Gabriel Quadros Silva

Geração Automática de Exploits para Vulnerabilidades de Heap Overflow

Vitória da Conquista

Janeiro de 2011

Gabriel Quadros Silva

Geração Automática de Exploits para Vulnerabilidades de Heap Overflow

Monografia apresentada para conclusão do curso de Ciência da Computação na Universidade Estadual do Sudoeste da Bahia, no período letivo 2010.2.

Orientador:
Prof. Dr. Marlos André Marques Simões de Oliveira

UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
DEPARTAMENTO DE CIÊNCIAS EXATAS

Vitória da Conquista

Janeiro de 2011

Monografia sob o título *"Geração Automática de Exploits para Vulnerabilidades de Heap Overflow"*, defendida por Gabriel Quadros Silva e aprovada em 20 de janeiro de 2010, em Vitória da Conquista, Bahia, pela banca examinadora constituída pelos professores:

Prof. Dr. Marlos André Marques Simões de Oliveira
Orientador

Prof.ª Me. Lidiana de França Martins
Instituto Federal de Educação, Ciência e Tecnologia da Bahia

Resumo

A pesquisa de vulnerabilidades é uma das atividades-chave da Segurança Computacional, pois é por meio dela que as vulnerabilidades que são usadas em grande parte dos ataques à redes e sistemas computacionais são descobertas e/ou corrigidas. Entretanto, a descoberta de uma vulnerabilidade não representa o fim de uma avaliação de segurança. Saber de que forma ela pode ser explorada e, em certos casos, desenvolver um artefato capaz de realizar esse ato, que é chamado de *exploit*, é necessário tanto para avaliar o impacto de sua exploração quanto para fornecer informações para sua correção.

O desenvolvimento de *exploits* vem se tornando uma atividade cada vez mais complexa, devido à implementação de mecanismos de proteção contra exploração nos principais sistemas operacionais. Com isso, o tempo gasto nessa atividade tem aumentado consideravelmente nos últimos anos, demandando o desenvolvimento de técnicas e ferramentas para facilitar sua realização.

Este trabalho inicialmente apresenta um estudo sobre a exploração de vulnerabilidades de *heap overflow* em programas baseados na *GNU C Library*. Posteriormente, são apresentados alguns algoritmos criados para auxiliar a construção de *exploits* para vulnerabilidades desse tipo, utilizando a técnica de exploração conhecida como *The House of Mind*. Finalmente, é apresentada uma ferramenta desenvolvida para testar o funcionamento de tais algoritmos em programas vulneráveis, juntamente com os resultados obtidos.

Abstract

Vulnerability research is one of the key activities of Computer Security, because it is through it that the vulnerabilities that are used in most of the attacks on networks and computer systems are discovered and/or patched. However, the discovery of a vulnerability does not mean the end of a security assessment. Knowing how it could be exploited and, in some cases, to develop an artifact capable of performing this act, which is called an *exploit*, is necessary both to assess the impact of their exploitation and to provide information for their correction.

Exploit development is becoming a task increasingly more complex, due to the implementation of mechanisms to protect against exploitation in the main operating systems. Thus, the time spent on this activity has increased considerably in recent years, requiring the development of techniques and tools to facilitate its realization.

This paper initially presents a study on the exploitation of heap overflow vulnerabilities in programs based on *GNU C Library*. Later, some algorithms created to support the exploit development process for this kind of vulnerability are presented, using the exploitation technique known as *The House of Mind*. Finally, a tool developed to test the functioning of such algorithms against vulnerable programs is presented, along with the results.

Sumário

Lista de Figuras

Lista de Listagens

Lista de Tabelas

1 Introdução

1.1 Contexto

O tema dessa pesquisa é a geração automática de *exploits* para vulnerabilidades em software. São abordadas vulnerabilidades de corrupção de memória, que estão presentes em um grande número de aplicações construídas em C e C++, devido aos erros introduzidos pelos programadores ao lidar com o gerenciamento de memória.

Um *exploit* pode ser definido como um programa de computador ou conjunto de dados que se aproveita de uma vulnerabilidade para executar ações maliciosas. Como as vulnerabilidades de corrupção de memória são geralmente exploradas para conseguir execução de código arbritário na máquina alvo, elas costumam trazer vários prejuízos quando são exploradas, seja em ataques direcionados à um alvo específico ou ataques em massa, como em um *worm* que explora uma vulnerabilidade como forma de infectar mais computadores. Isso faz com que elas sejam as mais procuradas por atacantes e empresas da área de segurança de software, que compram informações sobre vulnerabilidades para proteger os seus clientes [1, 2].

Esses *exploits* são comumente construídos manualmente, após serem identificadas as causas da vulnerabilidade. Analisar como os dados de entrada do programa podem ser manipulados para permitir a execução de código arbritário é uma tarefa geralmente complexa e algumas pesquisas [3–5] já foram feitas para automatizar esse processo.

1.2 Formulação do Problema

Como automatizar a geração de *exploits* para vulnerabilidades de *heap overflow* em programas que usam a *GNU C Library* (*glibc*) com a técnica de exploração conhecida como *The House of Mind*, dada uma entrada para o programa que o faça terminar sua execução de forma inesperada?

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é contribuir para a solução do problema da geração automática de *exploits* para vulnerabilidades de *heap overflow*, demonstrando a hipótese de pequisa. Para isso, foram desenvolvidos algoritmos para detecção, controle do *layout* da memória e geração de *exploits*.

Esses algoritmos foram implementados em uma ferramenta que foi usada na realização de testes com programas vulneráveis. Não faz parte do objetivo do trabalho lidar com os mecanismos de proteção contra exploração descritos no *Referencial Teórico*.

O sistema operacional escolhido para a implementação e teste da ferramenta foi o Ubuntu 9.10 de 32 bits, que é baseado na distribuição Debian GNU/Linux e usa o kernel Linux versão 2.6. A versão da *glibc* instalada por padrão no sistema é 2.10.1, que é suscetível à várias técnicas de exploração [6].

A pesquisa foca na geração de *exploits* utilizando a técnica *The House of Mind* [7].

1.3.2 Objetivos Específicos

- Mostrar que é possível gerar um *exploit* para o programa vulnerável mais simples.
- Mostrar que é possível gerar um *exploit* para variações do programa vulnerável mais simples.

1.4 Justificativa

Atualmente, os *exploits* para as vulnerabilidades de corrupção de memória são construídos manualmente com a ajuda de algumas ferramentas que permitem visualizar o estado do programa vulnerável. Vários mecanismos de proteção contra a exploração têm sido implementados nos principais sistemas operacionais e isso tem causado um aumento no tempo de construção dos *exploits*. Estima-se que a construção de um *exploit* confiável para uma vulnerabilidade de *heap overflow* em um sistema operacional que implementa alguns desses mecanismos demore de um à três meses [8].

A geração automática de *exploits* traz benefícios para as empresas que precisam avaliar o impacto de um *bug* encontrado em seus programas ou em programas de terceiros, no caso de

empresas que prestam serviços na área de segurança de software. Geralmente, a única forma de saber se um *bug* é realmente explorável é tentar escrever um *exploit* para ele. Outras organizações que podem se beneficiar da geração automática de *exploits* são as militares, como as agências de segurança e os serviços de inteligência. Para elas isso seria um instrumento para o desenvolvimento rápido de *exploits* que podem ser usados para conseguir acesso aos sistemas computacionais dos inimigos.

Embora pesquisas recentes [5, 9] tenham conseguido aplicar com êxito técnicas de análise dinâmica de software na geração de *exploits* para vulnerabilidades de *stack overflow*, a geração automática de *exploits* para vulnerabilidades de *heap overflow* continua sendo um problema em aberto.

1.5 Hipótese

É possível automatizar a geração de *exploits* para vulnerabilidades de *heap overflow* usando a técnica de exploração *The House of Mind* seguindo os seguintes passos:

Primeiramente é detectada a ocorrência de um *heap overflow* dada uma entrada para o programa vulnerável que o faça terminar sua execução de forma inesperada. Exemplos de entradas são arquivos e pacotes de rede. Para realizar essa detecção, foi desenvolvido um algoritmo de monitoramento das estruturas armazenadas na *heap* que, a cada chamada à função *free()*, verifica se existe dados derivados da entrada do usuário nas localizações exigidas pela técnica *The House of Mind*.

Após a detecção da vulnerabilidade, o segundo passo é coletar as restrições obtidas pela execução da *Taint Analysis* [10, 11]. Essas restrições são usadas na verificação da explorabilidade da vulnerabilidade. A técnica *The House of Mind*, assim como outras técnicas de exploração, exige o controle do *layout* da memória para ser executada com sucesso.

Finalmente, foi desenvolvido um algoritmo que usa um solucionador de restrições [11] para calcular quais posições no arquivo original terão seus valores sobrescritos e quais serão os seus novos valores para que a execução de código arbritário (*shellcode*) seja atingida.

1.6 Limitações do Trabalho

Esse trabalho não avalia a solução proposta sob o efeito dos mecanismos de proteção contra exploração apresentados no *Referencial Teórico*, devido ao limitado tempo disponível para sua

realização. A avaliação desses mecanismos é um interessante tópico de pesquisa para trabalhos futuros.

Outra limitação é que os testes foram executados apenas no programa vulnerável mais simples e em suas variações.

1.7 Organização

O capítulo 2 apresenta um breve histórico sobre a exploração de vulnerabilidades de *heap overflow* em sistemas *Unix-like* e o estado-da-arte na exploração de vulnerabilidades de *heap overflow* em programas baseados na *GNU C Library*. No capítulo 3, são apresentadas atualizações para a técnica de exploração conhecida como *The House of Mind* para que funcione em versões mais novas da *glibc*, bem como novas estratégias para burlar os mecanismos de proteção contra exploração implementados no Ubuntu. O capítulo 4 apresenta os algoritmos criados para automatização da exploração por meio da técnica *The House of Mind*. No capítulo 5, os detalhes da implementação da ferramenta são apresentados. O capítulo 6 traz os resultados obtidos no teste da ferramenta. Finalmente, o capítulo 7 apresenta as conclusões e principais contribuições do trabalho, bem como ideias para trabalhos futuros.

