OWASP
AppSecResearch 2012
Heap Exploitation
Abstraction by Example

Patroklos Argyroudis, Chariton Karamitas
{argp, huku}@census-labs.com
Census, Inc.
Who are we

- Patroklos Argyroudis, 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

- Example: FreeBSD kernel memory allocator (UMA)
- Example: Linux kernel memory allocator (SLUB)
- Example: jemalloc userland memory allocator
- Abstracting heap exploitation
Related Work

- “Attacking the Core: Kernel Exploiting Notes” [1]
  - twiz, sgrakkyu, Phrack, 2007
  - Linux (heap), Solaris (stack)

- “Kernel Wars” [2]
  - signedness.org, Black Hat EU, 2007
  - *BSD (mbuf), Windows (stack)
Related Work

- “Exploitation in the Modern Era (Blueprint)” [3]
  - Chris Valasek, Ryan Smith, Black Hat EU, 2011
  - First attempt to abstract exploitation

- “Patras Heap Massacre” [4]
  - Chariton Karamitas, Patroklos Argyroudis, Fosscomm, 2011
  - Attempt to abstract heap exploitation
Example: FreeBSD UMA
Universal Memory Allocator

- FreeBSD’s kernel memory allocator
- Funded by Nokia for a proprietary project
- The IPSO firewall/security appliance (thanks FX!)
- Donated to FreeBSD
- Functions like a traditional slab allocator
- Large areas, or slabs, of memory are pre-allocated
- `malloc(9)` returns a free slot
UMA Architecture
UMA Architecture

- Each zone (uma_zone) holds buckets (uma_bucket) of items
- The items are allocated on the zone's slabs (uma_slab)
- Each zone is associated with a keg (uma_keg)
- The keg holds the corresponding zone's slabs
- Each slab is of the same size as a page frame (usually 4096 bytes)
- Each slab has a slab header structure (uma_slab_head) which contains management metadata
### vmstat(8)

<table>
<thead>
<tr>
<th>ITEM</th>
<th>SIZE</th>
<th>LIMIT</th>
<th>USED</th>
<th>FREE</th>
<th>REQ</th>
<th>FAIL</th>
<th>SLEEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMA Kregs:</td>
<td>208</td>
<td>0</td>
<td>84</td>
<td>1</td>
<td>84</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UMA Zones:</td>
<td>512</td>
<td>0</td>
<td>84</td>
<td>0</td>
<td>84</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UMA Slabs:</td>
<td>568</td>
<td>0</td>
<td>872</td>
<td>3</td>
<td>3612</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UMA RCntSlabs:</td>
<td>568</td>
<td>0</td>
<td>195</td>
<td>1</td>
<td>195</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UMA Hash:</td>
<td>256</td>
<td>0</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16 Bucket:</td>
<td>152</td>
<td>0</td>
<td>39</td>
<td>11</td>
<td>39</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32 Bucket:</td>
<td>280</td>
<td>0</td>
<td>23</td>
<td>5</td>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64 Bucket:</td>
<td>536</td>
<td>0</td>
<td>15</td>
<td>6</td>
<td>15</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>128 Bucket:</td>
<td>1048</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td>19</td>
<td>569</td>
<td>0</td>
</tr>
<tr>
<td>VM OBJECT:</td>
<td>216</td>
<td>0</td>
<td>846</td>
<td>72</td>
<td>15779</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAP:</td>
<td>232</td>
<td>0</td>
<td>7</td>
<td>25</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>KMAP ENTRY:</td>
<td>120</td>
<td>15004</td>
<td>33</td>
<td>122</td>
<td>5952</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MAP ENTRY:</td>
<td>120</td>
<td>0</td>
<td>543</td>
<td>108</td>
<td>35870</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fakepg:</td>
<td>120</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mt_zone:</td>
<td>4112</td>
<td>0</td>
<td>301</td>
<td>10</td>
<td>301</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16:</td>
<td>16</td>
<td>0</td>
<td>2485</td>
<td>203</td>
<td>38918</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32:</td>
<td>32</td>
<td>0</td>
<td>2780</td>
<td>250</td>
<td>19562</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64:</td>
<td>64</td>
<td>0</td>
<td>5223</td>
<td>153</td>
<td>34106</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128:</td>
<td>128</td>
<td>0</td>
<td>7156</td>
<td>123</td>
<td>10268</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256:</td>
<td>256</td>
<td>0</td>
<td>830</td>
<td>40</td>
<td>3359</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>512:</td>
<td>512</td>
<td>0</td>
<td>347</td>
<td>52</td>
<td>3699</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024:</td>
<td>1024</td>
<td>0</td>
<td>50</td>
<td>142</td>
<td>6246</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
An offpage slab of the “512” zone

