio-Resposta pode ser utilizado para prevenção contra CSRF, podendo citar como exemplo para uso deste método o *captcha*.

#### 4.4.3 Prevenção pelo usuário

A seguir algumas práticas recomendadas para os usuários se protegerem de ataques CSRF (OWASP):

- Fazer *logout* imediatamente após usar alguma aplicação *web*;
- Não salvar nomes de usuários e senhas nos navegadores;
- Usar navegadores diferentes para operações diferentes, não utilizando o mesmo navegador que se navega livremente na *internet* do que se utiliza, por exemplo, para acessar a página *web* de um banco;
- Ter cuidado com leitores de *e-mail*, pois podem possuir códigos maliciosos que executarão sem que o usuário saiba, explorando algum ponto vulnerável da aplicação para realizar um ataque CSRF.

#### 4.4.4 OWASP CSRFGuard

A OWASP CSRFGuard é uma biblioteca desenvolvida pela OWASP para prevenção de ataques CSRF que utiliza o padrão Synchronizer Token. Essa biblioteca é integrada através da utilização de um filtro JavaEE, que permite através de vários métodos automáticos ou manuais, introduzir um *token* pseudoaleatório em páginas *html*.

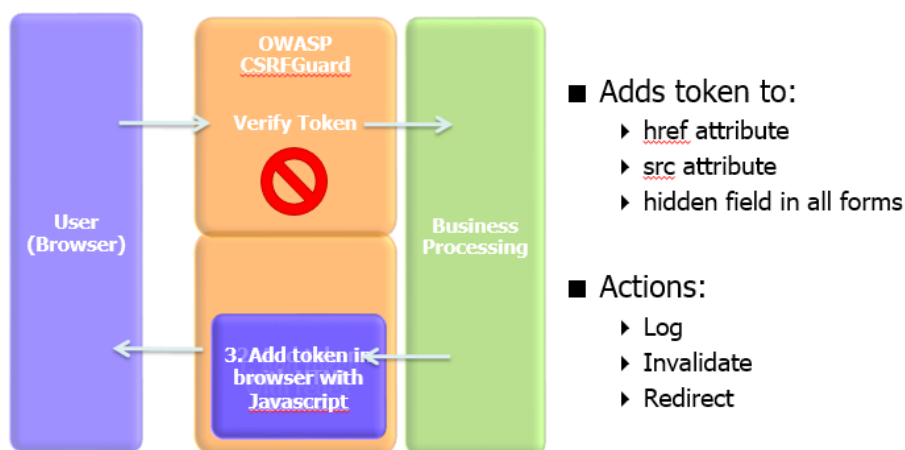


Figura 9: Arquitetura da biblioteca CSRFGuard (OWASP).

A Figura 9 mostra o fluxo e inserção de um *token* utilizando a biblioteca CSRFGuard. São

inseridos *tokens* em formulários e elementos *html* que contenham atributos *href*<sup>20</sup> e *src*<sup>21</sup>. O uso desta biblioteca é específico para aplicações desenvolvidas na linguagem Java.

## 4.5 Considerações Finais do Capítulo

Este capítulo detalhou o que é CSRF desde o conceito geral, possíveis formas de explorar o ataque e como as aplicações. Ao final do capítulo foi apresentado uma biblioteca desenvolvida pela OWASP que serve para prevenir CSRF em aplicações desenvolvidas na linguagem Java.

---

20 É um atributo presente nas seguintes *tags html*: `<a>`, `<area>`, `<base>`, `<link>`. Serve para indicar a URL para onde o *link* dessas *tags* ira realizar uma requisição (W3SCHOOLS).

21 É um atributo presente nas seguintes *tags html*: `<audio>`, `<embed>`, `<iframe>`, `<img>`, `<input>`, `<script>`, `<source>`, `<track>`, `<video>`. Serve para especificar a URL de um arquivo.



## 5 EVITANDO CSRF COM UM PROXY REVERSO

Neste capítulo será apresentada uma introdução sobre *proxy* e *proxy* reverso, justificando a escolha de um *proxy* reverso para prevenir CSRF, apresentando as tecnologias e ferramentas relacionadas com o projeto e desenvolvimento do *proxy* reverso e detalhada as etapas do projeto.

### 5.1 Conceito de proxy

Primeiramente, para que se entenda como funciona um *proxy* reverso, é preciso que se entenda o conceito por trás de um *proxy*.

#### 5.1.1 Funcionamento de um Proxy

O termo *proxy* significa um representante, procurador, alguém que vai fazer algo em nome de outros que poderá agir em seu nome (SILVA, 2014).

Os servidores *proxy* ficam situados entre um cliente e um servidor *web*, fazendo a comunicação entre uma requisição do cliente para o servidor de destino (SILVA, 2014).

Em uma forma mais simples de implementação, um servidor *proxy* facilita a comunicação de um cliente com um servidor. Quando o cliente faz a requisição ao servidor *web*, o *proxy* se passa pelo cliente perante o servidor *web*, assim se a resposta for bem-sucedida ele traz para o cliente que solicitou (SILVA, 2014).

Em formas mais avançadas de suas implementações, um servidor *proxy* pode servir como um filtro perante a determinadas regras configuradas no *proxy*. Essas regras são geralmente baseadas em endereço de IP, protocolo de acesso, tipos de conteúdos, entre outros (SILVA, 2014).

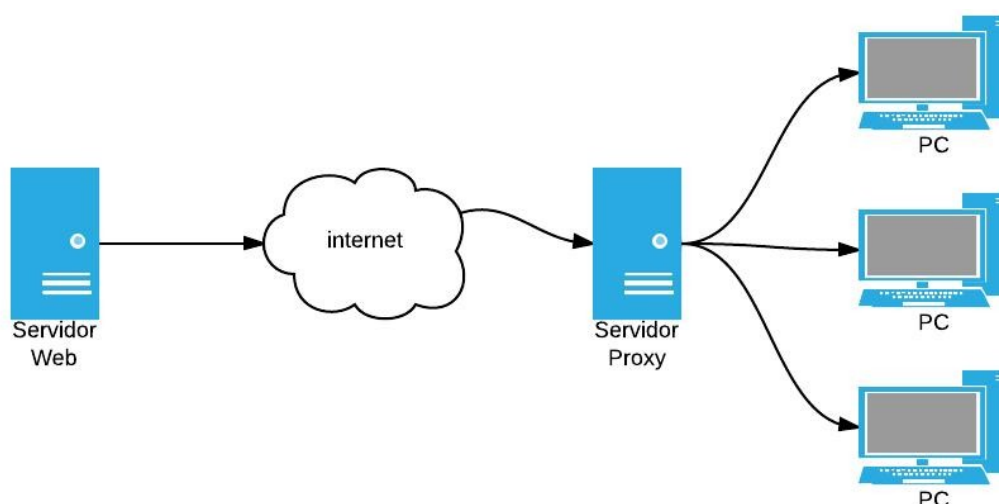


Figura 10: Imagem mostrando o funcionamento de um *proxy* convencional.  
Fonte: SILVA pg. 20.

Como pode ser observado na Figura 10 o servidor *proxy* fica instalado no cliente, desta forma as requisições para o servidor *web* passam antes por ele.

### 5.1.2 Exemplos de *proxys* disponíveis

Existe hoje uma vasta gama de *proxys* disponíveis para uso, cada um com suas particularidades. Alguns exemplos de servidores *proxy* são:

#### 5.1.2.1 SQUID

O Squid é um *proxy* de armazenamento para *cache*<sup>22</sup> de páginas *web*, suportando os protocolos de rede HTTP, HTTPS, FTP, entre outros. Ele melhora o tempo de resposta de uma solicitação e também otimiza a largura de banda utilizada, fazendo o *cache* das páginas mais solicitadas. É suportado pelos mais conhecidos sistemas operacionais, por exemplo Linux e Windows. (SQUID-CACHE, 2015).

#### 5.1.2.2 Winconnection

É um *proxy* desenvolvido no Brasil pela empresa Winco Tecnologia e Sistemas. Lançado em 1998 inicialmente para plataforma Microsoft Windows, funciona com uma licença proprietária, podendo ser usado no máximo cinco computadores na rede (SILVA, 2014).

#### 5.1.2.3 Omne Wall

O Omne Wall é um *firewall* criado pela BRConnection que conta com diversas camadas de

<sup>22</sup> É um recurso utilizado para guardar dados que são muito utilizados para um acesso futuro dos mesmos. Os dados são armazenados em algum local onde ficam disponível para o acesso, o *cache* está presente em navegadores, processadores, entre outros.

segurança, uma delas conta com um *proxy* que possui controle de acesso e filtro de dados (BRConnection, 2015).

## 5.2 O que é um servidor *proxy* reverso

O *proxy* reverso é um servidor de rede que fica instalado do lado do servidor *web*, sendo responsável por filtrar as requisições feitas pelos clientes antes delas chegarem ao servidor *web*. A Figura 11 mostra um exemplo de servidor *proxy* reverso.

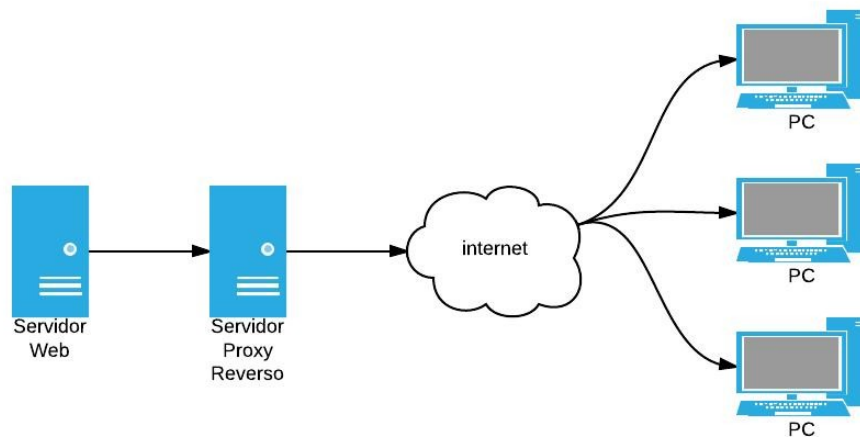


Figura 11: Servidor *proxy* reverso. Fonte: SILVA pg. 20

### 5.2.1 Benefícios de um Proxy Reverso

Com a utilização de um *proxy* reverso podemos ter diversos benefícios, por exemplo (FIDELIS, 2013):

- **Roteamento de requisições:** o *proxy* reverso pode definir as rotas das requisições;
- **Segurança:** como o *proxy* é a única *interface* externa da rede, ele esconde os demais servidores. É o caso desse trabalho;
- **Criptografia:** a criptografia SSL pode ser delegada ao *proxy* ao invés dos servidores internos;
- **Balanceamento de carga:** o servidor pode distribuir a carga para vários servidores da rede;
- **Cache:** assim como o servidor *web*, o *proxy* reverso pode manter em *cache* o conteúdo estático das requisições realizadas, ajudando a diminuir a carga;
- **Compressão:** o *proxy* reverso pode tornar o acesso mais rápido através da compressão do conteúdo acessado.

### 5.3 Justificativa de escolha do Proxy Reverso

Existem hoje soluções de bibliotecas para prevenir ataques CSRF, como por exemplo a CSRFGuard já citada neste trabalho. No entanto, essas bibliotecas geralmente são específicas de alguma linguagem de programação, como é o caso da própria CSRFGuard que limita-se a linguagem de programação Java para a plataforma *web*. Desta forma uma das justificativas de escolha de um *proxy* reverso para prevenir aplicações de ataques CSRF é de criar uma ferramenta capaz de prevenir ataques contra aplicações desenvolvida para qualquer linguagem de programação. Um outro ponto que será positivo com a utilização do *proxy* é não precisar ser acoplado nenhuma biblioteca externa nas aplicações que forem protegidas por ele, pois toda a lógica para realizar a prevenção dos ataques fica encapsulada no *proxy*.

### 5.4 Ferramentas utilizadas no desenvolvimento do *proxy* reverso

Neste tópico serão explicadas as ferramentas utilizadas durante o projeto e desenvolvimento do *proxy* reverso. Durante o tópico, serão dados exemplos de como instalar e utilizar essas ferramentas.