2 *Referencial Teórico*

2.1 *Buffer Overflows*

Segundo a referência 12 “(...) um *buffer overflow* é um defeito de software no qual dados copiados para uma localização na memória excedem o tamanho da área de destino reservada.” (tradução nossa).

Esse é o tipo mais comum de corrupção de memória e sua exploração bem sucedida pode permitir que o atacante execute código arbritário no computador da vítima.

2.1.1 *Stack Overflows*

Stack overflows são *buffer overflows* que ocorrem quando o *buffer* está alocado na pilha do programa [12]. Esse é o tipo mais entendido e mais fácil de explorar. As técnicas iniciais de exploração foram detalhadas em 1996 em um artigo [13] escrito por Aleph One na Phrack 49.

2.1.2 *Heap Overflows*

Heap overflows são *buffer overflows* que ocorrem quando o *buffer* está alocado na *heap* do programa [12]. As técnicas de exploração de vulnerabilidades desse tipo só começaram a aparecer no ano 2000 e são mais complexas que as técnicas para explorar um *stack overflow*.

2.2 *Shellcode*

As vulnerabilidades de *buffer overflow* são geralmente exploradas por meio do redirecionamento da execução para uma localização na memória na qual o atacante tem controle sobre os dados armazenados. A execução de código arbritário é conseguida por meio da inserção de trechos de código de máquina nessa localização, que são chamados de *shellcode*. Uma das ações

mais comuns encontradas em *shellcodes* é fornecer uma porta de entrada na máquina afetada para permitir o seu controle pelo atacante.

2.3 Exploração de *Heap Overflow* na *GNU C Library*

2.3.1 Histórico

A primeira demonstração pública de exploração de uma vulnerabilidade de *heap overflow* ocorreu em 2000, quando um pesquisador conhecido como Solar Designer publicou um *exploit* [14] para uma vulnerabilidade no processamento de imagens JPEG do navegador Netscape. Solar utilizou uma técnica de manipulação do estado da *heap* que permitia escrever 4 bytes em uma posição de memória arbitrária, e que posteriormente seria chamada de técnica *unlink()*.

Em 2001, na Phrack 57, foram publicados dois artigos sobre exploração de vulnerabilidades de *heap overflow* que contribuíram para a divulgação das novas técnicas de exploração. No primeiro artigo [15], MaXX descreve o alocador de memória usado pela *glibc* e duas técnicas de exploração: a técnica *unlink()*, apresentada originalmente por Solar Designer e uma nova técnica chamada *frontlink()*. Ele apresenta um *exploit* para uma vulnerabilidade no *Sudo* e explica detalhadamente quais as ações que executou para controlar o estado da *heap*. O segundo artigo [16], escrito por um autor anônimo, descreve a implementação da *heap* na *glibc* e no *System V*, criado pela AT&T.

A próxima contribuição apareceu em 2003, com um artigo [17] na Phrack 61 onde "jp" apresenta várias técnicas para facilitar a exploração das vulnerabilidades, usando a técnica *unlink()* como base. Seu objetivo era criar técnicas mais genéricas e confiáveis que fossem fáceis de automatizar.

Até 2004, a exploração da *glibc* usando essas técnicas era possível, mas as verificações de integridade implementadas no final do ano fizeram com que deixassem de funcionar.

O próximo avanço nas técnicas de exploração veio em 2005 com um artigo [7] publicado na lista de e-mails *Bugtraq*. Phantasmal Phantasmagoria apresentou cinco novas técnicas que poderiam ser usadas para explorar *heap overflows* nas novas versões da *glibc*. Seu artigo foi considerado muito teórico por não apresentar provas de conceito para demonstrar as técnicas.

Em 2007, a revista eletrônica ".aware EZine Alpha" incluiu um artigo [18] de k-sPecial onde ele analisa uma técnica de Phantasmal conhecida como *The House of Mind* e apresenta uma prova de conceito. Ele escolheu essa técnica por ela ser considerada como a mais geral de

Listagem 2.1: Estrutura *malloc_chunk*

```
1 #define INTERNAL_SIZE_T size_t
2
3 struct malloc_chunk {
4     INTERNAL_SIZE_T prev_size;
5     INTERNAL_SIZE_T size;
6     struct malloc_chunk * fd;
7     struct malloc_chunk * bk;
8 };
```

todas e ser a mais parecida com a técnica *unlink()* inicial.

Outro artigo publicado em 2007, na Phrack 64, revelou uma nova técnica de exploração. g463 descobriu como atacar um elemento especial da *heap* enquanto tentava explorar uma vulnerabilidade no utilitário *file* [19].

Os dois artigos mais recentes foram publicados na Phrack 66, em 2009. No primeiro deles [20], blackngel analisa detalhadamente o artigo teórico de Phantasmal Phantasmagoria, apresentando provas de conceitos e correções para as técnicas. No segundo [21], huku apresenta uma nova técnica de exploração, que não é baseada nos trabalhos de Phantasmal.

2.3.2 Informações Básicas sobre a Implementação

Chunks, *Flags*, *Heaps* e *Arenas*

Chunks são os blocos de memória gerenciados pelo alocador. Logo no início de cada *chunk* é armazenada uma *boundary tag*, definida como mostrado na Listagem 2.1.

Os campos dessa estrutura são usados de forma diferente, dependendo se o *chunk* correspondente e seu *chunk* anterior estão livres ou não:

1. **prev_size**

   O campo prev_size armazena o tamanho do *chunk* anterior ao *chunk* atual, caso o primeiro esteja livre. Se estiver alocado, esse campo pode armazenar dados do *chunk* anterior.

2. **size**

   O campo size armazena o tamanho do *chunk* atual.

3. **fd**

O campo fd armazena o ponteiro para o *chunk* seguinte na lista circular duplamente encadeada de *chunks* livres, se o *chunk* atual estiver livre. Caso contrário, é nele que se inicia a área de armazenamento de dados do *chunk*.

4. **bk**

   O campo bk armazena o ponteiro para o *chunk* anterior na lista circular duplamente encadeada de *chunks* livre, se o *chunk* atual estiver livre. Caso contrário, faz parte da área de armazenamento de dados do *chunk*.

O tamanho de um *chunk* é sempre um múltiplo de 8 bytes, de forma que os 3 bits menos significativos do campo size seriam sempre iguais à zero. Entretanto, para aproveitar esses 3 bits, a implementação os usa para armazenar as *flags* de status do *chunk*.

O bit de mais baixa ordem controla a *flag* PREV_INUSE, que indica se o *chunk* anterior ao *chunk* atual está alocado. O segundo bit de mais baixa ordem controla a *flag* IS_MMAPPED, que indica se o *chunk* atual foi alocado através do mecanismo de mapeamento de memória *mmap*. O terceiro bit controla a *flag* NON_MAIN_ARENA, que indica se o *chunk* atual faz parte de uma região de memória independente.

O alocador tenta fazer com que os *chunks* sejam alocados de forma contígua sempre que possível. Em algumas situações, quando são feitas muitas requisições de alocação e a *heap* principal está bloqueada ou ocupada, o alocador pode criar outra *heap* via *mmap()* para atendê-las. Cada nova *heap* criada contém uma *arena*.

Funções Públicas

Segundo a referência 15, as funções públicas principais que fazem o gerenciamento das alocações e desalocações de memória na *glibc* são:

1. **malloc(size_t n)**

   Retorna um ponteiro para um novo *chunk* alocado de no mínimo n bytes, ou null se não há espaço disponível.

2. **free(Void_t* p)**

   Libera o *chunk* de memória apontado por p, ou não tem efeito se p é null.

3. **realloc(Void_t* p, size_t n)**

Listagem 2.2: Macro *unlink()*

```
1 #define unlink(P, BK, FD) {            \
2     BK = P->bk;                        \
3     FD = P->fd;                        \
4     FD->bk = BK;                       \
5     BK->fd = FD;                       \
6 }
```

Retorna um ponteiro para um *chunk* de tamanho n que contém os mesmos dados que o *chunk* p até seu tamanho anterior, ou null se não há espaço disponível. O ponteiro retornado pode ou não ser o mesmo que p. Se p é null, a função se torna equivalente à malloc().

4. **calloc(size_t unit, size_t quantity)**

Retorna um ponteiro para memória alocada de tamanho quantity * unit, com todas as posições com valor zero.

2.3.3 Técnicas de Exploração

Apenas duas técnicas foram selecionadas para serem descritas. A técnica *unlink()* foi selecionada por ser a primeira divulgada publicamente e assim tem sua importância histórica. A técnica *The House of Mind* é considerada a mais geral das técnicas de exploração conhecidas atualmente e foi escolhida por ser interessante para ser automatizada.

Técnica *unlink()*

Segundo a referência 15,

> Se um atacante consegue enganar a *glibc* para processar um *chunk* falso de memória cuidadosamente manipulado (ou um *chunk* cujos campos fd e bk tenham sido corrompidos) com a macro *unlink()*, ele será capaz de sobrescrever um inteiro arbritário na memória com o valor de sua escolha, e portanto será capaz de eventualmente executar código arbritário. (tradução nossa).

A macro *unlink()* é chamada dentro da função *free()* e remove um *chunk* da lista duplamente encadeada de *chunks* em uso. A Listagem 2.2 mostra como ela era definida.

Se o atacante conseguisse colocar o endereço de um ponteiro de função menos 12 bytes no ponteiro FD, e o endereço de um *shellcode* no ponteiro BK, a macro sobrescreveria o ponteiro de função localizado em FD mais 12 bytes com o valor de BK.

Listagem 2.3: Macro *unlink()* com verificação de integridade

```
#define unlink(P, BK, FD) {                                          \
    FD = P->fd;                                                      \
    BK = P->bk;                                                      \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))            \
      malloc_printerr (check_action,                                 \
              "corrupted double-linked list", P);                    \
    else {                                                           \
        FD->bk = BK;                                                 \
        BK->fd = FD;                                                 \
    }                                                                \
}
```

Quando o programa lia o ponteiro de função e chamava a "função" apontada, o *shellcode* era executado.

Essa técnica de exploração funcionou até 2004, quando a verificação de integridade mostrada na Listagem 2.3 foi adicionada à *glibc* 2.3.5 [20].