A non-offpage slab of the “256” zone
struct uma_slab_head {
    uma_keg_t       us_keg;         /* Keg we live in */
    union {
        LIST_ENTRY(uma_slab)    _us_link;     /* slabs in zone */
        unsigned long          _us_size;      /* Size of allocation */
    } us_type;
    SLIST_ENTRY(uma_slab)   us_hlink;  /* Link for hash table */
    u_int8_t                *us_data;    /* First item */
    u_int8_t                us_flags;    /* Page flags see uma.h */
    u_int8_t                us_freecount; /* How many are free? */
    u_int8_t                us_firstfree; /* First free item index */
};
struct uma_keg {
    LIST_ENTRY(uma_keg) uk_link;   /* List of all kegs */
    struct mtx uk_lock;            /* Lock for the keg */
    struct uma_hash uk_hash;
    char *uk_name;                 /* Name of creating zone. */
    LIST_HEAD(, uma_zone) uk_zones;   /* Keg's zones */
    LIST_HEAD(, uma_slab) uk_part_slab;   /* Partially allocated slabs */
    LIST_HEAD(, uma_slab) uk_free_slab;   /* Empty slab list */
    LIST_HEAD(, uma_slab) uk_full_slab;   /* Full slabs */
    u_int32_t uk_recycle;           /* Allocation recursion count */
    u_int32_t uk_align;             /* Alignment mask */
    u_int32_t uk_pages;             /* Total page count */
    u_int32_t uk_free;              /* Count of items free in slabs */
    u_int32_t uk_size;              /* Requested size of each item */
    u_int32_t uk_rsize;             /* Real size of each item */
    u_int32_t uk_maxpages;          /* Maximum number of pages to alloc */
    uma_init uk_init;               /* Keg's init routine */
    uma_fini uk_fini;               /* Keg's fini routine */
    uma_alloc uk_allocf;            /* Allocation function */
    uma_free uk_freef;              /* Free routine */
    struct vm_object *uk_obj;       /* Zone specific object */
    vm_offset_t uk_kva;             /* Base kva for zones with objs */
    uma_zone_t uk_slabzone;         /* Slab zone backing us, if OFFPAGE */
    u_int16_t uk_pgoff;             /* Offset to uma_slab struct */
    u_int16_t uk_ppera;             /* Pages per allocation from backend */
    u_int16_t uk_ipers;             /* Items per slab */
    u_int32_t uk_flags;             /* Internal flags */
};
struct uma_zone {
  char *uz_name;       /* Text name of the zone */
  struct mtx *uz_lock; /* Lock for the zone (keg's lock) */

  LIST_ENTRY(uma_zone) uz_link;   /* List of all zones in keg */
  LIST_HEAD(,uma_bucket) uz_full_bucket; /* full buckets */
  LIST_HEAD(,uma_bucket) uz_free_bucket; /* Buckets for frees */

  LIST_HEAD(,uma_klink) uz_kegs;   /* List of kegs. */
  struct uma_klink uz_klink;       /* klink for first keg. */

  u_int32_t uz_slab;              /* Allocate a slab from the backend. */
  u_int32_t uz_ctor;              /* Constructor for each allocation */
  u_int32_t uz_dtor;              /* Destructor */
  u_int32_t uz_init;              /* Initializer for each item */
  u_int32_t uz_fini;              /* Discards memory */

  u_int32_t uz_flags;             /* Flags inherited from kegs */
  u_int32_t uz_size;              /* Size inherited from kegs */

  u_int64_t uz_allocs UMA_ALIGN;  /* Total number of allocations */
  u_int64_t uz_frees;             /* Total number of frees */
  u_int64_t uz_fails;             /* Total number of alloc failures */
  u_int64_t uz_sleeps;            /* Total number of alloc sleeps */
  uint16_t uz_fills;              /* Outstanding bucket fills */
  uint16_t uz_count;              /* Highest value ub_ptr can have */