#### 5.4.1 Linguagem de Programação Ruby

A Linguagem de programação foi criada por Yukihiro Matz Matsumoto, unindo partes de suas linguagens favoritas (Perl, Smalltalk, Eiffel, Ada e Lisp) para formar uma nova linguagem que equilibra a programação funcional com a programação imperativa (RUBY-LANG, 2015).

Ruby é uma linguagem totalmente livre, podendo ser copiada ou modificada. Seu foco é na simplicidade e produtividade. Tem uma sintaxe elegante de leitura natural e fácil escrita. É totalmente orientada a objetos, a seguir algumas de suas características (RUBY-LANG,2015):

- Possui tratamento de exceções como Java ou Python para facilitar o tratamento de erros (RUBY-LANG, 2015);
- Possui *mark-and-sweep garbage collector* para todos os objetos (RUBY-LANG, 2015);
- Pode carregar bibliotecas de extensões se o sistema operacional permitir (RUBY-LANG, 2015);
- Tem um sistema de *threading* independente do Sistema Operacional. (RUBY-LANG, 2015);
- É altamente portátil, desenvolvida principalmente em ambiente GNU/Linux, mas trabalha em muitos tipos de ambientes UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, entre outros (RUBY-LANG, 2015).

O motivo de escolha da linguagem Ruby para o desenvolvimento do *proxy* reverso é porque o autor utilizou a linguagem em alguns outros momentos e pode constatar sua simplicidade e produtividade. Deste modo foi visto uma oportunidade de aprender uma nova linguagem de programação que pode contribuir futuramente para a carreira de trabalho.

#### 5.4.2 Rack

Rack é uma *interface* para diversos tipos de servidores que suportam a linguagem Ruby (CAELUM, 2015).

Para instalar a *interface* Rack basta digitar no terminal do Ubuntu o comando “*gem install rack*” (após confirmar será instalado a *gem*).

No trecho de código a seguir será demonstrado como criar um servidor *web* utilizando a *interface* Rack:

```
run Proc.new {|env| [200, {"Content-Type" => "text/html"},
                    ["Hello World"]]}
```

Após instalar a *gem* deve-se criar um arquivo e copiar o código do exemplo acima. O seguinte passo é salvar o arquivo como, “*servidorComRack.ru*” e executar no terminal o comando “*rackup servidorComRack.ru*”, o nome do arquivo pode ser qualquer um mas a extensão precisa ser “*ru*”. Seguidos estes passos é só acessar via navegador pelo endereço “*localhost:9292*” e conferir o retorno (CAELUM,2015).

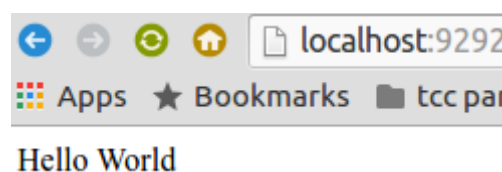


Figura 12: Requisição para o servidor utilizando a *interface* Rack. Fonte: autoria própria.

A Figura 12 mostra o retorno do servidor *web* criado utilizando a *interface* Rack.

A justificativa de escolha do Rack se deu pelo fato dele oferecer uma *interface* única para vários servidores desenvolvidos para aplicações da linguagem Ruby.

#### 5.4.3 HTTPClient

HTTPClient é uma biblioteca desenvolvida por Hiroshi Nakamura que contém uma API para aplicações escritas em Ruby realizarem requisições GET, POST, PUT, DELETE. Neste trabalho

esta biblioteca será utilizada para realizar as requisições `GET` e `POST` para os servidores das aplicações em que o *proxy* estiver protegendo. (HTTPCLIENT, 2015).

#### 5.4.4 Sistema Operacional Ubuntu

O Ubuntu é um sistema operacional com versões para *desktop*, *notebooks* e servidores. É um *software* totalmente livre e gratuito. Possui instalado diversos programas como por exemplo, editores de texto, navegadores e leitores de *e-mails* (UBUNTU-BR ORG, 2015).

O Ubuntu foi desenvolvido visando a segurança, sendo que o usuário tem atualizações de graça por, pelo menos, dezoito meses. Na versão LTS a garantia das atualizações são de três anos (UBUNTU-BR ORG, 2015).

A escolha deste sistema operacional para o projeto é pelo motivo do autor já utilizá-lo ao longo do curso e se sentir mais familiarizado com o mesmo para o desenvolvimento de trabalhos e projetos (UBUNTU-BR ORG, 2015).

#### 5.4.5 GitHub

O GitHub é um *site* onde podem ser hospedados, gratuitamente ou com modalidades pagas, os códigos de um projeto. Ele utiliza para controle de versões o Git (ferramenta responsável por gerenciar as versões de um projeto) desenvolvido por Linus Torvalds e utilizado no desenvolvimento do Kernel do Linux (GITHUB. FEATURES, 2015).

O GitHub possui recursos interessantes semelhantes a redes sociais. Nele podemos seguir e compartilhar projetos com outras pessoas e além disso possui uma série de ferramentas que ajudam no desenvolvimento de um projeto de *software* (GITHUB. FEATURES, 2015).

Algumas vantagens de usar o GitHub (Bell, Beer, 2015):

- Documentar requisitos;
- Colaborar com linhas independentes no código de um projeto;
- Revisar um trabalho em progresso;
- Ver o progresso de uma equipe em um projeto.

Conforme o projeto avançar e for identificado a necessidade de novas ferramentas, elas serão citadas em outros capítulos posteriormente.

### 5.5 Projeto do proxy reverso

Neste tópico será explicado a estrutura e funcionamento do *proxy* reverso, qual licença será

utilizada e apresentado o fluxograma do projeto.

### 5.5.1 Especificações do projeto

A seguir algumas especificações do *proxy* para que ele cumpra seu objetivo de prevenir as aplicações contra CSRF:

- Será a única *interface* entre o servidor *web* real e o cliente;
- Filtrará as requisições do cliente para evitar ataques CSRF a formulários *html*;
- Usará alguma técnica estudada durante o trabalho e baseada no uso de *tokens* para prevenir CSRF.

### 5.5.2 Licença

Este projeto foi disponibilizado sob a licença MIT. Criada pelo Massachusetts Institute of Technology é uma licença permissiva para software possibilitando seu uso de forma livre para copiar, distribuir, vender ou modificar da maneira que for necessário o *software*. (TLDRLEGAL, 2014).

### 5.5.3 Fluxograma do projeto

O fluxograma do projeto foi dividido em duas partes. O primeiro fluxograma demonstrará o comportamento do *proxy* ao receber uma requisição `GET`, que é quando o cliente está buscando os dados no servidor antes de acontecer alguma ação. O segundo fluxograma demonstrará uma requisição `POST` que é quando o usuário realizaria o envio de dados para o servidor.

Fluxograma da requisição `GET`:

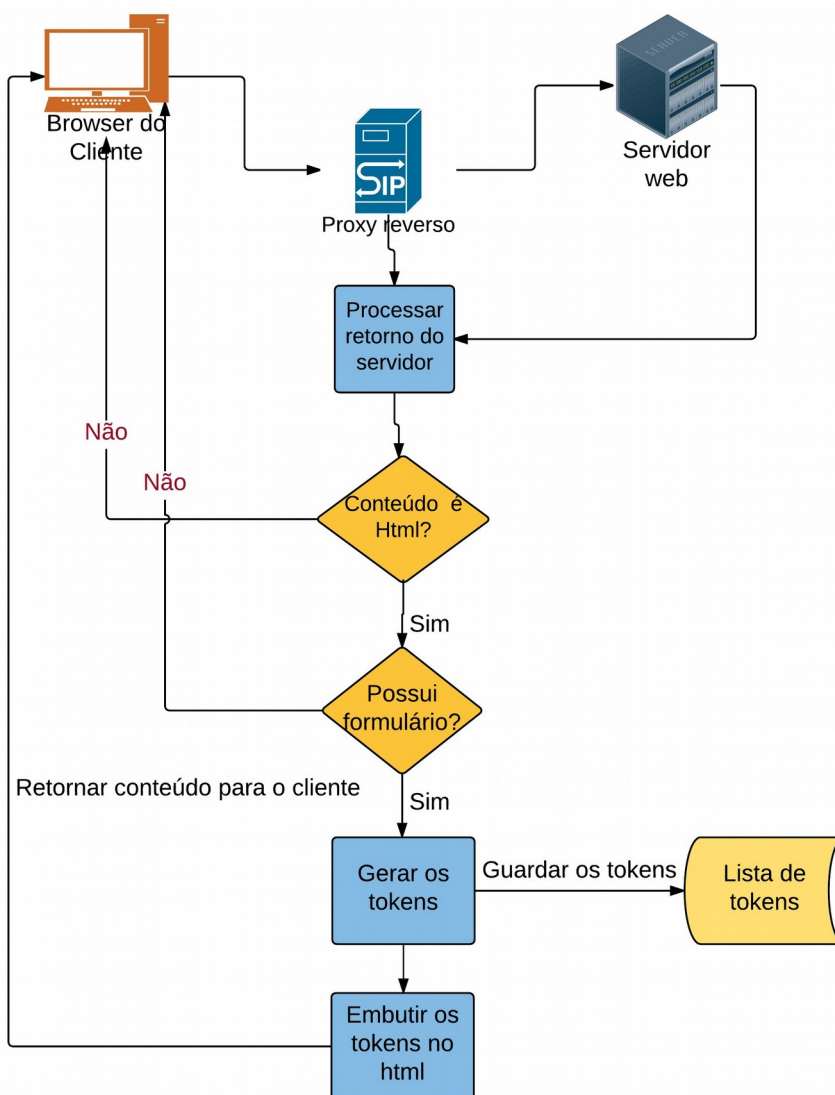


Figura 13: Fluxograma de uma requisição GET para o *proxy* reverso. Fonte: autoria própria.

A Figura 13 demonstra o fluxograma de uma requisição GET para o servidor *proxy* reverso. Nas requisições GET o *proxy* reverso fará uma solicitação para o servidor *web* real onde a aplicação encontra-se hospedada. Após receber os dados retornados do servidor *web* real será feito um filtro a fim de identificar os elementos *html* que precisam ser inseridos os *tokens*, caso seja encontrado algum elemento, o *proxy* irá gerar e embutir um *token* secreto no *html* retornado para o cliente. Este *token* será guardado na lista de *tokens* do *proxy* para uma posterior validação quando for feito o envio dos dados por parte do cliente para o servidor.

Fluxograma da requisição POST:



