# Memory Eye

Yoann Guillot

Sogeti - ESEC

**Résumé** L'analyse de programmes inconnus passe traditionnellement par le désassemblage du code et l'étude du listing assembleur ainsi généré. Nous proposons ici une approche complémentaire basée sur l'analyse de la zone de mémoire dynamique (heap), qui s'intéresse uniquement aux données. Cette approche permet de s'abstraire de modifications du code, notamment lors de mises à jour du logiciel, ou encore lors de l'analyse d'une version pour une architecture différente. Cet article détaille principalement l'étape initiale d'énumération des zones allouées. Nous concluons en appliquant cette approche sur un jeu de simulation : Dwarf Fortress.

## 1 La mémoire

### 1.1 Taxinomie

Du point de vue du programmeur, une variété d'espace de stockage d'information est disponible.

Outre les sources externes, comme les fichiers ou le réseau, différentes classes d'espace de stockage sont offertes au sein du programme en cours d'exécution.

On peut citer les **registres** du processeur, qui sont généralement considérés comme volatiles, et ne stockent une information que le temps de travailler dessus. L'espace offert par ceux-ci est très réduit (plus ou moins selon l'architecture considérée). Un processeur de la famille *Ia32* aura environs $16 * 8$ octets disponibles dans les registres « généraux », et jusqu'à $16 * 32$ octets supplémentaires en allant fouiller dans les registres *ymm* ; ce qui reste extrêmement limité.

La ressource principale reste la mémoire vive.

Celle-ci est divisée logiquement en différentes sections, que l'on considère généralement disjointes et indépendantes, bien qu'en pratique il ne s'agisse que d'un seul et même espace d'adressage virtuel. Je veux parler de la **pile**, de la zone de **mémoire dynamique**, et de la **mémoire statique**.

En pratique, la pile sert à stocker des données locales à une fonction ; celles-ci disparaissent dès lors que la fonction retourne. La mémoire statique contient les variables globales, définies au moment de la compilation du programme. Celles-ci sont disponibles et accessibles à tout instant de l'exécution, mais restent en nombre limité. Enfin, la mémoire dynamique rassemble tout le reste. C'est sur celle-ci que nous allons nous concentrer dans cet article.

### 1.2 Le Heap

**Heap** (ou **tas** en langue de Molière) est le nom donné à la zone mémoire dédiée aux allocations dynamiques, typiquement gérées par les fonctions de la bibliothèque standard **malloc()** et **free()**.

Un programme peut ainsi demander une zone de mémoire d'une certaine taille au système d'exploitation, l'utiliser pendant un certain temps, et finalement la libérer quand il n'en a plus l'utilité. *malloc* demande l'allocation d'une zone d'une certaine taille en octets, et renvoie un pointeur vers celle-ci (le système peut fournir une zone de taille supérieure). Ce segment de l'espace d'adressage du processus est traditionnellement nommé **chunk**. *free* permet de libérer cette ressource.

En pratique, l'allocation de mémoire au niveau du système d'exploitation se fait sur la base de pages entières (4096 octets). La bibliothèque standard du système fournit une interface plus conviviale, qui permet de gérer des allocations de taille arbitraire, et inclue toute la comptabilité permettant de gâcher le moins de mémoire possible. Ainsi deux allocations de 8 octets seront généralement servies par des pointeurs situés dans une seule page physique.

Les routines d'allocation sont des éléments critiques quant à la performance des processus, aussi les algorithmes d'allocation et de libération sont souvent complexes, de manière à avoir un temps d'exécution minimal, tout en assurant une utilisation optimale de la mémoire physique. La requête de nouvelles pages au noyau du système est considérée comme coûteuse, aussi les bibliothèques vont-elles souvent implémenter un mécanisme de cache de la mémoire libérée, afin de la réutiliser en évitant de la rendre puis la re-demander au noyau quelques instants plus tard.

De plus, afin d'améliorer les performances des processus multi-threadés, la bibliothèque standard va généralement allouer les requêtes de différents threads dans des parties distinctes de l'espace d'adressage. Cela permet d'éviter qu'un thread, en faisant une requête d'allocation, et donc en manipulant les données de controle du heap, ne bloque un autre thread qui voudrait faire une requête similaire en même temps. On va alors se retrouver avec l'ensemble des données de contrôle dupliquées, de manière à créer deux heaps (ou plus) au sein du même espace d'adressage, qui pourront alors servir des requêtes indépendemment.

