==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x09 of 0x12

|=----------=[Once upon a free()...]=----------=|

|=----------=[anonymous <d45a312a@author.phrack.org>]=----------=|

On the Unix system, and later in the C standard library there are functions to handle variable amounts of memory in a dynamic way. This allows programs to dynamically request memory blocks from the system. The operating system only provides a very rough system call 'brk' to change the size of a big memory chunk, which is known as the heap.

On top of this system call the malloc interface is located, which provides a layer between the application and the system call. It can dynamically split the large single block into smaller chunks, free those chunks on request of the application and avoid fragmentation while doing so. You can compare the malloc interface to a linear file system on a large, but dynamically sized raw device.

There are a few design goals which have to be met by the malloc interface:

- stability
- performance
- avoidance of fragmentation
- low space overhead

There are only a few common malloc implementations. The most common ones are the System V one, implemented by AT&T, the GNU C Library implementation and the malloc-similar interface of the Microsoft operating systems (RtlHeap*).

Here is a table of algorithms and which operating systems use them:

Algorithm	Operating System
BSD kingsley	4.4BSD, AIX (compatibility), Ultrix
BSD phk	BSDI, FreeBSD, OpenBSD
GNU Lib C (Doug Lea)	Hurd, Linux
System V AT&T	Solaris, IRIX
Yorktown	AIX (default)
RtlHeap*	Microsoft Windows *

It is interesting to see that most of the malloc implementations are very easy to port and that they are architecture independent. Most of those implementations just build an interface with the 'brk' system call. You can change this behaviour with a #define. All of the implementations I have come across are written in ANSI C and just do very minimal or even no sanity checking. Most of them have a special compilation define that

includes asserts and extra checks. Those are turned off by default in the final build for performance reasons. Some of the implementations also offer extra reliability checks that will detect buffer overflows. Those are made to detect overflows while development, not to stop exploitation in the final release.

Storing management info in-band

Most malloc implementations share the behaviour of storing their own management information, such as lists of used or free blocks, sizes of memory blocks and other useful data within the heap space itself. Since the whole idea of malloc/free is based on the dynamic requirements the application has, the management info itself occupies a variable amount of data too. Because of this, the implementation can seldomly just reserve a certain amount of memory for its own purposes, but stores the management information "in-band", right after and before the blocks of memory that are used by the application.

Some applications do request a block of memory using the malloc interface, which later happens to be vulnerable to a buffer overflow. This way, the data behind the chunk can be changed. Possibly the malloc management structures can be compromised. This has been demonstrated first by Solar Designer's wizard-like exploit [1].

The central attack of exploiting malloc allocated buffer overflows is to modify this management information in a way that will allow arbitrary memory overwrites afterwards. This way pointers can be overwritten within the writeable process memory, hence allowing modification of return addresses, linkage tables or application level data.

To mount such an attack, we have to take a deep look within the internal workings of the implementation we want to exploit. This article discusses the commonly used GNU C Library and the System V implementation and how to gain control over a process using buffer overflows which occur in malloced buffers under Linux, Solaris and IRIX systems.

System V malloc implementation

IRIX and Solaris use an implementation which is based on self-adjusting binary trees. The theoretical background of this implementation has been described in [2].

The basic idea of this implementation is to keep lists of equally sized malloc chunks within a binary tree. If you allocate two chunks of the same size, they will be within the same node and within the same list of this node. The tree is ordered by the size of its elements.

The TREE structure

The definition of the TREE structure can be found in the mallint.h, along with some easy-to-use macros to access its elements. The mallint.h file

can be found in the source distribution of the Solaris operating system [4]. Although I cannot verify that IRIX is based on the same source, there are several similarities which indicated this. The malloc interface internally creates the same memory layout and functions, besides some 64 bit alignments. You can utilize the Solaris source for your IRIX exploits, too.

To allow each tree element to be used for a different purpose to avoid overhead and force an alignment, each TREE structure element is defined as a union:

```
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
        size_t          w_i;            /* an unsigned int */
        struct _t_      *w_p;           /* a pointer */
        char            w_a[ALIGN];     /* to force size */
} WORD;
```

Central TREE structure definition:

```
/* structure of a node in the free tree */
typedef struct _t_ {
        WORD    t_s;    /* size of this element */
        WORD    t_p;    /* parent node */
        WORD    t_l;    /* left child */
        WORD    t_r;    /* right child */
        WORD    t_n;    /* next in link list */
        WORD    t_d;    /* dummy to reserve space for self-pointer */
} TREE;
```

The 't_s' element of the chunk header contains the rounded up value of the size the user requested when he called malloc. Since this size is always rounded up to a word boundary, at least the lower two bits of the 't_s' elements are unused - they normally would have the value of zero all the time. Instead of being zero, they are ignored for all size-related operations. They are used as flag elements.