Essa verificação garante que o *chunk* P que está sendo removido faz parte da lista duplamente encadeada. Como P->fd aponta para o *chunk* seguinte (FD), então FD->bk deve apontar para P. Da mesma forma P->bk aponta para o *chunk* anterior (BK), e BK->fd deve apontar para P.

The House of Mind

A técnica *The House of Mind* permite sobrescrever o valor armazenado em um endereço de memória arbritário com apenas uma chamada à função *free()*. Por isso, é considerada como a técnica que mais se assemelha à técnica *unlink()* dentre aquelas publicadas por Phantasmal em 2005 [7].

Essa técnica foi pesquisada posteriormente por k-sPecial [18] e blackngel [20], que fizeram correções e apresentaram provas de conceito. Os trechos de código mostrados são referentes à *glibc* 2.3.6.

Deve-se considerar que a função *free()* é chamada para remover um *chunk* que pôde ser sobrescrito previamente através de um *overflow* ocorrido em um *chunk* que está alocado atrás dele.

Quando a função *free()* da *glibc* é chamada, o controle passa para um *wrapper* chamado *public_fREe()*. A técnica consiste em enganar esse *wrapper* para poder controlar o ponteiro de *arena* que é passado para a função interna *_int_free()*. O trecho de código relevante de

Listagem 2.4: Trecho de código do *wrapper public_fREe()*

```
1  void
2  public_fREe(Void_t* mem)
3  {
4      mstate ar_ptr;
5      mchunkptr p;     /* chunk corresponding to mem */
6      ...
7      p = mem2chunk(mem);
8      ...
9      ar_ptr = arena_for_chunk(p);
10     ...
11     _int_free(ar_ptr, mem);
12     ...
```

Listagem 2.5: Macro *arena_for_chunk()*

```
1  #define HEAP_MAX_SIZE (1024*1024) /* must be a power of two */
2
3  #define heap_for_ptr(ptr) \
4    ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
5
6  #define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
7
8  #define arena_for_chunk(ptr)     \
9    (chunk_non_main_arena(ptr)?heap_for_ptr(ptr)->ar_ptr:&main_arena)
```

public_fREe() é mostrado na Listagem 2.4.

O ponteiro para a memória alocada que se deseja liberar que é passado para a função *free()* aponta para a porção de dados de um *chunk*. A macro *mem2chunk()* recebe esse ponteiro e retorna o endereço do *chunk* correspondente.

Após isso, o ponteiro para o *chunk* é passado para a macro *arena_for_chunk()*, que é definida como mostrado na Listagem 2.5.

Esse código é usado quando a *glibc* é compilada com suporte à *arenas* (USE_ARENAS), o que ocorre por padrão. A macro *arena_for_chunk()* recebe um ponteiro para um *chunk* e retorna a *arena* apropriada para ele. Quando o bit NON_MAIN_ARENA no campo size não está ativado, ar_ptr receberá o endereço de main_arena.

Como o atacante controla o valor do campo size, o bit NON_MAIN_ARENA pode ser ativado, fazendo com que a macro *chunk_non_main_arena()* retorne verdadeiro e ar_ptr receba o ponteiro retornado por heap_for_ptr(ptr)->ar_ptr.

A macro *heap_for_ptr()* recebe o endereço do *chunk* e o retorna alinhado à um múltiplo de HEAP_MAX_SIZE, dado que novas *heaps* são criadas alinhadas à um múltiplo dessa constante. Uma estrutura heap_info se localiza logo no início de uma *heap* e contém um elemento chamado

Listagem 2.6: Estrutura *heap_info*

```
typedef struct _heap_info {
    mstate ar_ptr; /* Arena for this heap. */
    struct _heap_info *prev; /* Previous heap. */
    size_t size; /* Current size in bytes. */
    size_t pad; /* Make sure the following data is properly aligned. */
} heap_info;
```

Listagem 2.7: Estrutura *malloc_state*

```
struct malloc_state {
    mutex_t mutex;
    INTERNAL_SIZE_T max_fast;   /* low 2 bits used as flags */
    mfastbinptr     fastbins[NFASTBINS];
    mchunkptr       top;
    mchunkptr       last_remainder;
    mchunkptr       bins[NBINS * 2];
    unsigned int    binmap[BINMAPSIZE];
    ...
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
}
...
static struct malloc_state main_arena;
```

ar_ptr, que é um ponteiro para a *arena* dessa *heap*. A Listagem 2.6 mostra sua definição.

O atacante precisa manipular a *heap*, geralmente fazendo com que a aplicação aloque memória repetidamente até que um bloco de memória que ele tenha controle seja alocado em um endereço alinhado à HEAP_MAX_SIZE. Controlando esse bloco de memória, o atacante tem controle sobre o valor do campo ar_ptr, que é passado para a função *_int_free()*, e com isso consegue forjar uma *arena*.

Depois de controlar o valor do campo ar_ptr, o atacante pode escolher entre duas formas de explorar o *heap overflow*.

Explorando *unsorted_chunks()*

A primeira delas explora a função *unsorted_chunks()*. Para isso, é necessário entender a estrutura malloc_state (Listagem 2.7), pois ar_ptr é um ponteiro para essa estrutura.

O trecho de código que será explorado está localizado na função *_int_free()* e é mostrado na Listagem 2.8.

O parâmetro av recebe o valor de ar_ptr, que aponta para a falsa *arena*. A macro *unsorted_chunks()* é mostrada na Listagem 2.9.

Listagem 2.8: Trecho de código da função *_int_free()* que executa a macro *unsorted_chunks()*

```
1  void _int_free(mstate av, Void_t *mem) {
2      ...
3      bck = unsorted_chunks(av);
4      fwd = bck->fd;
5      p->bk = bck;
6      p->fd = fwd;
7      bck->fd = p;
8      fwd->bk = p;
9      ...
10 }
```

Listagem 2.9: Macro *unsorted_chunks()*

```
1 #define bin_at(m, i) ((mbinptr)((char*)&((m)->bins[(i)<<1]) \
2             - (SIZE_SZ<<1)))
3 ...
4 #define unsorted_chunks(M)          (bin_at(M, 1))
```

Sabendo-se que *unsorted_chunks()* retorna o endereço de av->bins[0], a exploração segue como mostrado no exemplo a seguir:

```
bck = &av->bins[0];
fwd = bck->fd = *(&av->bins[0] + 8);
fwd->bk = *(&av->bins[0] + 8) + 12 = p;
```

Portanto, como o atacante controla o valor armazenado em &av->bins[0] + 8, o endereço de um ponteiro de função - 12 bytes pode ser armazenado lá e terá seu valor sobrescrito com o valor de p, que é o ponteiro para o *chunk* que está sendo removido.

O atacante também controla os valores localizados nesse *chunk*, e pode colocar uma instrução *JMP* para pular para um *shellcode* localizado adiante.

Entretanto, para que a execução do *shellcode* aconteça, algumas condições verificadas na função *_int_free()* devem ser satisfeitas. Essas condições foram revisadas e são descritas no trabalho de blackngel [20].

Explorando *fastbins*

blackngel apresentou uma solução para o método *fastbin*, proposto inicialmente por Phantasmal e discutido posteriormente por k-sPecial. Ele resolveu os problemas referentes às condições de exploração e propôs uma nova forma de explorar *fastbins*, que será descrita a seguir.

Listagem 2.10: Trecho de código da função *_int_free()* que acessa *fastbins*

```
1  if ((unsigned long)(size) <= (unsigned long)(av->max_fast)) {
2      if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ,
           0) ||
3          __builtin_expect (chunksize(chunk_at_offset(p, size))
4                  >= av->system_mem, 0))
5      {
6          errstr = "free(): invalid next size (fast)";
7          goto errout;
8      }
9
10     set_fastchunks(av);
11     fb = &(av->fastbins[fastbin_index(size)]);
12     if (__builtin_expect (*fb == p, 0))
13     {
14         errstr = "double free or corruption (fasttop)";
15         goto errout;
16     }
17     p->fd = *fb;
18     *fb = p;
19 }
```

Listagem 2.11: Macro *fastbin_index()*

```
1  #define fastbin_index(sz) ((((unsigned int)(sz)) >> 3) - 2)
```

A técnica consiste em explorar o código mostrado na Listagem 2.10.

O atacante deve colocar em fb o endereço de memória cujo conteúdo queira sobrescrever. Na atribuição da linha 19, esse endereço terá seu valor trocado para p. Se o endereço de memória escolhido armazenar um ponteiro de função, ele passará a apontar para a "função" p. Quando esse ponteiro de função for lido para fazer a chamada da função correspondente, será executado o código armazenado no endereço p. Como p é o endereço do *chunk* que está sendo removido, e que foi previamente sobrescrito através de um *overflow* em um *chunk* que estava armazenado antes dele, seu conteúdo é controlado pelo atacante, podendo conter o código arbritário (*shellcode*) que deseja executar.

Para controlar o valor de fb, o atacante deve controlar o valor de retorno da expressão na linha 11. A macro *fastbin_index()* é mostrada na Listagem 2.11.

Se o argumento sz for passado com o valor 16 bytes, essa macro retorna (16 >> 3) − 2 = 0. Assim, fb recebe &(av->fastbins[0]).

Como o atacante controla o conteúdo da *arena*, pode controlar o valor armazenado nesse endereço, mas para que a exploração ocorra com sucesso, um conjunto de condições devem ser respeitadas, assim como no método *unsorted_chunks()*.

blackngel sugeriu que fb fosse usado para alterar o valor do endereço de retorno armazenado na pilha. Dessa forma, quando a função corrente que contém o código vulnerável fosse finalizada, seria executado o código localizado em p.

2.4 Mecanismos de Proteção

A maioria dos sistemas operacionais modernos implementam alguns mecanismos de proteção contra a exploração de vulnerabilidades. Esses mecanismos tornam a exploração mais difícil e devem ser abordados numa pesquisa com maior disponibilidade de tempo.

2.4.1 *Stack Hardening*

Os dois métodos mais comuns de dificultar a exploração de vulnerabilidades de *stack overflow* são a implementação de *stack cookies* e a reordenação das variáveis alocadas na pilha. Esses métodos estão implementados nos principais compiladores de C/C++.