## 2 Énumération du Heap

« *L'image montre un heap et des chunks. Les chunks entourent le heap.* »

Dans cet article, nous nous plaçons du point de vue d'un développeur qui débogue un programme en cours d'exécution. Nous disposons donc de primitives permettant de

controller l'exécution des différents threads d'exécution du processus cible, notamment la possibilité de les stopper ; ainsi que d'un accès illimité à l'espace d'adressage virtuel de ce processus.

Notre premier objectif est, à partir de ces primitives, d'énumerer l'ensemble des **chunks** en cours d'utilisation par le programme. Or cette énumération n'a généralement pas été envisagée lors de l'écriture des routines de gestion du tas. Nous allons donc devoir retrouver et interpreter les structures internes de l'allocateur ; ce qui présente un certain nombre de difficultés.

La littérature actuelle sur ce sujet est plutôt restreinte ; la plupart des articles se concentant sur l'aspect *exploitation d'un buffer overflow dans le heap*. On y retrouve toutefois la description de structures et d'algorithmes qui ne sont pas documentés ailleurs, et qui nous seront d'une aide précieuse.

## 2.1 Linux/glibc

Sous Linux, nous nous intéresserons à l'allocateur de la *glibc* version *2.11*, basé sur *ptmalloc2*[1].

Dans ce cadre, la taille et l'alignement des chunks est toujours égal à 2x la taille d'un pointeur (*i.e.* 8 octets en x86, et 16 octets en x64).

Un chunk alloué est précédé d'un entier qui indique la taille du chunk, ainsi que 3 bits :

- `PREV_INUSE` (1), indique si le chunk précédent est alloué
- `IS_MMAPPED` (2), indique si le chunk est alloué via `mmap()`
- `NON_MAIN_ARENA` (4), indique que le chunk dépend d'une arena secondaire

Ils sont stockés dans les 3 bits de poids faible de la taille du chunk, celle-ci étant toujours un multiple de 8.

Si le chunk précédent est libre, la taille de celui-ci est stockée juste avant.

On peut ainsi décrire le chunk grâce à la structure suivante (le programme utilisateur recoit l'adresse du champ `data` comme valeur de retour de `malloc()`) :

```
struct malloc_chunk {
    uintptr_t prev_size;
    uintptr_t size;
    char data[];
};
```

Quand le chunk précédant est alloué, le champ `prev_size` fait partie du buffer utilisateur, et peut donc contenir une valeur arbitraire.

Le champ `size` indique la taille du chunk courant en octets, du `prev_size` au `prev_size` du chunk qui suit en mémoire (une fois masqué les bits de meta-data). Celui-ci correspond à la taille passée en argument de `malloc()`, plus 4, arrondie au multiple de 8 supérieur (8 et 16 en 64-bit).

Les chunks de petite taille (la limite varie dynamiquement entre 512ko et quelques Mo) sont alloués dans un espace mémoire dédié nommé `arena`. Les chunks au-delà de cette limite sont alloués individuellement via l'appel `mmap()`.

**Le heap** Par défaut la libc utilise une seule arena (`main_arena`), mais il est possible de créer des arena supplémentaires. Sous linux, l'arena principale est gérée à travers l'appel système `sbrk()`, et peut grandir. Les arenas secondaires sont allouées via `mmap()`, et ont une taille fixe.

Les arenas sont décrites par la structure suivante :

```
#define NFASTBINS 10
#define NBINS 128
#define BINMAPSIZE (NBINS/32)

struct malloc_state { // size = 0x44C
    mutex_t mutex;
    int flags;
#if THREAD_STATS
    long stat_lock_direct;
    long stat_lock_loop;
    long stat_lock_wait;
#endif
    void *fastbinsY[NFASTBINS];
    void *top;
    void *last_remainder;
    void *bins[NBINS * 2 - 2];
    unsigned int binmap[BINMAPSIZE];
    struct malloc_state *next;
#ifdef PER_THREAD
    struct malloc_state *next_free;
#endif
    uintptr_t system_mem;
    uintptr_t max_system_mem;
};
```

L'arena principale est décrite par une de ces structures définie statiquement par la libc. Les arenas secondaires commencent par une structure qui permet entre autres de retrouver le `malloc_state` correspondant.