From the malloc.c source it reads:

   BIT0: 1 for busy (block is in use), 0 for free.

   BIT1: if the block is busy, this bit is 1 if the preceding block in
         contiguous memory is free. Otherwise, it is always 0.

TREE Access macros:

```
/* usable # of bytes in the block */
#define SIZE(b)         (((b)->t_s).w_i)

/* free tree pointers */
#define PARENT(b)       (((b)->t_p).w_p)
```

```
#define LEFT(b)         (((b)->t_l).w_p)
#define RIGHT(b)        (((b)->t_r).w_p)

/* forward link in lists of small blocks */
#define AFTER(b)        (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */
#define LINKFOR(b)      (((b)->t_n).w_p)
#define LINKBAK(b)      (((b)->t_p).w_p)
```

For all allocation operations a certain alignment and minimum size is enforced, which is defined here:

```
#define WORDSIZE        (sizeof (WORD))
#define MINSIZE         (sizeof (TREE) - sizeof (WORD))
#define ROUND(s)        if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))
```

The tree structure is the central element of each allocated chunk. Normally only the 't_s' and 't_p' elements are used, and user data is stored from 't_l' on. Once the node is freed, this changes and the data is reused to manage the free elements more efficiently. The chunk represents an element within the splay tree. As more chunks get freed, the malloc implementation tries to merge the free chunks right next to it. At most FREESIZE (32 by default) chunks can be in this dangling free state at the same time. They are all stored within the 'flist' array. If a call to free is made while the list is already full, the old element at this place falls out and is forwarded to realfree. The place is then occupied by the newly freed element.

This is done to speed up and avoid defragmentation in cases where a lot of calls to free are made in a row. The real merging process is done by realfree. It inserts the chunk into the central tree, which starts at the 'Root' pointer. The tree is ordered by the size of its elements and is self-balancing. It is a so called "splay tree", in which the elements cycle in a special way to speed up searches (see google.com "splay tree" for further information). This is not much of importance here, but keep in mind that there are two stages of free chunks: one being within the flist array, and one within the free-elements tree starting at 'Root'.

There are some special management routines for allocating small chunks of memory, which happen to have a size below 40 bytes. Those are not considered here, but the basic idea is to have extra lists of them, not keeping them within a tree but in lists, one for each WORD matching size below 40.

There is more than one way to exploit a malloc based buffer overflow, however here is one method which works against both, IRIX and Solaris.

As a chunk is realfree'd, it is checked whether the neighbor-chunks are already within the realfree'd tree. If it is the case, the only thing that has to be done is to logically merge the two chunks and reorder its position within the tree, as the size has changed.

This merging process involves pointer modification within the tree, which consists of nodes. These nodes are represented by the chunk header itself. Pointers to other tree elements are stored there. If we can overwrite them, we can possibly modify the operation when merging the chunks.

Here is, how it is done in malloc.c:
(modified to show the interesting part of it)

```
static void
realfree(void *old)
{
        TREE    *tp, *sp, *np;
        size_t  ts, size;

        /* pointer to the block */
        tp = BLOCK(old);
        ts = SIZE(tp);
        if (!ISBIT0(ts))
                return;
        CLRBITS01(SIZE(tp));

        /* see if coalescing with next block is warranted */
        np = NEXT(tp);
        if (!ISBIT0(SIZE(np))) {
                if (np != Bottom)
                        t_delete(np);
                SIZE(tp) += SIZE(np) + WORDSIZE;
        }
```

We remember NEXT points to the chunk directly following the current one. So we have this memory layout:

```
  tp               old              np
  |                |                |
  [chunk A header] [chunk A data] | [chunk B or free ....]
                                  |
                                  chunk boundary
```

In the usual situation the application has allocated some space and got a pointer (old) from malloc. It then messes up and allows a buffer overflow of the chunk data. We cross the chunk boundary by overflowing and hit the data behind, which is either free space or another used chunk.

```
        np = NEXT(tp);
```

Since we can only overflow data behind 'old', we cannot modify the header of our own chunk. Therefore we cannot influence the 'np' pointer in any way. It always points to the chunk boundary.