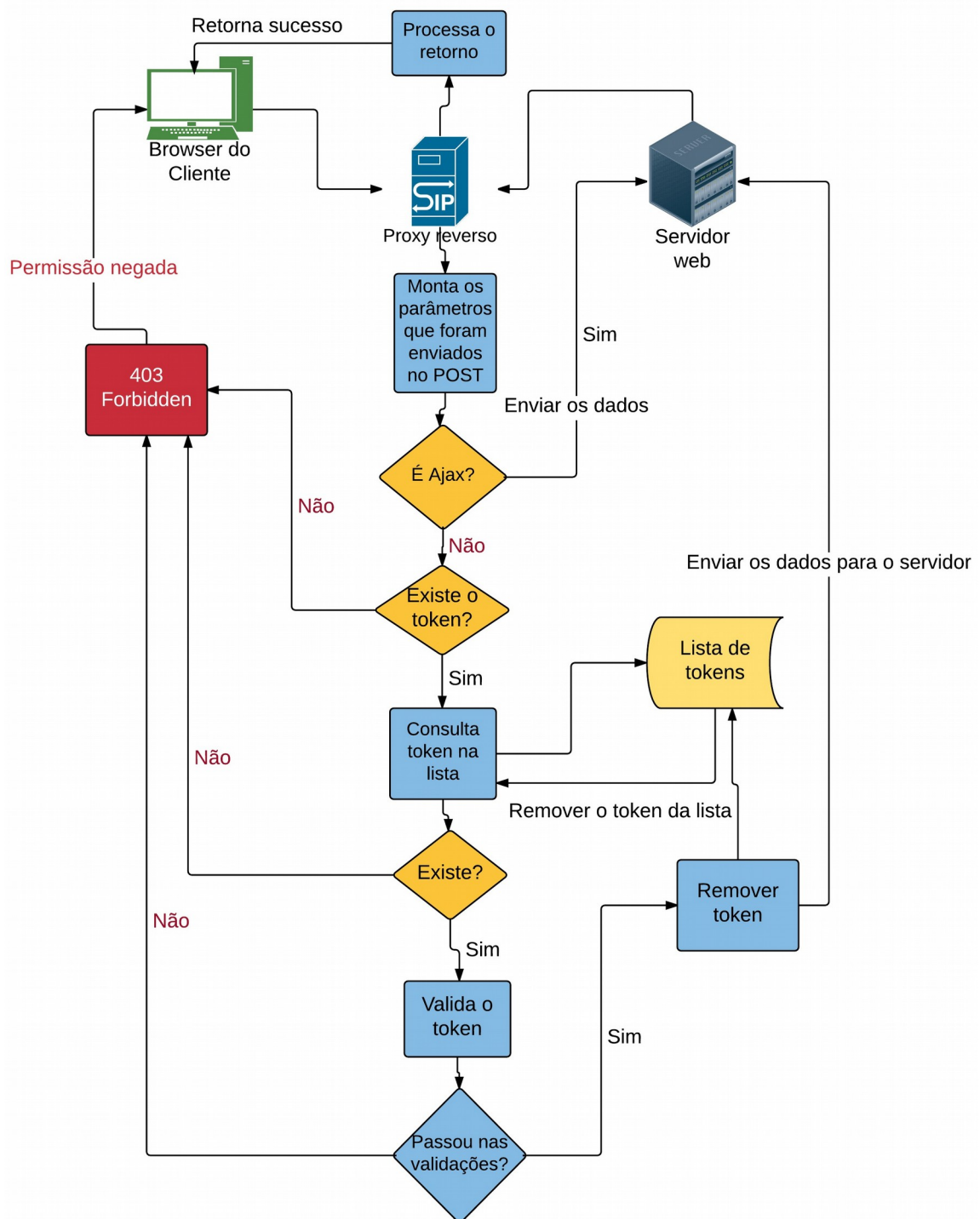


Figura 14: Requisição POST para o *proxy* reverso.

A Figura 14 demonstra como o *proxy* valida o envio dos dados para o servidor. Ao receber a requisição do cliente são preparados os parâmetros, após é feita uma validação a fim de verificar se a requisição é *ajax*, se for *ajax* a requisição é enviada para o servidor, caso não seja é verificado nos parâmetros recebidos do cliente se o mesmo possui um *token*, caso o *token* esteja presente são feitas as validações necessárias para enviar os dados para o servidor, se o *token* encontrado não passar nas validações necessárias a requisição é negada ao cliente e os dados não são enviados para o servidor.

Os detalhes de como o *proxy* se comporta em uma requisição GET ou POST serão explicados posteriormente no capítulo de desenvolvimento do projeto.

## 5.6 Considerações finais desse capítulo

Neste capítulo foi explicado o que é um *proxy* mostrando exemplos de servidores *proxy* disponíveis. Foi explicado o que é um *proxy* reverso mostrando alguns benefícios do seu uso e apresentada a justificativa de escolha de um *proxy* reverso para prevenção de ataques CSRF. Também foram apresentadas as ferramentas que o autor utilizará no desenvolvimento do projeto e, por fim, foram apresentados dois fluxogramas que mostram o comportamento esperado do *proxy* reverso em nas requisições GET e POST.

## 6 DESENVOLVIMENTO

Neste capítulo será explicado como ocorreu o desenvolvimento do *proxy* reverso.

### 6.1 Bibliotecas auxiliares no desenvolvimento

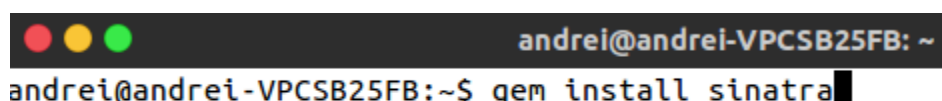
Após iniciar o desenvolvimento do projeto, foi identificado a necessidade de utilizar algumas ferramentas não citadas anteriormente, assim será explicado a seguir sobre o uso dessas ferramentas e o motivo para o qual elas foram necessárias.

#### 6.1.1 Sinatra

Sinatra é uma DSL (Domain-Specific Language) para a criação de *web services*, aplicações Ruby e aplicações *web* (Harris,A.;Haase,K. pg 1).

O Sinatra foi escolhido para criar a camada de servidor do *proxy* reverso. Ao realizar os testes necessários a ferramenta atendeu a necessidade de criar um servidor *web* para receber as requisições GET e POST vindas do cliente. Um outro motivo da escolha do Sinatra foi o fato do autor ter definido que utilizaria a *interface* Rack na camada de servidor do *proxy* e como o Sinatra utiliza esta *interface* este requisito foi atendido.

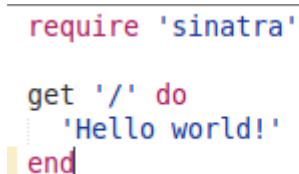
Para instalar o Sinatra deve-se usar o seguinte comando no terminal:

A screenshot of a terminal window with a dark background. The prompt is 'andrei@andrei-VPCSB25FB: ~'. The command 'gem install sinatra' has been entered and executed, as indicated by the cursor at the end of the line.

```
andrei@andrei-VPCSB25FB: ~$ gem install sinatra
```

Figura 15: Comando para instalar o Sinatra no terminal. Fonte: autoria própria.

Após instalar a *gem* do Sinatra deve-se criar um arquivo e definir as rotas que serão utilizadas. No exemplo a seguir será mostrado uma requisição GET utilizando o Sinatra:

A code snippet showing the Ruby code for a Sinatra GET route. The code is color-coded: 'require' is pink, 'sinatra' is blue, 'get' is blue, '/' is blue, 'do' is blue, 'Hello world!' is blue, and 'end' is pink.

```
require 'sinatra'

get '/' do
  'Hello world!'
end
```

Figura 16: Requisição GET utilizando Sinatra. Fonte: autoria própria.

Após colocar este código em um arquivo, deve-se somente nomeá-lo e executar no terminal

o seguinte comando “`ruby nomeDoarquivo.rb`”.



```

andrei@andrei-VPCSB25FB: ~/guadianOfRequests/codigos de testes
andrei@andrei-VPCSB25FB:~/guadianOfRequests/codigos de testes$ ruby testeSi
.rb
[2016-04-09 16:06:37] INFO  WEBrick 1.3.1
[2016-04-09 16:06:37] INFO  ruby 1.9.3 (2013-11-22) [x86_64-linux]
== Sinatra (v1.4.7) has taken the stage on 4567 for development with backup
WEBrick
[2016-04-09 16:06:37] INFO  WEBrick::HTTPServer#start: pid=15909 port=4567

```

Figura 17: Exemplo de servidor *web* Utilizando Sinatra. Fonte: autoria própria.

A Figura 17 mostra o um servidor *web* esperando uma requisição na porta 4567.

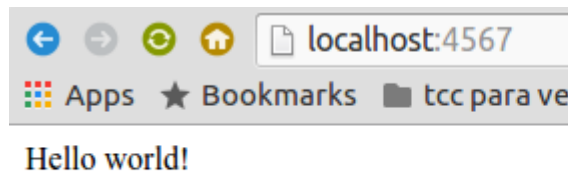


Figura 18: Requisição GET pelo servidor do Sinatra. Fonte: autoria própria.

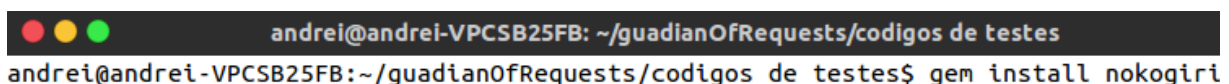
A Figura 18 mostra como acessar via navegador o servidor *web* criado utilizando o Sinatra. A frase que aparece no navegador na Figura 18 foi definida no arquivo “`testeSinatra.rb`”.

### 6.1.2 Nokogiri

A *gem* Nokogiri é uma biblioteca escrita em Ruby capaz de ler e modificar arquivos *html* e *xml*. Neste projeto ela será utilizada para pesquisar elementos no conteúdo *html* requisitado pelos usuários e inserir novos elementos no conteúdo retornado conforme a necessidade. Algumas de suas funcionalidades são (Nokogiri):

- Pesquisa por seletores CSS3;
- Pesquisa por XPath 1.0;
- Pesquisa em arquivos *xml* e *html*.

Para instalar esta *gem* deve-se executar o comando no terminal conforme a Figura 19:



```

andrei@andrei-VPCSB25FB: ~/guadianOfRequests/codigos de testes
andrei@andrei-VPCSB25FB:~/guadianOfRequests/codigos de testes$ gem install nokogiri

```

Figura 19: Instalando a *gem* Nokogiri pelo terminal. Fonte: autoria própria.

Após realizar a instalação da *gem* deve-se instalar o pacote `build-essential`<sup>23</sup> utilizando

<sup>23</sup> É um pacote que contém ferramentas para realizar a compilação de programas desenvolvidos na linguagem C.

o seguinte comando no terminal: “`sudo apt-get install build-essential patch`”.

Para demonstrar o uso da *gem* será feita uma pesquisa por formulários em um arquivo com conteúdo *html*, segue a seguir o código fonte do formulário utilizado no exemplo:

```
<div class="col-6">
  <form class="form-signin" action="logado.php" method="post">
    <h2 class="form-signin-heading">Sign in</h2>
    <label for="inputEmail" class="sr-only">Email</label>
    <input type="email" id="inputEmail" class="form-control" name="email" placeholder="Email" required autofocus>
    <label for="inputPassword" class="sr-only">Senha</label>
    <input type="password" id="inputPassword" class="form-control" name="senha" placeholder="Senha" required>
    <div class="checkbox">
      <label>
        <input type="checkbox" value="remember-me" name="lembrar">Lembrar-me
      </label>
    </div>
    <button class="btn btn-lg btn-primary btn-block" type="submit">Logar</button>
  </form>
</div>
```

Figura 20: Código fonte de um arquivo *html*. Fonte: autoria própria.

A Figura 20 mostra o conteúdo de um arquivo *html* contendo um formulário. A Figura 21 mostra como realizar uma pesquisa por esse formulário utilizando a *gem* Nokogiri:

```
1 require 'nokogiri'
2
3
4 doc = Nokogiri::HTML(File.open("/var/www/html/form.html"))
5
6 doc.search("form").each do |form|
7   puts form
8 end
9
10
```

---

```
<form class="form-signin" action="logado.php" method="post">
  <h2 class="form-signin-heading">Sign in</h2>
  <label for="inputEmail" class="sr-only">Email</label>
  <input type="email" id="inputEmail" class="form-control" name="email" placeholder="Email" required autofocus>
  <label for="inputPassword" class="sr-only">Senha</label>
  <input type="password" id="inputPassword" class="form-control" name="senha" placeholder="Senha" required>
  <div class="checkbox">
    <label>
      <input type="checkbox" value="remember-me" name="lembrar">Lembrar-me
    </label>
  </div>
  <button class="btn btn-lg btn-primary btn-block" type="submit">Logar</button>
</form>
```

[Finished in 0.1s]

Figura 21: Pesquisa por formulário em um arquivo *html* utilizando a *gem* Nokogiri.

Fonte: autoria própria.

Como pode ser demonstrado na Figura 21, foi realizado um teste utilizando o editor de texto Sublime Text. No console do editor é possível ver que foi encontrado o formulário que estava no arquivo *html*.

Com esta *gem* também é possível inserir elementos em conteúdos *html*. A Figura 22 demonstra como inserir um elemento:

```
require 'nokogiri'

doc = Nokogiri::HTML(File.open("/var/www/html/form.html"))

doc.search("form").each do |form|
  input = Nokogiri::HTML::Document.new.create_element
    |"input",
      :type => "hidden", :name => "token",
      'value' => 'aqui vai ser o valor do token'
  form.add_child input
  puts form
end
```

---

```
<form action="index.html" method="post">
  <span class="label label-default">form 3dd</span>
  <input type="text">
  <input type="submit">submeter
  <input type="hidden" name="token" value="aqui vai ser o valor do token">
</form>
```

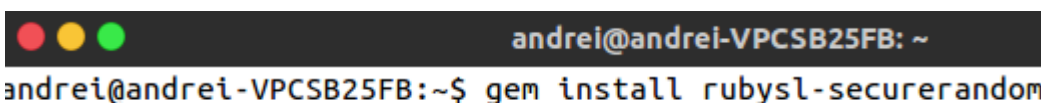
Figura 22: Inserindo elemento no *html* utilizando a *gem* Nokogiri. Fonte: autoria própria.