Cette structure est :

```
struct _heap_info {
    struct malloc_state *ar_ptr;
    struct _heap_info *prev;
    size_t size;
    size_t mprotect_size;
};
```

Détaillons un peu la structure décrivant l'arena. On y retrouve une liste de *fastbins*, qui sont de petits chunks récemment libérés. Chaque élément de ce tableau est le pointeur de début d'une liste chaînée de chunks d'une taille donnée. L'index $i$ correspond aux chunks de taille $8 * (i + 2)$ (16x en 64-bit). Ces chunks ont la particularité d'être toujours marqués comme alloués (le bit `PREV_INUSE` correspondant reste à 1).

`Bins` est un tableau de liste doublement chaînée qui recense tous les chunks libres. Ces listes sont décrites en détail dans la littérature, et ne nous concernent pas.

`Top` est le seul pointeur utile vers un chunk que nous trouvons dans cette structure. Il s'agit en particulier d'un pointeur vers le dernier chunk existant au sein de la zone réservée à l'allocation mémoire. Celui-ci est toujours libre, et c'est en le fragmentant que `malloc()` peut créer de nouveaux chunks à renvoyer à l'utilisateur. Dans le cadre du heap principal, si *top* devient trop petit, une requête est faite au système d'exploitation pour augmenter la taille du heap disponible.

Or d'après la structure des chunks que nous avons vue précedemment, on peut passer d'un chunk au chunk suivant, mais il s'agit d'un sens unique. Pour retrouver le début de la zone d'allocation, et donc le premier chunk, nous pouvons utiliser le champs `system_mem` : celui-ci indique la taille totale de la zone d'allocation. Ainsi, à partir du *top* chunk, nous récupérons sa taille, ce qui nous mène à la fin de la zone, et nous pouvons ensuite retrancher `system_mem` pour retrouver le début. Dans le cas du heap principal, nous y trouverons directement le premier chunk, et donc la liste complète (en passant au suivant via le champs `size` du chunk). Dans le cas d'un heap secondaire, il nous faut déréferencer le premier pointeur, afin de trouver le descripteur de l'arena, qui est placé juste après ; nous connaissons la taille de cette structure, et le premier chunk lui est accolé.

À partir de l'arena principale, nous pouvons également trouver toutes les arenas secondaires en parcourant la liste via le pointeur `next`.

**Extraction de l'adresse de `main_arena`** Malheureusement, la structure décrivant l'arena principale n'est pas publique ; elle se trouve quelquepart au milieu de la section `.bss` de la libc.

Pour la retrouver, nous pouvons rechercher une fonction qui la déréference, et l'analyser pour en extraire le pointeur. Une fonction qui s'y prête particulièrement est `malloc_trim()`. Elle commence ainsi :

```
int public_mTRIm(size_t s) {
  int result = 0;
  if(__malloc_initialized < 0)
    ptmalloc_init ();
  mstate ar_ptr = &main_arena;
  do {
      (void) mutex_lock (&ar_ptr->mutex);
  /* ... */
```

Elle est intéressante car :

1. C'est une fonction exportée.
2. Elle utilise l'adresse de `mutex`, qui est l'adresse de la structure elle-même.
3. Cette adresse est passée à `mutex_lock()`, qui a une implémentation particulière.
4. Cet appel vient très tôt dans la fonction, si l'on ignore le test d'initialisation.

Concernant le point 3, la fonction `mutex_lock()` est implémentée au niveau binaire en utilisant l'instruction assembleur `cmpxchg` ; elle est donc très simple à repérer au sein du code de la fonction. Le premier argument de cette instruction contient l'adresse du champs `mutex`, qui est également l'adresse de la structure elle-même puisqu'il s'agit de son premier membre. Cette heuristique fonctionne bien pour l'architecture Ia32, aussi bien en 32 qu'en 64-bit.

Nous utilisons les primitives de débugging du framework *metasm*[2]. Nous allons mettre à profit les fonctionnalités d'émulation de code pour retrouver la valeur du pointeur qui nous intéresse. (La libc étant une librairie ELF, elle est compilée de manière indépendante de la position, ce qui rend l'extraction du pointeur non triviale si l'on se base simplement sur le listing de la fonction).

L'adresse de la fonction est déduite des symboles exportés par le module *libc*, dont l'adresse est disponible dans le fichier `/proc/<pid>/maps`.

