MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

1º Ten BRUNO MATISSEK WORM
Cap NARCELIO RODRIGUES DE MEDEIROS
1º Ten PEDRO LUCAS PORTO ALMEIDA

# UM FRAMEWORK PARA AUTOMAÇÃO DE PENTEST

Rio de Janeiro
2017

**INSTITUTO MILITAR DE ENGENHARIA**

**1º Ten BRUNO MATISSEK WORM**
**Cap NARCELIO RODRIGUES DE MEDEIROS**
**1º Ten PEDRO LUCAS PORTO ALMEIDA**

# UM FRAMEWORK PARA AUTOMAÇÃO DE PENTEST

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Maj Anderson Fernandes Pereira dos Santos - D.Sc.
Co-Orientador: Prof. Ricardo Choren Noya - D.Sc.

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270



005 WORM, BRUNO MATISSEK
W928f Um Framework para automação de PENTEST / BRUNO MATISSEK WORM, NARCELIO RODRIGUES DE MEDEIROS, PEDRO LUCAS PORTO ALMEIDA, orientado por Anderson Fernandes Pereira dos Santos e Ricardo Choren Noya - Rio de Janeiro: Instituto Militar de Engenharia, 2017.

48p.: il.

Projeto de Fim de Curso (graduação) - Instituto Militar de Engenharia, Rio de Janeiro, 2017.

1. Curso de Graduação em Engenharia de Computação - projeto de fim de curso. 1. *Framework*. 2. *Pentest*. I. dos Santos, Anderson Fernandes Pereira . II. Noya, Ricardo Choren. III. Título. IV. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

1° Ten BRUNO MATISSEK WORM
Cap NARCELIO RODRIGUES DE MEDEIROS
1° Ten PEDRO LUCAS PORTO ALMEIDA

# UM FRAMEWORK PARA AUTOMAÇÃO DE PENTEST

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Maj Anderson Fernandes Pereira dos Santos - D.Sc.

Co-Orientador: Prof. Ricardo Choren Noya - D.Sc.

Aprovado em 28 de Setembro de 2017 pela seguinte Banca Examinadora:

Maj Anderson Fernandes Pereira dos Santos - D.Sc. do IME - Presidente

Prof. Ricardo Choren Noya - D.Sc. do IME

Ten Cel Sérgio dos Santos Cardoso Silva - M.Sc. do IME

Prof. Ronaldo Ribeiro Goldschmidt - D.Sc. do IME

Rio de Janeiro
2017

Ao Instituto Militar de Engenharia, alicerce da minha formação e aperfeiçoamento.

## AGRADECIMENTOS

Agradeço a todas as pessoas que me incentivaram, apoiaram e possibilitaram esta oportunidade de ampliar meus horizontes.

Meus familiares, colegas e mestres.

Em especial ao meu Professor Orientador Maj. Anderson Fernandes Pereira dos Santos e ao Professor Co-orientador Dr. Ricardo Choren Noya, por suas disponibilidades e atenções.

"If you spend more on coffee than on IT security, you will be hacked. What's more, you deserve to be hacked."

RICHARD CLARKE

**SUMÁRIO**

## LISTA DE ILUSTRAÇÕES

## LISTA DE SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| CERT | Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança |
| CSRF | Cross-Site Request Forgery |
| DoS | Denial of Service |
| DDoS | Distributed Denial of Service |
| HTML | HyperText Markup Language |
| HTTP | HyperText Transfer Protocol |
| ITU | International Telecommunication Union |
| OWASP | Open Web Application Security Project |
| SQL | Structured Query Language |
| XSS | Cross-Site Scripting |

## RESUMO

O desenvolvimento de aplicações seguras é um dos maiores desafios da indústria de tecnologia. Para auxiliar em tal tarefa, diversas técnicas de detecção e correção de vulnerabilidades foram desenvolvidas, dentre elas o teste de invasão, que consiste em simular uma invasão ao sistema para identificar as falhas de segurança existentes. Esse trabalho tem como objetivo propor um *framework* automatizado de testes de invasão para auxiliar na análise e correção de vulnerabilidades, visando aprimorar a segurança de aplicações existentes.

## ABSTRACT

The development of secure applications is one of the biggest challenges in the technology industry. To assist in such a task, several vulnerability detection and correction techniques have been developed, such as the penetration test, which simulates an attack to discover existing security flaws. This work aims to propose an automated penetration testing framework to assist in the analysis and correction of vulnerabilities, aiming to improve the security of existing applications.

# 1 INTRODUÇÃO

O número de usuários de Internet do mundo é cada vez maior. Em 2000, de acordo com dados da ITU (2015), aproximadamente 400 milhões de pessoas possuíam acesso à Internet no mundo todo. Já em 2015, a estimativa subiu para 3,2 bilhões de indivíduos, correspondendo à 43,4% da população mundial

Paralelamente com o aumento de usuários da Internet, também cresce o número de incidentes relacionados à segurança. Segundo dados fornecidos pelo CERT (2015), mais de 700 mil incidentes de segurança foram reportados à organização em 2015. A existência de vulnerabilidades em *software* é uma fonte de ameaças à infraestrutura dos mais diversos setores da nossa sociedade, como os de defesa, saúde, energia e financeiro.

O desenvolvimento de aplicações seguras figura entre um dos maiores desafios da atualidade. Analisando dados do CERT.br, pode-se perceber que os incidentes de rede têm aumentado de forma exponencial, fato agravado em anos de realização de grandes eventos, como a Copa do Mundo, realizada no Brasil em 2014, conforme representado na FIG. 1.1.

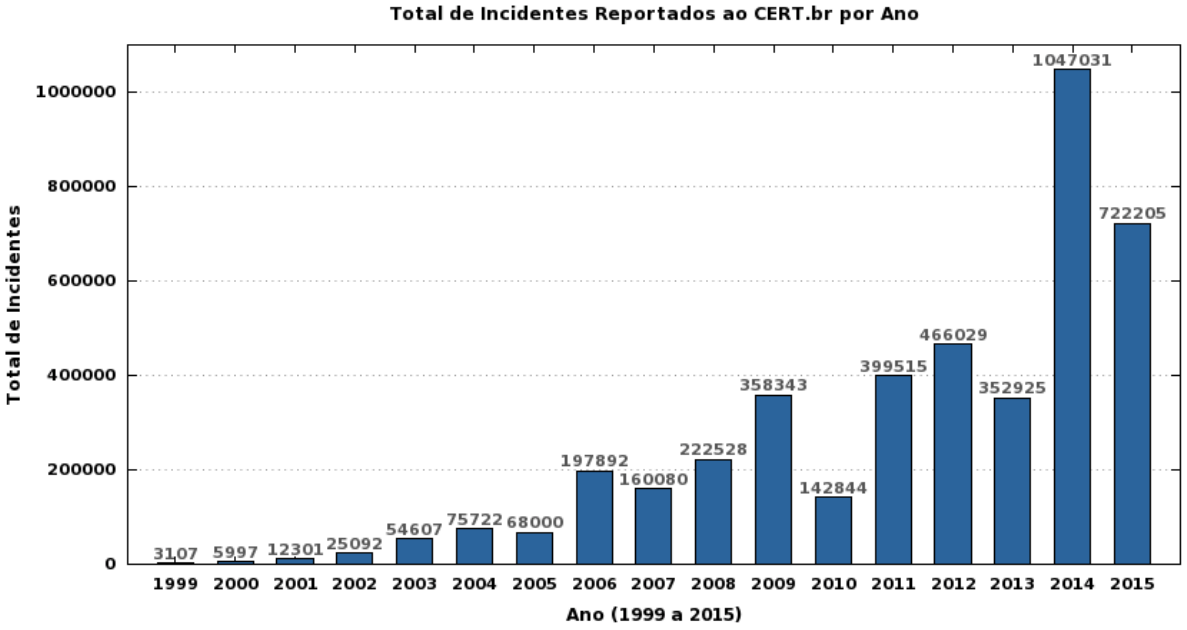


FIG. 1.1: Incidentes de Rede por Ano (CERT, 2015)

A Estratégia Nacional de Defesa estabeleceu diversas diretrizes para desenvolvimento de capacidades nas Forças Armadas para longo prazo. Em uma delas, o governo tratou a

dimensão cibernética como uma preocupação da nação e atribuiu a sua responsabilidade ao Exército Brasileiro (BRASIL, 18 dez. 2008).

Tal assunto carrega grande importância no cenário atual, pois um sistema com vulnerabilidades pode ser explorado e causar danos aos ativos e à reputação de grandes organizações. Um exemplo disto ocorreu em 2015, (GLOBO, 2015) onde houve um acesso não autorizado a um banco de dados de um servidor do Exército, gerando a exposição de diversas informações de militares e de seus dependentes. Este incidente foi extensivamente explorando pela mídia, tendo em vista a responsabilidade do Exército em promover a pesquisa na área.

## 1.1 OBJETIVO

O objetivo do presente trabalho é desenvolver um *framework* de automação de testes de invasão, servindo de interface única entre o usuário e as ferramentas que normalmente são utilizadas durante as etapas de um teste de invasão. Tal *framework* deverá ser escalável, de forma que módulos considerados relevantes sejam facilmente integrados ao sistema.

Como prova de conceito, serão implementados dois módulos referentes às vulnerabilidades *WEB* de *SQL Injection* e *Cross-Site Scripting*, que serão testados em uma aplicação que esteja sendo executada em um ambiente controlado a fim de conter impactos negativos decorrentes das práticas.

## 1.2 JUSTIFICATIVA

O presente projeto se traveste de alta prioridade em termos de manutenção da imagem da força, pois pretende-se desenvolver uma ferramenta que irá relacionar as brechas de segurança, informando aos administradores os pontos vulneráveis antes que a plataforma esteja disponível na internet, ou sempre que for necessário. Fato, que por si só, reduzirá a probabilidade de invasões e acessos não autorizados.

Por outro lado, este projeto agrega um alto valor acadêmico, tendo em vista que contribui com pesquisas na área de defesa cibernética. Além de envolver grandes áreas de conhecimento como redes de computadores, segurança da informação e linguagens de programação instigando estudos ainda mais aprofundados em cada uma dessas áreas.

## 1.3 METODOLOGIA

A metodologia utilizada para atingir o objetivo proposto neste trabalho foi dividida em:

- *Fundamentação Teórica.* Para a correta implementação do sistema em questão, é necessário compreender os conceitos de segurança da informação, com ênfase em vulnerabilidades *WEB*; testes de invasão e desenvolvimento de *frameworks*.

- *Engenharia de Requisitos*

  *Análise de cenários.* Baseado na peculiaridade de cada fase do teste de invasão, decidiu-se que o núcleo deveria ser capaz de executar módulos individuais e que, independente da função executada por este, o núcleo seria capaz de gerar um relatório contendo a atividade realizada. Além disso, devido a sua importância, optou-se por implementar os módulos *XSS* e *SQL injection*.

  *Análise de Requisitos dos Ataques.* Com base o estudo mais aprofundado de cada vulnerabilidade, decidiu-se as funcionalidades presentes em cada um dos módulos.