A Figura 22 mostra um teste via console do editor de texto Sublime Text no qual foi inserido um elemento *input* em um formulário.

### 6.1.3 SecureRandom

SecureRandom é um módulo da linguagem Ruby que possui uma interface para gerar números aleatórios. Neste projeto será utilizado o método `uuid` disponível pelo módulo para gerar o valor aleatório dos *tokens* gerados pelo *proxy* reverso. Este método retorna um valor padronizado chamado UUID<sup>24</sup>.

Para instalar a *gem* SecureRandom deve-se executar no terminal o comando demonstrado na Figura 23.



```
andrei@andrei-VPCSB25FB: ~
andrei@andrei-VPCSB25FB:~$ gem install rubysl-securerandom
```

Figura 23: Comando para instalar a *gem* SecureRandom. Fonte: autoria própria.

Para utilizar a *gem* basta chamar os métodos disponíveis pelo módulo SecureRandom. Na Figura 24 pode ser observado como utilizar método `uuid`:

<sup>24</sup> O UUID é um valor de 128 bits único em um espaço de tempo. Este padrão é utilizado nos sistemas da plataforma Microsoft (<http://www.rfc-base.org/rfc-4122.html>).

```

1  require "securerandom"
2
3  puts SecureRandom.uuid

```

```
"4f5f3613-4308-4649-9699-c8cc9450fcb3"
```

Figura 24: Utilizando o método `uuid` do módulo `SecureRandom`. Fonte: autoria própria.

A Figura 24 mostra um UUID no console do editor de texto Sublime Text gerado com a utilização do módulo `SecureRandom`.

#### 6.1.4 Rufus-Scheduler

Rufus-Scheduler é uma *gem* utilizada para fazer agendamentos de tarefas. Neste projeto esta *gem* será utilizada para realizar um agendamento da lista de *tokens* do *proxy* reverso para fazer a exclusão dos *tokens* expirados.

Para definir agendamentos é possível utilizar alguns métodos disponíveis na API da *gem*, como por exemplo (Rufus-Scheduler):

- Método `in`: recebe um parâmetro que vai representar um tempo. Após passar este tempo será executado a tarefa;
- Método `every`: recebe um parâmetro que vai representar um tempo. Diferentemente do método `in` este método usa o tempo recebido no parâmetro como intervalo para rodar a tarefa. Este método fica executando uma tarefa até ser interrompido;
- Método `cron`: executa uma tarefa diariamente conforme o tempo passado como parâmetro.

```

1  require 'rufus-scheduler'
2
3  scheduler = Rufus::Scheduler.new
4
5  scheduler.every "1s" do
6    p Time.new.sec
7  end
8
9  scheduler.join

```

Figura 25: Utilizando o método `every` para realizar uma tarefa. Fonte: autoria própria.

Para este projeto foi escolhido o método `every` disponível na API da *gem* para executar a tarefa de limpar os *tokens* expirados da lista de *tokens* do *proxy* reverso. Na Figura 25 pode ser observado a utilização deste método. Ele recebe o parâmetro “1s” que indica que ele deve executar um código programado a cada segundo de tempo passado.

```
andrei@andrei-VPCSB25FB:~/guardianOfRequests$ ruby scheduler.rb
"segundo atual 19"
"segundo atual 20"
"segundo atual 21"
```

Figura 26: Execução do método `every` no terminal. Fonte: autoria própria.

Na Figura 26 é exibido uma mensagem no terminal que corresponde a execução programada utilizando o método `every`. Como foi definido no parâmetro, a execução ocorre a cada segundo de tempo passado, exibindo uma mensagem no terminal.

## 6.2 Classes do projeto

A lista a seguir explica a responsabilidade de cada classe do *proxy* reverso bem como detalha os métodos presentes nas mesmas:

- Classe `GuardianOfRequests` é a classe responsável por configurar e iniciar o servidor do *proxy* reverso. Possui os seguintes métodos: `initialize`, `configuraOptionsProxyBase`, `options`, `start`. O método `initialize` é visível (*public*) e é chamado na instância do objeto da classe. Este método recebe um parâmetro chamado `options` do tipo `Hash`. O método `initialize` chama o método `configuraOptionsProxyBase` para configurar os valores passados no parâmetro `options`. O método `options` é visível (*public*) e retorna um `Hash`. O método `configuraOptionsProxyBase` não é visível (*private*) e configura na classe `ProxyBase` os valores do parâmetro `options` passados na instância do objeto da classe `GuardianOfRequests`. O método `start` é visível (*public*) e é responsável por iniciar o servidor do *proxy* reverso;
- Classe `ProxyBase` é a classe responsável pela lógica principal do *proxy* reverso. Ela estende a classe `Base` do Sinatra e implementa a lógica para realizar as requisições GET e POST para o servidor real. Possui os seguintes métodos, sendo todos eles de visibilidade privada (*private*): `html`, `cliente`, `listaDeTokens`, `dataHoraAtual`, `montaParametrosParaPost`, `limpaArquivosTemporarios`, `podePostar`, `arquivosEnviados`. O método `html` retorna um objeto da classe `Html`. O método



`cliente` retorna um objeto da classe `Cliente`. O método `listaDeTokens` retorna um objeto da classe `ListaDeTokens`. O método `dataHoraAtual` retorna um objeto da classe `DataHora`. O método `podePostar` realiza as lógicas para verificar se os dados podem ser enviados para o servidor real e retorna um `Boolean`. O método `monstaParametrosParaPost` prepara corretamente os parâmetros para serem enviados ao servidor real, ele retorna um `Hash` com os parâmetros a serem enviados no `POST`. O método `limpaArquivosTemporarios` limpa uma pasta contendo os arquivos enviados para o servidor real. O método `arquivosEnviados` retorna um `Array` com o nome de todos os arquivos enviados para o servidor real;

- Classe `DataHora` é uma classe utilizada no auxílio da data e hora. Esta classe possui os seguintes métodos: `dataHoraAtual`, `emMinutos`. Os dois métodos são visíveis (*public*). O método `dataHoraAtual` retorna um objeto da classe `Time`. O método `emMinutos` calcula o horário atual e retorna convertido para minutos;
- Classe `Html` é responsável por auxiliar a manipulação do conteúdo *html* enviado nas requisições para o *proxy* reverso. Esta classe possui os seguintes métodos: `ehHtml`, `possuiElemento`, `criaInputHidden`, `insereInputNosFormularios`, `parsing`, `numeroDeFormularios`, `geraHtmlRetorno`. O método `ehHtml` é visível (*public*) e recebe um parâmetro chamado `contentType` do tipo `String`. Este método é responsável por verificar se o conteúdo é do tipo *html*. O retorno é do tipo `Boolean`. O método `possuiElemento` recebe os parâmetros chamados `html` e `elemento`, os dois parâmetros são do tipo `String`. Este método verifica no conteúdo *html* passado no parâmetro `html`, se o mesmo possui o elemento que foi passado no parâmetro `elemento`, o retorno do método é do tipo `Boolean`. O método `criaInputHidden` não é visível (*private*) e recebe dois parâmetros chamados `valorAtributo` e `nomeAtributo`, os dois são do tipo `String`. Este método é responsável por criar um elemento *input* do tipo *hidden* para ser inserido em um formulário *html*. O retorno deste método é um objeto da classe `Element`. O método `insereInputNosFormularios` é visível (*public*) e recebe dois parâmetros chamados `conteudoHtml` e `tokens`, os dois são do tipo `String`. Este método é responsável por percorrer um conteúdo *html* e inserir em cada formulário encontrado um *input hidden*. O retorno deste método é um objeto da classe `Document`. O método `parsing` não é visível (*private*) e recebe um parâmetro chamado `html` do tipo `String`. Este método é responsável por fazer uma conversão do

conteúdo *html* para um objeto da classe `Document`. O retorno deste método é um objeto da classe `Document`. O método `numeroDeFormularios` é visível (*public*) e recebe um parâmetro chamado `conteudoHtml` do tipo `String`. Este método conta quantos formulários foram encontrados no conteúdo *html*. O retorno é um número inteiro. O método `geraHtmlRetorno` é visível (*public*) e recebe os parâmetro chamados `documento` e `options`. Este método é responsável por converter um objeto da classe `Document` em um objeto da classe `String`. O retorno é um objeto da classe `String`;

- Classe `Cliente` é responsável por fazer requisições `POST` e `GET` para um servidor *web*; Possui os seguintes métodos: `initialize`, `consultarUrl`, `postarUrl`;
- Classe `Token` é utilizada pela classe `ListaDeTokens` para gerar os *tokens* inseridos nos formulários *html*. Possui os seguintes métodos: `initialize`, `time`, `time=`, `expirado`, `expirado=`, `uuid`, `uuid=`;
- Classe `ListaDeTokens` é responsável por gerar e armazenar os *tokens* inseridos nos formulários. Possui os seguintes métodos: `initialize`, `tempoVerificacaoTokens`, `tempoVerificacaoTokens=`, `excluiTokensExpirados`, `tokens`, `tokens=`, `tempoMaxToken`, `tempoMaxToken=`, `gerarTokens`, `pegarToken`, `removerToken`.

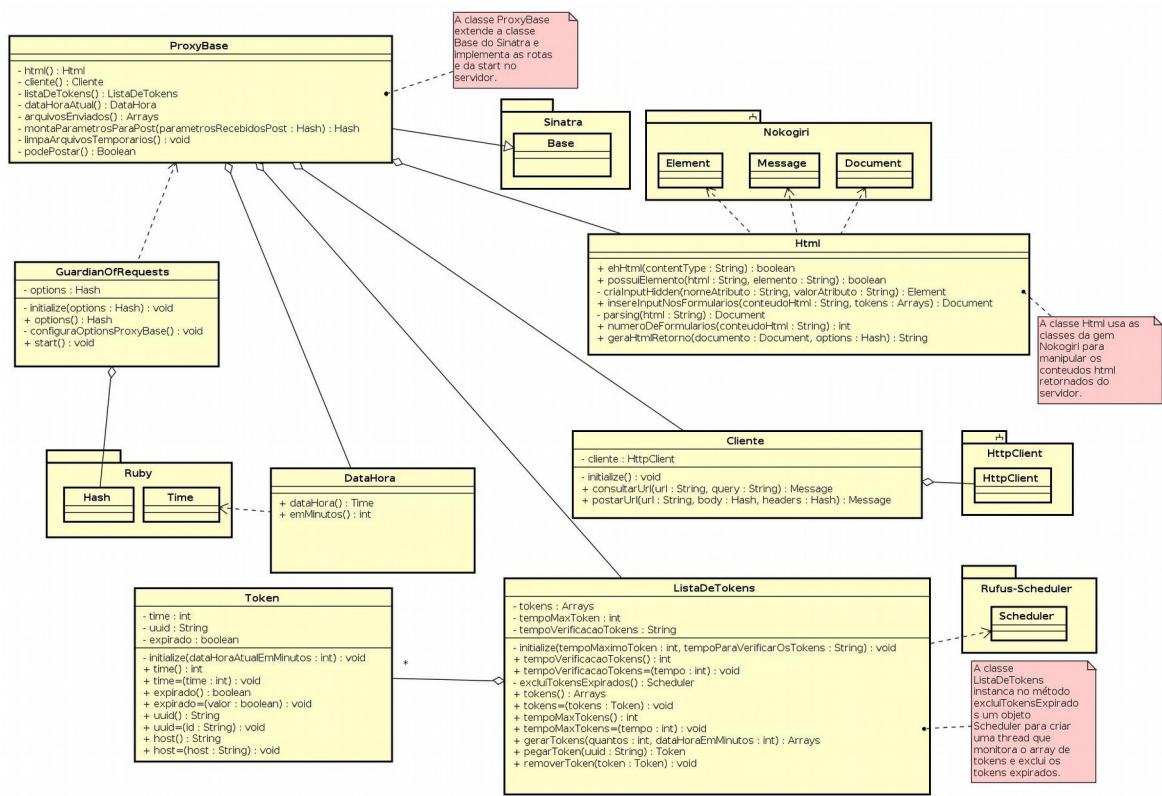
### 6.2.1 Classes auxiliares

As seguintes classes apresentadas foram classes nativas da linguagem Ruby ou desenvolvidas por terceiros:

- Classe `Time` é nativa da linguagem Ruby, responsável pela abstração de datas e tempos. Neste projeto ela é utilizada pela classe `DataHora`;
- As classes `HTML`, `Document`, `Message` e `Element` são classes da *gem* `Nokogiri`. São utilizadas pela classe `Html` para manipular o conteúdo *html*;
- Classe `HTTPClient` faz parte da *gem* `HTTPClient`. É utilizada pela classe `Cliente`;
- Classe `SecureRandom` é responsável por gerar a identificação única de cada *token*. Faz parte da *gem* `SecureRandom`. Esta classe é utilizada na classe `Token`;
- Classe `Scheduler` faz parte da *gem* `Rufus-Scheduler`. É utilizada pela classe `ListaDeTokens`.