La portion de script suivant nous permet de récuperer l'adresse de la `main_arena` :

```
dasm.disassemble_fast(malloc_trim)

if dasm.block_at(malloc_trim).list.last.opcode.name == 'call'
    # getip-style x86 PIC function
    dasm.disassemble dasm.block_at(malloc_trim).to_normal.first
end

cmpxchg = dasm.decoded.values.find { |di| di.kind_of?(DecodedInstruction)
    and di.opcode.name == 'cmpxchg' }
raise 'no cmpx' if not cmpxchg

main_arena_ptr = dasm.backtrace(cmpxchg.instruction.args.first.symbolic.
    pointer, cmpxchg.address)

if main_arena_ptr.length == 1
    main_arena_ptr = main_arena_ptr[0].reduce
end
raise 'cant find mainarena' if not main_arena_ptr.kind_of?(Integer)
main_arena_ptr += libc_base_addr
```

On commence par désassembler le code de `malloc_trim`, sans rentrer dans les sous-fonctions (`ptmalloc_init()` est particulièrement complexe[1]). En x86, le compilateur va générer un appel de fonction particulier pour récuperer l'adresse effective de chargement du code ; cet appel sera dans le premier bloc d'instructions. Dans cette situation, on demande le désassemblage de cette fonction, afin que l'émulateur de code puisse fonctionner correctement et gère le code indépendant de la position.

Enfin, on récupére l'instruction `cmpxchg`, et on demande la résolution de la valeur utilisée comme pointeur dans le premier argument.

Si cette résolution réussie, la valeur du pointeur nous permet alors de parcourir les arenas, et l'ensemble des chunks qu'elles contiennent.

De plus, pour chaque chunk marqué alloué, nous sommes en mesure de parcourir la liste des fastbins pour déceler certains chunks qui sont en réalité libres.

**Mmap** Enfin, il nous reste à récuperer les chunks de grande taille, qui ont été alloués directement via `mmap()`.

Ceux-ci ne sont référencés nulle part ; nous allons donc parcourir tous les mappings anonymes de l'espace d'adressage, et utiliser une heuristique basé sur la structure décrivant les chunks.

Pour un chunk mmappé, le premier dword (`prev_size`) sera nul, et le second contiendra la taille du chunk. Cette taille devra être un multiple de la taille d'une page (4096), et avoir uniquement le bit `IS_MMAPED` (2) positionné.

1. « the disassembler has gone stark raving mad ! »

Attention, une seule ligne décrivant un mapping anonyme dans `proc/<pid>/maps` peut contenir une séquence de mappings différents consécutifs en mémoire. Pour chacune de ces lignes, il faudra donc scanner chaque page concernée.

Au niveau de la taille du chunk, une demande d'allocation de taille $n$ sera transformée en $n + 12$ (+24 en x64), et arrondie au multiple de 4096 supérieur.

**Approximation** Cette méthode complexe nous permet de lister précisément tous les chunks, ainsi que leur taille (arrondie).

Toutefois une seconde méthode, beaucoup plus simple, nous permet d'avoir une vision plus approximative du tas.

La zone utilisée pour l'arena principale étant gérée, sous Linux, par l'appel système `sbrk()`, celle-ci sera marquée distinctement du label `[heap]` dans le listing des mappings du processus.

À partir de là, nous sommes en mesure d'énumerer les chunks du heap principal, et ce même si nous n'avions pas l'heuristique permettant de retrouver le descripteur de la `main_arena` en mémoire. La technique pour les chunks alloués via `mmap()` reste utilisable ; par contre nous n'accederions pas au listing des arenas secondaires, et ne pourrions pas non plus éliminer les chunks référencés dans les *fastbins* (qui sont toujours marqués comme alloués), ce qui pourrait conduire à des résultats un peu moins précis, mais potentiellement intéressants (comme on dit, *un tas vaut mieux que deux « tu le désassembles, tu le déréferences et tu l'auras »*).

### 2.2 Windows XP

Sous Windows, la situation est un peu plus claire.

L'API `CreateToolhelp32Snapshot()` permet de lister les heaps d'un processus, ce qui nous évite d'ores et déjà le recours à des manœvres désespérées comme le désassemblage à la volée d'une méthode en espérant que les planètes seront correctement alignées et que l'on pourra en extraire un offset vers une structure plus ou moins connue.

Ensuite, même si les structures décrivant un heap ne sont pas officiellement documentées, on peut les récupérer dans les symboles de déboguage du kernel. Enfin, celles-ci ont été largement analysées par la communauté[3], ainsi que les algorithmes qui les manipulent[4]. Ces structures sont par ailleurs un peu mieux pensées, on y retrouve notamment un pointeur direct sur le premier chunk alloué.

L'allocateur mémoire Windows est divisé en deux parties. La première, dite *backend*, est fixe, et similaire à l'allocateur de la libc décrit précédemment. La partie correspondant aux *fastbins* est déportée dans la partie *frontend*, qui est paramétrable.