Now a check is made to test if it is possible to merge forward, that is our chunk and the chunk behind it. Remember that we can control the chunk to the right of us.

```
        if (!ISBIT0(SIZE(np))) {
```

```
                if (np != Bottom)
                        t_delete(np);
                SIZE(tp) += SIZE(np) + WORDSIZE;
        }
```

BIT0 is zero if the chunk is free and within the free elements tree. So if it is free and not the last chunk, the special 'Bottom' chunk, it is deleted from the tree. Then the sizes of both chunks are added and later in the code of the realfree function the whole resized chunk is reinserted into the tree.

One important part is that the overflowed chunk must not be the last chunk within the malloc space, condition:

        1. Overflowed chunk must not be the last chunk

Here is how the 't_delete' function works:

```
static void
t_delete(TREE *op)
{
        TREE    *tp, *sp, *gp;

        /* if this is a non-tree node */
        if (ISNOTREE(op)) {
                tp = LINKBAK(op);
                if ((sp = LINKFOR(op)) != NULL)
                        LINKBAK(sp) = tp;
                LINKFOR(tp) = sp;
                return;
        }
```

There are other cases, but this is the one easiest to exploit. As I am already tired of this, I will just explain this one here. The others are very similar (look at malloc.c).

ISNOTREE compares the 't_l' element of the TREE structure with -1. -1 is the special marker for non-tree nodes, which are used as doubly linked list, but that does not matter.

Anyway, this is the first condition we have to obey:

        2. fake->t_l = -1;

Now the unlinking between FOR (t_n) and BAK (t_p) is done, which can be rewritten as:

```
        t1 = fake->t_p
        t2 = fake->t_n
        t2->t_p = t1
        t1->t_n = t2
```

Which is (written in pseudo-raw-assignments which happen at the same time):

```
        [t_n + (1 * sizeof (WORD))] = t_p
```

```
        [t_p + (4 * sizeof (WORD))] = t_n
```

This way we can write to arbitrary addresses together with valid addresses at the same time. We choose to use this:

```
        t_p = retloc - 4 * sizeof (WORD)
        t_n = retaddr
```

This way retloc will be overwritten with retaddr and *(retaddr + 8) will be overwritten with retloc. If there is code at retaddr, there should be a small jump over the bytes 8-11 to not execute this address as code. Also, the addresses can be swapped if that fits the situation better.

Finally our overwrite buffer looks like this:

```
  | <t_s> <t_p> <t_l> <j: t_r> <t_n> <j: t_d>
  |
  chunk boundary
```

Where: t_s = some small size with lower two bits zeroed out
t_p = retloc - 4 * sizeof (WORD)
t_l = -1
t_r = junk
t_n = retaddr
t_d = junk

Note that although all of the data is stored as 32 bit pointers, each structure element occupies eight bytes. This is because of the WORD union, which forces at least ALIGN bytes to be used for each element. ALIGN is defined to eight by default.

So a real overflow buffer behind the chunk boundary might look like:

```
ff ff ff f0 41 41 41 41  ef ff fc e0 41 41 41 41  | ....AAAA....AAAA
ff ff ff ff 41 41 41 41  41 41 41 41 41 41 41 41  | ....AAAAAAAAAAAA
ef ff fc a8 41 41 41 41  41 41 41 41 41 41 41 41  | ....AAAAAAAAAAAA
```

All 'A' characters can be set arbitrarily. The 't_s' element has been replaced with a small negative number to avoid NUL bytes. If you want to use NUL bytes, use very few. Otherwise the realfree function will crash later.

The buffer above will overwrite:

```
        [0xeffffce0 + 32] = 0xeffffca8
        [0xeffffca8 +  8] = 0xeffffce0
```

See the example code (mxp.c) for a more in-depth explanation.