## 6.3 Diagrama de classes

A figura a seguir mostra o diagrama das classes descritas no tópico anterior:



powered by Astah

Figura 27: Diagrama de classes do *proxy* reverso. Fonte: autoria própria

A Figura 27 é a representação do diagrama de classes do *proxy* reverso. É possível verificar no diagrama o relacionamento das classes do projeto. A descrição e os métodos das mesmas já foram explicados nos tópicos 6.2 e 6.2.1.

## 7 ESTUDO DE CASO

Foi montado um cenário de teste para simular um sistema acadêmico para atualização de notas de uma turma de alunos. O cenário foi montado com base no que foi estudado durante o projeto para mostrar o *proxy* reverso atuando na prevenção de alguma aplicação vulnerável a ataques CSRF.

Neste projeto, em específico, o *proxy* reverso vai prevenir ataques CSRF somente a formulários *html*. Requisições via *ajax* e outros elementos *html* que contenham atributos *src* e *href* não serão abordados neste projeto e ficarão como sugestão de melhorias futuras.

### 7.1 Sistema acadêmico

O cenário de testes montado visa mostrar a postagem de um formulário para atualização de notas de uma turma de alunos simulando um sistema acadêmico. A simulação consiste na postagem de um formulário por um usuário autenticado no sistema e com permissão de atualização das notas dos alunos. O sistema foi construído utilizando a linguagem de programação PHP para gerenciar a parte de autenticação, atualização de notas e controlar a sessão do usuário. Para a parte de *layout* do sistema foi utilizado o Bootstrap (um *framework* para o desenvolvimento de aplicações com *interfaces web* responsivas) e para a persistência dos dados foi utilizado o banco de dados MySQL.

#### 7.1.1 Login

Como característica de um ataque CSRF o usuário precisa estar autenticado na aplicação vulnerável, desse modo o primeiro acesso ao sistema é uma tela de *login*:

A interface de login do sistema acadêmico. No topo, há um logotipo com o texto "PORTAL ACADÊMICO" ao lado de um ícone de livro aberto. Abaixo, há dois campos de entrada: "Usuário" e "Senha", ambos com bordas arredondadas e um ícone de lupa à direita. Abaixo dos campos, há um botão verde com o texto "Entrar" em branco.

Figura 28: Tela de *login* do sistema acadêmico. Fonte: autoria própria.

A Figura 28 mostra o *login* ao sistema acadêmico utilizado na simulação. Nesta tela serão preenchidos o usuário e a senha para realizar a autenticação.

### 7.1.2 Atualizando notas

Após realizar o *login* no sistema, o usuário será redirecionado para a página inicial onde deverá acessar um menu para editar as notas dos alunos.

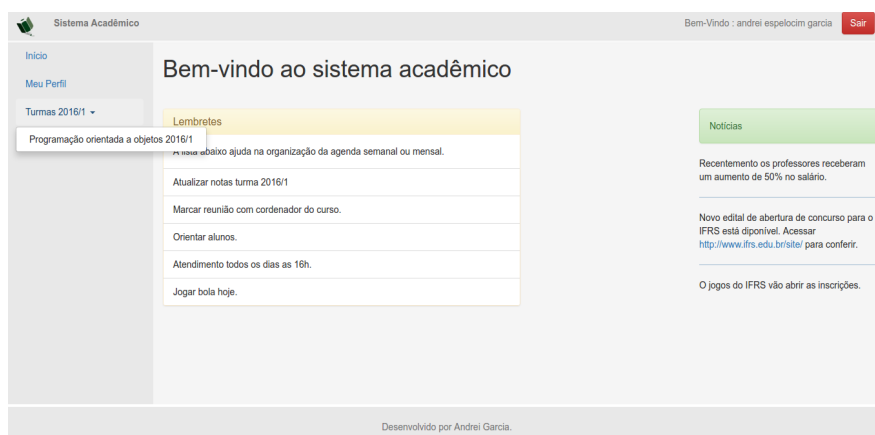
A interface principal do sistema acadêmico. No topo, há uma barra de navegação com o logotipo "Sistema Acadêmico" à esquerda, o texto "Bem-Vindo : andrei.espelocim.garcia" no centro e um botão "Sair" à direita. Abaixo, há uma barra lateral esquerda com links: "Início", "Meu Perfil", "Turmas 2016/1" (com uma seta para baixo) e "Lembretes" (destacado). O conteúdo principal mostra o texto "Bem-vindo ao sistema acadêmico" e uma lista de lembretes: "Programação orientada a objetos 2016/1", "Atualizar notas turma 2016/1", "Marcar reunião com coordenador do curso", "Orientar alunos", "Atendimento todos os dias as 16h", e "Jogar bola hoje". À direita, há uma seção "Notícias" com o texto: "Recentemente os professores receberam um aumento de 50% no salário.", "Novo edital de abertura de concurso para o IFRS está disponível. Acessar [http://www.ifrs.edu.br/site](\"http://www.ifrs.edu.br/site\") para conferir.", e "O jogos do IFRS vão abrir as inscrições.". No rodapé, há o texto "Desenvolvido por Andrei Garcia."

Figura 29: Tela inicial do sistema. Fonte: autoria própria.

A Figura 29 mostra como acessar pelo *menu* a página para editar as notas de uma determinada turma de alunos. Após acessar no *menu* a opção “Programação orientada a objetos 2016/1” conforme mostra na Figura 29, o usuário será redirecionado para uma página onde é possível submeter um formulário para editar as notas.




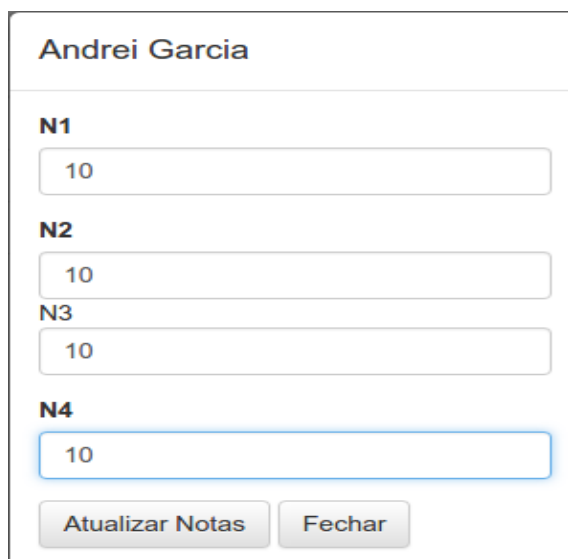
Aluno	N1	N2	N3	N4	Editar Notas
Andrei Garcia	11	6,8	1	1	
Leandro Marques	5	5	2	5	
Rodrigo Braga	5	5	2	5	
Anderson Souza	2	2	2	2	

Figura 30: Página de listagem das notas dos alunos. Fonte: autoria própria.

A Figura 30 mostra a página de listagem das notas dos alunos. Para atualizar a nota de um aluno é preciso acessar o *link* presente na figura do lápis na coluna editar notas, onde será aberto um formulário para edição e envio das notas.



**Andrei Garcia**

**N1**

**N2**

**N3**

**N4**

Figura 31: Formulário para atualizar as notas de um aluno. Fonte: autoria própria.

A Figura 31 mostra o formulário para edição e atualização das notas do aluno selecionado. Ao submeter este formulário as notas do aluno selecionado serão atualizadas com os valores que foram inseridos nos campos N1, N2, N3 e N4 do formulário.

```

<div class="modal-body">
  <form method="post" action="turmas/atualizaNotas.php">
    <input type="hidden" name="id" value="1">
    <div class="form-group">
      <label for="n1">N1</label>
      <input type="text" class="form-control" id="n1" name="n1" value="10">
    </div>
    <div class="form-group">
      <label for="n2">N2</label>
      <input type="text" class="form-control" id="n2" name="n2" value="10">
    </div>
    <div class="form-group">
      <label for="n3">N3</label>
      <input type="text" class="form-control" id="n3" name="n3" value="10">
    </div>
    <div class="form-group">
      <label for="n4">N4</label>
      <input type="text" class="form-control" id="n4" name="n4" value="10">
    </div>
    <button type="submit" class="btn btn-default">Atualizar Notas</button>
    <button type="button" class="btn btn-default" data-dismiss="modal">Fechar</button>
  </form>
</div>

```

Figura 32: Código fonte do formulário da Figura 31. Fonte: autoria própria.

A Figura 32 mostra o código fonte do formulário submetido para atualizar as notas do aluno selecionado. Como pode ser observado o formulário contém um campo escondido o com o nome “id”, que identifica o aluno que terá suas notas atualizadas no sistema.

```

<?php
session_start();
$dsn = 'mysql:dbname=academico;host=127.0.0.1';
$user = $_SESSION['nomeUsuario'];
$password = $_SESSION['senha'];

try {
    $con = new PDO($dsn,$user,$password);
    $sql = "UPDATE notas SET n1=?,n2=?,n3=?,n4=? WHERE id_aluno=?";
    $res = $con->prepare($sql);

    $res->bindParam(1,$_POST['n1']);
    $res->bindParam(2,$_POST['n2']);
    $res->bindParam(3,$_POST['n3']);
    $res->bindParam(4,$_POST['n4']);
    $res->bindParam(5,$_POST['id']);
    $res->execute();
    $_SESSION['buscarTurma'] = true;
    header("Location: ../index.php");
} catch (PDOException $e) {
    header("Location: ../login/login.php");
}

?>

```

Figura 33: Código fonte mostrando como é feita a atualização de nota de um aluno no servidor. Fonte: autoria própria.

A Figura 33 exibe a lógica para atualizar as notas de um aluno quando o formulário é submetido para o servidor pelo usuário. São utilizados os dados de sessão para conectar e gravar no banco de dados, os dados recebidos do formulário submetido pelo usuário.

A Figura 34 mostra a listagem de notas do sistema após submeter o formulário da Figura 31.


Aluno	N1	N2	N3	N4	Editar Notas
Andrei Garcia	10	10	10	10	

Figura 34: Tela de listagem de notas do sistema mostrando as notas atualizadas conforme o formulário da Figura 31. Fonte: autoria própria.

Pode ser observado na Figura 34 que após submeter o formulário exibido na Figura 31 o aluno teve todas as suas notas atualizadas conforme os dados informados nos campos N1, N2, N3 e N4.

## 7.2 O problema

Conforme foi demonstrado na Figura 33 o sistema não possui nenhum tipo de validação para atualizar as notas de um aluno no banco de dados, bastando somente que um usuário esteja autenticado para que seja feita uma conexão com o banco de dados e após seja realizado a atualização das notas com as informações enviadas. Para simular o problema foi construído um formulário com os dados de um aluno cadastrado no banco de dados. O formulário utilizado como exemplo para forjar um POST para o sistema é só um exemplo simples. Geralmente as vítimas de ataques CSRF são atacadas com técnicas de Engenharia Social, sendo elaborados ataques bastante sofisticados para induzir o usuário a realizar uma ação indevida para a aplicação vulnerável. O formulário construído para realizar a requisição forjada tem as mesmas configurações do formulário enviado na página verdadeira, assim garantindo que a submissão dos dados vai resultar em sucesso.

### Post Fake

N1

N2

N3

N4

Figura 35: Formulário para realizar um POST forjado para o sistema. Fonte: autoria própria.