**Chunks** Sous Windows XP, les chunks mémoire ont la structure suivante :

```
struct _HEAP_ENTRY { // size = 8
    UINT16 Size;
    UINT16 PrevSize;
    UINT8 SmallTagIndex;
    UINT8 Flags;
    UINT8 UnusedBytes;
    UINT8 SegmentIndex;
#ifdef _WIN64
    VOID *pad;
#endif
    char data[];
};

struct _HEAP_VIRTUAL_ALLOC_ENTRY { // size = 0x20
    struct _LIST_ENTRY Entry;
    struct _HEAP_ENTRY_EXTRA ExtraStuff;
    ULONG32      CommitSize;
    ULONG32      ReserveSize;
    struct _HEAP_ENTRY BusyBlock;
};
```

Comme sous linux, le programme utilisateur reçoit un pointeur sur le membre `data`.

`Size` représente la taille du chunk courant, en multiples de la taille du `HEAP_ENTRY`. Ainsi en 32-bit, la taille du chunk est $8 * Size$, 16 en 64-bit.

Le champ `UnusedBytes` indique la différence entre la taille allouée (multiple de 8) et la taille requise par l'utilisateur ; ce qui nous permet de savoir exactement combien le programme a demandé à l'origine.

Le champ `Flags` indique l'état du chunk, comme sous linux. Les valeurs suivantes existent :

- `HEAP_ENTRY_BUSY` (1), indique si le chunk est alloué.
- `HEAP_ENTRY_EXTRA_PRESENT` (2), indique si certaines méta-data (de debugging) sont disponibles pour ce chunk. Elles sont ajoutées à la fin du chunk.
- `HEAP_ENTRY_FILL_PATTERN` (4), indique si l'allocateur doit réécrire la zone de contenu au moment de l'allocation/libération pour détecter les erreurs mémoires (activé pour le déboguage).
- `HEAP_ENTRY_VIRTUAL_ALLOC` (8), indique un chunk de grande taille, alloué directement via `VirtualAlloc()`.
- `HEAP_ENTRY_LAST_ENTRY` (0x10), indique que ce chunk est le dernier du segment courant.

**Backend** Un heap est décrit par une structure principale, qui référence différents segments (64 sous Windows XP), qui contiennent eux-mêmes les chunks à proprement parler.

Initialement un heap contient un unique segment de taille raisonnable (et fixe). Une fois que celui-ci est épuisé, un nouveau segment est alloué, de taille le double du précédent, et est utilisé pour satisfaire les requêtes du programme. Si celui-ci est épuisé, un nouveau segment, à nouveau deux fois plus grand, est créé, et ainsi de suite.

La structure de contrôle du heap est inclue au début du premier segment. On notera également que chaque segment n'est pas intégralement alloué en mémoire réelle ; le mécanisme de réservation et de *commit* de mémoire virtuelle permet à Windows de gérer finement l'utilisation de la mémoire physique en assurant une répartition maitrisée dans l'espace d'adressage virtuel.

Le rôle du backend est la gestion des segments, l'allocation de chunks dans ceux-ci, et la gestion des chunks libérés (notamment la fusion de chunks libres adjacents).

La structure décrivant le heap est celle-ci (seuls les champs intéressants sont notés) :

```
struct _HEAP { // size = 0x588
[...]
    struct _LIST_ENTRY VirtAllocdBlocks;  // 0x050
    struct _HEAP_SEGMENT* Segments[64];   // 0x058
[...]
    VOID*         FrontEndHeap;           // 0x580
    UINT16        FrontHeapLockCount;     // 0x584
    UINT8         FrontEndHeapType;       // 0x586
[...]
};

struct _HEAP_SEGMENTS { // size = 0x3c
[...]
    struct _HEAP_ENTRY* FirstEntry;       // 0x20
[...]
};
```

Le frontend est paramétrable, en fonction de la valeur du champs `FrondEndHeapType`.

- À 0, il est désactivé, et le backend gère toutes les allocations directement.
- À 1, le frontend **LookAside** gère les chunks entre 0 et 1024 octets.
- À 2, le frontend **LowFragmentationHeap** gère les chunks entre 0 et 16ko (depuis XP SP3 seulement).

**LookAside frontend** Le frontend `LookAside` est similaire aux *fastbins* de la glibc.

Lorsqu'un chunk de taille approprié est libéré, le frontend peut choisir de le prendre en charge. Dans ce cas, il va le stocker tel quel dans une liste chaînée de chunks de la même taille. Le chunk sera toujours marqué comme alloué, afin d'éviter que le backend ne le fusionne avec d'autres chunks libres adjacents.