- *Análise e Projeto.* Definiram-se aspectos mais detalhados da modelagem do projeto, interfaceamento entre os módulos e o *framework* e a interface de uso com o usuário

- *Codificação.* Inicialmente foi implementado o núcleo juntamente com um módulo para realizar um teste simples da funcionalidade de suas funcionalidades, bem como a geração de relatórios, após isso, foi implementado o módulo de *SQL Injection* e, por fim, o módulo de *XSS*.

- *Teste.* Fase na qual ocorreu a implementação de testes das funcionalidades previstas ao se aplicar os módulos de análise de vulnerabilidades *WEB* em aplicações executadas em um ambiente controlado.

## 1.4 ESTRUTURA DO TEXTO

A organização deste trabalho é feita de acordo com o seguinte formato:

- *Capítulo 2*: realiza-se a discussão sobre definições e conceitos teóricos utilizados na concepção e desenvolvimento do projeto.

- *Capítulo 3*: apresenta e delimita as características do *framework* construído, bem como de seus módulos.

- *Capítulo 4*: discute-se os testes e resultados obtidos a partir da utilização dos módulos implementados.

- *Capítulo 5*: expõe-se a conclusão do projeto e propõe-se temas de trabalhos futuros.

# 2 CONCEITOS TEÓRICOS

## 2.1 SEGURANÇA DA INFORMAÇÃO

Para realizar o desenvolvimento de uma aplicação que automatize o teste de invasão, é necessário realizar algumas definições que servirão de fundamentação teórica para o desenvolvimento deste projeto. Segundo Herzog (2010), segurança é uma função de separação, ou seja, é a separação entre o ativo e qualquer que seja a ameaça, sendo essa existente ou não. Além disso, propõe três formas lógicas e proativas para criar essa separação:

- Mudança do ativo a fim de criar uma barreira física ou lógica entre ele e as ameaças;
- Alterar a ameça para um estado inofensivo; e
- Destruir a ameaça.

Segurança da informação pode ser caracterizada como a proteção de informações individuais ou organizacionais, seja no processo de armazenamento em diversas mídias ou em seu trânsito, a fim de garantir a continuidade do negócio. São pilares fundamentais da segurança da informação a confidencialidade, integridade e disponibilidade (ABNT, 2016). Herzog (2010) ainda projeta uma relação entre as três características, definindo-as como os "objetivos garantidores da informação"e com os "controles de operação". A confidencialidade é a propriedade para assegurar que um bem exibido ou trocado entre partes que interagem entre si não pode ser disponibilizado para outrem que não tenha autorização para tal. A confidencialidade depende da autenticação, que é a propriedade através de algum desafio de credenciais baseadas em identificação e autorização, ou seja, saber se quem está acessando determinado documento está autenticado, é quem diz ser, e se está autorizado a acessar o mesmo.

Integridade é a propriedade para assegurar que as partes que interagem sabem que os ativos e os processos não foram alterados por pessoas não autorizadas, princípio da salvaguarda da exatidão. Herzog (2010) ainda complementa com o conceito de subjugação que é um controle assegurando que as interações ocorrem apenas de acordo com processos definidos.

Disponibilidade trata-se da propriedade de estar acessível sempre quando for necessário por pessoas, sistemas ou entidades autorizadas. Continuidade é a propriedade que

atua sobre todas as interações a fim de manter a interatividade com os ativos mesmo em eventos de falha ou corrupção.

## 2.2 INCIDENTES E AMEÇAS

Segundo CERT (2012), incidente de segurança pode ser definido como qualquer evento adverso ao sistema computacional ou à rede de computadores. Consideram-se incidentes, as atividades sob suspeita ou as confirmadas. Dentre os exemplos mais comuns de incidentes de segurança estão:

a) Tentativas de uso ou ganho de acesso ilegal a sistemas ou dados;

b) Ataques de negação de serviço, ou seja, tentativa de tornar serviços indisponíveis; e

c) Modificações ilícitas no sistema alvo.

Esses incidentes podem ser entendidos como a exploração das vulnerabilidades de um sistema com intuito de causar dano ou operar de forma ilícita o mesmo.

Ameaça é a causa potencial gerada de um incidente, podendo ou não resultar num dano para os ativos ou sistemas de uma organização (ABNT, 2016).

### 2.2.1 TIPOS DE ATAQUES MAIS COMUNS

A FIG 2.1 mostra o número de incidentes reportados ao CERT.br de janeiro a dezembro de 2015, divididos por tipo de ataque. A maior quantidade de incidentes reportados foram ataques de varredura (*Scan*) que consistem em requisições feitas aos servidores do sistema com o intuito de identificar quais computadores estão ativos e quais serviços estão sendo disponibilizados por eles. Constitui-se um artifício amplamente utilizado com o objetivo de coletar informações de alvos em potenciais e as possíveis vulnerabilidades dos serviços ativos.

Em seguida estão as fraudes, ataques que consistem em se passar por outrem a fim de realizar alguma transação não autorizada, auferindo lucros a si próprio, tendo como seus representantes mais comuns as fraudes bancárias e fraudes de violações de direitos autorais.

Em terceiro lugar, estão os ataques *WEB*, nos quais o atacante visa o comprometimento de servidores disponíveis na internet para obter acesso a informações privilegiadas ou apenas desfigurar páginas disponíveis na internet (*Defacement*). Em 2015, foram reportados ao CERT.br 65.647 ataques *WEB*, ou seja, uma média de sete ataques e meio por

hora a servidores no Brasil, demonstrando a importância de um servidor bem configurado para que não tenha seus ativos vazados para pessoas não autorizadas.

Tendo em vista a grande quantidade de aplicações *WEB* que o Exército possui, aliado ao incidente já citado, decidiu-se pela priorização do teste de invasão baseado nas vulnerabilidades *WEB* com o intuito de produzir algo de relevância para a Força.

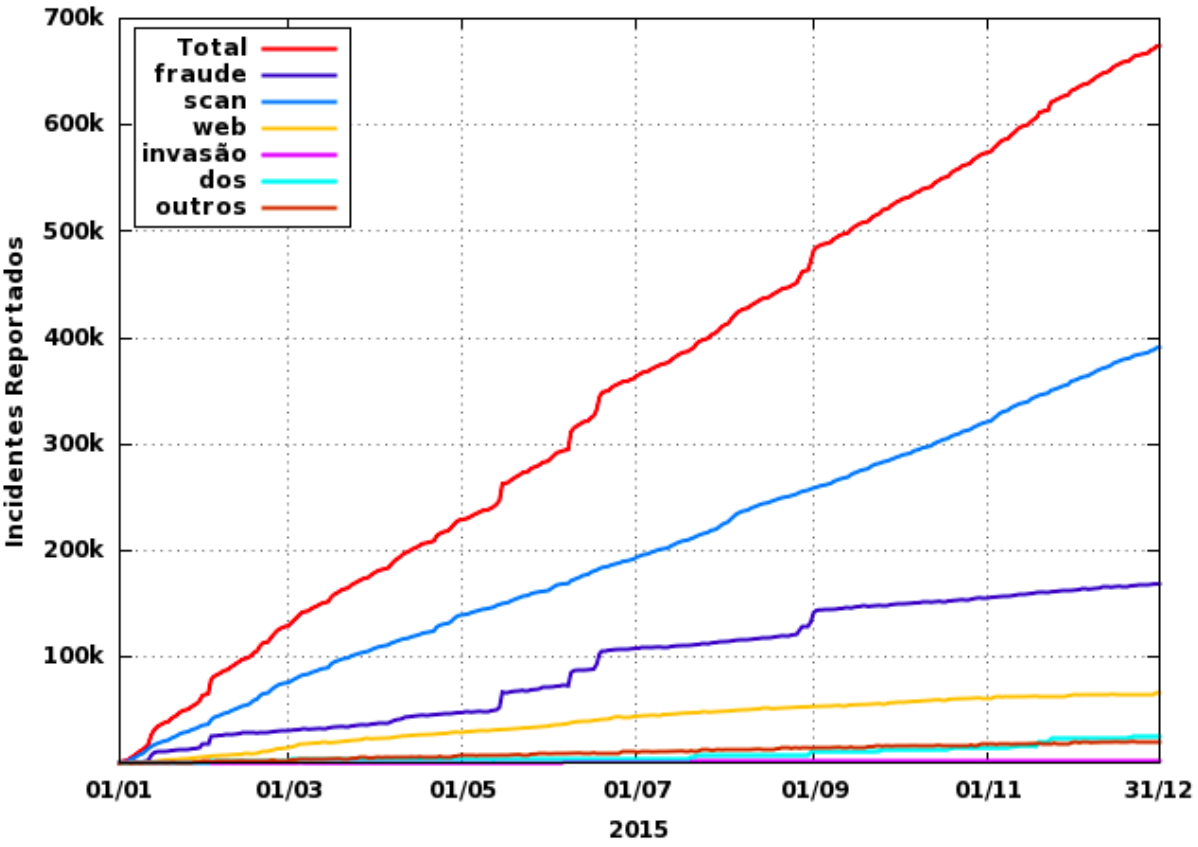


FIG. 2.1: Incidentes Reportados:Tipos de Ataques Acumulado (CERT, 2015)

Existem ainda os ataques de negação de serviço("*DoS*"ou "*DDoS*") , onde o atacante utiliza um computador ou um conjunto de computadores para sobrecarregar e tornar indisponível os serviços ou até a rede de uma organização. Por fim, outro ataque que também merece destaque é de engenharia social que tem como centro das atenções o usuário do sistema se valendo de diversos artifícios para que este revele informações, intencionalmente ou não, para obter informações privilegiadas ou até mesmo a extorsão.

## 2.3 TESTES DE INVASÃO

São testes realizados a fim de colocar a prova os dispositivos e políticas de segurança da empresa, realizados para identificar possíveis vulnerabilidades e seus impactos, serem

usados a fim de se obter uma certificação de organismos reguladores e, por fim, visam melhorar a infraestrutura organizacional e pessoal.

Cabe ressaltar que os testes são limitados, não constituindo em uma garantia de total segurança do sistema ou uma prevenção de futuros ataques, porém estes podem identificar vulnerabilidades que, ao serem corrigidas, reduzem significativamente a probabilidade de um ataque bem sucedido. Devido a descoberta constantes de novas vulnerabilidades, os testes de invasão se tornam rapidamente ultrapassados e, desse modo, devem ser realizados periodicamente.

### 2.3.1 TIPOS DE TESTE DE INVASÃO

FeOIS (2004) possui critérios detalhados quanto a classificação dos testes de invasão. Cada tipo de classificação se dá conforme o cenário onde o teste será realizado, a saber:

1. Quanto a Base de Informações: aquilo que é informado inicialmente sobre o alvo. E subdivide-se em:

    1.1. *Black-Box*: o teste deve ser realizado sem nenhum conhecimento prévio do sistema alvo. As informações devem ser adquiridas no decorrer do teste. É o teste mais aproximado ao que um atacante enfrentaria, porém necessita de um tempo maior para descoberta e exploração.

    1.2. *White-Box*: possui informações detalhadas sobre algumas partes ou totalidade do sistema, inclusive com acesso a fonte de códigos e a documentação. É um teste mais profundo, pois tem-se acesso a todas informações e possibilita um maior estudo no cerne do sistema, além de ter uma maior cobertura.

