BSD heap smashing

05/14/2003

BlackHat Briefings Europe

$ cd bullshit && ls -tr1

```
overview
algorithms detailed
sample exploitation techniques
real flaw exploitation
```

$ cd overview

```
$ for file in `ls -tr`; do
> clear;
> echo $file;
> [ -x $file ] &&
>     ./$file ||
>     cat $file;
> read foo;
> done
```

memory_regions.txt

```
[+] brk region
    the break
        FreeBSD/i86 and NetBSD/i86: 0x0804????
        OpenBSD/i86: 0x0000????
    brk(2) and sbrk(2)
[+] mmap region
    mmap(2)
    FreeBSD/i86 and NetBSD/i86: 0x480E????
    OpenBSD/i86: 0x400D????
```

logical_layers.txt

```
[+] bottom layer
    handles memory pages
[+] top layer
    handles chunks, including:
        user chunks
        internal use chunks
```

pages_referencing.txt

```
[+] user pages reside in the brk region
[+] they are referenced in an array
    ptr2idx (page address) = array index
    MALLOC_NOT_MINE (0)
    MALLOC_FIRST (2), MALLOC_FOLLOW (3)
    MALLOC_FREE (1)
[+] these values may be overwritten and
    restored by the top layer
```

free_pages_handling.txt

[+] free pages are referenced in a doubly linked list

```
struct pgfree {
    struct pgfree *next;
    struct pgfree *prev;
    void          *page;
    void          *end;
    size_t         size;
}
```

[+] static struct pgfree free_list;

[+] list elements are allocated with imalloc()

[+] the list is sorted

chunks_overview.txt

```
[+] three categories of chunks
    large chunks: > (malloc_pagesize/2)
    medium-sized chunks
    tiny chunks
[+] large chunks reside in dedicated
    pages
[+] other chunks are grouped in pages
    where each chunk has the same size
    (rounded up to a power of 2)
```

chunks_referencing.txt

[+] pages containing tiny and medium-sized
    chunks are referenced in linked lists

```
struct pginfo {
    struct pginfo *next;
    void          *page;
    u_short       size;
    u_short       shift;
    u_short       free;
    u_short       total;
    u_int         bits[1];
}
```

[+] the lists are sorted (page field)

[+] bits has in fact a variable length

bits_field.txt

[+] the appropriate bit is set to one if the associated chunk is free

[+] chunk number j is associated with the bit given by:

bits[i] & (1<<n) where:

i = j / (8 * sizeof(u_int))

n = j % (8 * sizeof(u_int))

sample-i86.out

```
[+] i86: 32 bits, little endian

[+] no chunk allocated

       bits[0]   |1|1|1|1|1|1|1|1| LSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| MSB

       bits[1]   |1|1|1|1|1|1|1|1| LSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| MSB
```

sample-i86.out

```
[+] i86: 32 bits, little endian
[+] 1st chunk allocated
        bits[0]   |1|1|1|1|1|1|1|0| LSB
                  |1|1|1|1|1|1|1|1|
                  |1|1|1|1|1|1|1|1|
                  |1|1|1|1|1|1|1|1| MSB
        bits[1]   |1|1|1|1|1|1|1|1| LSB
                  |1|1|1|1|1|1|1|1|
                  |1|1|1|1|1|1|1|1|
                  |1|1|1|1|1|1|1|1| MSB
```

sample-i86.out

```
[+] i86: 32 bits, little endian
[+] 1st and 23rd chunks allocated
       bits[0]   |1|1|1|1|1|1|1|0| LSB
                 |1|1|1|1|1|1|1|1|
                 |1|0|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| MSB

       bits[1]   |1|1|1|1|1|1|1|1| LSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| MSB
```

sample-i86.out

```
[+] i86: 32 bits, little endian
[+] 1st, 23rd and 42nd chunks allocated
       bits[0]   |1|1|1|1|1|1|1|0| LSB
                 |1|1|1|1|1|1|1|1|
                 |1|0|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| MSB

       bits[1]   |1|1|1|1|1|1|1|1| LSB
                 |1|1|1|1|1|1|0|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| MSB
```

sample-PPC.out

```
[+] PPC 7450: 32 bits, big endian

[+] no chunk allocated

       bits[0]   |1|1|1|1|1|1|1|1| MSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| LSB

       bits[1]   |1|1|1|1|1|1|1|1| MSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| LSB
```