La structure décrivant le lookaside est très simple, il s'agit d'un tableau de 128 entrées, chaque entrée correspondant à une taille donnée (pour chaque multiple de 8 jusqu'à 1024).

```
struct _HEAP_LOOKASIDE { // size = 0x30
    struct _SLIST_HEADER {
        struct _SINGLE_LIST_ENTRY {
            struct _SINGLE_LIST_ENTRY *Next;
        };
        UINT16   Depth;
        UINT16   Sequence;
    } ListHead;
    UINT16       Depth;
    UINT16       MaximumDepth;
    ULONG32      TotalAllocates;
    ULONG32      AllocateMisses;
    ULONG32      TotalFrees;
    ULONG32      FreeMisses;
    ULONG32      LastTotalAllocates;
    ULONG32      LastAllocateMisses;
    ULONG32      Counters[2];
    UINT8        _PADDING0_[0x4];
};
```

Le seul élément intéressant ici est le premier, qui constitue en fait la tête d'une liste chaînée de chunks libres gardés de côté. Les autres champs servent à maintenir des statistiques sur l'utilisation du cache, afin d'optimiser dynamiquement la taille optimale des listes, etc.

Dans le cas du `LookAside` frontend, le champs `FrontEndHeap` pointe directement sur le tableau contenant ces 128 structures.

Ce frontend est activé par défaut pour les heaps.

Dans cette situation, nous pouvons donc facilement énumérer les chunks du processus : pour chaque segment de chaque heap, on liste les chunks existants en utilisant le champ `Size`. On peut récuperer la taille exacte de l'allocation requise initialement par le programme pour ce chunk grâce au champ `UnusedBytes`. On stoppe dès qu'on rencontre le dernier chunk, que l'on reconnaît au flag `HEAP_ENTRY_LAST_ENTRY`.

Enfin on élimine les chunks référencés dans les listes du `LookAside`.

De plus on peut directement énumérer les gros chunks alloués via `VirtualAlloc()` grâce à la liste doublement chaînée `VirtualAllocdBlocks`. La taille exacte requise pour ces blocs est égale à la valeur de `ReserveSize` à laquelle on retranche le champ `BusyBlock.Size`.

**Low Fragmentation Heap frontend** Le frontend *Low Fragmentation* à été introduit dans Windows XP SP3, où il est disponible en *optin* via l'API `HeapSetInformation()` ou lors de la création d'un nouveau heap, via un flag passé à `HeapCreate()`.

Il s'agit d'une architecture relativement complexe que l'on ne détaillera pas ici.

### 2.3 Windows 7

Le heap a complètement été remanié sous Windows 7 (et Vista). Nous n'avons pas été en mesure de l'analyser en détail au moment de la rédaction de cet article.

Les principaux changements sont l'utilisation du *Low Fragmentation Heap* frontend par défaut, et la suppression du frontend `LookAside` ; ainsi que le chiffrement de la plupart des méta-données. Ainsi les informations comme la taille de chaque chunk sont obfusquées au moyen d'un *xor* avec une constante choisie dynamiquement à l'initialisation du tas. Les méta-données d'un chunk son ensuite re-xorées avec l'adresse du chunk lui-même.

Ce chiffrement intervient dans le cadre du hardening du heap contre les attaques de type *heap overflow*, double free, etc.

La nouvelle définition de la structure des chunks est :

```
struct _HEAP_ENTRY {
    UINT16       Size;
    UINT8        Flags;
    UINT8        SmallTagIndex;
    UINT8        _PADDING0_[0x4];
};
```

La structure du heap est totalement remaniée, et fait notamment intervenir une liste chaînée de segments au lieu du tableau ; ainsi que des *SubSegments*.

### 2.4 Applications

Notons que ces techniques d'énumération du heap ne fonctionnent que si le programme ciblé utilise les routines standards de gestion de la mémoire.

Lors de nos tests, il est apparu qu'un certain nombre de « grosses » applications, comme les browsers web, utilisent leurs propres routines ; Firefox par exemple utilise jusqu'a 7 classes d'allocateurs différentes suivant les contextes (core, plugins, xpcom, ...), et n'a jamais recours au `malloc()` standard, rendant notre approche inefficace.

Dans ce cas de figure, il sera nécessaire d'adapter notre code pour gérer les routines d'allocation spécifiques utilisées.