    1.3. *Grey-Box*: Têm-se um certo conhecimento de parte do sistema. Trata-se de um misto do *White-box* com o *Black-box* sendo possível combinar aspectos de ambos.

2. Quanto a abordagem: determina qual a visibilidade de quem realiza o teste. Subdivide-se em:

    2.1. Dissimulado: faz-se uso de métodos furtivos a fim de que o teste não seja identificado como uma tentativa de invasão. Há um alto custo e tempo associado a esse tipo de teste, pois faz-se necessário uma discrição, por este motivo esta abordagem não é frequentemente utilizada.

2.2. Ostensivo: não há preocupação com a detecção de invasão ou tentativa de invasão por parte da equipe de segurança da organização alvo.

3. Quanto a agressividade: determina o quão intrusivo o teste poderá ser.

   3.1. Passivo: todas as vulnerabilidades que são detectadas durante o teste não são exploradas, apenas reportadas.

   3.2. Cauteloso: as vulnerabilidades que foram encontradas passa por um processo de análise e, apenas aquelas onde o sistema testado não sofrerá algum dano, são testadas.

   3.3. Calculado: diferentemente do cauteloso, as vulnerabilidades que podem causar dano ao sistema, podem ser testadas, porém calcula-se previamente o quão grave será a consequência, bem como, as possibilidade de sucesso.

   3.4. Agressivo: levando em conta as possíveis consequências nos sistemas, a despeito de qualquer informação sobre o mesmo, todas as potenciais vulnerabilidades são testadas.

4. Quanto ao escopo: definição dos sistemas que serão testados. Subdivide-se em:

   4.1. Específico: define-se exatamente aquilo que deve ser testado.

   4.2. Limitado: há uma limitação clara dos sistemas que podem ser testados.

   4.3. Completo: todo os sistemas são testados.

5. Quanto ao conhecimento da Equipe de Segurança do alvo: determina se a equipe de segurança saberá ou não do teste.

   5.1. *Red Teaming*: nesse tipo, a equipe de segurança dos sistemas alvos não sabe sobre a ocorrência do teste, entretanto um indivíduo superior hierarquicamente a equipe deverá ter pleno conhecimento do teste a ser realizado. Este método possui a vantagem de ter a possibilitar a verificação dos procedimentos de resposta aos incidentes de segurança da equipe, podendo, ainda, avaliar o conhecimento da equipe com relação as políticas e práticas da empresa.

   5.2. *Blue Teaming*: ao contrário do *Red Teaming*, este tipo de teste é realizado apenas com o conhecimento e consentimento da equipe de segurança da organização alvo.

6. Quanto ao Ponto de Vista: definição da posição física em que é realizado o teste. Podendo ser:

    6.1. Interno: neste caso, o teste é realizado em um ponto interno à rede da organização alvo. Por isso, o teste já inicia com um bom nível de acesso e conhecimento sobre a rede.

    6.2. Externo: nesta modalidade, o teste é realizado num ponto externo à rede da organização alvo, possuindo acesso inicial limitado. No caso de realização de testes nos dois pontos de vista, o primeiro ponto a ser realizado é o interno.

## 2.4 METODOLOGIA

Inicialmente, deve-se partir do princípio que o um teste de invasão deve ser compatível com a organização que se deseja testar. Ou seja, deve-se avaliar cuidadosamente e classificar uma abordagem apropriada, bem como ter em mente quais são os resultados esperados.

Existem diferentes metodologias para os testes de invasão. Conforme Weidman (2014) um teste de invasão pode-se dividir em preparação, reconhecimento, escaneamento, exploração, pós-exploração e relatório.

A fase de preparação consiste em definir os objetivos para o teste, o mapeamento do escopo, além de definição do contrato do serviço. Esta fase não será nosso objeto de estudo. A seguir, abordaremos em mais detalhes as demais fases expostas. O reconhecimento é tido como a fase mais longa e importante, podendo, às vezes, durar semanas ou meses (WEIDMAN, 2014). Nesta fase, não há interação direta com o alvo. Deve-se utilizar diversas fontes para aprender o máximo possível sobre o este e como ele opera. Os passos mais comuns para se encontrar informações publicamente disponíveis segundo McClure et al. (2014), podem ser, mas não se limitam a:

- Busca nas páginas *WEB* da empresa, examinando inclusive, códigos-fonte *HTML*;
- Verificar quais são as organizações associadas, buscando por desenvolvimento terceirizado;
- Obter informações sobre funcionários, como e-mails, por exemplo;
- Verificar se existe algum grande evento, como fusões e aquisições, pois estes podem fornecer indícios e oportunidade que antes não existiam;
- Descobrir políticas de privacidade e segurança e/ou detalhes técnicos que indiquem o tipo de mecanismo de segurança em vigor;

- Procurar por informações arquivadas, recuperar cópias arquivadas utilizando, por exemplo, *WayBack Machine*, site que guarda *caches* guardados pelo site *Google* ao longo dos anos;

- Anúncios de emprego da organização os quais informam sobre a tecnologia utilizada pela mesma; e

- Determinação dos blocos de IP através de pesquisas relacionada ao domínio. Através de, por exemplo, pesquisas aos bancos de dados dos servidores *WHOIS*, nos quais guardam diversas informações do alvo.

Após cercar-se de todas informações suficientes para entender como o alvo funciona e quais ativos podem estar disponíveis, passa-se para a fase de escaneamento onde é realizada a primeira interação ativa com o alvo. O escaneamento equivale-se a inspecionar por portas como pontos de entradas em potencial, procurando por (MCCLURE et al., 2014):

- Portas abertas;

- Serviços ativos; e

- Aplicações vulneráveis.

A fase de exploração, que também é conhecida como obtenção efetiva de acesso ao sistema, pode ser considerada como fundamental em todo o teste. Seu objetivo principal consiste em extrair informações de grande valor agregado para a organização, demonstrando o quão impactante são as vulnerabilidades. Para isto é necessário obter algum nível de acesso ao serviço ou sistema e a partir daí escalar os níveis até ter acesso ao ativo da organização.

Conforme Weidman (2014), esta fase pode ser divida em outras três:

- Modelagem das ameaças: desenvolver planos de ataques e estratégias para a invasão de acordo com as informações coletadas nas fases anteriores;

- Análise de vulnerabilidades: descoberta ativa de vulnerabilidades para determinar se as estratégias de exploração de falhas serão bem-sucedidas. Uma falha mal sucedida pode ativar alertas de detecção de invasão e comprometer todo o teste. Nesta fase, são usados com frequência rastreadores de vulnerabilidades que se valem de banco de dados e outras verificações ativas para determinar com uma maior acurácia e as falhas do sistema alvo; e

- Exploração de falhas: por fim, a exploração de falhas é efetivamente utilizada contra o sistema de forma a ganhar o acesso.

Apesar da importância da fase de exploração, os passos que serão executados na fase pós-exploratória determinarão o sucesso de um teste bem realizado. Conforme já discutido, as informações coletadas devem ser importantes para a lógica do negócio da organização, portanto invadir a máquina de recursos humanos, por exemplo, pode não significar algo significativo, ou ainda, adquirir credenciais de acesso a um sistema que não possua permissões de alteração, execução, ou até mesmo visualização pode ser algo insignificante em determinados casos.

Na fase de pós-exploração de falhas, reunem-se informações sobre o sistema invadido, procura-se por arquivos relevantes, além de elevar o nível de acesso conquistado pois, com acessos de administradores, é possível alterar qualquer configuração que se queira, bem como visualizar todas as senhas e usuários disponíveis na máquina. Tal procedimento facilita a tentativa de acesso de outros sistemas, tendo em vista o conhecimento das credenciais. Outra possibilidade é realizar um *pivoteamento*, que consiste em utilizar o sistema invadido para atacar outros sistemas que não estavam disponíveis para elementos externos a rede (WEIDMAN, 2014). Por fim, deve-se gerar um relatório onde devem constar todas as informações realizadas ao cliente de maneira significativa. Devem ser abordados os aspectos positivos, ou seja, as boas práticas, mas principalmente os aspectos a serem melhorados, detalhando neste ponto a forma como a invasão ocorreu, o que foi descoberto e como corrigir os problemas.

É importante ser um relatório que aborde os tópicos de forma elucidativa, clara e técnica, pois servirá de apoio tanto para que as equipes responsáveis pela segurança possam entender o passo-a-passo da invasão e corrigir os tópicos necessários, quanto para a alta gerência que será a responsável pelas mudanças da política de segurança. Uma sugestão de Weidman (2014) é que um relatório deve conter:

- Sumário técnico: descreve-se os objetivos do teste e fornece uma visão geral das descobertas. Deve abordar o propósito do teste, uma visão geral da eficiência do teste, uma classificação geral da postura da empresa quanto à segurança, uma sinopse dos problemas identificados, um resumo das recomendações e oferecer objetivos de curto e longo prazo para que a organização possa melhorar sua postura; e

- Relatório técnico: fornece detalhes técnicos sobre o teste. Deve-se abordar os detalhes das descobertas feitas na fase de reconhecimento, de escaneamento, de exploração e de pós-exploração. Além de realizar uma descrição quantitativa do risco

identificado, fazendo uma estimativa, se for o caso, das perdas da organização caso as vulnerabilidades sejam exploradas por um invasor. E por fim, uma visão geral do teste.

## 2.5 VULNERABILIDADES EM APLICAÇÕES *WEB*

O OWASP é uma comunidade aberta cujo objetivo é possibilitar organizações a desenvolver, comprar e manter aplicações e APIs confiáveis. Dentro dos seus diversos projetos, destacamos o OWASP Top 10 *Most Critical Web Application Risks*(OWASP, 2017), sendo um *rank* feito pela organização com as dez principais classes de vulnerabilidades presentes em aplicações *WEB*, analisadas e classificadas em 2017 levando em conta os critérios de facilidade de exploração, detecção, difusão e impacto no negócio.

- *Injection:* considerada ainda a vulnerabilidade mais crítica pelos critérios do OWASP, a injeção de código ocorre quando dados de entrada não confiáveis são enviados a um interpretador como parte de uma consulta (por exemplo, SQL). Caso a rotina de tratamento dessa entrada não faça a sua devida validação, um usuário malicioso pode fazer com que o interpretador execute comandos não-intencionados ou acesse dados restritos.

- *Broken Authentication and Session Management:* a implementação incorreta de funções relacionadas à autenticação e gerenciamento de sessão acaba por permitir o comprometimento de senhas, chaves e/ou *tokens* de sessão por atacantes. Isso torna possível que um usuário malicioso assuma a identidade de outros usuários (temporariamente ou permanentemente).

- *Cross-Site Scripting (XSS):* o XSS é uma vulnerabilidade que permite que *scripts* arbitrários sejam executados no navegador da vítima permitindo possibilidades como o roubo de sessão e redirecionamento a um site malicioso. Ela pode ocorrer de diversas formas, como quando uma aplicação insere uma entrada fornecida em uma nova página sem a devida validação ou escape (muito comum em páginas do tipo fórum de discussão, por exemplo).