sample-PPC.out

```
[+] PPC 7450: 32 bits, big endian

[+] 1st chunk allocated

       bits[0]   |1|1|1|1|1|1|1|1| MSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|0| LSB

       bits[1]   |1|1|1|1|1|1|1|1| MSB
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1|
                 |1|1|1|1|1|1|1|1| LSB
```

sample-PPC.out

```
[+] PPC 7450: 32 bits, big endian

[+] 1st and 23rd chunk allocated

      bits[0]  |1|1|1|1|1|1|1|1| MSB
               |1|0|1|1|1|1|1|1|
               |1|1|1|1|1|1|1|1|
               |1|1|1|1|1|1|1|0| LSB

      bits[1]  |1|1|1|1|1|1|1|1| MSB
               |1|1|1|1|1|1|1|1|
               |1|1|1|1|1|1|1|1|
               |1|1|1|1|1|1|1|1| LSB
```

sample-PPC.out

```
[+] PPC 7450: 32 bits, big endian

[+] 1st, 23rd and 42nd chunk allocated

      bits[0]  |1|1|1|1|1|1|1|1| MSB
               |1|0|1|1|1|1|1|1|
               |1|1|1|1|1|1|1|1|
               |1|1|1|1|1|1|1|0| LSB

      bits[1]  |1|1|1|1|1|1|1|1| MSB
               |1|1|1|1|1|1|1|1|
               |1|1|1|1|1|1|0|1|
               |1|1|1|1|1|1|1|1| LSB
```

pginfo_location.txt

[+] tiny chunks pages: the pginfo is
    located at the beginning of the page

[+] medium-sized chunks pages: the
    pginfo is allocated through a call
    to imalloc()

[+] a chunk is medium-sized if the
    pginfo structure effective size is
    less than half the size of a chunk

pgfree_location.txt

[+] the px cache is defined as:
    static struct pgfree *px;

[+] it is set to the address of a ready to use pgfree structure sized chunk or to 0

[+] pgfree structures are allocated through a call to imalloc()

[+] when they become useless, they are freed through a call to ifree()

[+] one of them is most of the time kept in px

pages_directory.txt

```
[+] the pages directory is defined as:
        static struct pginfo **page_dir;

[+] it is split into two parts
        for i < malloc_pageshift:
            page_dir[i] -> (1<<i) bytes pginfos
        for i >= malloc_pageshift:
            page_dir[i] == MALLOC_FIRST or
                           MALLOC_FOLLOW or
                           MALLOC_FREE or
                           MALLOC_NOT_MINE or
                           address of a pginfo

[+] page_dir[0], page_dir[1], page_dir[2] and
    page_dir[3] are unused
```

pgdir_handling.txt

[+] page_dir is initially set to the address of one mmap()ed page

[+] it is extended whenever required, one page at a time

[+] it is accessed thanks to the ptr2idx macro, defined as:

```
#define ptr2idx(i) \
 (((size_t)(i)>>malloc_pageshift)-malloc_origo)
```

[+] if ptr is the address of a tiny or medium-sized chunk, then page_dir[ptr2idx(ptr)] is the associated pginfo pointer (it is in the second part of page_dir)

```
$ cd ../algorithms\ \
detailed

 $ for file in `ls -tr`; do
> >     clear;
> >     echo $file;
> >     [ -x $file ] &&
> >         ./$file ||
> >         cat $file;
> >     read foo;
> > done
```

malloc.txt

```
[+] concurrent call check
        if (malloc_active++) {
            wrtarning("recursive call.\n");
            malloc_active--;
            return 0;
        }
[+] call to imalloc() to do the real job
        if (malloc_sysv && !size)
            r = 0;
        else
            r = imalloc(size);

[+] cleanup
        malloc_active--;
```

imalloc.txt

```
[+] if the chunk is tiny or medium-
    sized, call malloc_bytes() to
    allocate it

[+] otherwise, call malloc_pages()
    to allocate the proper number
    of pages
```

malloc_pages.txt

```
[+] round size up to a multiple of malloc_pagesize
[+] look for a sufficient number of adjacent free
    pages in free_list
[+] if there is a perfect match, remove the area
    from the list and mark its pgfree for freeing
[+] if the first match is too large, eat its first
    pages
[+] if there was no match, call map_pages() to
    request new pages in the brk region
[+] update the pages directory
[+] if a pgfree has been marked for freeing, and
    the px cache is empty, then it becomes the new
    px cache, otherwise, it is freed through a call
    to ifree()
```