To summarize down the guts if you happen to exploit a malloc based buffer overflow on IRIX or Solaris:

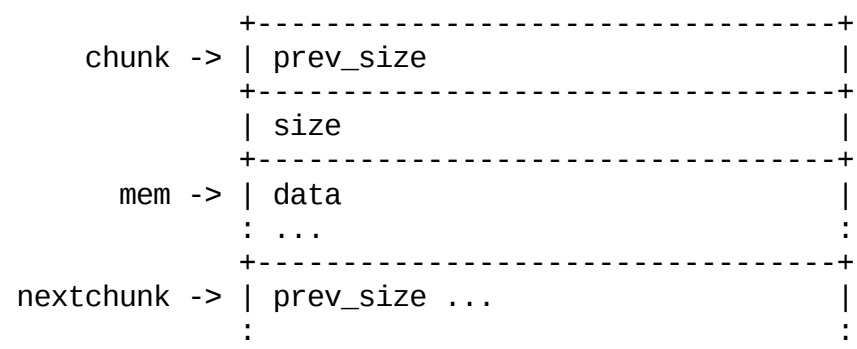
1. Create a fake chunk behind the one you overflow
2. The fake chunk is merged with the one you overflow as it is passed to realfree
3. To make it pass to realfree it has to call malloc() again or

there have to be a lot of successive free() calls
4. The overflowed chunk must not be the last chunk (the one before Bottom)
5. Prepend the shellcode/nop-space with jump-aheads to not execute the unavoidable unlink-overwrite address as code
6. Using the t_splay routines attacks like this are possible too, so if you cannot use the attack described here (say you cannot write 0xff bytes), use the source luke.

There are a lot of other ways to exploit System V malloc management, way more than there are available in the GNU implementation. This is a result of the dynamic tree structure, which also makes it difficult to understand sometimes. If you have read until here, I am sure you can find your own ways to exploit malloc based buffer overflows.

GNU C Library implementation

The GNU C library keeps the information about the memory slices the application requests in so called 'chunks'. They look like this (adapted from malloc.c):



Where mem is the pointer you get as return value from malloc(). So if you do a:

```
unsigned char * mem = malloc (16);
```

Then 'mem' is equal to the pointer in the figure, and (mem - 8) would be equal to the 'chunk' pointer.

The 'prev_size' element has a special function: If the chunk before the current one is unused (it was free'd), it contains the length of the chunk before. In the other case - the chunk before the current one is used - 'prev_size' is part of the 'data' of it, saving four bytes.

The 'size' field has a special meaning. As you would expect, it contains the length of the current block of memory, the data section. As you call malloc(), four is added to the size you pass to it and afterwards the size is padded up to the next double-word boundary. So a malloc(7) will become a malloc(16), and a malloc(20) will become malloc(32). For malloc(0) it will be padded to malloc(8). The reason for this behaviour will be explained in the latter.

Since this padding implies that the lower three bits are always zero and are not used for real length, they are used another way. They are used to indicate special attributes of the chunk. The lowest bit, called PREV_INUSE, indicates whether the previous chunk is used or not. It is set if the next chunk is in use. The second least significant bit is set if the memory area is mmap'ed -- a special case which we will not consider. The third least significant bit is unused.

To test whether the current chunk is in use or not, we have to check the next chunk's PREV_INUSE bit within its size value.

Once we free() the chunk, using free(mem), some checks take place and the memory is released. If its neighbour blocks are free, too (checked using the PREV_INUSE flag), they will be merged to keep the number of reuseable blocks low, but their sizes as large as possible. If a merge is not possible, the next chunk is tagged with a cleared PREV_INUSE bit, and the chunk changes a bit:

```
            +---------------------------------+
    chunk -> | prev_size                       |
            +---------------------------------+
            | size                            |
            +---------------------------------+
      mem -> | fd                              |
            +---------------------------------+
            | bk                              |
            +---------------------------------+
            | (old memory, can be zero bytes)  |
            :                                 :

nextchunk -> | prev_size ...                   |
            :                                 :
```

You can see that there are two new values, where our data was previously stored (at the 'mem' pointer). Those two values, called 'fd' and 'bk' - forward and backward, that is, are pointers. They point into a double linked list of unconsolidated blocks of free memory. Every time a new free is issued, the list will be checked, and possibly unconsolidated blocks are merged. The whole memory gets defragmented from time to time to release some memory.

Since the malloc size is always at least 8 bytes, there is enough space for both pointers. If there is old data remaining behind the 'bk' pointer, it remains unused until it gets malloc'd again.

The interesting thing regarding this management, is that the whole internal information is held in-band -- a clear channeling problem.
(just as with format string commands within the string itself, as control channels in breakable phonelines, as return addresses within stack memory, etc).