- *Broken Access Control:* políticas de restrição de acesso a usuários autenticados não corretamente configuradas possibilitam a atacantes acessar funcionalidades ou dados sem autorização, como contas de usuários, arquivos sensíveis, ou até mesmo modificar direitos de acesso.

- *Security Misconfiguration:* as configurações de segurança de aplicações *WEB*, devem ser propriamente definidas, implementadas e mantidas, pois muitas vezes o uso dos parâmetros default é inseguro e pode fornecer acesso não autorizado a dados ou funcionalidades.

- *Sensitive Data Exposure:* algumas aplicações e APIs armazenam de forma vulnerável informações sensíveis, como dados financeiros, de saúde e pessoais. Atacantes podem explorar tais vulnerabilidades para roubar tais informações e cometer crimes com elas, tais como fraude de cartão e roubo de identidade.

- *Insufficient Attack Protection:* é considerado que em muitas condições, uma API poderia impedir que uma vulnerabilidade fosse explorada se possuissem meios de detectar, responder e até mesmo bloquear os ataques, realizando a análise de atividade suspeita.

- *Cross-Site Request Forgery (CSRF):* é um tipo de ataque que se baseia em fazer a vítima involuntariamente enviar uma requisição HTTP forjada, com os *cookies* de sessão e outras informações de autenticação inclusas a uma aplicação vulnerável, fazendo-a acreditar que seja uma requisição real. Tal ataque geralmente envolve um *site* de controle do atacante que redireciona a requisição ao site alvo.

- *Using Components with Known Vulnerabilities:* em uma aplicação *WEB*, as bibliotecas, *frameworks* e módulos podem possuir vulnerabilidades conhecidas, possibilitando que um atacante detecte a presença de tais componentes expostos e realize a exploração, podendo causar desde danos mínimos até o total comprometimento do servidor e roubo informações sensíveis.

- *Underprotected APIs:* as aplicações complexas modernas envolvem aplicações e APIs que muitas vezes são vulneráveis e necessitam de proteção extra.

Descritas as vulnerabilidades acima, destacamos os itens *Injection* e *Cross-Site Scripting*, que foram escolhidos como os temas dos dois módulos que serão implementados como prova de conceito para o *framework*. Justificamos tais escolhas devido à relevância das vulnerabilidades na classificação do OWASP e por ambos se basearem em vetores de entrada de dados na aplicação sendo passíveis de automatização.

### 2.5.1 INJECTION

As falhas de injeção ocorrem quando atacantes enviam dados não validados para um interpretador. É comum que sistemas *WEB* realizem comandos externos como chamadas de sistema, comandos de *shell* e consultas SQL, assim aplicações vulneráveis podem permitir que um usuário mal intencionado realize ataques baseados em texto explorando a sintaxe do interpretador base para realizar ações não-intencionadas.

Dentro do conjunto das falhas de injeção, a categoria de *SQL Injection* é um tipo particular dessa vulnerabilidade que permite que um atacante influencie nas consultas SQL que são passadas a um banco de dados no *backend* de uma aplicação (CLARKE, 2009). A forma mais comum de exploração deste tipo de vulnerabilidade consiste na inserção direta de código dentro da aplicação, que é então concatenado com uma consulta e executado.

O motivo mais comum deste tipo de vulnerabilidade ocorrer é a aplicação *WEB* não realizar a validação dos dados recebidos do usuário, passando-os diretamente para as consultas que serão executadas no banco de dados no *backend*.

#### 2.5.1.1 EXEMPLO DE ATAQUE DE SQL INJECTION

Considera o cenário onde uma aplicação construa uma consulta SQL da seguinte forma:

```
String query = "SELECT * FROM accounts WHERE
                custID='" + request.getParameter("id") + "'";
```

Se um usuário mal-intencionado modificar o parâmetro "id"em seu navegador para *' or '1'='1* e enviar a requisição ao servidor, a seguinte consulta será construída e transmitida para ser executada pelo interpretador SQL:

```
SELECT * FROM accounts WHERE custID='' or '1'='1'
```

A consulta acima irá retornar todos os registros da tabela de contas, pois a condição '1'='1' é sempre satisfeita para qualquer registro.

Há casos onde não serão mostrados os resultados da consulta maliciosa diretamente na página *WEB* retornada. Entretanto, isso não significa que a aplicação não esteja vulnerável a uma injeção de comandos SQL (CLARKE, 2009). Nestes casos onde há a vulnerabilidade, o ataque se denomina como *Blind SQL Injection*.

As duas imagens a seguir demonstram este conceito com clareza. Na FIG. 2.2, percebe-se que o erro acusado é de senha inválida. Já na FIG. 2.3, vemos a mensagem de usuário e senhas inválidos.

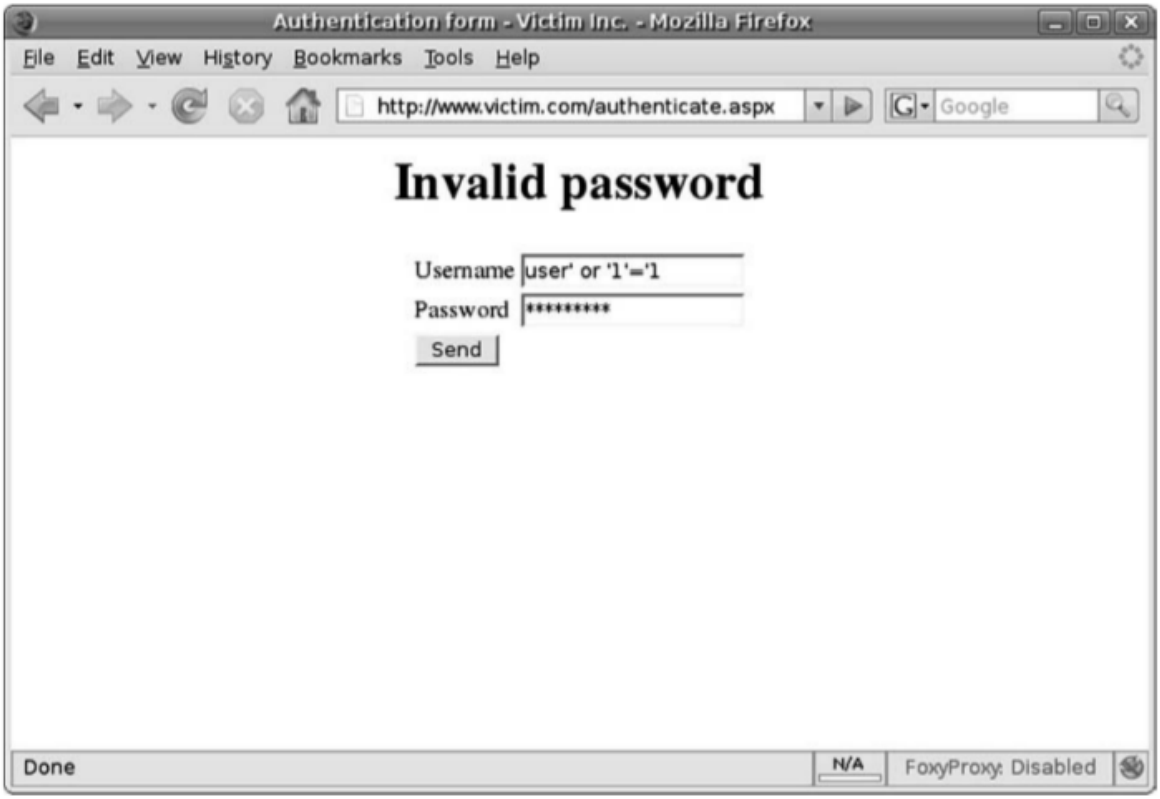


FIG. 2.2: Blind SQL Injection - Condição sempre verdadeira (CLARKE, 2009)

O erro acusado nas duas telas é diferente. Na primeira imagem, a condição *or '1'='1'* concatenada à consulta para o campo *Username* é considerada sempre verdadeira, e nenhum erro é acusado para o campo *Username* - somente o campo *Password* é inválido. Ao mudarmos para uma condição sempre falsa *and '1'='2'*, percebe-se que a mensagem muda, informando que o usuário também é inválido.

Neste caso, é a vulnerabilidade é caracterizada como *Blind SQL Injection*, pois a aplicação retorna respostas diferentes para valores verdadeiros e falsos de uma condição. Este tipo de vulnerabilidade é bem comum, entretanto mais difícil de se detectar, pois deve-se analisar o comportamento da aplicação para cada entrada.

### 2.5.2 CROSS-SITE SCRIPTING (XSS)

A vulnerabilidade a ataques *Cross-Site Scripting* ocorre quando uma aplicação permite carregar uma página *WEB* com código fornecido pelo cliente, sem a devida precaução quanto a impedir que tal código seja executado. Assim, o navegador, ao não saber se pode ou não confiar no código fornecido irá executá-lo, alcançando o objetivo do atacante.

Segundo OWASP (2016) existem duas categorias primárias de XSS:


FIG. 2.3: Blind SQL Injection - Condição sempre falsa (CLARKE, 2009)

- *Stored:* nessa categoria, o código malicioso fica permanentemente armazenado no servidor alvo. Geralmente envolve fóruns, comentários ou outras aplicações que necessitem armazenar mensagens do usuário em bancos de dados e exibi-las na aplicação.

- *Reflected:* esse tipo de ataque envolve refletir o código por meio de respostas como mensagens de erro ou resultados de pesquisas, que incluam parte ou toda a entrada do usuário. Assim, o atacante pode utilizar um link malicioso para fazer a vítima submeter tal código ao site vulnerável que retorna o ataque ao usuário.

Os ataques XSS podem causar desde incômodos na utilização do site pelo usuário até comprometimento total de sua conta. Um dos principais objetivos dos atacantes é roubar os *cookies* de sessão da vítima, o que possibilita o usuário malicioso a se passar pelo usuário alvo. Conforme explicitado na FIG 2.4, o atacante injeta um código malicioso no site e aguarda que a vítima interaja com o mesmo, ao requisitar o site a vítima fornece seus dados ao atacante sem saber.

O código a seguir exemplifica o uso de um *script* XSS inserido em uma *tag* que tenta


FIG. 2.4: Esquema demonstrativo de ataque XSS baseado em (INCAPSULA, 2017)

carregar uma imagem inexistente.

```
<img src="http://url.to.file.which/not.exist" onerror=alert(document.cookie);>
```

Tal código tentaria carregar uma imagem que não existe, de forma que, ao verificar o erro, os *cookies* armazenados no navegador seriam mostrados em uma tela de alerta.

## 2.6 *FRAMEWORKS*

Segundo Junior (2006), uma das principais características dos *frameworks* é sua capacidade de reuso, uma vez que permite a criação de uma família de produtos a partir de uma única estrutura. Além disso, *frameworks* permitem estender a aplicação, de forma a garantir que funcionalidades extras possam ser adicionadas conforme necessárias, definindo uma arquitetura comum aos subsistemas e construtores básicos para sua criação.