Toutefois, à partir du moment où l'on a accès à la liste des chunks alloués et à leur taille, la suite peut s'appliquer.

## 3 Dwarf Fortress

*« La gravure représente un programme. Le programme commande une nuée de chunks. »*

Dwarf Fortress [2] est un jeu gratuit de simulation d'un univers médiéval fantastique, où le joueur incarne principalement une colonie de nains qu'il doit faire prospérer dans un monde hostile. L'univers du jeu est très détaillé, incluant un grand nombre de plantes et d'animaux, prenant en compte la psychologie et l'anatomie individuelle de chaque créature du jeu, ainsi que la dernière blessure qu'il a reçue au petit orteil du pied gauche.

Le jeu, closed source, est le centre d'intérêt d'une communauté de hackers qui en ont déjà analysé une grande partie, principalement à l'aide du désassembleur IDA. Notons que l'auteur du jeu n'interdit pas ce genre de pratiques, et offre même à l'occasion ses conseils.

À partir de cette analyse, les structures internes du programme ont été documentées, ainsi que le moyen d'y accéder (il s'agit généralement de l'offset du pointeur racine ou de la variable dans le binaire). Un certain nombre d'outils d'instrumentation ont été développés. Ils permettent aussi bien d'interagir avec le jeu au-delà des possibilités offertes par l'interface native, autorisant par exemple la réalisation d'actions en masse (changer les affectations de toute une catégorie de créatures) ; ils permettent également de tricher (modification arbitraire des caractéristiques des objets et des personnages).

Une des difficultés rencontrée par la communauté est que le jeu est activement développé, et lors de la publication (à peu près bimensuelle) d'une nouvelle version, bien que la majorité des structures considérées restent inchangées, il est nécessaire de refaire toute une analyse afin de mettre à jour les offsets des pointeurs racines, qui permettent ensuite aux différents outils de retrouver la liste des créatures, la liste des

2. http://www.bay12games.com/dwarves/

items, les données géologiques, etc.).

Ce jeu constitue un excellent cas d'usage de notre outil.

En effet, il est d'une part codé en C++, et d'autre part extrêmement détaillé, ce qui se traduit par une myriade de structures relativement distinctes présentes en mémoire.

Par nature, le code C++ va générer un chunk par objet créé. De plus, le programmeur ne manipule pas directement les primitives mémoire, ce qui nous laisse penser que les liens logiques entre objets se feront via des pointeurs sur le début des chunks.

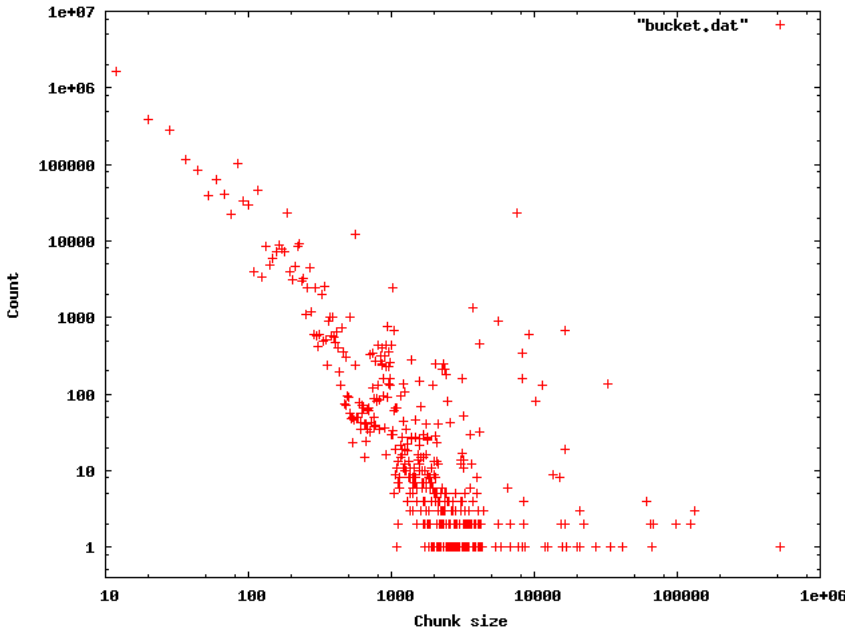
Nous allons aborder ici quelques résultats préliminaires, qui seront approfondis lors de la présentation.