malloc_bytes.txt

```
[+] make sure size is at least 16

[+] find j such that size is (1<<j)

[+] if page_dir[j] is 0, make a new (1<<j)
    bytes chunks page thanks to
    malloc_make_chunks():
        map one page with malloc_pages()
        tiny or medium-sized chunks?
        imalloc() a new pginfo if necessary
        initialize the pginfo fields
        update the pages directory

[+] choose the lowest address free chunk

[+] remove the pginfo from the list if
    necessary
```

free.txt

```
[+] malloc-style concurrent call check
[+] call to ifree() to perform the real
    job
        check the pointer is in the brk
        region
        if the associated page_dir entry
        is MALLOC_FIRST, call
        free_pages(), otherwise, call
        free_bytes()
```

free_pages.txt

[+] sanity checks: pointer to the beginning of a page whose entry in page_dir is MALLOC_FIRST

[+] mark the pages as free in the pages directory

[+] make sure the px cache is not empty

[+] insert the freed area in free_list, with two constraints:
    enforcing its sorting policy
    performing areas merges when possible
    if no merge is possible, the px cache is used

[+] possibly unmap pages in the brk region

[+] if two merges were performed, call ifree() to get rid of the pgfree of the highest of the three areas

free_bytes.txt

[+] sanity checks

    the pointer really points to the beginning of a chunk

    the chunk is not already free

[+] chunk is marked as free in the bits field

[+] if the page was full of allocated chunks, it is reinserted in the pages directory (this operation enforces the sorting policy of the pages directory)

realloc.txt

[+] the same memory area is used if:

the chunk is a large chunk and the operation doesn't change the number of necessary pages

the chunk is tiny or medium-sized and the operation doesn't change its effective size

[+] otherwise, a new chunk is allocated (imalloc()), data is copied with memcpy(), and the former chunk is freed (ifree())

```
$ cd ../sample\ \
> exploitation\ techniques

 $ for file in `ls -tr`; do
> >     clear;
> >     echo $file;
> >     [ -x $file ] &&
> >         ./$file ||
> >         cat $file;
> >     read foo;
> > done
```

i86_parameters.txt

```
[+] pginfo structures: 16 bytes + bits field
        16 bytes tiny chunks: 48 bytes
        32 bytes tiny chunks: 32 bytes
        64 bytes medium-sized chunks: 24 bytes

[+] pgfree structures: 20 bytes

[+] page size: 4096 bytes
        2 2048 bytes chunks per page
        128 32 bytes chunks per page
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "vuln.c"

int main () {
    char buf[1024];

    while (1) {
        fgets ( buf, sizeof(buf), stdin );
        if ( *buf != '+' && *buf != '-' )
            exit( 42 );
        vuln_inside ( *buf, atoi ( buf + 1 ) );
    }

    return 0;
}
```

vuln-1.c

```
void vuln_inside ( char op, unsigned int i ) {
    char *p;

    if ( op == '+' ) {
        p = malloc ( i );
        gets( p );
    } else {
        free ( (void *) i );
    }
}
```

expl-1.out

```
initial heap state

  32 |i|p|


  i: pginfo structure
  p: px cache
```

expl-1.out

```
allocation of a 32 bytes chunk

  32 |i|p|x|



  i: pginfo structure
  p: px cache
  x: allocated chunk
```

expl-1.out

```
allocation of a 2048 bytes chunk


  32 |i|p|x|i|
2048 |x|


   i: pginfo structure
   p: px cache
   x: allocated chunk
```

expl-1.out

```
freeing of the 32 bytes chunk


  32 |i|p| |i|
2048 |x|


  i: pginfo structure
  p: px cache
  x: allocated chunk
```

expl-1.out

```
reallocation and overflow of the 32
bytes chunk
  32 |i|p|x|o|
2048 |x|


  i: pginfo structure
  p: px cache
  x: allocated chunk
  o: overwritten structure
```

fake_pginfo.txt

```
[+] next allocated chunk at:
        page + n * ( 1 << shift )
        where n depends on bits

[+] size may matter if malloc_junk
    is set (not the default)

[+] other fields do not matter
```