Ao se analisar a estrutura geral de um *framework*, percebe-se a subdivisão em parte fixa (*frozen spot*) e parte variável (*hot spot*). Segundo Markiewicz e Lucena (2001), a parte variável corresponde aos pontos de flexibilização do *framework* nos quais o usuário pode inserir o próprio código e gerar uma nova instância específica à sua necessidade, enquanto a parte fixa contém a parte de código imutável do *framework*, ou seja, a parte que desempenha as funções comuns a todas as instâncias implementadas e que carrega as partes variáveis desenvolvidas. Assim, um *framework* pode ter a seguinte organização:

- Núcleo: consiste da parte central responsável por carregar os diferentes módulos e executar suas funções (parte fixa);
- Arquivos de Configuração: são responsáveis por inicializar as configurações básicas a serem utilizadas durante a execução do *Framework*; e
- Módulos: permitem estender a aplicação adicionando funcionalidades ao *Framework* (parte variável).

Quanto ao tipo, os *frameworks* são geralmente divididos em:

- *Framework* de Aplicações Orientado a Objetos: geram famílias de aplicações orientadas a objetos. São estendidas a partir de interfaces ou classes abstratas.
- *Framework* de Componentes: é uma entidade de software que define determinado modelo, permitindo conectar componentes que se baseiam nele. Além disso, também é responsável por gerenciar a interação entre as instâncias desses componentes.

Os benefícios da utilização de *frameworks* advém de suas características principais como modularidade, reusabilidade, extensibilidade e inversão de controle, uma vez que assumem o controle da execução, invocando métodos da aplicação quando necessário Junior (2006).

O esforço para entender e manter a aplicação é reduzido, uma vez que *frameworks* encapsulam detalhes de implementação voláteis, definindo pontos de extensão e interfaces; de forma a garantir a localização dos pontos de mudança e implementação. Além disso, o reuso de módulos desenvolvidos, possibilitando um aumento de produtividade no desenvolvimento de novas aplicações ou na adição de novas funcionalidades; a possibilidade de estender a aplicação para suprir necessidades do desenvolvedor e do usuário; tornam optar por utilizar um *framework* vantajoso.

Entretanto, para garantir as vantagens de implementação do *framework*, deve-se atentar para a resolução de certos desafios, como curva de aprendizado, uma vez que a utilização do *framework* e sua extensão devam ser vantajosas ao usuário; manutenção, uma vez que a adição de novos módulos incorre em adição de novos vetores de erro; e validação, uma vez que por ser uma aplicação em desenvolvimento, só se pode validar a instância atual do *framework*.

Para resolver tais problemas, define-se uma interface de simples entendimento e desenvolvimento, garantindo uma curva de aprendizado baixa na extensão da aplicação, além

do fato da alta modularidade do *framework* reduzir as implicações de erros no núcleo da aplicação, só sendo necessária a remoção de erros no módulo a ser adicionado.

# 3 O *FRAMEWORK* PROPOSTO

## 3.1 NÚCLEO

O *framework* desenvolvido é composto pelo núcleo (parte fixa), com seu arquivo de configuração; e pelos módulos implementados (partes variáveis) e seus respectivos arquivos de configuração e de resultados.

O Núcleo do *Framework* foi modelado de forma que seu fluxo permita as seguintes ações:

- Recarregar Configurações: permite recarregar as configurações do núcleo durante a execução do programa;
- Execução de Módulo: são listados os módulos disponíveis e o usuário pode escolher um desses módulos para ser executado; e
- Geração de Relatório: o usuário escolhe quais módulos serão executados e, após sua execução, é gerado um relatório com os resultados obtidos.

O arquivo de configuração do núcleo apresenta os módulos que estarão disponíveis ao usuário, definindo o nome que será apresentado, o nome do arquivo principal do módulo e o diretório no qual se encontra, conforme apresentado na FIG. 3.1 abaixo:

```
"nome_do_módulo1":{"name":"nome_do_arquivo_principal1",
                   "directory":"diretório_do_módulo1"},
"nome_do_módulo2":{"name":"nome_do_arquivo_principal2",
                   "directory":"diretório_do_módulo2"},
...
```

FIG. 3.1: Modelo do arquivo de configuração

## 3.2 MÓDULOS

Para que o módulo possa ser corretamente carregado, é necessário que esse seja estruturado de acordo com o interfaceamento proposto que prevê duas funções básicas para o módulo, *BEGIN* e *RUN*, como explicado na FIG. 3.2 abaixo:

```
def BEGIN():
    #função que tem por finalidade informar os parâmetros necessários à
    #execução do módulo, de forma a permitir que tais parâmetros sejam
    #solicitados ao usuário. Deve retornar uma lista com tais parâmetros.

def RUN(file_name):
    #função que tem por finalidade executar a funcionalidade principal
    #do módulo, tendo como parâmetro o nome do arquivo que contém os
    #valores a serem utilizados para sua execução. Deve retornar o
    #resultado obtido de forma a apresentá-lo ao usuário
```

FIG. 3.2: Modelo de interface com o Núcleo

Assim, a execução do módulo pelo núcleo ocorre da seguinte forma:

a) O núcleo realiza a chamada o *BEGIN* do módulo a ser executado;

b) O usuário preenche os parâmetros pedidos que são salvos em um arquivo;

c) O núcleo realiza a chamada *RUN* do módulo passando como parâmetro o nome do arquivo com os parâmetros necessários; e

d) O módulo é executado e os resultados obtidos são salvos em outro arquivo.

A FIG. 3.3 apresenta o fluxograma de execução de um módulo genérico:

### 3.2.1 MÓDULO I: TESTE DE INJEÇÃO SQL

Conforme já explicado, SQL é a linguagem utilizada para se comunicar com os servidores de banco de dados relacionais. Este módulo consiste na busca de atributos que sejam construídos dinamicamente na aplicação *WEB*. Banco de dados interpretam alguns caracteres especiais como limite entre código e dado, ao inserir este, espera-se um erro. Nessa premissa, implementou-se um módulo que, munido de um *payload* de diferentes caracteres especiais, pois diferentes tipos de banco de dados possuem diferente tipos de caracteres de fechamento, fosse possível verificar a existência da vulnerabilidade buscando abranger uma gama de possibilidades.

#### 3.2.1.1 METODOLOGIA USADA

Conforme proposto por Clarke (2009) existem três aspectos chaves para se encontrar uma vulnerabilidade SQL *injection*, a saber:

- Identificar os pontos de entrada de dados que serão enviados para o servidos para realizar algum tipo de consulta;
- Modificar os dados de entrada incluindo valores de forma a tentar manipular a *query* SQL que será enviada ao banco de dados da aplicação; e
- Detectar anomalias que porventura possam retornar do servidor.

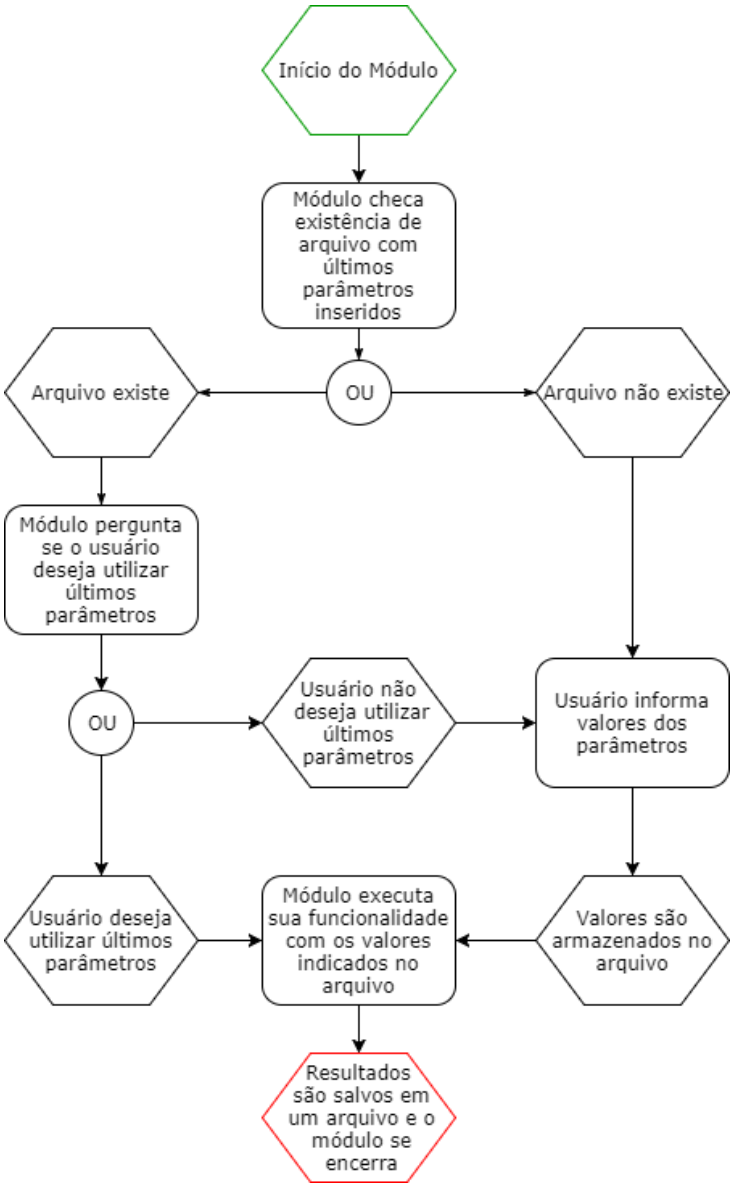


FIG. 3.3: Fluxograma de execução de módulo geral

Inicialmente, é necessário realizar uma requisição da página a ser testada. Para tal tarefa utilizou-se a biblioteca *open source Requests* de *Python* para a confecção, manipulação e envio de requisições e respostas *HTTP* de forma rápida e intuitiva. Por exemplo,

com apenas as duas linhas de código abaixo, é possível realizar uma requisição *HTTP* de uma página pelo método *GET*:

```
import requests
r = requests.get('https://google.com')
```

Após receber a resposta da requisição não maliciosa enviada, é necessário identificar as *tags HTML* da página, decidir quais formulários devem ser preenchidos com os *payloads* e enviar a requisição maliciosa produzida.

Respostas do servidor com códigos de erro *HTTP* ou que incluíssem erros de banco de dados, normalmente, sinalizam a existência de vulnerabilidade para uma injeção SQL. Entretanto, nem todas as requisições retornam erros e isso não é condição suficiente de que o serviço esteja totalmente protegido, conforme descrito no capítulo 2.

#### 3.2.1.2 FLUXO DA APLICAÇÃO

Na FIG 3.4 abaixo é apresentado o fluxo de execução da funcionalidade do módulo *SQL*:

### 3.2.2 MÓDULO II: TESTE DE XSS

Uma vulnerabilidade *XSS* é detectada quando um usuário malicioso consegue inserir código (*Script*) que será executado pelo navegador como se fosse originário da página alvo. Assim, para auxiliar na identificação de tal vulnerabilidade, foram utilizados *payloads* que apresentam código executável pelo navegador, como um *Alert* de *Javascript*, e tentou-se verificar sua execução na resposta do servidor.