Segundo a referência 12:

> *Stack cookies* funcionam por meio da inserção de valores aleatórios de 32 bits (geralmente gerados em tempo de execução) na pilha imediatamente depois do endereço de retorno e ponteiro de frame salvos, mas antes das variáveis locais em cada frame da pilha (...). Este *cookie* é inserido quando a função é executada e é verificado imediatamente antes que a função retorne. Se o valor do *cookie* foi alterado, o programa pode inferir que a pilha foi corrompida e toma as ações apropriadas. Esta resposta geralmente envolve logar o problema e terminar o programa imediatamente. (tradução nossa)

O outro método faz a reordenação das variáveis alocadas na pilha, de forma que os buffers fiquem acima das outras variáveis. Isso dificulta o trabalho do atacante em sobrescrever variáveis como ponteiros de função para executar código arbritário.

2.4.2 *Heap Hardening*

Várias verificações de integridade das estruturas da *heap* têm sido implementadas nos últimos anos. Essas verificações visam dificultar a exploração de *heap overflows* analisando os meta-dados da *heap* a cada operação executada.

2.4.3 *Stack* e *Heap* Não-Executáveis

Esse mecanismo de proteção previne a execução de código a partir da pilha ou *heap*, considerando que essas áreas da memória não contêm o código original da aplicação sendo executada. É geralmente chamado de Data Execution Prevention (DEP) ou NoExec, e pode ser implementado por software ou hardware.

2.4.4 *Address Space Layout Randomization*

Segundo a referência 12, "Address space layout randomization (ASLR) é uma tecnologia que tenta mitigar a ameaça de *buffer overflows* escolhendo de forma aleatória onde os dados e código da aplicação serão mapeados" (tradução nossa).

2.4.5 Mecanismos Implementados no Ubuntu 9.10

O Ubuntu 9.10 implementa todos os mecanismos de segurança descritos anteriormente por padrão, além de alguns outros como descrito em 22. Dentre tais mecanismos, o ASLR e a *heap* não-executável são aqueles que dificultam a exploração das vulnerabilidades de *heap overflow* utilizando a técnica *The House of Mind*.

2.5 Técnicas de Análise de Software

2.5.1 *Taint Analysis*

Segundo a referência 11,

> O propósito da *taint analysis* dinâmica é seguir o fluxo de informação entre origens e destinos. Qualquer valor em um programa cuja computação dependa de dados derivados de uma origem afetada é considerado afetado (*tainted*). Qualquer outro valor é considerado não-afetado (*untainted*). Uma regra especifica exatamente como os dados afetados fluem com a execução do programa, quais tipos de operações introduzem novos dados afetados e quais verificações são executadas nesses dados. (tradução nossa)

A etapa de *taint analysis* do trabalho 5, que usa instrumentação binária, cria um conjunto inicial de localizações de memória e registradores afetados e analisa os elementos em cada instrução executada pelo processador para adicionar ou remover elementos nesse conjunto.

O exemplo abaixo mostra como poderia funcionar a *taint analysis* quando aplicada à um trecho de código com instruções da arquitetura x86, considerando que, inicialmente, somente o registrador EDX faça parte do conjunto afetado:

```
MOV EAX, 0x1234 ; EAX = 0x1234
MOV EBX, EAX    ; EBX = EAX
ADD EBX, EDX    ; EBX = EBX + EDX (EBX é adicionado ao conjunto)
MOV ECX, EDX    ; ECX = EDX (ECX é adicionado ao conjunto)
MOV EBX, 0x4321 ; EBX = 0x4321 (EBX é removido do conjunto)
```

2.5.2 *Constraint Solving*

A resolução de restrições (*Constraint Solving*) tem sido aplicada com sucesso em pesquisas sobre geração automática de casos de teste [23, 24] e na geração automática de *exploits* [3, 5, 9, 25].

Ela consiste em construir fórmulas durante a execução do programa, que são posteriormente resolvidas com um solucionador de restrições.

2.6 Trabalhos Relacionados

Os seguintes trabalhos estão relacionados com geração automática de *exploits*:

- *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilites* - Heelan [5] explora a geração automática de *exploits* para vulnerabilidades de *buffer overflow* nas quais a entrada do usuário corrompe um ponteiro de instrução armazenado, um ponteiro de função ou a localização de destino e valor de origem de uma instrução de escrita na memória. O problema geral de geração de *exploits* é formalizado e vários algoritmos usando *Taint Analysis* e *Constraint Solving* são apresentados. Esse trabalho não lida com *overflows* nos meta-dados da *heap* e só leva em conta a mitigação conhecida como ASLR.

- *Automated Exploit Development, The future of exploitation is here* - Medeiros [4] descreve os algoritmos implementados na ferramenta Prototype-8 para o desenvolvimento automatizado de *exploits*. A ferramente contém regras pré-definidas que permitem gerar *exploits* para alguns tipos de vulnerabilidades de *stack* e *heap overflow*. Essas regras tentam detectar que partes dos dados de entrada do programa sobrescrevem dados em

localizações interessantes, que são posteriormente usados para gerar um *exploit* baseado em um modelo. Essa pesquisa não tenta resolver as restrições que podem existir sobre os dados de entrada.

- *Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach* - Lin et al. [25] também ataca o problema da geração automática de *exploits*. Sua técnica consiste em iniciar com uma entrada suspeira e executar uma análise de fluxo de dados para obter restrições que são então resolvidas com um provador de teoremas. É preciso a interação do usuário para guiar as mutações executadas nos dados de entrada e a pesquisa não lida com o problema da execução de *shellcode*.

- *Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications* - Brumley et al. [3] apresenta uma técnica que consiste em analisar um programa vulnerável e sua versão corrigida para identificar as restrições introduzidas pelo segundo. Assim, essas restrições são resolvidas com o solucionador de restrições para gerar dados de entrada para o programa vulnerável capazes de disparar o código vulnerável. Os autores não lidam com o problema de conseguir execução de *shellcode* automaticamente.

- *AEG: Automatic Exploit Generation* - Avgerinos et al. [9] apresenta várias técnicas para a geração automática de *exploits*, indo desde a descoberta da vulnerabilidade até a produção do *exploit* final. O tipo de vulnerabilidade considerado é o *stack overflow* e é apresentado um novo modelo de execução simbólica. Esse novo modelo é usado para coletar restrições e construir fórmulas que são resolvidas com um solucionador de restrições. As análises requerem que o código-fonte do programa vulnerável esteja disponível, o que limita de certo modo a aplicação prática da técnica.

- *Byakugan* - Essa extensão [26] para o *WinDbg* usa casamento de padrão para descobrir onde os dados de entrada foram colocados na memória do programa. Apesar de levar em conta simples modificações, a extensão não sabe lidar com as restrições sobre a entrada para o programa.

3 *Atualização da Técnica* The House of Mind *para as Versões 2.10.1 e 2.11.1 da* glibc

A técnica *The House of Mind* permite sobrescrever o valor armazenado em um endereço arbitrário de memória com apenas uma chamada à função *free()*. Uma atualização da técnica foi necessária, pois o último trabalho sobre ela [20] aborda a versão 2.8.90 da *glibc*, enquanto o sistema operacional Ubuntu 9.10, que foi escolhido para a implementação da ferramenta, usa a versão 2.10.1. Além disso, algumas modificações introduzidas na versão 2.11.1 usada pelo Ubuntu 10.04 também são descritas.

A atualização dessa técnica é uma das contribuições deste trabalho e a ordem como os passos são apresentados é baseada na apresentação da técnica feita no *Referencial Teórico*.

3.1 Informações Básicas sobre a *Heap*

As *boundary tags*, que são armazenadas no início de cada *chunk* são definidas como mostrado na Listagem 3.1.

Os novos campos fd_nextsize e bk_nextsize têm o mesmo propósito que os campos fd e bk,

Listagem 3.1: Estrutura *malloc_chunk*

```
1  struct malloc_chunk {
2      INTERNAL_SIZE_T       prev_size;
3      INTERNAL_SIZE_T       size;
4
5      struct malloc_chunk* fd;
6      struct malloc_chunk* bk;
7
8      struct malloc_chunk* fd_nextsize;
9      struct malloc_chunk* bk_nextsize;
10 }
```

Listagem 3.2: Wrapper *public_fREe()*

```
1  void
2  public_fREe(Void_t* mem)
3  {
4      mstate ar_ptr;
5      mchunkptr p;
6      ...
7      p = mem2chunk(mem);
8      ...
9      ar_ptr = arena_for_chunk(p);
10     ...
11     _int_free(ar_ptr, p);
12     ...
13 }
```

que são usados somente se o *chunk* está livre e armazenam os ponteiros para o *chunk* seguinte e o anterior em uma lista duplamente encadeada. A diferença é que são usados somente para blocos grandes.

3.2 *The House of Mind*

Deve-se considerar que a função *free()* é chamada para remover um *chunk* que pôde ser sobrescrito previamente por meio de um *overflow* ocorrido em um *chunk* que está alocado atrás dele.

Quando a função *free()* da *glibc* é chamada, o controle passa para um *wrapper* chamado *public_fREe()*. A técnica consiste em enganar esse *wrapper* para poder controlar o ponteiro de *arena* que é passado para a função interna *_int_free()*. O trecho de código relevante de *public_fREe()* não sofreu alteração em relação à versão anterior da *glibc*, e é mostrado na Listagem 3.2.

O ponteiro passado para *free()* aponta para os dados do *chunk*. A macro *mem2chunk()* recebe esse ponteiro e retorna o endereço do *chunk* correspondente. Após isso, o ponteiro para o *chunk* é passado para a macro *arena_for_chunk()*, que está definida como mostrado na Listagem 3.3.

Esse trecho de código não apresenta mudanças em relação à versão anterior da *glibc* e é usado quando ela é compilada com suporta à *arenas* (USE_ARENAS), o que ocorre por padrão. A macro *arena_for_chunk()* recebe um ponteiro para um *chunk* e retorna a *arena* correspondente. Quando o bit NON_MAIN_ARENA no campo size não está ativado, ar_ptr recebe o endereço de main_arena.