vuln-2.c

```
void vuln_inside ( char op, unsigned int i ) {
    int j;
    char *p;

    if ( op == '+' ) {
        p = malloc ( i );
        j = fread ( p, 1, i, stdin );
        p[j] = 0;
    } else {
        free ( (void *) i );
    }
}
```

expl-2.out

```
initial heap state

  32 |i|p|

  i: pginfo structure
  p: px cache
```

expl-2.out

```
allocation of a 32 bytes chunk

  32 |i|p|x|

  i: pginfo structure
  p: px cache
```

expl-2.out

```
allocation of two 2048 bytes chunks


  32 |i|p|x|i|
2048 |x|x| (page is complete)

  i: pginfo structure
  p: px cache
  x: allocated chunk
```

expl-2.out

```
leakage of some 32 bytes chunks

      |<- 256 bytes ->|
   32 |i|p|x|i|x|x|x|x|x|
2048 |x|x| (page is complete)

   i: pginfo structure
   p: px cache
   x: allocated chunk
```

expl-2.out

```
allocation of a 2048 bytes chunk

      |<- 256 bytes ->|
   32 |i|p|x|i|x|x|x|x|x|i|
2048 |x|x| (page is complete)
2048 |x|
   i: pginfo structure
   p: px cache
   x: allocated chunk
```

expl-2.out

```
freeing of the first 2048 bytes
chunk
      |<- 256 bytes ->|
   32 |i|p|x|i|x|x|x|x|x|i|
2048 | |x|
2048 |x|
   i: pginfo structure
   p: px cache
   x: allocated chunk
```

expl-2.out

```
freeing of the appropriate 32 bytes
chunk
      |<- 256 bytes ->|
   32 |i|p| |i|x|x|x|x|x|i|
2048 | | |x|
2048 |x|
   i: pginfo structure
   p: px cache
   x: allocated chunk
```

expl-2.out

```
reallocation and overflow of the 32
bytes chunk
      |<- 256 bytes ->|
   32 |i|p|x|o|x|x|x|x|x|i|
2048 | |x|
2048 |x|
   i: pginfo structure
   p: px cache
   x: allocated chunk
   o: partly overwritten structure
```

```
$ cd ../real \
> flaw\ exploitation

 $ for file in `ls -tr`; do
> >     clear;
> >     echo $file;
> >     [ -x $file ] &&
> >         ./$file ||
> >         cat $file;
> >     read foo;
> > done
```

background.txt

[+] CVS flaw reported by Stefan Esser in a
    VulnWatch posting (20/01/03):

    http://archives.neohapsis.com/archives/
                vulnwatch/2003-q1/0028.html

[+] oversimplified main server loop:

```
        buf_read_line( buf_from_net, &cmd, NULL );
        call server_*( cmd + something );
        free( cmd );
```

vulnerability.txt

```
[+] adapted from serve_directory( char *arg ):
    buf_read_line( buf_from_net, &repos, NULL );
    dirswitch( arg, repos );
    free( repos );

[+] flaw in dirswitch( char *dir, char *repos ):
    if ( dir_name != NULL )
      free( dir_name );
    dir_len = 80 + strlen(dir);
    if ( dir_len > 0 && dir[dir_len-1] == '/' ) {
        if ( alloc_pending( 80 + dir_len ) )
            sprintf( pending_error_text, ...
        return;
    }
    dir_name=malloc(strlen(srv_tmp_dir)+dir_len+40);
```

CVS_buffers_handling.txt

[+] 4k buffers allocated 16 at a time:
        malloc(17 * 4k - 1)

[+] CVS keeps track of the available buffers
    for incoming data in a linked list of 16
    bytes structures allocated 16 at a time
    before the associated 4k buffers:
        malloc(16 * 16)

[+] buf_read_line uses these buffers to store
    incoming data, and allocates a new buffer
    to copy it when a '\n' is found, the
    buffers are then recycled

CVS_useful_functions.txt

```
[+] serve_noop() sends and frees the pending
    errors

[+] serve_set() sets a CVS variable value,
    variables are kept in a hash table using 32
    bytes structures

[+] serve_max_dotdot(char *arg) sets
    srv_tmp_dir to

    "/tmp/cvs????/d/d/d/d/d/d.../d",

    allocated with

    malloc(strlen(srv_tmp_dir)+2*atoi(arg)+10)
```