#### 3.2.2.1 METODOLOGIA USADA

Por ser um tipo de *Injection*, a metodologia utilizada para identificar vulnerabilidades *XSS* pode ser adaptada da utilizada pelo módulo *SQL* da seguinte forma:

- Identificar os pontos de entrada de dados que serão enviados para o servidos para realizar algum tipo de consulta;
- Enviar requisição preenchendo os pontos de entrada com valores que contenham código que possa ser executado pelo navegador; e
- Detectar, na resposta, se os códigos enviados foram efetivamente executados pelo navegador.

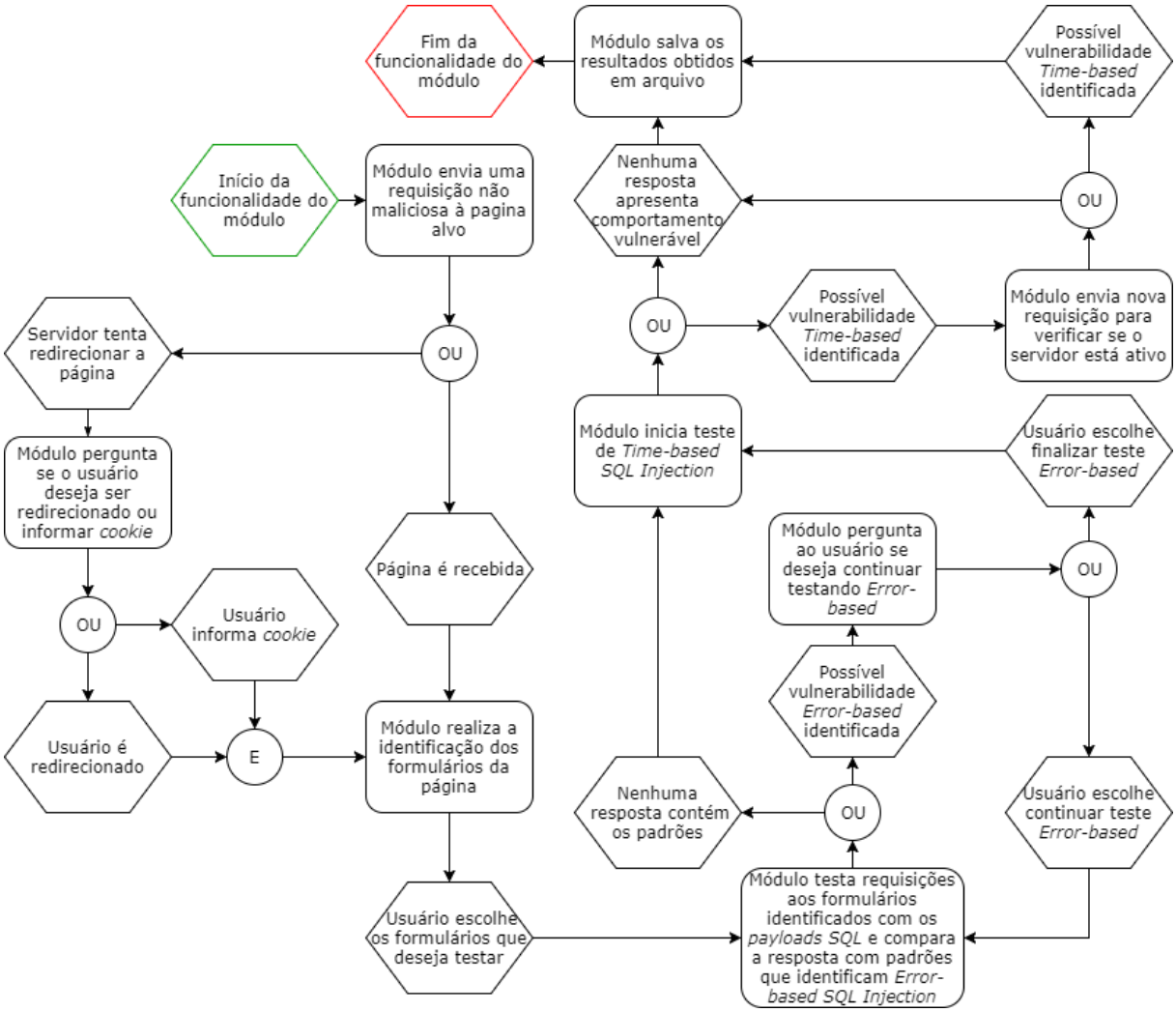


FIG. 3.4: Fluxograma de execução de módulo SQL

Inicialmente, assim como o módulo *SQL*, realiza-se uma requisição da página a ser testada e a identificação de seus formulários. Após isso, requisições são realizadas preenchendo tais formulários com os *payloads XSS* e armazenando sua resposta em um arquivo temporário. Esse arquivo é então carregado pelo *Selenium WebDriver*.

O *Selenium* é um conjunto de ferramentas de *software open source* utilizadas para o controle automatizado de navegadores. Inicialmente foi desenvolvido para o uso em projetos *Java*, mas atualmente existem amarrações para seu uso em diversas linguagens, incluíndo *Python*. Na sua arquitetura, é utilizada a *API* WebDriver responsável por fornecer uma interface única com cada navegador, possuindo suporte para os principais navegadores utilizados atualmente no mercado como *Google Chrome*, *Mozilla Firefox*, *Opera*, *Safari* e *Internet Explorer*. Durante os testes, utilizou-se o *ChromeDriver* para a integração com o navegador *Google Chrome*, porém com poucas modificações é possível suportar o uso de outros navegadores.

Com o exemplo de código abaixo, é possível abrir uma janela do navegador *Chrome* e realizar uma requisição *GET* de uma página de forma automatizada, sendo necessário para isso o arquivo *ChomeDriver*.

```
from selenium import webdriver
driver = webdriver.Chrome('/path/to/chromedriver')
driver.get('https://google.com')
```

O *Selenium* é utilizado para verificar se o navegador executou o código inserido na requisição. Em caso positivo, uma possível vulnerabilidade *Reflected XSS* é detectada. Após isso, ocorre um envio de requisição não malicioso, para tentar identificar se o código inserido persistiu, o que indica uma possível vulnerabilidade *Stored XSS*.

#### 3.2.2.2 FLUXO DA APLICAÇÃO

A FIG 3.5 mostra o fluxograma de execução do módulo *XSS*.

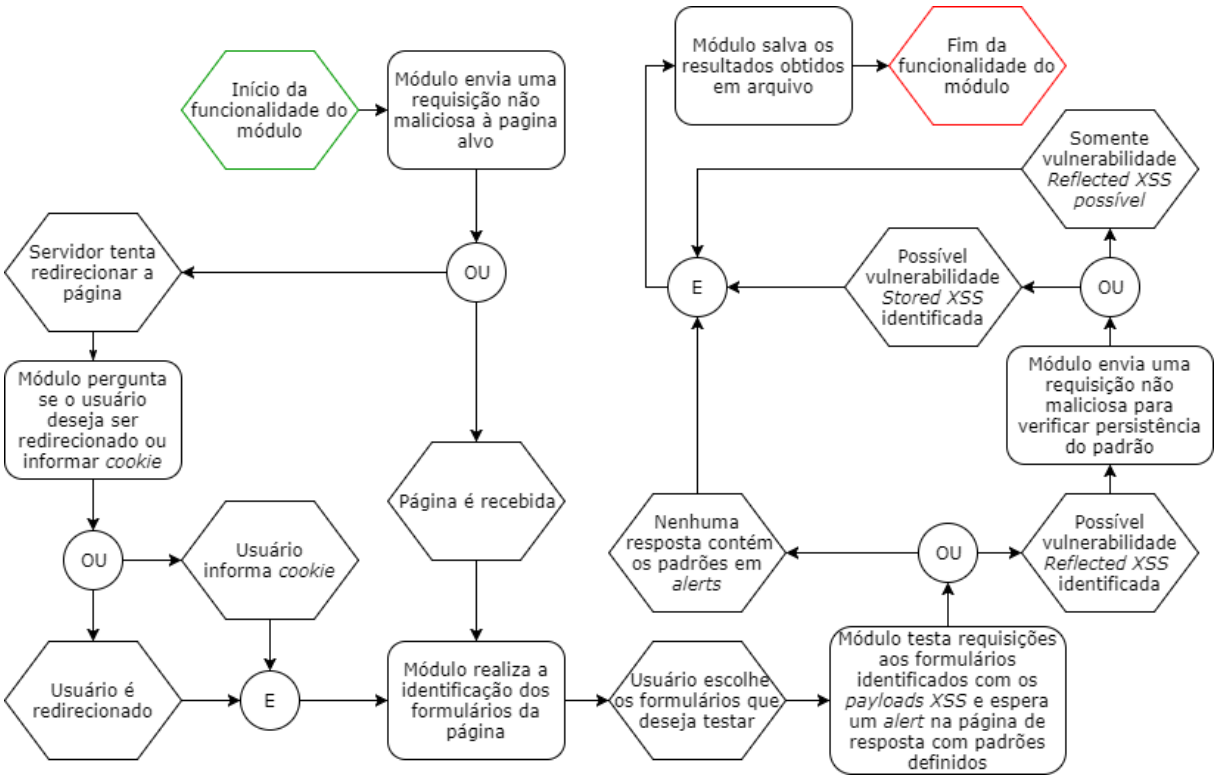


FIG. 3.5: Fluxograma de execução de módulo XSS

## 3.3 ESTRUTURA DE CÓDIGO

A FIG 3.6 demonstra a estrutura de código do *Framework* e dos módulos implementados:

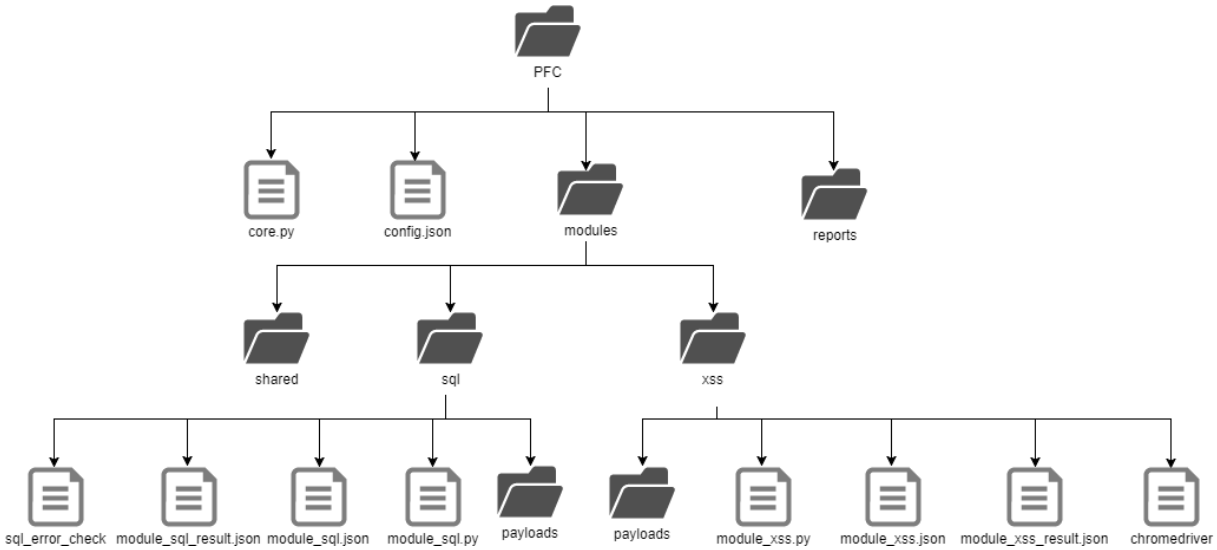


FIG. 3.6: Estrutura de Código do *Framework*

### 3.3.1 FRAMEWORK

O código que constitui o *framework* proposto está dividido em:

- Arquivo *core.py*: representa o núcleo da aplicação, no qual estão definidas as funcionalidades básicas do *framework*, como execução de módulos e geração de relatório;
- Arquivo *config.json*: representa as configurações do núcleo do *framework*;
- Diretório *modules*: diretório base onde os módulos adicionais devem ser implementados, além do diretório *shared* que contém código comum e reutilizável por módulos diferentes;
- Diretório *reports*: diretório criado pela aplicação, no qual ficam armazenados os relatórios gerados pelo *framework*.