Listagem 3.3: Macro *arena_for_chunk()*

```
#define HEAP_MAX_SIZE (1024*1024)

#define heap_for_ptr(ptr) \
     ((heap_info *)((unsigned long)(ptr) & \
               ~(HEAP_MAX_SIZE-1)))

#define chunk_non_main_arena(p) \
     ((p)->size & NON_MAIN_ARENA)

#define arena_for_chunk(ptr) \
     (chunk_non_main_arena(ptr) ? \
        heap_for_ptr(ptr)->ar_ptr : &main_arena)
```

Listagem 3.4: Estrutura *heap_info*

```
typedef struct _heap_info {
    mstate ar_ptr;
    struct _heap_info *prev;
    size_t size;
    size_t mprotect_size;
    char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

A macro *heap_for_ptr()* recebe o ponteiro para um *chunk* e o retorna alinhado à um múltiplo de HEAP_MAX_SIZE, dado que novas heaps são criadas alinhadas à um múltiplo dessa constante. Uma estrutura heap_info reside no início de uma *heap* e contém um elemento nomeado ar_ptr que é um ponteiro para a *arena* dessa *heap*. A Listagem 3.4 mostra sua nova definição.

Essa estrutura foi modificada nas novas versões e traz alguns novos campos. O atacante precisa manipular o estado da *heap*, geralmente forçando a aplicação à alocar memória repetidamente até que um *chunk* que ele controla seja alocado em um endereço alinhado à HEAP_MAX_SIZE . Controlando esse *chunk*, o atacante tem controle sobre o valor do campo ar_ptr, que é passado para a função *_int_free()*, e assim pode forjar uma *arena*.

Após controlar o valor do campo ar_ptr, o atacante pode escolher entre dois modos de explorar o *heap overflow*: método *unsorted_chunks()* e método *fastbinsY* [7, 18, 20].

Método *unsorted_chunks()*

O primeiro método explora a função *unsorted_chunks()*. Para isso, é necessário entender a nova definição da estrutura malloc_state, mostrada na Listagem 3.5, pois ar_ptr é um ponteiro para ela [20].

O trecho de código que será explorado está localizado na função *_int_free()* e é mostrado

Listagem 3.5: Estrutura *malloc_state*

```
struct malloc_state {
    mutex_t mutex;
    int flags;
    mfastbinptr fastbinsY[NFASTBINS];
    mchunkptr top;
    mchunkptr last_remainder;
    mchunkptr bins[NBINS * 2 - 2];
    unsigned int binmap[BINMAPSIZE];
    ...
    INTERNAL_SIZE_T system_mem;
    INTERNAL_SIZE_T max_system_mem;
};
```

Listagem 3.6: Código de *_int_free()* que executa a macro *unsorted_chunks()*

```
static void
_int_free(mstate av, mchunkptr p)
{
    ...
    bck = unsorted_chunks(av);
    fwd = bck->fd;
    p->fd = fwd;
    p->bk = bck;
    ...
    bck->fd = p;
    fwd->bk = p;
    ...
}
```

na Listagem 3.6. O parâmetro av recebe o valor de ar_ptr, que aponta para a *arena* falsa. A nova definição da macro *unsorted_chunks()* é mostrada na Listagem 3.7.

Sabendo que *unsorted_chunks()* retorna o endereço de av->bins[0], a exploração segue como mostrado a seguir:

Listagem 3.7: Macro *unsorted_chunks()*

```
#define bin_at(m, i) \
 (mbinptr) \
  (((char *) &((m)->bins[((i) - 1) * 2])) \
          - offsetof (struct malloc_chunk, fd))

#define unsorted_chunks(M)   (bin_at(M, 1))
```

```
bck = unsorted_chunks(av);
// bck = &av->bins[0];
fwd = bck->fd;
// fwd = *(&av->bins[0] + 8);
fwd->bk = p;
// *((&av->bins[0] + 8) + 12) = p;
```

Assim, como o atacante controla o valor armazenado em &av−>bins[0] + 8, ele pode colocar lá o endereço de um ponteiro de função menos 12 bytes e o valor armazenado nesse endereço será sobrescrito com o valor de p, que é o endereço do *chunk* que está sendo removido.

Entretanto, para que a execução do *shellcode* aconteça, algumas condições verificadas na função *_int_free()* devem ser satisfeitas. Essas condições foram descritas em trabalhos anteriores [7, 18, 20], mas as modificações introduzidas no código das novas versões da *glibc* requerem atualizações nas condições 3 e 6. As novas condições são:

1. O valor negativo do tamanho do *chunk* p precisa ser menor que o endereço do próprio *chunk*.
2. O tamanho do *chunk* precisa ser no mínimo MINSIZE bytes.
3. O tamanho do *chunk* precisa ser maior que get_max_fast().
4. O bit IS_MMAPPED precisa estar desativado no campo size.
5. O *chunk* p não pode ser o *chunk* av−>top.
6. O bit NONCONTIGUOUS_BIT de av−>flags precisar estar ativado.
7. O bit PREV_INUSE do próximo *chunk* precisa estar ativado.
8. O tamanho do próximo *chunk* precisa ser maior que 2 ∗ SIZE_SZ e menor que av−>system_mem.
9. O bit PREV_INUSE do *chunk* precisa estar ativado.
10. O próximo *chunk* precisa ser diferente de av−>top.
11. O bit PREV_INUSE do *chunk* posterior ao próximo *chunk* precisa estar ativado.

Listagem 3.8: Nova verificação adicionada à função *_int_free()*

```
1  static void
2  _int_free(mstate av, mchunkptr p)
3  {
4      ...
5      bck = unsorted_chunks(av);
6      fwd = bck->fd;
7      if (__builtin_expect (fwd->bk != bck, 0))
8      {
9        errstr = "free(): corrupted unsorted chunks";
10       goto errout;
11     }
12     p->fd = fwd;
13     p->bk = bck;
14     ...
15     bck->fd = p;
16     fwd->bk = p;
17     ...
18 }
```

Os trabalhos anteriores propõe que o atacante sobrescreva um ponteiro de função como aqueles localizados em *.dtors* ou *GOT*, por exemplo, com o endereço de p and coloque uma instrução *JMP* no campo p->prev_size para pular para um *shellcode* localizado adiante nos dados do *chunk*. Este método funciona caso a *heap* esteja marcada como executável.

Outro ponto para notar é que esta técnica só funciona até a versão 2.10.1 da *glibc*, pois a versão 2.11.1 vem com uma nova verificação que foi adicionada à função *_int_free()* e que previne a exploração [6]. A verificação é mostrada na Listagem 3.8.

Método *fastbinsY*

Esse método, com uma solução prática apresentada em [20], funciona em ambas as versões 2.10.1 e 2.11.1 da *glibc*. Ele consiste em explorar o código mostrado na Listagem 3.9.

O atacante deve colocar em fb o endereço de memória que ele quer sobrescrever. Na atribuição *fb = p;, esse endereço terá seu valor alterado para p. Se o endereço de memória escolhido contém um ponteiro de função, ele passará a apontar para a "função" p. Quando esse ponteiro de função é dereferenciado para chamar a função correspondente, a instrução armazenada em p é executada. Como p é o endereço do *chunk* que está sendo removido, que foi previamente sobrescrito por um overflow ocorrido em um chunk alocado antes dele, seu conteúdo pode ser controlado pelo atacante que pode armazenar um *shellcode* lá.

Para controlar o valor de fb, o atacante deve controlar o valor retornado da expressão &fastbin (av, fastbin_index (size)). As macros *fastbin()* e *fastbin_index()* são mostradas na Listagem 3.10.

Listagem 3.9: Código de *_int_free()* que acessa *fastbin*

```
if ((unsigned long)(size) <=
    (unsigned long)(get_max_fast ()))
  {
    if (__builtin_expect
        (chunk_at_offset (p, size)->size <=
            2 * SIZE_SZ, 0)
    || __builtin_expect
        (chunksize (chunk_at_offset (p, size))
            >= av->system_mem, 0))
    {
      errstr = "free(): invalid next size (fast)";
      goto errout;
    }
    ...
    set_fastchunks(av);
    fb = &fastbin (av, fastbin_index(size));
    if (__builtin_expect (*fb == p, 0))
    {
      errstr = "double free or corruption (fasttop)";
      goto errout;
    }

    p->fd = *fb;
    *fb = p;
```

Listagem 3.10: Macros *fastbin()* e *fastbin_index()*

```
#define fastbin(ar_ptr, idx) \
  ((ar_ptr)->fastbinsY[idx])

#define fastbin_index(sz) \
  ((((unsigned int)(sz)) >> \
        (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

Se o argumento sz é passado para a macro *fastbin_index()* com o valor de 16 bytes, esta macro retorna $(16 >> 3) - 2 = 0$. Assim, fb recebe &(av−>fastbinsY[0]).

Como o atacante controla os campos da *arena* falsa, ele também pode controlar o valor armazenado nesse endereço, mas para a exploração ocorrer com sucesso, um conjunto de condições precisam ser satisfeitas, assim como no método *unsorted_chunks()*, como descrito em [20]. A condição 3 foi atualizada para lidar com as novas versões da *glibc*. As novas condições são:

1. O valor negativo do tamanho do *chunk* p precisa ser menor que o endereço do próprio *chunk*.
2. O tamanho do *chunk* precisa ser no mínimo MINSIZE bytes.
3. O tamanho do *chunk* precisa ser menor ou igual à get_max_fast().
4. O tamanho do próximo *chunk* precisa ser maior que $2 * SIZE\_SZ$ e menor que av−>system_mem.

O trabalho mais recente sugere que fb seja usado para sobrescrever o endereço de retorno armazenado na pilha [20]. Assim, quando a função atual que contém o código vulnerável é finalizada, o *shellcode* localizado em p é executado.

3.3 Novas Estratégias de Ataque

Apesar de não abordar a geração de *exploits* quando as proteções do sistema estão habilitadas, algumas estratégias para burlá-las foram desenvolvidas durante a etapa de pesquisa e são apresentadas nesta seção.