  /*
   * This HAS to be the last item because we adjust the zone size
   * based on NCPU and then allocate the space for the zones.
   */

  struct uma_cache uz_cpu[1];     /* Per cpu caches */
};
uma_zfree_arg(uma_zone_t zone, void *item, void *udata)
{
    uma_cache_t cache;
    uma_bucket_t bucket;
    int bflags;
    int cpu;

    #ifdef UMA_DEBUG_ALLOC_1
    printf("Freeing item %p to %s(%p)\n", item, zone->uz_name, zone);
    #endif

    CTR2(KTR_UMA, "uma_zfree_arg thread %x zone %s", curthread,
        zone->uz_name);

    /* uma_zfree(..., NULL) does nothing, to match free(). */
    if (item == NULL)
        return;

    if (zone->uz_dtor)
        zone->uz_dtor(item, zone->uz_size, udata);
uz_dtor Hijacking

1. System call
   - fake uma_kseg { uk_zones }
2. A slab of the "256" zone
   - addr
3. Kernel can dereference userland
   - uma_slab_head { us_kseg }
4. 4.
   - Kernel shellecode
5. 5.
   - fake uma_zone { uz_dtor }
6. 6. Restore us_kseg pointer

UMA managed kernel memory
Userland memory
Example: Linux SLUB
SLUB

- Organizes physical memory frames in “caches” (UMA: kegs)
- Each cache holds slabs (UMA: slab) of objects (UMA: items) of the same size
  - kmalloc-32, kmalloc-64, task_struct, mm_struct
- Objects on a slab are contiguous
- A slab may have both allocated (used) and deallocated (free) objects
SLUB’s slabs

- Each slab is \textit{at least} PAGE\_SIZE bytes (default 4096 bytes)
- A slab \textit{may} span many pages
  - \texttt{kmalloc-32}: 128 objects * 32 bytes == 4096 bytes
  - \texttt{task\_struct (1088 bytes)}: 30 objects * 1088 bytes == 32640
- A \texttt{task\_struct} slab spans 8 pages
- Each CPU core has its own slabs
Metadata?

- No separate/dedicated metadata structures stored on the slabs
- Each free object stored on a slab has a next-free-object pointer
- Each slab has a page structure (struct page) that has a pointer (freelist) to the slab's first free object
SLUB’s behavior

- Partial slabs: some free and some used objects
- New requests satisfied from partial slabs
  - Least-recently-used (LRU) policy
  - No partial slabs $\rightarrow$ allocation of new slab
- Generic slabs (e.g. kmalloc-32) are used to store different objects of the same size
  - Different kernel structures, buffers, etc
- Contiguous
SLUB Exploitation

- Attack alternatives
  - Corrupt metadata of free objects on a slab
  - Corrupt adjacent objects on a slab
- We need a suitable kernel structure to corrupt
- We can allocate/deallocate from userland
- Same size as the object/structure we can overflow from
- Bring target slab to a predictable state in order to have the victim structure after the structure we can overflow from
SLUB Exploitation Algorithm

- Find free objects on target slab:
  - cat /proc/slabinfo
- Ensure allocations/deallocation happen on the slabs of the same CPU: sched_setaffinity(2)
- Consume a large number of objects that go on the target slab (reducing fragmentation)
- Deallocate a small number of objects from the target slab
- Allocate a smaller number of our selected victim objects
- Trigger the heap overflow bug overflowing onto the victim object
SLUB Exploitation