A submissão do formulário da Figura 35 deve ser feita de uma outra aplicação quando o usuário estiver autenticado no sistema, desta forma caracterizando um ataque CSRF. A seguir segue o código fonte do formulário forjado:

```
<h1>Post Fake</h1>
<form method="post" action="http://localhost/academico/turmas/atualizaNotas.php">
  <input type="hidden" name='id' value="1">
  <div class="form-group">
    <label for="n1">N1</label>
    <input type="text" class="form-control" id="n1" name="n1" value="0">
  </div>
  <div class="form-group">
    <label for="n2">N2</label>
    <input type="text" class="form-control" id="n2" name="n2" value="0">
  </div>
  <div class="form-group">
    <label for="n3">N3</label>
    <input type="text" class="form-control" id="n3" name="n3" value="0">
  </div>
  <div class="form-group">
    <label for="n4">N4</label>
    <input type="text" class="form-control" id="n4" name="n4" value="0">
  </div>
  <button type="submit" class="btn btn-default">Enviar</button>
</form>
```

Figura 36: Código fonte do formulário da Figura 35. Fonte: autoria própria.

Após submeter os dados do formulário da Figura 35, a nota do aluno ficou conforme os dados informados nos campos do formulário.


Aluno	N1	N2	N3	N4	Editar Notas
Andrei Garcia	0	0	0	0	

Figura 37: Resultado do envio dos dados do formulário da Figura 35. Fonte: autoria própria.

A Figura 37 mostra que os dados do aluno foram atualizados através após ser submetido o formulário da Figura 35, zerando todas as suas notas. Se aproveitando dessa vulnerabilidade e se o sistema fosse real, vários alunos poderiam ser prejudicados caso alguém descobrisse essa vulnerabilidade e resolvesse aplicar ataques para zerar as notas de qualquer aluno. Outro possível problema seria um aluno conseguir atualizar as próprias notas para se beneficiar. Desta forma pode ser observado que o sistema é vulnerável a ataques CSRF podendo vir a comprometer as informações dos alunos caso este fosse um sistema real.

### 7.3 Prevenindo ataque CSRF a formulário com um *proxy* reverso

Neste tópico será explicado como configurar e iniciar o *proxy* reverso desenvolvido durante o projeto e demonstrado como o mesmo evita ataques CSRF a formulários.

#### 7.3.1 Criando o arquivo para iniciar o *proxy*

Antes de iniciar o *proxy* reverso deve-se criar um arquivo. O nome do arquivo pode ser de

qualquer preferência, porém a extensão deve ser “rb”, que é a extensão de arquivos executados pela linguagem Ruby, segue um exemplo de nome: “testandoProxyReverso.rb”. Dentro deste arquivo deve-se utilizar o comando `require_relative` seguido do nome do arquivo que contém a classe `GuardianOfRequests` do *proxy* reverso.

```
|require_relative "guardianOfRequests.rb"
```

Figura 38: Referência para o arquivo da classe `GuardianOfRequests` do *proxy* reverso. Fonte: autoria própria.

A Figura 38 mostra a referência para o arquivo que contém a classe `GuardianOfRequests`. Neste exemplo os arquivos-fonte do *proxy* estão no mesmo diretório do arquivo de testes, caso estivessem em outro diretório o caminho deve ser ajustado relacionando o diretório correto dos arquivos.

### 7.3.2 Configurando o *proxy* reverso

Após criar o arquivo de testes, podem ser definidas algumas configurações para o *proxy* reverso, segue as opções disponíveis:

- Tempo máximo de um *token*: é definido por um número inteiro;
- Domínio do servidor *proxy*: é o domínio onde o servidor responderá as requisições GET e POST, exemplo: “localhost”;
- Porta do servidor do *proxy*: é um número inteiro;
- Servidor do *proxy* reverso: os servidores disponíveis pelo Sinatra. No *proxy* reverso será utilizado o *webrick*;
- tempo de verificação dos *tokens* na lista de *tokens* do *proxy*: são os tempos aceitos pelo método `every` da *gem* *Rufus-Scheduler*, mais informações em: (<https://github.com/jmettraux/rufus-scheduler>).

A Figura 39 mostra como definir as configurações citadas anteriormente para o *proxy* reverso:

```
require_relative "guardianOfRequests.rb"

options = Hash.new
options[:tempoMaxToken] = 1
options[:bind] = 'localhost'
options[:port] = 4999
options[:server] = :webrick
options[:tempoParaVerificarOsTokens] = "10s"

proxy = GuardianOfRequests.new options
```

Figura 39: Definindo as configurações do *proxy* reverso. Fonte: autoria própria.

Para definir as configurações do *proxy* reverso deve-se criar uma variável do tipo Hash, definir as configurações necessárias e passar a variável criada como parâmetro na instância do objeto da classe `GuardianOfRequests`.

### 7.3.3 Iniciando o *proxy* reverso

Após configurar o *proxy* deve-se chamar o método `start` da classe `GuardianOfRequests` e utilizar o comando “`ruby testandoProxyReverso.rb`” no terminal para iniciar o servidor do *proxy* reverso. A figura 40 mostra a chamada ao método `start` pra iniciar o servidor do *proxy* reverso.

```
require_relative "guardianOfRequests.rb"

options = Hash.new
options[:tempoMaxToken] = 1
options[:bind] = 'localhost'
options[:port] = 4999
options[:server] = :webrick
options[:tempoParaVerificarOsTokens] = "10s"

proxy = GuardianOfRequests.new options
proxy.start
```

Figura 40: Iniciando o *proxy* reverso. Fonte: autoria própria.

Após realizar o comando “`ruby testandoProxyReverso.rb`” no terminal aparecerá as informações referentes ao servidor do *proxy* reverso.

```

andrei@andrei-VPCSB25FB:~/guadianOfRequests$ ruby testandoProxyReverso.rb
[2016-04-24 20:43:03] INFO WEBrick 1.3.1
[2016-04-24 20:43:03] INFO ruby 1.9.3 (2013-11-22) [x86_64-linux]
== Sinatra (v1.4.7) has taken the stage on 4999 for development with backup from WEBrick
[2016-04-24 20:43:03] INFO WEBrick::HTTPServer#start: pid=5297 port=4999

```

Figura 41: Informações no terminal sobre o servidor do *proxy* reverso. Fonte: autoria própria.

A Figura 41 mostra no terminal, o servidor do *proxy* reverso esperando as requisições GET e POST na porta 4999. Estas configurações foram definidas antes de iniciar o *proxy* no arquivo que foi criado. Foi definido como domínio para o *proxy* receber as requisições o “localhost”, desta forma requisições feitas para o domínio “localhost:4999” passarão pelo *proxy* reverso antes de chegar no servidor *web* real.

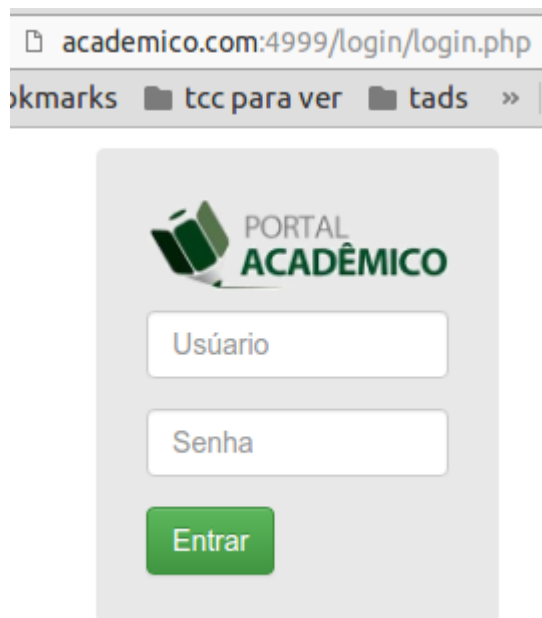


Figura 42: Fazendo *login* no sistema acadêmico através do *proxy* reverso. Fonte: autoria própria.

Como pode ser observado na Figura 42, foi feita uma requisição no navegador para a tela de autenticação do sistema acadêmico passando pelo *proxy* reverso. Via terminal é possível verificar o GET que foi feito pelo servidor do *proxy* reverso para a página de autenticação do sistema, segue o essas informações na Figura 43:

```

tempo 0.000 1902
localhost - - [26/Apr/2016:22:42:33 BRT] "GET /academico/ HTTP/1.1" 302 537
- -> /academico/
localhost - - [26/Apr/2016:22:42:33 BRT] "GET /academico/login/login.php HTTP/1.1" 200 1928
- -> /academico/login/login.php
localhost - - [26/Apr/2016:22:42:33 BRT] "GET /academico/css/stylo.css HTTP/1.1" 200 284
http://localhost:4999/academico/login/login.php -> /academico/css/stylo.css
localhost - - [26/Apr/2016:22:42:33 BRT] "GET /academico/imagens/loginLogo.png HTTP/1.1" 200 32784
http://localhost:4999/academico/login/login.php -> /academico/imagens/login.png
localhost - - [26/Apr/2016:22:42:34 BRT] "GET /favicon.ico HTTP/1.1" 404 28
http://localhost:4999/academico/login/login.php -> /favicon.ico

```

Figura 43: Requisições GET para o servidor do proxy reverso.

A Figura 43 mostra as informações das requisições feitas pelo servidor do *proxy* reverso no terminal do sistema. São realizadas várias requisições GET a fim de trazer imagens, arquivos *css* e a página principal de autenticação do sistema acadêmico.

### 7.3.4 Inserindo um *token* em um formulário com o *proxy* reverso

Para proteger as aplicações *web* contra ataques CSRF o *proxy* reverso utilizará como técnica um padrão chamado Sincronizador de Tokens (Synchronizer Token). Este padrão já foi explicado mais detalhadamente no capítulo 4 do trabalho onde foi detalhado o estudo sobre CSRF. Esta técnica consiste em inserir *tokens* ocultos em um formulário *html* e validar no POST.

O processo de inserção de um *token* pelo *proxy* reverso em formulários consiste em: identificar se possui formulários no conteúdo *html* requisitado, gerar os *tokens*, embutir esses *tokens* nos formulários e retornar o conteúdo *html* para o cliente.

```

def possuiElemento(html, elemento)
  if(Nokogiri::HTML(html).at(elemento).nil?)
    return false
  end
  true
end

```

Figura 44: Método `possuiElemento` da classe `Html`. Fonte: autoria própria.

A Figura 44 mostra o método `possuiElemento` da classe `Html`. Nas requisições GET este método é executado para verificar se o conteúdo *html* retornado do servidor *web* real contém algum formulário. Caso o conteúdo contenha algum formulário são gerados os *tokens*.

```

def gerarTokens(quantos,dataHoraEmMinutos,host)
  Array.new(quantos){|indice,elemento|
    self.tokens = (token = Token.new dataHoraEmMinutos,host)
    elemento = token.uuid
  }
end

```

Figura 45: Método `gerarTokens` da classe `ListaDeTokens`.

Fonte: autoria própria.

A Figura 45 mostra como são gerados os *tokens* para os formulários presentes no conteúdo *html* de uma requisição GET. Cada *token* recebe um tempo que será a sua validade para uso, um *host* (domínio de onde veio a requisição) e um UUID. Essas informações são importantes, pois serão validadas na requisição POST para permitir que os dados sejam enviados para o servidor *web* real. Após gerar os *tokens*, é embutido um campo *input* do tipo *hidden* nos formulários *html* retornados para o cliente. O valor do campo *input* é um UUID de um *token* válido que foi gerado no *proxy*.

```

def insereInputNosFormularios(conteudoHtml,tokens)
  documentoHTML = parsing(conteudoHtml)
  posicaoToken = 0
  documentoHTML.search("form").each do |form|
    form.add_child criaInputHidden 'value', tokens[posicaoToken]
    posicaoToken = posicaoToken + 1
  end
  documentoHTML
end

```

Figura 46: Método `insereInputsNosFormularios` da classe `Html`.

Fonte: autoria própria.

A Figura 46 mostra o código do método `insereInputNosFomulario`. Este método é executado para inserir em cada formulário retornado para o cliente, um campo *input* do tipo *hidden*. Esse campo *input* terá o valor do UUID de um *token* gerado pelo *proxy* reverso. Posteriormente na requisição POST o valor deste campo será validado a fim de permitir o envio dos dados para o servidor, a seguir a Figura 47 é utilizada como exemplo para demonstrar o *proxy* reverso inserindo um *token* em um formulário.