## 3.1 Enumeration

« *L'image montre un script et des chunks. Le script compte les chunks.* »

En utilisant les techniques décrites précedemment, on peut énumérer les différents chunks en utilisation.

En débutant une partie standard, on se retrouve avec un heap d'environ 350Mo, qui contient plus de 3 millions de chunks. La moitié d'entre eux fait moins de 20 octets.



### 3.2 Maillage du tas

« *La statue représente le chunk* `map_data`*. Le chunk prend une pose menaçante.* »

À partir du listing des chunks, nous pouvons construire un graphe dans lequel les arcs traduisent la présence d'un pointeur vers le début d'un autre chunk. Nous trouvons ainsi un tiers des chunks qui contiennent au moins un pointeur, et nous constatons également que la plupart des chunks (2.850.000) sont pointés par au moins un autre chunk.

Une étude un peu plus poussée nous permet de tenter d'identifier les listes chaînées et les tableaux. Nous considérons avoir affaire à une liste chaînée si l'on trouve un ensemble de chunks de même taille, telle que chaque chunk contienne à un offset fixe un pointeur vers l'élément suivant de la chaîne. Pour chaque élément, cet offset doit soit contenir un pointeur vers un autre élément, soit la valeur `NULL` (0) ; et la chaîne doit être de longueur au moins 4.

Ceci nous permet d'identifier 26.000 chaînes, la plupart de taille entre 4 et 6 éléments ; l'ensemble des chaînes contient 125.000 éléments distincts. Nous constatons également que seuls 1000 éléments appartiennent à plus d'une chaîne.

Enfin, nous définissons `tableau` comme un chunk constitué en majorité de pointeurs vers des structures ayant toutes la même taille. Il doit contenir au moins 6 pointeurs vers une structure de taille fixe.

Cette définition nous permet d'identifier 25.000 tableaux, de taille variable entre 6 et 23000 entrées. Au total, 840.000 éléments distincts font partie d'un tableau, dont 100.000 sont recensés dans plusieurs tableaux. En comptant les éléments qui font soit partie d'un tableau, soit sont pointés par un élément d'un tableau, nous arrivons au chiffre de 1.830.000 éléments distincts ; et 2.230.000 en autorisant un déréférencement supplémentaire.

Il apparait donc que les tableaux sont la structure centrale utilisée dans le programme.

Ces deux classifications, listes et tableaux, permettent de définir des classes d'équivalence dans les structures : on peut ainsi supposer que tous les éléments d'un même tableau sont similaires. Nous pouvons alors effectuer une analyse sur ces classes d'équivalence, afin de dégager des similitudes. Par exemple, « tous les éléments contiennent à l'offset 8 un pointeur vers une chaîne de caractères ». Cela peut constituer la base d'un algorithme de reconstruction de types.

Cette partie sera développée dans la présentation, veuillez vous référer aux slides pour plus de détails.

Strike the earth !

## 4 Conclusion

L'analyse de programme sans analyser le code est un domaine intéressant, qui permet notamment de s'abstraire des détails de l'architecture spécifique considérée.

Dans le cadre de l'instrumentation de programmes, cette technique permet également de gérer facilement les mises à jour du code cible qui ne changent pas fondamentalement la structure des données manipulées ; ainsi si l'on sait qu'un élément est manipulé via 3 niveaux de tableaux (une représentation 3D discrète par exemple), on pourra facilement retrouver le pointeur sur cette structure dans toutes les versions du binaire sans avoir à auditer une seule ligne de code.

Nous n'avons considéré pour l'instant que les pointeurs internes au(x) heap(s) ; cependant prendre en compte les pointeurs du heap vers d'autres parties de l'espace d'adressage, notamment des pointeurs de fonctions, serait probablement très instructif. On peut notamment penser à l'analyse d'objets C++, à travers leurs v-tables, qui pourrait permettre d'identifier des objets de la même classe, ou du moins partageant une partie de leur hiérarchie de classe.

## Références

1. http://www.eglibc.org/cgi-bin/viewcvs.cgi/branches/eglibc-2_13/libc/malloc/malloc.c?rev=12752&view=markup.
2. http://metasm.cr0.org/.
3. Ben Hawkes. Attacking the vista heap, 2008. http://lateralsecurity.com/downloads/hawkes_ruxcon-nov-2008.pdf.
4. IBM John McDonald, Chris Valasek. Practical windows xp/2003 heap exploitation, 2009. http://www.blackhat.com/presentations/bh-usa-09/MCDONALD/BHUSA09-McDonald-WindowsHeap-PAPER.pdf.