Step #1

USED  FREE  USED  USED  FREE  FREE

Step #3

USED  USED  USED  USED  USED  USED  USED

Step #4

USED  FREE  FREE  FREE  FREE  FREE  FREE

Step #5

USED  FREE  FREE  USED  USED  USED  USED

Step #6

USED  USED  USED  USED  ED  USED  USED

overflow
Victim Structure

- Traditionally struct shmid_kernel
- Allocations/deallocations controlled from userland
  - Allocation: shmget(2)
  - Deallocation: ipcrm(1)
- Leads to structure with yummy function pointers
struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm
    struct file *
    unsigned long
    unsigned long
    time_t
    time_t
    time_t
    pid_t
    pid_t
    struct user_struct
    shm_perm;
    shm_file;
    shm_nattch;
    shm_segsz;
    shm_atim;
    shm_dtim;
    shm_ctim;
    shm_cprid;
    shm_lprid;
    *mlock_user;
};
struct file {
    /*
     * fu_list becomes invalid after file_free is called and queued via
     * fu_rcuhead for RCU freeing
     */
    union {
        struct list_head fu_list;
        struct rcu_head fu_rcuhead;
    } f_u;
    struct path f_path;
#define f_dentry f_path.dentry
#define f_vfsmnt f_path.mnt
    const struct file_operations *f_op;
    spinlock_t f_lock; /* f_ep_links, f_flags, no IRQ */
#endif CONFIG_SMP
    int f_sb_list_cpu;
#endif CONFIG_SECURITY
    void *f_security;
#endif CONFIG_EPOLLP
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head f_ep_links;
#endif /* ifdef CONFIG_EPOLLP */
    struct address_space *f_mapping;
#endif CONFIG_DEBUG_WRITECOUNT
    unsigned long f_mnt_write_state;
#endif
};
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, f1_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags) (int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write) (struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read) (struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
    int (*setlease) (struct file *, long, struct file_lock *);
    long (*fallocate) (struct file *, int mode, loff_t offset,
    loff_t len);
};
Example: jemalloc
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 overview

- 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 Architecture
jemalloc Exploitation

- Adjacent memory overwrite
- Metadata overwrite
  - Run header corruption
  - Chunk header corruption
  - Magazine (a.k.a thread cache) corruption
- For the details attend our Black Hat USA 2012 talk!
Abstracting Heap Exploitation
UMA - SLUB - jemalloc

- End-user allocations: UMA - items, SLUB - objects, jemalloc - regions
- Allocation containers: UMA - slabs, SLUB - slabs, jemalloc - runs
- Container groupings: UMA - kegs, SLUB - caches, jemalloc - chunks
- Execution-specific metadata:
  - UMA - zone, Linux kernel - zone, jemalloc - arena
  - UMA - buckets, SLUB - N/A, jemalloc - bins
Value of Abstraction

- Chris Valasek's and Ryan Smith's Black Hat EU 2011 talk on abstracting exploitation through primitives [3]
- Back in CS 101 we were taught that abstraction is the most important skill of a computer scientist
- Specific exploitation techniques will become obsolete
- Our 2 drachmas are to abstract heap exploitation and have “primitives” that can be applied to new targets
Memory Allocators as Weird Machines

- Weird machine: The state machine of the target program after memory corruption [5, 6]
- In our case
  - State machine: Memory allocator
  - Weird machine: Post-corruption memory allocator
  - New states, unexpected by the developer
  - However reachable due to the memory corruption
Heap Weird Machines
Heap Weird Machines

- Our memory allocator model: deterministic automaton (threads not taken into account)
- Metadata corruption abstraction
  - Corruption of the automaton’s transition function
  - New states are reachable - most dead but not all
- Data (e.g. adjacent item) corruption abstraction
  - Manipulation of the automaton’s determinacy
- We control the order of transitions
The Weirding Module ;)

- The target heap manager should be treated as a high level API
  - For allocations and deallocations
- “Applications” that use the allocator (Javascript, system calls, incoming packets) provide a way to proxy these API calls

Attacker  Application (Proxy)  Allocator
The Weirding Module ;)
Conclusion

- Future work
  - Operational semantics (formal notation)
  - More examples on both allocators and exploits

- Acknowledgments
  - Dr ;) Dimitris Glynos
  - Chris Valasek
  - Sergey Bratus
References