Com as estratégias atuais de exploração descritas em trabalhos anteriores [7, 18, 20], a técnica *The House of Mind* é facilmente detida pela proteção NX. A habilidade fornecida pela técnica de escrever quatro bytes em qualquer lugar na memória não permite controlar diretamente o valor escrito no endereço escolhido. Esse valor é sempre um endereço da área da *heap* e se o atacante colocar o *shellcode* lá, o programa dará uma falha de segmentação ao tentar executá-lo. As seguintes estratégias foram desenvolvidas neste trabalho e podem ser usadas para burlar as mitigações:

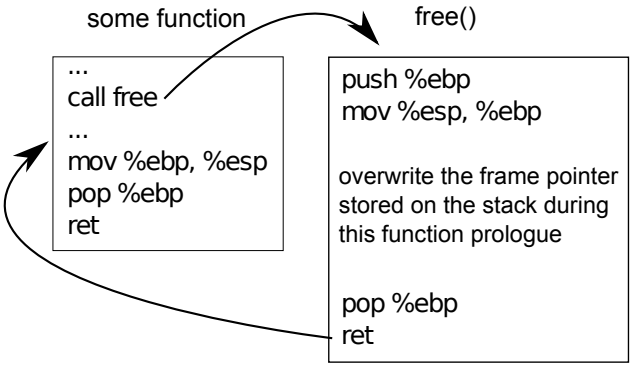


Figura 3.1: Cenário de exploração

3.3.1 Sobrescrevendo um "ponteiro de ponteiro de função"

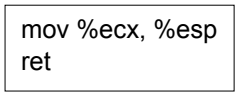
O atacante pode sobrescrever um "ponteiro de ponteiro de função" localizado na pilha ou na *heap* e indiretamente ganhar o controle sobre o valor escrito. Inicialmente, o "ponteiro de ponteiro de função" recebe o valor de p. Quando este ponteiro é lido duas vezes para chamar a função correspondente, o fluxo de controle é alterado para o endereço representado pelo valor armazenado em p−>prev_size. As vtables do C++ são um alvo em potencial.

3.3.2 Sobrescrevendo um ponteiro de *frame* armazenado na pilha

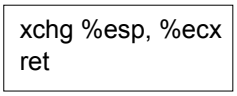
Se o atacante consegue sobrescrever um ponteiro de *frame* armazenado na pilha, ele pode controlar o valor removido desta pela instrução RET executada no epílogo da função. A Figura 3.1 mostra um possível cenário de exploração. O atacante poderia sobrescrever o ponteiro de *frame* localizado na pilha durante o prólogo da função *free()*. No epílogo dessa função, esse ponteiro sobrescrito é colocado no registrador EBP. O epílogo do chamador de *free()* move esse valor (p) para o registrador ESP (ponteiro da pilha), pega os primeiros 4 bytes localizados em ESP (p−>prev_size) para colocá-los em EBP e pega os próximos 4 bytes (p−>size) como o endereço para o qual o controle será desviado.

Embora esse cenário ilustre o método, ele é limitado porque o campo size do *chunk* que está sendo removido é restringido por várias condições. O atacante deve encontrar um epílogo de função que remove mais bytes até que RET seja capaz de pular para uma localização cujo valor seja totalmente controlável.

Usando as duas estratégias descritas acima, o atacante é capaz de controlar o fluxo de execução para executar instruções presentes nas seções executáveis do programa vulnerável. Antes de proceder com um método de *return-oriented programming* [27, 28], ele precisa encontrar um pivô na pilha (*stack pivot*) [29, 30] para fazer o ponteiro da pilha (ESP) apontar para a memória



(a) Move ECX para ESP (ponteiro da pilha) e pula para o endereço apontado por ESP.



(b) Troca os valores de ECX e ESP e pula para o endereço apontado por ESP.

Figura 3.2: Dois exemplos de pivôs na pilha

que ele controla na *heap*. Geralmente, um dos registradores irá apontar para a *heap*, próximo à localização onde o *return-oriented shellcode* deve ser colocado. A Figura 3.2 mostra dois exemplos de pivôs.

4 *Geração Automática de* Exploits

4.1 Visão Geral

Foram desenvolvidos vários algoritmos para automatizar a geração dos *exploits*. Esses algoritmos utilizam as técnicas de análise de software conhecidas como *Taint Analysis* [10, 11] e *Constraint Solving* [11, 23, 24, 31, 32]. A Figura 4.1 mostra uma visão geral da solução.

Figura 4.1: Visão geral da solução proposta

A solução recebe como entrada o programa vulnerável e uma entrada que o faz terminar sua execução de forma inesperada, que pode ter sido descoberta por meio da execução de testes de *fuzzing*. Como saída, é produzido um *exploit* válido capaz de executar código arbritário no sistema alvejado.

4.2 Instrumentação Binária

Para obter as informações necessárias sobre o programa vulnerável, é executada uma análise dinâmica, que acontece em tempo de execução. A técnica escolhida para fazer essa análise neste trabalho é a instrumentação binária das instruções executadas, pois além de não requerer o acesso ao código-fonte do programa que está sendo analisado, permite que todos os detalhes necessários para a exploração sejam obtidos. Essa técnica funciona por meio da inserção de código adicional no fluxo de execução do programa, permitindo que sejam realizadas várias análises em tempo de execução.

Existem vários *frameworks* para instrumentação binária, sendo que os mais conhecidos são Valgrind [33], PIN [34] e DynamoRIO [35]. O *framework* escolhido para implementação da

ferramenta foi o PIN [34], da Intel, por ser multi-plataforma e oferecer uma performance superior aos outros *frameworks*. O PIN intercepta a execução da primeira instrução do executável e gera um novo código para o bloco de código que a contém. Ele então transfere o controle para esse novo bloco de código, que é quase idêntico ao original, mas contém o código de análise e garante que o PIN ganhe o controle novamente quando um desvio for feito para fora do bloco. Após ganhar o controle novamente, ele gera um novo bloco de código para o destino do desvio e continua a execução. Esse processo é repetido durante a execução do programa e todo o código gerado é armazenado na memória para que possa ser reusado posteriormente.

A instrumentação é usada como ponto de partida para a execução de todos os outros algoritmos da solução. É através dela que são inseridas as *callbacks* responsáveis por interceptar as chamadas de sistema para leitura de dados externos, propagar as informações da *Taint Analysis*, coletar as restrições adicionadas por cada instrução e resolver as restrições para gerar o *exploit* final. O algoritmo de instrumentação é apresentado em Algoritmo 1.

Algoritmo 1 instrumentAll()

$insertCallback(BEFORE, instruction, instrumentInstruction)$

$mallocRtn \leftarrow findRoutine(malloc)$
$freeRtn \leftarrow findRoutine(free)$
$insertCallback(BEFORE, mallocRtn, analyzeMallocBefore)$
$insertCallback(AFTER, mallocRtn, analyzeMallocAfter)$
$insertCallback(BEFORE, freeRtn, analyzeFreeBefore)$

$insertCallback(BEFORE, syscallEntry, registerSyscallArguments)$
$insertCallback(AFTER, syscallExit, taintInput)$

$insertCallback(BEFORE, threadStart, createRegistersState)$
$insertCallback(AFTER, threadFini, removeRegistersState)$

A instrumentação por instrução é responsável pela etapa de propagação da *Taint Analysis* e extração da semântica de cada instrução executada, como mostrado no Algoritmo 2.

Algoritmo 2 instrumentInstruction(ins)

if $hasPassedEntryPoint() \wedge hasStartedTaintPropagation()$ **then**
  if $isWriteInstruction(ins)$ **then**
    $insertCallback(BEFORE, ins, taintAnalysis)$
    $insertCallback(BEFORE, ins, convertToFormula)$
  end if
end if

4.3 Monitoramento de Alocações e Desalocações de Memória na *Heap*

Todas as alocações e desalocações de memória realizadas pelo programa vulnerável são registradas. Isso é feito por meio da interceptação das chamadas às funções *malloc()* e *free()*. Uma tabela é usada para armazenar os tamanhos de memória alocados para cada endereço durante a execução do programa.

Além disso, é a partir da interceptação de uma chamada à função *free()* que se inicia a análise realizada para a geração do *exploit*. Os algoritmos de monitoramento são mostrados em Algoritmo 3, 4 e 5.

Algoritmo 3 analyzeMallocBefore(address)

$allocationAddress \leftarrow address$

Algoritmo 4 analyzeMallocAfter(retCode)

$allocatedSize \leftarrow retCode$
$registerAllocation(allocationAddress, allocatedSize)$

Algoritmo 5 analyzeFreeBefore(address)

$registerDeallocation(address)$

$generateExploit(address)$

4.4 *Taint Analysis*

A técnica de *Taint Analysis* é usada para identificar quais bytes da entrada do programa devem ser alterados para que a exploração ocorra com sucesso. O processo tem início com a definição de um conjunto de dados iniciais que são marcados como *tainted*. Nesse trabalho, a fonte de dados *tainted* é obtida por meio da interceptação de chamadas do sistema como *read()*, *recv()* e *recvfrom()*. Após serem interceptadas, o endereço de destino na memória e a quantidade de bytes lidos são obtidos para que os endereços de memória correspondentes sejam marcados como *tainted*.

Somente após a definição de uma área de dados na memória marcada como *tainted* é que se inicia a instrumentação de cada instrução de máquina executada. À medida que as instruções são executadas, elas vão sendo analisadas para que os dados *tainted* sejam propagados. Se uma localização deriva diretamente de uma outra que está marcada como *tainted*, então ela

também será marcada como tal. Essa técnica tem sido usada em vários projetos onde é preciso saber se uma certa localização de memória ou registrador deriva de dados controlados pelo usuário [10, 11].

Uma forma de representação é usar dois conjuntos disjuntos, um para dados *tainted* e outro para dados *untainted*, sendo que os elementos de cada conjunto podem ser endereços de memória ou registradores. Como formalização, dada uma instrução i, uma localização de memória ou registrador x, um conjunto de dados *tainted* T e um conjunto de dados *untainted* U, as equações 4.1 e 4.2 definem as operações realizáveis.

$$x \in i_{dsts} \wedge \exists y (y \in i_{srcs} \wedge y \in T) \Leftrightarrow x \in T \tag{4.1}$$

$$x \in i_{dsts} \wedge \forall y (y \in i_{srcs} \wedge y \notin T) \Leftrightarrow x \notin T \tag{4.2}$$

Os dois algoritmos mostrados em Algoritmo 6 e 7 são responsáveis por detectar a leitura de dados externos para a memória do programa e marcá-la como *tainted*.