### 3.3.2 MÓDULO SQL

O código que constitui o módulo *SQL* está dividido em:

- Arquivo *module_ sql.py*: representa o arquivo principal do módulo *sql*;
- Arquivo *module_ sql.json*: representa o arquivo com os valores dos parâmetros necessários à execução do módulo;
- Arquivo *module_ sql_ result.json*: representa o arquivo com os resultados obtidos na última execução do módulo;

- Arquivo *sql_error_check*: representa o arquivo com os padrões de resultados possíveis caso a aplicação testada apresente vulnerabilidade *sql*;
- Diretório *payloads*: diretório que armazena os *payloads* a serem testados.

### 3.3.3 MÓDULO XSS

O código que constitui o módulo *XSS* está dividido em:

- Arquivo *module_xss.py*: representa o arquivo principal do módulo *xss*;
- Arquivo *module_xss.json*: representa o arquivo com os valores dos parâmetros necessários à execução do módulo;
- Arquivo *module_xss_result.json*: representa o arquivo com os resultados obtidos na última execução do módulo;
- Arquivo *chromedriver*: representa o arquivo com os *drivers* necessários à execução do *Selenium Webdriver* utilizando como base o navegador *Google Chrome*;
- Diretório *payloads*: diretório que armazena os *payloads* a serem testados.

# 4 UTILIZAÇÃO E TESTES DO *FRAMEWORK*

No presente capítulo, realizam-se demonstrações do funcionamento do *framework* e dos módulos implementados em aplicações contendo vetores de entrada para possíveis requisições maliciosas.

## 4.1 AMBIENTE DE TESTE

Durante o desenvolvimento foi necessário criar um ambiente de testes para que fosse possível verificar a efetividade do módulo. Testes deste tipo normalmente realizam uma alta quantidade de requisições ao servidor, além da possibilidade de expor vulnerabilidades, portanto pode ser considerado crime quando da sua realização sem o consentimento.

Foram realizadas pesquisas para se encontrar uma aplicação que fosse vulnerável e que fosse possível executar testes educativos a fim de aperfeiçoar o *framework*. Com estes aspectos em mente, foi utilizada a plataforma *bWAPP* (ITSECGAMES, 2017) por ser uma aplicação *WEB* de fonte aberta deliberadamente insegura que abrange todos os principais erros conhecidos, incluindo os riscos citados pelo projeto *OWASP*. A FIG 4.1 mostra a página inicial da aplicação com o menu para a seleção de vulnerabilidade a ser explorada.

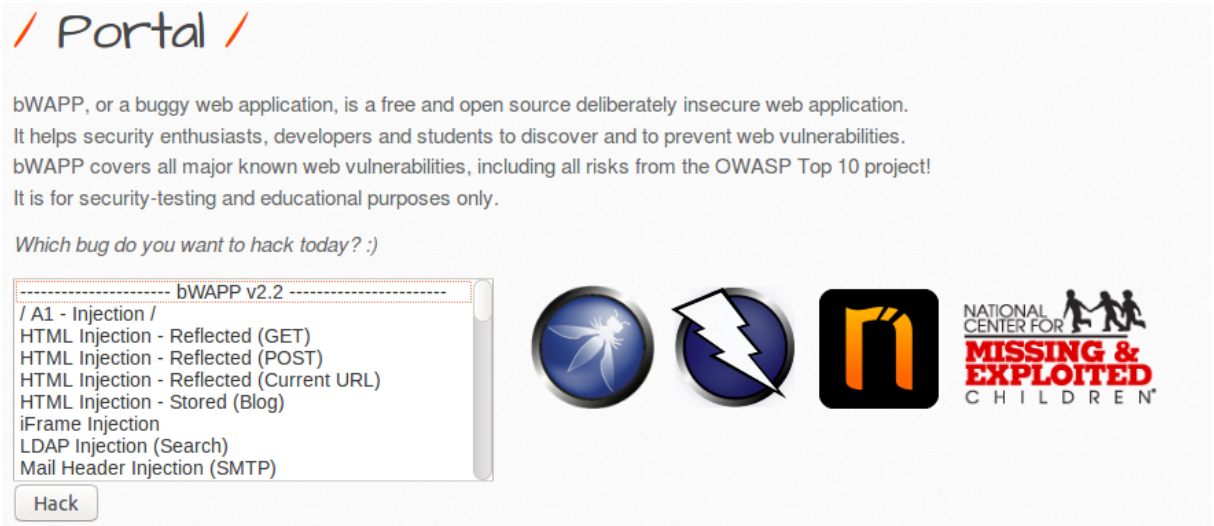


FIG. 4.1: Portal da aplicação *bWAPP* com o menu da seleção de vulnerabilidades

## 4.2 FUNCIONAMENTO DA APLICAÇÃO

Esta seção busca demonstrar o fluxo de funcionamento da aplicação através da ilustração do uso em exemplos práticos. Ao inicializar o *framework*, é exibido ao usuário o menu de ações, conforme a FIG 4.2.

```
Choose an action:
 -> reload_configuration
 -> execute_module
 -> generate_report
 -> exit
Action:
```

FIG. 4.2: Menu de opções da aplicação

Escolhendo-se a opção para executar um módulo, será exibida a lista de opções, conforme FIG 4.3.

```
Choose an action:
 -> reload_configuration
 -> execute_module
 -> generate_report
 -> exit
Action: execute_module

Choose one of the modules:
 -> test
 -> sql_injection
 -> xss_check
 -> Return to Main Menu
Module:
```

FIG. 4.3: Lista de possíveis módulos para execução

### 4.2.1 MÓDULO SQL

Será demonstrado como o *framework* proposto realiza a inspeção da aplicação usando o módulo *SQL*. Inicialmente, é fornecido como parâmetro a *URL* que será verificada. Em seguida, é feita uma requisição para obtenção do documento *HTML* referente ao endereço fornecido. Para este teste, a vulnerabilidade explorada é referente à página *sqli_ 1.php* da aplicação *bWAPP*.

Em alguns casos, há o redirecionamento da requisição para outra *URL*. Isso é comum nas situações onde há o controle de acesso por um *login* prévio para acesso à funcionalidade desejada, e nestes casos é possível fornecer ao módulo um *cookie* de sessão para permitir o acesso ou simplesmente seguir o redirecionamento normalmente. A FIG 4.4 ilustra a interface com usuário para a entrada de *URL* e questionamento sobre a opção desejada diante de um redirecionamento.

```
Choose one of the modules:
 -> xss_check
 -> sql_injection
 -> test
 -> Return to Main Menu
Module: sql_injection
Use the last used parameters? [yes/no] no
URL: http://169.254.2.99/bWAPP/sqli_1.php
[!] It seems the URL is being redirected ((<Response [302]>,))
Redirection URL: http://169.254.2.99/bWAPP/login.php
If a login is required, then is recommended to inform a cookie for the session
Do you want to follow the redirect URL or inform a cookie? (f/c)
```

FIG. 4.4: Entrada de *URL* e alerta sobre redirecionamento

Como o controle de acesso via *login* é feito no *bWAPP*, nestes exemplos sempre será fornecido um *cookie* de sessão para efetuar o acesso à página. O módulo então realiza o *parsing* do documento *HTML* recebido e exibe os formulários detectados com um identificador associado, conforme ilustrado na FIG 4.5.

```
If a login is required, then is recommended to inform a cookie for the session
Do you want to follow the redirect URL or inform a cookie? (f/c)
c
Inform the cookie
security_level=0; PHPSESSID=c75e7834ea92c7b38cd2112e5e50278b
[+] Sending request for the URL web page
[+] Response received. Parsing HTML Document in order to detect FORMS
[+] Parsing completed
[+] The following FORMS were found in the HTML document:
 ID: #0 : GET form with the following fields:
{'action': 'search', 'title': ''}
 ID: #1 : POST form with the following fields:
{'form_security_level': 'submit', 'security_level': ''}
 ID: #2 : POST form with the following fields:
{'form_bug': 'submit', 'bug': ''}
Which of the forms will be tested? (space-separated ID list) or (all)
```

FIG. 4.5: Exibição dos formulários detectados no documento *HTML* recebido

Ao usuário, é oferecida a escolha de quais formulários serão testados, baseado no seu

identificador. Para cada formulário selecionado, é realizado o envio de uma requisição contendo os devidos valores maliciosos em todos campos do formulário, contendo os *payloads* da base de dados do módulo. Estre procedimento ocorre até que haja detecção de padrões de erros conhecidos, onde então o módulo informará ao usuário sobre a possível descoberta de uma vulnerabilidade e fornece a opção de continuar o teste para outros valores. A FIG 4.6 e a FIG 4.7 mostram o envio das requisições e as mensagens suspeitas encontradas para cada uma.

```
Which of the forms will be tested? (space-separated ID list) or (all)
0
[+] Sending non malicious request for FORM #0
[+] Testing form 0 for error-based SQL injection detection
[+] Sending request for FORM #0. PAYLOAD: '
[-] Detected "error in your sql syntax" in the response, possible sql error-base
d vulnerability
[-] Detected "you have an error in your sql syntax" in the response, possible sq
l error-based vulnerability
[-] Detected "error" in the response, possible sql error-based vulnerability
[!] Possible error-based vulnerability detected for form 0 using payload '
Do you want to continue testing (y/n)?
```

FIG. 4.6: Envio de requisição maliciosa para um formulário escolhido e detecção de possíveis vulnerabilidades

```
If a login is required, then is recommended to inform a cookie for the session
Do you want to follow the redirect URL or inform a cookie? (f/c)
c
Inform the cookie
security_level=0; PHPSESSID=c75e7834ea92c7b38cd2112e5e50278b
[+] Sending request for the URL web page
[+] Response received. Parsing HTML Document in order to detect FORMS
[+] Parsing completed
[+] The following FORMS were found in the HTML document:
 ID: #0 : GET form with the following fields:
{'action': 'search', 'title': ''}
 ID: #1 : POST form with the following fields:
{'form_security_level': 'submit', 'security_level': ''}
 ID: #2 : POST form with the following fields:
{'form_bug': 'submit', 'bug': ''}
Which of the forms will be tested? (space-separated ID list) or (all)
```

FIG. 4.7: Continuação dos testes para outros valores maliciosos

Finalizados os testes para *error-based SQL injection*, caso sejam detectados erros, é facultativa a escolha para adotar a verificação de *time-based SQL injection*, conforme FIG 4.8.