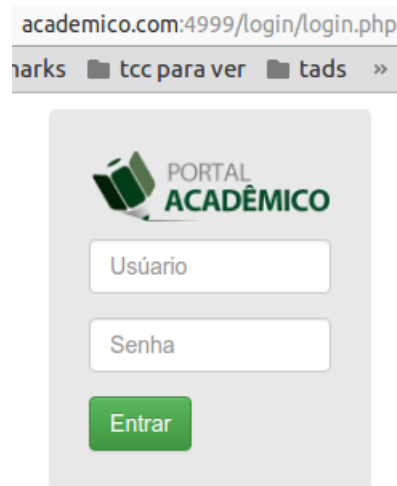


Figura 47: Login do sistema acadêmico através do *proxy* reverso. Fonte: autoria própria.

A Figura 47 mostra uma requisição GET para o formulário de autenticação do sistema acadêmico. Após inspecionar no navegador o código fonte do formulário, podemos ver o *token* que foi inserido pelo *proxy* reverso, segue o código na Figura 48.

```
<form method="post" action="autenticacao.php">
  <div class="form-group">
    <input type="text" class="form-control" name="usuario" placeholder="Usuário" required="">
  </div>
  <div class="form-group">
    <input type="password" class="form-control" name="senha" placeholder="Senha" required="">
  </div>
  <button type="submit" class="btn btn-success">Entrar</button>
  <input type="hidden" name="token" value="ca6dfb2c-edeb-46b6-8f8a-13655f8d7f2d">
</form>
```

Figura 48: Código fonte de um Formulário com *token*. Fonte: autoria própria.

A Figura 48 mostra o código fonte do formulário de autenticação do sistema acadêmico retornado após ser realizada uma requisição GET para o *proxy* reverso. Como pode ser observado, o formulário foi retornado contendo um *input* oculto com o valor de um UUID de um *token* válido gerado através do *proxy*, que servirá para permitir o envio dos dados posteriormente na requisição POST.

### 7.3.5 Negando POST com o *proxy* reverso

Para demonstrar como evitar um ataque CSRF através de uma requisição POST utilizando o *proxy* reverso desenvolvido durante este trabalho, foi criado um formulário para realizar uma requisição POST para a página de atualização de notas de alunos do sistema acadêmico, segue o exemplo do formulário na Figura 49.

## Post Fake

N1

N2

N3

N4

Enviar

Figura 49: Formulário para realizar uma requisição POST para o sistema acadêmico. Fonte: autoria própria.

A Figura 49 mostra o formulário para realizar uma requisição POST para a página de atualização de notas do sistema acadêmico. Este formulário contém quatro campo para simular a atualização de notas de um aluno conforme o formulário utilizado pela página verdadeira do sistema. Na Figura 50 pode ser observado o código fonte do formulário.

```
<h1>Post Fake</h1>
<form method="post" action="http://localhost:4999/academico/turmas/atualizaNotas.php">
  <input type="hidden" name='id' value="1">
  <div class="form-group">
    <label for="n1">N1</label>
    <input type="text" class="form-control" id="n1" name="n1" value="0">
  </div>
  <div class="form-group">
    <label for="n2">N2</label>
    <input type="text" class="form-control" id="n2" name="n2" value="0">
  </div>
  <div class="form-group">
    <label for="n3">N3</label>
    <input type="text" class="form-control" id="n3" name="n3" value="0">
  </div>
  <div class="form-group">
    <label for="n4">N4</label>
    <input type="text" class="form-control" id="n4" name="n4" value="0">
  </div>
  <button type="submit" class="btn btn-default">Enviar</button>
</form>
```

Figura 50: Código fonte do formulário da Figura 49. Fonte: autoria própria.

Como pode ser observado na Figura 50, o atributo *action*<sup>25</sup> do formulário indica que ao ser submetido será feita uma requisição POST para o domínio e porta onde está respondendo o servidor do *proxy* reverso. Por realizar os testes localmente e estar em desenvolvimento, foi configurado manualmente o domínio e a porta para realizar a requisição para o servidor do *proxy* reverso, ficando esta configuração visível para o usuário como pode ser observado na Figura 50. No entanto,

<sup>25</sup> É um atributo da tag `<form>` da linguagem *html*. Serve para especificar para onde será enviado os dados de um formulário após o mesmo ser submetido.



se o *proxy* reverso estivesse em produção teria que ser configurado corretamente pelo administrador do servidor os direcionamentos para as requisições cheguem primeiro ao *proxy* reverso.

Conforme as técnicas estudadas durante o projeto as aplicações protegidas pelo *proxy* reverso, receberão nos seus formulários um *input* do tipo *hidden* no qual seu valor que é um UUID de um *token*, gerado pelo *proxy* reverso. O UUID será validado posteriormente na requisição POST. Na Figura 51 pode ser observado a lógica feita no *proxy* quando recebe uma requisição GET para inserir os *tokens*.

```

if(html.ehHTML contentType)
  if(html.possuiElemento(conteudo, "form"))
    tokensGerados = listaDeTokens.gerarTokens(html.numeroDeFormularios(conteudo), dataHoraAtual.emMinutos, host)
    documentoHTML = html.inserirInputNosFormularios(conteudo, tokensGerados)
    conteudo = html.geraHtmlRetorno(documentoHTML)
  end
end

```

Figura 51: Inserção de *tokens* na requisição GET. Fonte: autoria própria.

A Figura 51 mostra um trecho de código *proxy* reverso. Nesse ponto de código é feita a inserção dos *tokens*. Os *tokens* gerados são únicos e tem um tempo máximo definido para serem aceitos posteriormente em uma requisição POST. Caso não for definido nas configurações, o tempo do máximo de cada *token* é de uma hora. Após passado esse tempo o *token* é removido da lista de *tokens* e uma requisição POST com esse *token* não é mais aceita pelo *proxy*. Na Figura 52 é demonstrado a lógica do método `excluiTokensExpirados` da classe `ListaDeTokens` que é responsável por excluir os *tokens* que tiveram o tempo expirado.

```

def excluiTokensExpirados()
  scheduler = Rufus::Scheduler.new
  scheduler.repeat tempoVerificacaoTokens do
    time = Time.new
    tempoAtualEmMinutos = (time.hour*60)+time.min+((time.sec/60))
    tokens.each do |token|
      if (tempoAtualEmMinutos - token.time) >= tempoMaxToken
        removerToken token
      end
    end
  end
end

```

Figura 52: Método `excluiTokensExpirados` da classe `ListaDeTokens`. Fonte: autoria própria.

A figura 52 contém um trecho de código da classe `ListaDeTokens` onde mostra o código fonte do método `excluiTokensExpirados`. Este método é chamado assim que a lista de *tokens* é criada no *proxy*. Sua responsabilidade é procurar no *array* de *tokens* da lista de *tokens* do *proxy* reverso, os *tokens* que tiveram seu tempo de vida expirado e removê-los. Este método tem um

tempo definido de intervalo para realizar a monitoração da lista de *tokens*, o tempo padrão é de dez segundos, podendo ser definido um tempo nas configurações do *proxy* reverso.

```

if podePostar(request.xhr?,parametros,tempoAtualPost,host)
  token = listaDeTokens.pegarToken parametros["token"]
  listaDeTokens.removerToken token
  parametros.delete "token"
  consulta = cliente.postarUrl(url,parametros,
    cabecalhosDeConsulta
  )
  limpaArquivosTemporarios
  contentType = consulta.headers['Content-Type']
  conteudo = consulta.body

  if(html.ehHTML contentType)
    if(html.possuiElemento(conteudo,"form"))
      tokensGerados = listaDeTokens.gerarTokens(
        html.numeroDeFormularios(conteudo),
        dataHoraAtual.emMinutos,host
      )
      documentoHTML = html.insereInputNosFormularios(
        conteudo,
        tokensGerados
      )
      conteudo = html.geraHtmlRetorno(documentoHTML)
    end
  end

  status consulta.status
  conHeaders = consulta.headers
else
  status 403
  conHeaders = {}
  conteudo = erb :erroPermissao, :format => :html5
end

```

Figura 53: Lógica executada pela classe ProxyBase nas requisições POST. Fonte: autoria própria.

A Figura 53 mostra um trecho de código da classe ProxyBase. Este código contém a lógica para realizar uma requisição POST pelo *proxy* reverso. Como pode ser observado na Figura, antes de realizar a requisição para o servidor real, é chamado o método `podePostar` para verificar os parâmetros da requisição a fim de enviar dos dados para o servidor *web* real ou não. A Figura 54 contém o código fonte deste método.

```

def podePostar(ehAjax,parametrosRecebidosPost,tempoAtualPost,host)
  negarPost = false
  if ehAjax
    negarPost = true
  elsif parametrosRecebidosPost.has_key? "token"
    token = listaDeTokens.pegarToken parametrosRecebidosPost["token"]
    if !token.nil?
      if((tempoAtualPost - token.time) <
        listaDeTokens.tempoMaxToken)
        | && token.host == host
          negarPost = true
        end
      end
    end
  end
  negarPost
end

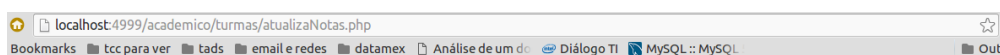
```

Figura 54: Método `podePostar` da classe `ProxyBase`. Fonte: autoria própria.

A Figura 54 mostra a lógica do método `podePostar` da classe `ProxyBase`. Este método verifica se os dados enviados para o servidor possuem um *token* e faz algumas validações a fim de autorizar a requisição `POST`.

Ao executar este método são passados os parâmetros recebidos durante uma requisição `POST`. Caso seja encontrado um *token* nesses parâmetros é verificado se o mesmo encontra-se expirado e se o domínio é o mesmo de onde vem a requisição, caso o *token* passe nessas validações a requisição `POST` é autorizada, se alguma dessas validações falharem ou o *token* não for encontrado, a requisição não é autorizada. A execução deste método libera as requisições que vierem por *ajax*, pois neste trabalho não serão desenvolvidas proteções para este tipo de requisição por questões de tempo e falta de estudo.

Na Figura 55 é demonstrado o retorno do *proxy* reverso após do envio dos dados do formulário da Figura 49.



Permissão negada!

Figura 55: Mensagem de retorno do *proxy* após a tentativa de enviar o formulário da Figura 49. Fonte: autoria própria.

Como pode ser observado na Figura 55 o *proxy* reverso retornou uma mensagem de permissão negada após a tentativa do POST do formulário da Figura 49. Esta mensagem foi retornada porque ao ser executado o método `podePostar` da classe `ProxyBase`, o qual foi explicado na Figura 54, não foi encontrado o *token* nos parâmetros e o domínio de onde partiu a requisição também não era o mesmo atribuído ao *token* quando foi gerado pelo *proxy* reverso.

A seguir será demonstrado um POST utilizando o aplicativo do navegador Google Chrome chamado Postman. Com este aplicativo é possível simular requisições GET e POST para um servidor *web*. Neste teste poderá ser verificado o *status* de retorno do *proxy* reverso quando uma requisição POST for negada.

O teste será realizar uma requisição POST para a página de atualização de notas dos alunos junto com os parâmetros que são o *id* do aluno e as notas *n1*, *n2*, *n3* e *n4*.

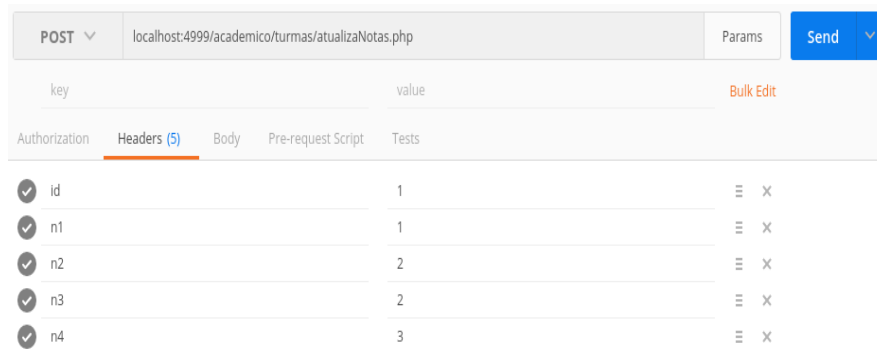


Figura 56: Interface do aplicativo Postman. Fonte: autoria própria.