Algoritmo 6 registerSyscallArguments(syscall)

```
if readExternalData(syscall) then
    dstAddress ← getDstAddress(syscall)
    readData ← true
end if
```

Algoritmo 7 taintInput(syscall)

```
ret ← getReturnCode(syscall)

if isGoodReturnValue(ret) ∧ readData == true then
    sizeRead ← getSizeRead(syscall)
    taint(dstAddress, sizeRead)
    startedPropagation ← true
    readData ← false
end if
```

O Algoritmo 8 descreve os passos para a execução da *Taint Analysis* usados para rastrear os dados de entrada do usuário e extrair as restrições adicionadas por cada instrução executada.

4.5 Geração do *Exploit*

Após detectar uma chamada à função *free()*, é preciso verificar se todas as localizações na memória que devem ser controladas estão marcadas como *tainted*. Caso estejam, são ge-

Algoritmo 8 taintAnalysis(ins)

```
dsts ← getDestinationOperands()

for dst ∈ dsts do
  srcs ← getSources(dst)

  for src ∈ srcs do
    if isTainted(src) then
      taint(dst, src)
    end if
  end for
end for
```

radas fórmulas contendo cada uma das restrições necessárias para a exploração com a técnica *The House of Mind*, que foram descritas anteriormente. Após isso, são adicionadas fórmulas contendo as condições do caminho de execução que fizeram com que o programa chegasse no estado em que está, e que devem se manter para que a exploração ocorra com sucesso.

Finalmente, um resolvedor *Satisfiability Modulo Theories* (SMT) é usado para avaliar a fórmula, e verificar se é satisfatível. Caso seja, os dados de entrada são modificados para que essas condições sejam satisfeitas. Se não, a geração do *exploit* não é feita.

Caso uma chamada à função *free()* não satisfaça todas as condições de exploração requeridas, uma condição mínima é calculada e resolvida para que tal chamada não cause uma finalização prematura da execução do programa vulnerável. Isso se repete até que uma chamada que atenda à todos os requisitos de exploração seja encontrada.

Para otimizar o processo de solução das fórmulas, a entrada original é copiada e apenas os dados necessários são alterados, ao invés de gerar fórmulas para construir todo o arquivo. O algoritmo é mostrado em Algoritmo 9.

Algoritmo 9 generateExploit(freedAddress)

```
requiredLocations ← calculeRequiredLocations(freedAddress)
minimumRequiredLocation ← calculeMinimumRequiredLocation(freedAddress)

if isAllTainted(requiredLocations) then
  solveConstraintsAndModifyInput(requiredLocations)
else if isTainted(minimumRequiredLocation) then
  solveConstraintsAndModifyInput(minimumRequiredLocation)
end if
```

O Algoritmo 10 mostra como a fórmula que contém as restrições para que a exploração seja conseguida é criada. Ele itera sobre todas as localizações nas quais devem haver dados marcados como *tainted*, calcula os valores requeridos pelo método escolhido da técnica *The House of*

Mind, adiciona as restrições para que essas localizações tenham esses valores e adiciona as condições do caminho que fazem com que o fluxo de execução chegue no estado em que se encontra atualmente. Após isso, a fórmula final é passada para o solucionador de restrições, que diz se é satisfatível ou não, e se for, a solução encontrada por ele é analisada para que os bytes correspondentes da entrada do programa sejam alterados.

Algoritmo 10 solveConstraintsAndModifyInput(requiredTaintedLocations)

```
formula ← EmptyFormula()

for requiredLocation ∈ requiredTaintedLocations do
    requiredValue ← getRequiredValue(requiredLocation)
    formula ← addConstraints(requiredLocation, requiredValue)
    pc ← buildPathCondition(requiredLocation)
    formula ← createConjunct(formula, pc)
end for

if isSolvable(formula) then
    modifyInputData()
end if
```

5 *Implementação do Sistema*

5.1 Instrumentação Binária

O sistema foi implementado como uma extensão do *framework* PIN [34] e contém pouco mais de 4000 linhas de código C++.

5.2 *Taint Analysis*

A *Taint Analysis* é executada a nível de byte, onde para cada byte *tainted* é mantida uma estrutura para armazenar informações como quais são as origens de onde ele deriva e qual o seu relacionamento com elas. A complexidade da arquitetura x86 e o limite de tempo para realização do trabalho impediram a implementação da lógica de propagação para todas as instruções suportadas por ela, mas a lógica foi implementada para as principais instruções de transferência de dados (MOV, REP MOV, etc) e instruções aritméticas (ADD, SUB, etc). Para evitar falsos positivos, as instruções não suportadas são analisadas para que seus operandos de destino sempre sejam marcados como *untainted*.

5.3 *Constraint Solving*

As restrições são coletadas com base nas informações obtidas durante a *Taint Analysis*, e então formatadas de acordo com o padrão SMT-LIB [36]. Esse formato de representação foi escolhido por permitir que qualquer solucionador de restrições seja usado no projeto sem requerer alterações no código de formatação.

O solucionador de restrições escolhido foi o Yices [37], que suporta o padrão SMT-LIB e produz os resultados em um formato que pode ser analisado facilmente realizar as alterações na entrada do programa. Os dois exemplos a seguir mostram um trecho de um arquivo contendo as fórmulas geradas durante a análise de um programa e o um trecho do resultado de sua avaliação

pelo solucionador de restrições.

Trecho de arquivo no formato SMT-LIB

```
(benchmark test
:status unknown
:logic QF_BV

:extrafuns ((i16 BitVec[8]) (i17 BitVec[8])
            (i18 BitVec[8]) (i19 BitVec[8]) ... )

:assumption (bvuge i16 bv0[8])
:assumption (bvuge i17 bv0[8])
:assumption (bvuge i18 bv0[8])
:assumption (bvuge i19 bv0[8])

...

:formula (= i16 bv2[8])
:formula (= i17 bv4[8])
:formula (= i18 bv0[8])
:formula (= i19 bv0[8])

...
```

Resultado da avaliação do arquivo pelo Yices

```
sat
(= i16 0b00000010)
(= i17 0b00000100)
(= i18 0b00000000)
(= i19 0b00000000)
...
```

6 *Resultados Experimentais*

6.1 Ambiente de Testes

Os testes foram realizados no sistema operacional Ubuntu 9.10. Essa versão tem instalada por padrão a versão 2.10.1 da *glibc*, que é suscetível à exploração de *heap overflows* por meio dos dois métodos da técnica *The House of Mind*: método *unsorted_chunks()* e método *fastbinsY()*. Além disso, ela suporta todos os mecanismos de segurança descritos anteriormente, além de alguns outros como descrito em 22.

Dentre tais mecanismos, o ASLR e a *heap* não-executável são aqueles que dificultam a exploração das vulnerabilidades de *heap overflow* utilizando a técnica escolhida. Como esse trabalho não abrange a geração de *exploits* sob efeito desses mecanismos, eles foram desabilitados para a execução dos testes.

Para desabilitar o ASLR, pode-se usar o comando seguinte como usuário *root*:

```
sysctl -w kernel.randomize_va_space=0
```

Para desabilitar a *heap* não-executável ao compilar o programa vulnerável, deve-se usar o seguinte comando:

```
gcc vulnerable.c -o vulnerable -z execstack
```

6.2 Exemplo

6.2.1 Programa Vulnerável

Como programa vulnerável para testes, foi escolhido aquele criado por k-sPecial [18] para demonstração da exploração feita em seu artigo. Esse programa é usado como exemplo na

Listagem 6.1: Programa vulnerável à *heap overflow* criado por *k-sPecial*

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

int main (void) {
    char *ptr  = malloc(1024);          /* First allocated chunk */
    char *ptr2;                         /* Second chunk          */
    /* ptr & ~(HEAP_MAX_SIZE-1) = 0x08000000 */
    int heap = (int)ptr & 0xFFF00000;
    _Bool found = 0;

    printf("ptr found at %p\n", ptr);  /* Print address of first chunk */

    // i == 2 because this is my second chunk to allocate
    for (int i = 2; i < 1024; i++) {
        /* Allocate chunks up to 0x08100000 */
        if (!found && (((int)(ptr2 = malloc(1024)) & 0xFFF00000) == \
                                    (heap + 0x100000))) {
            printf("good heap allignment found on malloc() %i (%p)\n", i,
                ptr2);
            found = 1; /* Go out */
            break;
        }

    }

    malloc(1024); /* Request another chunk: (ptr2 != av->top) */
    /* Incorrect input: 1048576 bytes */
    fread (ptr, 1024 * 1024, 1, stdin);

    free(ptr);   /* Free first chunk  */
    free(ptr2);  /* The House of Mind */

    printf("bye\n");
    return(0);   /* Bye */
}
```

Listagem 6.2: Variação do programa vulnerável que introduz uma estrutura de repetição contendo operações aritméticas

```
1     ...
2
3     fread (ptr, 1024 * 1024, 1, stdin);
4
5     for (int i = 0; i < 1024; i++) {
6         *(ptr + i) = *(ptr + i) + 10;
7     }
8
9     free(ptr);    /* Free first chunk  */
10    free(ptr2);   /* The House of Mind */
11
12    printf("bye\n");
13
14    ...
```

Listagem 6.3: Variação do programa vulnerável que introduz uma estrutura de decisão