Testou-se também o módulo na página *sqli_15.php* da aplicação *bWAPP*, contendo uma vulnerabilidade do tipo *blind SQL Injection*. Os testes para a estratégia *time-based*

```
[-] No errors detected
[+] Sending request for FORM #0. PAYLOAD: %
[-] No errors detected
[+] Sending request for FORM #0. PAYLOAD: *
[-] No errors detected
[+] Sending request for FORM #0. PAYLOAD: ';
[-] Detected "error in your sql syntax" in the response, possible sql error-base
d vulnerability
[-] Detected "you have an error in your sql syntax" in the response, possible sq
l error-based vulnerability
[-] Detected "error" in the response, possible sql error-based vulnerability
[!] Possible error-based vulnerability detected for form 0 using payload ';
Do you want to continue testing (y/n)?
n
[!] For FORM #0 there was an possible error-based SQL Injection vulnerability de
tected.
Do you want to continue and test for time-based? (y/n)
n
```

FIG. 4.8: Testes finalizados para a estratégia *error-based*

são realizados de forma análoga, entretanto deve-se considerar que um *timeout* não necessariamente indica que a aplicação é vulnerável ao *payload*, pois o mesmo pode ser consequência de outros fatores, como por exemplo a queda do servidor por motivos diversos. Por este motivo, é necessária uma verificação mais rigorosa de forma a evitar a detecção de falsos-positivos. Assim que detectada uma demora inesperada para resposta à requisição testada, inspeciona-se se o servidor continua operando. No caso positivo, é repetido o envio da entrada maliciosa, e ocorrendo novamente o atraso na resposta, o teste é considerado como contendo uma possível vulnerabilidade. A FIG 4.9 e a FIG 4.10 ilustram como o módulo interage com o usuário durante a verificação.

```
[+] Testing form 0 for time-based SQL injection detection
[+] Sending request for FORM #0. PAYLOAD: ' waitfor delay '0:0:15' --
[+] Sending request for FORM #0. PAYLOAD: ');waitfor delay '0:0:15' --
[+] Sending request for FORM #0. PAYLOAD: 1,'0');waitfor delay '0:0:15' --
[+] Sending request for FORM #0. PAYLOAD: ' waitfor time '0:0:15' --
[+] Sending request for FORM #0. PAYLOAD: ');waitfor time '0:0:15' --
[+] Sending request for FORM #0. PAYLOAD: 1,'0');waitfor time '0:0:15' --
[+] Sending request for FORM #0. PAYLOAD: ' sleep(15) --
[+] Sending request for FORM #0. PAYLOAD: ' or sleep(3)#
[+] Possible time-based SQL injection vulnerability detected. Increasing expecte
d-timeout to reduce the chance of false-positive
```

FIG. 4.9: Testes para *time-based blind SQL Injection*

```
[+] Possible time-based SQL injection vulnerability detected. Increasing expecte
d-timeout to reduce the chance of false-positive
[+] Sending a new request to check if the server is OK.
[+] Server is OK. Trying a new request to check
[+] Timeout reached, possible vulnerability detected. Yay!
[!] Possible time-based vulnerability detected for form 0 using payload ' or sle
ep(1)#
Do you want to continue testing (y/n)?
y
[+] Sending request for FORM #0. PAYLOAD: ' and sleep(1)#
[+] Sending request for FORM #0. PAYLOAD: ') sleep(15) --
[+] Sending request for FORM #0. PAYLOAD: 1,'0') sleep(15) --
[+] Sending request for FORM #0. PAYLOAD: ' pg_sleep(15) --
[+] Sending request for FORM #0. PAYLOAD: ') pg_sleep(15) --
[+] Sending request for FORM #0. PAYLOAD: 1,'0') pg_sleep(15) --
```

FIG. 4.10: Reenvio das requisições maliciosas para filtro de falsos-positivos

### 4.2.2 MÓDULO XSS

Escolhido o módulo *XSS* no menu inicial, a interface com usuário solicitando o parâmetro *URL*, o alerta de redirecionamento e a seleção dos formulários a serem verificados é semelhante à do módulo *SQL*. Após a seleção dos formulários verificados, abre-se a janela de navegador via *Selenium* que será usada para verificar se os *scripts* injetados nas requisições foram executados ou não.

A aplicação módulo na página *xss_get.php* do *bWAPP* para a detecção de XSS é ilustrado na FIG 4.11. Para cada *payload* na base de dados, é realizada a requisição maliciosa e armazenado o documento *HTML* da resposta em um arquivo lido pelo navegador. Caso o método de detecção de vulnerabilidade verifique que o código *javascript* injetado foi executado, conclui-se a presença de uma possível vulnerabilidade *reflected XSS*. É feita uma nova requisição não maliciosa de forma a verificar se houve a persistência do *script* inserido na página, caracterizando a vulnerabilidade *stored XSS*. No caso da FIG 4.11, é detectado somente a variante *reflected XSS*.

Já na FIG 4.12 o módulo é testado na página *xss_stored_1.php*, demonstrando como a funcionalidade detecta a persistência do código injetado.

```
Which of the forms will be tested? (space-separated ID list) or (all)
0
[+] Opening test browser window
[+] Sending non malicious request for FORM #0
[+] Testing form 0 for reflected XSS detection
[+] Sending request for FORM #0. PAYLOAD: <script>alert("XSS")</script>
[!] Possible reflected xss vulnerability detected for form 0 using payload <scrip
t>alert("XSS")</script>
[!] Sending non-malicious request to check for stored xss as well
[+] Sending non-malicious request for FORM #0.
[-] No errors detected
[!] No stored xss found
```

FIG. 4.11: Módulo *XSS* aplicado em uma página com uma vulnerabilidade *reflected*

```
Which of the forms will be tested? (space-separated ID list) or (all)
0
[+] Opening test browser window
[+] Sending non malicious request for FORM #0
[+] Testing form 0 for reflected XSS detection
[+] Sending request for FORM #0. PAYLOAD: <script>alert("XSS")</script>
[!] Possible reflected xss vulnerability detected for form 0 using payload <scri
pt>alert("XSS")</script>
[!] Sending non-malicious request to check for stored xss as well
[+] Sending non-malicious request for FORM #0.
[!] Possible stored xss vulnerability detected for form 0 using payload <script>
alert("XSS")</script>
```

FIG. 4.12: Módulo *XSS* aplicado em uma página com uma vulnerabilidade *stored*

# 5 CONCLUSÃO

Este trabalho teve como objetivo desenvolver um *framework* que automatize tarefas típicas de testes de invasão e que sirva como interface única para todas as fases do teste. Após o estudo conceitual sobre segurança da informação e testes de invasão, foi possível entender as possibilidades e limitações de escopo do projeto. Sendo assim, decididiu-se direcionar o foco do projeto a aplicações Web, devido a sua importância e disseminação tanto no meio civil quanto no âmbito do exército.

Considerou-se no desenvolvimento do *framework* a necessidade de escalabilidade, ficando como sugestão de projetos futuros a adição de novos módulos que complementem as funcionalidades das fases subsequentes do teste. Nos módulos implementados, demonstrou-se o acoplamento de ferramentas funcionais ao *framework* desenvolvido. Por fim, foi realizado uma prova de conceito, com o objetivo de testar e medir a eficiência das funcionalidades implementadas. Como é vedado testes em aplicações reais sem consentimento, criou-se um ambiente controlado no qual foi possível executar os módulos, realizar os testes e gerar os relatórios. Outro possível futuro projeto seria de alertas automáticos ao administrador apenas quando uma possível vulnerabilidade for encontrada.

O *framework* se mostrou bastante promissor e de fácil aprendizagem pois possui uma interface amigável. Devido as restrições de tempo e grande complexidade, apenas foi possível criar dois módulos.

# 6 REFERÊNCIAS BIBLIOGRÁFICAS


ABNT. **Tecnologia da informação - Técnicas de segurança - Código de prática para a gestão da segurança da informação**. Rio de Janeiro: ABNT NBR ISO/IEC 27017, 2016.

BRASIL. **Decreto nº 6.703, de 18 de dezembro de 2008. Aprova a Estratégia Nacional de Defesa.**. Diário Oficial: Brasília, DF, 18 dez. 2008.

CERT. **Cartilha de Segurança para Internet**. 2. ed. São Paulo: Comitê Gestor da Internet no Brasil, 2012. 140 p.

CERT. Estatísticas de Incidentes de Segurança Reportados ao CERT.br. Disponível em: <https://www.cert.br/stats/incidentes/>. Acesso em: 09 de maio de 2017.

CLARKE, J. **SQL Injection Attacks and Defense**. [S.l.]: Syngress, 2009. 473 p.

FEOIS. **Penetration Testing Model**. [S.l.]: Federal Office For Information Security (BSI, 2004. 111 p.

GLOBO. Hackers invadem servidores do Exército e vazam CPFs de militares - 2015. Disponível em: <http://g1.globo.com/tecnologia/noticia/2015/11/hackers-invadem-servidores-do-exercito-e-vazam-cpfs-de-militares.html>. Acesso em: 22 abril de 2017.

HERZOG, P. **OSSTMM 3.0:Open-Source Security Testing Methodology Manual**. [S.l.]: ISECOM, 2010. 213 p.

INCAPSULA. Cross-site Scripting (XSS) Attacks. Disponível em: <https://www.incapsula.com/web-application-security/cross-site-scripting-xss-attacks.html>. Acesso em: 10 maio de 2017.

ITSECGAMES. Cross-site Scripting (XSS) Attacks. Disponível em: <http://www.itsecgames.com/>. Acesso em: 20 julho de 2017.

ITU. ICT Facts and Figures in 2015. Disponível em: <http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf>. Acesso em: 09 de maio de 2017.

JUNIOR, C. G. B. **Agregando Frameworks de Infra-Estrutura em uma Arquitetura Baseada em Componentes: um Estudo de Caso no Ambiente AulaNet**. 2006. 210 f. Dissertação (Programa de Pós-Graduação em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2006.

MARCUS EDUARDO MARKIEWICZ AND CARLOS J.P. LUCENA. Object Oriented Framework Development. Disponível em: <http://www.cos.ufrj.br/ toacy/material/Object-OrientedAcesso em: 19 setembro de 2017.

MCCLURE, S.; SCAMBRAY, J. ; KURTZ, G. **Hackers Expostos**. 7. ed. Porto Alegre: Bookman, 2014. 738 p.

OWASP. Cross-site Scripting (XSS). Disponível em: <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>. Acesso em: 09 maio de 2017.

OWASP. OWASP Top 10 Application Security Risks - 2017. Disponível em: <https://www.owasp.org/index.php/Top_10_2017-Top_10>. Acesso em: 09 maio de 2017.

WEIDMAN, G. **Testes de Invasão**. 1. ed. São Paulo: Novatec, 2014. 576 p.