A Figura 56 mostra a *interface* do aplicativo Postman onde foi configurado uma requisição POST para a página de atualização de notas dos alunos.

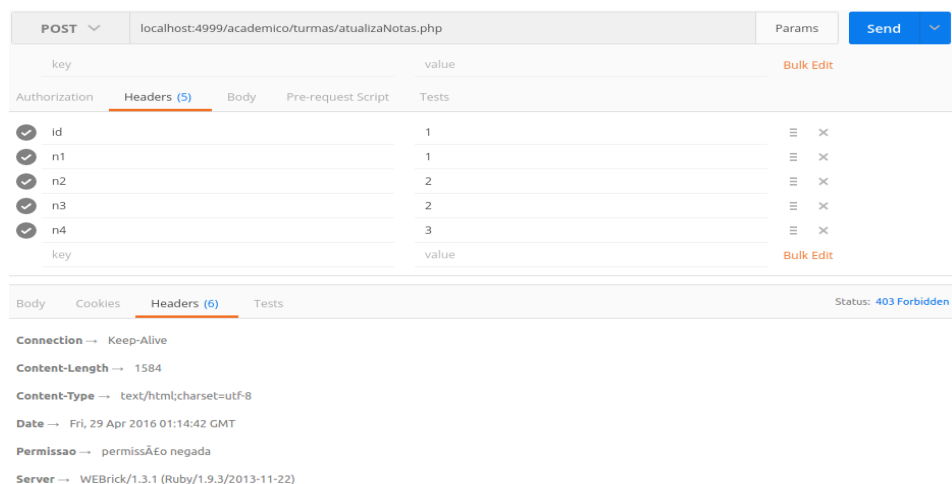


Figura 57: Status de retorno do servidor *proxy* reverso. Fonte: autoria própria.

A Figura 57 mostra *status* de retorno da requisição `POST` através do aplicativo Postman. O *status* retornado foi 403, que é um *status* de erro do protocolo HTTP onde o servidor não permite o acesso a um recurso. Sempre que um `POST` for negado pelo *proxy* reverso será retornado este *status*.

Com o teste de envio de dados através do formulário da Figura 49 e a requisição `POST` configurada no aplicativo Postman conforme demonstrado na Figura 56, pode ser demonstrado o uso do servidor *proxy* reverso desenvolvido durante o projeto para evitar um ataque CSRF.

## 8 CONCLUSÃO

Este presente trabalho teve como motivações realizar estudos sobre a área de Segurança da Informação e segurança para *internet*, pois estes assuntos não fizeram parte de nenhuma disciplina da grade obrigatória no durante o curso.

Com base nas motivações do trabalho, foi escolhido tratar uma vulnerabilidade explorada em aplicações desenvolvidas para a plataforma *web* chamada CSRF, que é considerada pela OWASP umas das dez mais críticas vulnerabilidades conhecidas para aplicações desta plataforma. Para prevenir essa vulnerabilidade foi definido como objetivo desenvolver um *proxy* reverso capaz prevenir ataques CSRF. Foram definidos como objetivos para o *proxy* reverso: prevenir aplicações desenvolvidas em qualquer linguagem de programação ou para qualquer servidor *web* e utilizar alguma técnica estuda durante o trabalho para evitar os ataques CSRF.

Após realizar levantamentos sobre Segurança da Informação, estudar detalhadamente o ataque CSRF e as técnicas necessárias para evitar o mesmo, foram escolhidas as ferramentas necessárias para desenvolver o *proxy* reverso e dado início ao desenvolvimento.

Na etapa de desenvolvimento do *proxy* reverso foi desenvolvido as classes, diagrama de classes e toda a lógica necessária para prevenir ataques CSRF a uma aplicação. Durante esta etapa por questões de tempo, ficou definido como escopo tratar somente formulários *html*, deixando outros elementos passíveis de ataques CSRF e requisições via *ajax* como sugestão de melhorias futuras para o *proxy* reverso.

Após a etapa de desenvolvimento foi elaborado para testar o *proxy*, um cenário de testes simples que envolveu o desenvolvimento de um sistema acadêmico para atualizar notas de alunos. O sistema conta com uma página de autenticação onde um usuário cadastrado faz a autenticação no sistema para poder ter permissão de atualizar as notas dos alunos cadastrados. Este sistema foi criado vulnerável a ataques CSRF para realizar os testes necessários.

Durante a etapa de testes do *proxy* reverso, foram requisitados os formulários presentes na aplicação de teste. Neste primeiro momento o *proxy* cumpriu o que era esperado que foi inserir nos formulários um *input* oculto com um *token* gerado pelo *proxy*. Logo após esta primeira etapa de testes foi realizada a tentativa de submeter um formulário desenvolvido para simular um ataque CSRF, neste teste o *proxy* conseguiu cumprir também o que era esperado, que foi negar a tentativa não permitindo o envio dos dados para o servidor na requisição `POST`.

Os testes realizados partiram do princípio que seriam utilizadas requisições `GET` apenas para trazer dados do servidor e requisições `POST` somente para enviar (indicado pela OWASP como uma boa prática para desenvolver aplicações para a plataforma *web*), que a segurança do navegador não será quebrada a ponto de vazarem o *token* oculto do formulário e também que não seja descoberto como burlar o algoritmo utilizado para gerar o *token*. Portanto se for utilizado o método `GET` para enviar dados para o servidor, o algoritmo de geração de *tokens* for quebrado e a segurança do navegador seja comprometida a ponto do *token* secreto ser vazado, o *proxy* reverso não funcionaria.

Portanto ficou concluído com o presente trabalho, que uma aplicação que esteja utilizando o método `POST` para enviar dados de formulário para o servidor, a segurança do navegador não seja comprometida e seja utilizada uma técnica estudada, testada e confiável para prevenir ataques CSRF é possível com um *proxy* reverso prevenir aplicações vulneráveis a ataques CSRF. Também pode ser concluído que o *proxy* reverso conseguiu cumprir alguns de seus objetivos que foram definidos durante o projeto que são: prevenir aplicações independentemente da linguagem de programação que foram desenvolvidas e realizar requisições para qualquer servidor *web*.

## 8.1 Trabalhos futuros

Por questões de tempo foi implementada proteção contra ataques CSRF somente para formulários *html* e desenvolvida lógica no *proxy* apenas para realizar requisições `GET` e `POST`, desta forma fica como trabalhos futuros:

- Requisições via *ajax*: neste trabalho as requisições via *ajax* passam de forma livre no *proxy*, desta forma fica como trabalho futuro um estudo de alguma técnica para que o *proxy* seja capaz de proteger aplicações contra ataques CSRF.
- Elementos *html* com os atributos *href* e *src*: neste projeto por questões de tempo só foi implementada proteção para formulários *html*, desta forma fica como trabalho futuro implementar proteção para elementos que contenham os atributos citados.
- Rotas `PUT`, `DELETE` entre outras: no *proxy* reverso foi implementada rota para requisições `POST` e `GET`, desta forma fica como trabalho futuro a implementação de outras possíveis rotas.

## REFERÊNCIAS

BELL,P.; BEER,B. Introdução ao GitHub. Primeira Edição. São Paulo: Novatec Editora Ltda. Janeiro de 2015. 136 p.

BRCONNECTION. Disponível em: <<http://www.omnewall.com.br/tecnologia/omne-wall.php>>. Acesso em 6 de junho de 2015.

BullGuard.Disponível em: <<http://www.bullguard.com/pt-br/bullguard-security-center/internet-security/security-tips/cybercrime.aspx>>. Acesso em 20 de junho de 2016.

CERT. Cartilha de Segurança para Internet. Disponível em: <<http://cartilha.CERT.br/>>. Acesso em 21 de abril de 2015.

CORREIO, BRAZILIENSE. Disponível em: <[http://www.correiobraziliense.com.br/app/noticia/economia/2015/01/11/internas\\_economia,465719/brasil-deve-superar-o-japao-em-numero-de-acessos-a-internet-em-2015.shtml](http://www.correiobraziliense.com.br/app/noticia/economia/2015/01/11/internas_economia,465719/brasil-deve-superar-o-japao-em-numero-de-acessos-a-internet-em-2015.shtml)>. Acesso em 05 de julho de 2015.

CRYPTOID. Disponível em : <<https://cryptoid.com.br/ssl/o-que-sao-ssl-ssh-https/>> Acesso em 20 de junho de 2016.

GITHUB. FEATURES. Disponível em:<<https://github.com/features>>. Acesso em 05 de julho de 2015.

HARRIS,A.; HAASE,K. Sinatra,Up and Running. Primeira Edição. Gavenstein Highway North: O'Reilly. 2012. 103 p.

HTTPCLIENT. Disponível em: <<https://github.com/nahi/httpclient>>. Acesso em: 05 de julho de 2015.

JQUERY. FOUNDATION. Disponível em: <<https://jquery.org/license/>>. Acesso em 22 de junho de 2015.

MANN,I. Engenharia Social. Primeira Edição. São Paulo: Editora Edgard Blücher, 2015. 233 p.

Nokogiri. Disponível em: <<http://www.rubydoc.info/github/sparklemotion/nokogiri>>. Acesso em 10 de junho de 2016.

OLHARDIGITAL. Disponivel em: <[http://olhardigital.uol.com.br/fique\\_seguro/noticia/qual-a-diferenca-entre-hacker-e-cracker/38024](http://olhardigital.uol.com.br/fique_seguro/noticia/qual-a-diferenca-entre-hacker-e-cracker/38024)>. Acesso em 29 de junho de 2016.

OWASP. CSRFGUARD PROJECT. Disponível em: <[https://www.owasp.org/index.php/Category:OWASP\\_CSRFGuard\\_Project](https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project)>. Acesso em 14 de junho de 2015.

OWASP. Cross-Site Request Forgery (CSRF) Prevenção Cheat Sheet. Disponível em: <<https://www.owasp.org/index.php/Cross->



Site\_Request\_Forgery\_(CSRF)\_Prevention\_Cheat\_Sheet#General\_Recommendation:\_Synchronize\_Token\_Pattern>. Acesso em 24 de maio 2015.

OWASP. CROSS-SITE SCRIPTING. Disponível em: <[https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))>. Acesso em 15 de junho de 2015.

OWASP. HTTPONLY. Disponível em: <<https://www.owasp.org/index.php/HttpOnly>>. Acesso em 21 junho de 2015.

OWASP. Top 10 2013. Disponível em:

<[https://www.owasp.org/images/9/9c/OWASP\\_Top\\_10\\_2013\\_PT-BR.pdf](https://www.owasp.org/images/9/9c/OWASP_Top_10_2013_PT-BR.pdf)>. Acesso em 21 de maio de 2015.

POLITICA. DE. MESMA. ORIGEM. Disponível em: <[http://pt.wikipedia.org/wiki/Pol%C3%Aadtica\\_de\\_mesma\\_origem](http://pt.wikipedia.org/wiki/Pol%C3%Aadtica_de_mesma_origem)>. Acesso em 10 de junho de 2015.

RACK.GITHUB.IO. Disponível em: <<http://rack.github.io/>>. Acesso em 10 de junho de 2015.

RUBY-LANG. Disponível em: <<https://www.ruby-lang.org/pt/>>. Acesso em 10 de junho de 2015.

Rufus-Scheduler. Disponível em: <<https://github.com/jmettraux/rufus-scheduler>>. Acesso em 11 de junho de 2016.

SQUID-CACHE. Disponível em: <<http://www.squid-cache.org/>>. Acesso em 6 de junho de 2015.

TLDRLEGAL. Disponível em: <<https://tldrlegal.com/license/mitlicense#changesets/active>>. Acesso em 22 de junho de 2015.

TANENBAUM, ANDREW.S.;. Redes de Computadores 4ª Edição. Primeira Edição. Editora , 2015. 233 p.

UBUNTU-BR. ORG. Disponível em: <<http://ubuntu-br.org/ubuntu>>. Acesso em 22 de junho de 2015.

W3SCHOOLS. Disponível em: <[http://www.w3schools.com/tags/ref\\_attributes.asp](http://www.w3schools.com/tags/ref_attributes.asp)>. Acesso em 21 de junho de 2016.

W3C. Disponível em: <<https://www.w3.org/Protocols/HTTP/HTTRQ-Headers.html>>. Acesso em 22 de julho.