```
1     ...
2
3     fread (ptr, 1024 * 1024, 1, stdin);
4
5     free(ptr);    /* Free first chunk  */
6
7     if (*(ptr + 0x300) == 0x25)
8         free(ptr2);  /* The House of Mind */
9
10    printf("bye\n");
11
12    ...
```

maioria das fontes sobre a técnica de exploração *The House of Mind* e simula o comportamento de uma aplicação vulnerável à *heap overflow*. O código é mostrado em Listagem 6.1.

Além do teste desse programa, algumas variações dele foram criadas por meio da inserção de estruturas para simular operações típicas realizadas em programas reais. Uma delas introduz um laço contendo comandos para alterar parte dos dados originais lidos da entrada do programa usando operações aritméticas (Listagem 6.2). Outra contém uma estrutura de decisão que só faz a chamada à função *free()* que permite a exploração da vulnerabilidade caso a condição seja satisfeita (Listagem 6.3).

6.2.2 Resultados

A ferramenta desenvolvida foi testada contra o programa vulnerável mais simples e foi capaz de gerar um *exploit* válido em tempo curto. A partir de uma entrada que provoca a

Programa vulnerável	Tempo para solucionar	*Exploit* funcional
Simples	<1s	Sim
Com laço e operações aritméticas	<1s	Sim
Com desvio condicional	<1s	Sim

Tabela 6.1: Tabela de resultados

quebra do processo do program vulnerável, é construído um *exploit* por meio da modificação dos bytes necessários para que as restrições impostas pela técnica de exploração escolhida sejam satisfeitas. O *exploit* gerado pode ser baixado na referência 38.

Além disso, a ferramenta foi capaz de gerar *exploits* válidos para as variações do programa vulnerável mais simples. Os tempos de geração dos *exploits* podem ser vistos na Tabela 6.1, que mostra os resultados obtidos no teste da ferramenta.

7 *Conclusões*

7.1 Conclusões

A solução proposta neste trabalho para o problema da geração de *exploits* para vulnerabilidades de *heap overflow* usando a técnica *The House of Mind*, apresentado no capítulo de *Introdução*, permitiu a criação de *exploits* de forma automática para alguns programas vulneráveis utilizados nos testes. Inicialmente, foi mostrado como o *exploit* foi gerado para o programa vulnerável mais simples, que apenas copia os dados lidos na chamada do sistema para a memória sem alterá-los. Depois, a solução mostrou ser eficaz em programas que fazem alterações nesses dados e/ou que executam desvios no fluxo de execução com base nos valores lidos, que são atividades típicas de um *parser* de arquivos ou de máquinas de estado para reconhecimento de protocolos de rede, encontrados em diversas aplicações que fazem o processamento de dados externos.

Um ponto negativo do trabalho foi o fato de não terem sido realizados testes com programas vulneráveis reais, o que seria uma oportunidade de demonstrar com maior propriedade a eficácia da solução proposta. Apesar disso, acredita-se que os programas testados apresentam operações comumente executadas em aplicações que fazem o processamento de dados externos.

7.2 Contribuições do Trabalho

7.2.1 Atualização da Técnica de Exploração *The House of Mind*

Neste trabalho foram apresentadas algumas técnicas de exploração de vulnerabilidades de *Heap Overflow*, com ênfase na técnica *The House of Mind*. Até a presente data, existem poucos recursos em português sobre o assunto e tais recursos não abordam o tema com o mesmo nível de detalhamento apresentado.

Além disso, a técnica foi atualizada para uso em versões mais novas da *glibc*, que estão presentes nas últimas versões das distribuições Linux. Vale ressaltar que essas atualizações não

foram apresentadas anteriormente na literatura.

7.2.2 Aplicação de Técnicas de Análise de Software na Construção de *Exploits* de *Heap Overflow* usando a Técnica *The House of Mind*

Foi realizado um estudo sobre técnicas de análise de software para selecionar quais delas poderiam ser usadas na automatização da criação de *exploits* para vulnerabilidades de *heap overflow* por meio da técnica *The House of Mind*. A literatura apresenta estudos sobre automatização da exploração de vulnerabilidades de *stack overflow*, mas ainda há grande espaço para estudos sobre automatização da exploração de *heap overflows* e outras vulnerabilidades.

Este trabalho descreve o funcionamento dessas técnicas de análise e como podem ser aplicadas na solução do problema.

7.2.3 Algoritmos para Automatizar a Geração de *Exploits*

A solução proposta usa algoritmos que descrevem os passos necessários para a geração automática dos *exploits*. Esses algoritmos são específicos para automatizar a criação de *exploits* por meio da técnica *The House of Mind* e fazem uso de várias técnicas de análise de software. Entretanto, devido às similaridades da técnica abordada com outras técnicas disponíveis para exploração da *heap*, acredita-se que pouca modificação seja necessária para lidar com elas.

7.2.4 Validação dos Algoritmos

Os algoritmos desenvolvidos foram implementados e testados em programas vulneráveis de amostra e os resultados obtidos foram apresentados.

7.3 Trabalhos Futuros

Como trabalho futuro, um tópico interessante para pesquisa é analisar a relação entre os dados de entrada do programa e as alocações e desalocações de memória que são feitas na *heap*, como discutido em 32. A integração desse tipo de análise na solução proposta poderia aumentar a taxa de sucesso na geração de *exploits* para diversos tipos de programas vulneráveis.

Um outro trabalho futuro é investigar como a solução apresentada se comporta quando as proteções do sistema contra exploração são ativadas e descobrir quais modificações são necessárias para lidar com elas. Uma forma de lidar com o *ASLR* seria usar força bruta para gerar

um grande conjunto de *exploits*, cada um tomando como base um dos endereços de memória possíveis.

Referências Bibliográficas

[1] MILLER, C. The legitimate vulnerability market. 2007.

[2] AMINI, P. Mostrame la Guita! Adventures in Buying Vulnerabilities. In: EKOPART, 2009, Buenos Aires - Argentina. Argentina, 2009.

[3] BRUMLEY, D. et al. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[4] MEDEIROS, J. *Automated Exploit Development*: The future of exploitation is here. [S.l.], 2007.

[5] HEELAN, S. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. Dissertação (Mestrado) — University of Oxford, 2009.

[6] ROHLF, C. *Glibc 2.11 stops the House of Mind*. Disponível em: <http://em386.blogspot.com/2010/01/glibc-211-stops-house-of-mind.html>.

[7] PHANTASMAGORIA, P. Malloc Maleficarum. *Bugtraq*, 2005. Disponível em: <http://seclists.org/bugtraq/2005/Oct/0118.html>. Acesso em: 18 de abril de 2010.

[8] WAISMAN, N. Apology of 0days. In: H2HC, 2008, Brasil. Brasil, 2008.

[9] AVGERINOS, T. et al. AEG: Automatic Exploit Generation. In: *Network and Distributed System Security Symposium*. [S.l.: s.n.], 2011.

[10] NEWSOME, J.; SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[11] SCHWARTZ, E. J.; AVGERINOS, T.; BRUMLEY, D. All You Ever Wanted to Know About Dynamic Taint Analysis and Foward Symbolic Execution (but might have been afraid to ask). *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[12] DOWD, M.; MCDONALD, J.; SCHUH, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. United States: [s.n.], 2007.

[13] ONE, A. Smashing The Stack For Fun And Profit. *Phrack*, v. 49, 1996. Disponível em: <http://www.phrack.com/issues.html?issue=49&id=14#article>. Acesso em: 18 de abril de 2010.

[14] DESIGNER, S. JPEG COM Marker Processing Vulnerability in Netscape Browsers. 2000. Disponível em: <http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt>. Acesso em: 18 de abril de 2010.

[15] MAXX. Vudo - An object superstitiously believed to embody magical powers. *Phrack*, v. 57, 2001. Disponível em: <http://www.phrack.org/issues.html?issue=57&id=8#article>. Acesso em: 18 de abril de 2010.

[16] ANONYMOUS. Once upon a free()... *Phrack*, v. 57, 2001. Disponível em: <http://www.phrack.org/issues.html?issue=57&id=9#article>. Acesso em: 18 de abril de 2010.

[17] JP. Advanced Doug lea's malloc exploits. *Phrack*, v. 61, 2003. Disponível em: <http://www.phrack.org/issues.html?issue=61&id=6#article>. Acesso em: 18 de abril de 2010.

[18] K-SPECIAL. The House of Mind. *.aware EZine Alpha*, 2007. Disponível em: <http://www.awarenetwork.org/etc/alpha/?x=4>. Acesso em: 18 de abril de 2010.

[19] G463. The use of set_head to defeat the wilderness. *Phrack*, v. 64, 2007. Disponível em: <http://www.phrack.org/issues.html?issue=64&id=9#article>. Acesso em: 18 de abril de 2010.

[20] BLACKNGEL. Malloc Des-Maleficarum. *Phrack*, v. 66, 2009. Disponível em: <http://www.phrack.org/issues.html?issue=66&id=10#article>. Acesso em: 18 de abril de 2010.

[21] HUKU. Yet another free() exploitation technique. *Phrack*, v. 66, 2009. Disponível em: <http://www.phrack.org/issues.html?issue=66&id=6#article>. Acesso em: 18 de abril de 2010.

[22] COOK, K. *Security/Features - Ubuntu Wiki*. Disponível em: <https://wiki.ubuntu.com/Security/Features>.

[23] CADAR, C. et al. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution. *Proceedings of the ACM Conference on Computer and Communications Security*, October 2006.

[24] MOLNAR, D. A. *Dynamic Test Generation for Large Binary Programs*. Tese (Doutorado) — University of California, Berkeley, 2009.

[25] LIN, Z.; ZHANG, X.; XU, D. Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach. *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2008.

[26] GRENIER, L.; LIN0XX. Byakugan: Increase your sight. In: TOORCON, 2007. [S.l.], 2007.

[27] BUCHANAN, E. et al. When good instruction go bad: Generalizing return-oriented programming to RISC. *Proceedings of the ACM Conference on Computer and Communications Security*, 2008.

[28] CHECKOWAY, S.; SHACHAM, H. Escape From Return-Oriented Programming: Return-oriented Programming without Returns (on the x86). 2010.

[29] GRENIER, L. *DEP and Heap Sprays*. Disponível em: <http://vrt-sourcefire.blogspot.com/2009/12/dep-and-heap-sprays.html>.

[30] ZOVI, D. D. Practical Return-Oriented Programming. In: SOURCE BOSTON, 2010, Boston. Boston, 2010.

[31] KING, J. C. Symbolic execution and program testing. *Commun. ACM*, ACM, v. 19, n. 7, p. 385–394, 1976.

[32] SAXENA, P. et al. *Loop-Extended Symbolic Execution on Binary Programs*. [S.l.], March 2009.

[33] NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of ACM SIGPLAN 2007*, 2007.

[34] LUK, C.-K. et al. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, p. 190–200, June 2005. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1064978.1065034>.

[35] BRUENING, D. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Tese (Doutorado) — MIT, 2004.

[36] RANISE, S.; TINELLI, C. The SMT-LIB Format: an Initial Proposal. *Proceedings of PDPAR'03*, 2003.

[37] DUTERTRE, B.; MOURA, L. de. The YICES SMT Solver. Disponível em: <http://yices.csl.sri.com/tool-paper.pdf>.

[38] SILVA, G. Q. *Exploit gerado para o programa vulnerável mais simples*. Disponível em: <https://sites.google.com/site/gquadrossilva/files/exploit>.