Since we can overwrite this internal management information if we can overwrite a malloced area, we should take a look at how it is processed later on. As every malloc'ed area is free()'d again in any sane program,

we take a look at free, which is a wrapper to chunk_free() within malloc.c (simplified a bit, took out #ifdef's):

```
static void
chunk_free(arena *ar_ptr, mchunkptr p)
{
  size_t     hd = p->size; /* its head field */
  size_t     sz;           /* its size */
  int        idx;          /* its bin index */
  mchunkptr  next;         /* next contiguous chunk */
  size_t     nextsz;       /* its size */
  size_t     prevsz;       /* size of previous contiguous chunk */
  mchunkptr  bck;          /* misc temp for linking */
  mchunkptr  fwd;          /* misc temp for linking */
  int        islr;         /* track whether merging with last_remainder */

  check_inuse_chunk(ar_ptr, p);

  sz = hd & ~PREV_INUSE;
  next = chunk_at_offset(p, sz);
  nextsz = chunksize(next);
```

Since the malloc management keeps chunks within special structures called 'arenas', it is now tested whether the current chunk that should be free directly borders to the 'top' chunk -- a special chunk. The 'top' chunk is always the top-most available memory chunk within an arena, it is the border of the available memory. The whole if-block is not interesting for typical buffer overflows within the malloc space.

```
  if (next == top(ar_ptr))                        /* merge with top */
  {
    sz += nextsz;

    if (!(hd & PREV_INUSE))                    /* consolidate backward */
    {
      prevsz = p->prev_size;
      p = chunk_at_offset(p, -(long)prevsz);
      sz += prevsz;
      unlink(p, bck, fwd);
    }

    set_head(p, sz | PREV_INUSE);
    top(ar_ptr) = p;

      if ((unsigned long)(sz) >= (unsigned long)trim_threshold)
        main_trim(top_pad);
    return;
  }
```

Now the 'size' of the current chunk is tested whether the previous chunk is unused (testing for the PREV_INUSE flag). If this is the case, both chunks are merged.

```
  islr = 0;
```

```
  if (!(hd & PREV_INUSE))                    /* consolidate backward */
  {
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -(long)prevsz);
    sz += prevsz;

    if (p->fd == last_remainder(ar_ptr))     /* keep as last_remainder */
      islr = 1;
    else
      unlink(p, bck, fwd);
  }
```

Now the same is done vice versa. It is checked whether the chunk in front of the current chunk is free (testing for the PREV_INUSE flag of the size two chunks ahead). If this is the case the chunk is also merged into the current one.

```
  if (!(inuse_bit_at_offset(next, nextsz)))   /* consolidate forward */
  {
    sz += nextsz;

    if (!islr && next->fd == last_remainder(ar_ptr))
                                                /* re-insert last_remainder */
    {
      islr = 1;
      link_last_remainder(ar_ptr, p);
    }
    else
      unlink(next, bck, fwd);

    next = chunk_at_offset(p, sz);
  }
  else
    set_head(next, nextsz);                  /* clear inuse bit */

  set_head(p, sz | PREV_INUSE);
  next->prev_size = sz;
  if (!islr)
    frontlink(ar_ptr, p, sz, idx, bck, fwd);
}
```

As Solar Designer showed us, it is possible to use the 'unlink' macro to overwrite arbitrary memory locations. Here is how to do:

A usual buffer overflow situation might look like:

```
        mem = malloc (24);
        gets (mem);
        ...
        free (mem);
```

This way the malloc'ed chunk looks like this:

```
[ prev_size ] [ size P] [ 24 bytes ... ] (next chunk from now)
       [ prev_size ] [ size P] [ fd ] [ bk ] or [ data ... ]
```

As you can see, the next chunk directly borders to our chunk we overflow. We can overwrite anything behind the data region of our chunk, including the header of the following chunk.

If we take a closer look at the end of the chunk_free function, we see this code:

```
  if (!(inuse_bit_at_offset(next, nextsz)))   /* consolidate forward */
  {
    sz += nextsz;

    if (!islr && next->fd == last_remainder(ar_ptr))
                                            /* re-insert last_remainder */
    {
      islr = 1;
      link_last_remainder(ar_ptr, p);
    }
    else
      unlink(next, bck, fwd);

    next = chunk_at_offset(p, sz);
  }
```

The inuse_bit_at_offset, is defined as macro in the beginning of malloc.c:

```
#define inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)
```

Since we control the header of the 'next' chunk we can trigger the whole if block at will. The inner if statement is uninteresting, except our chunk is bordering to the top-most chunk. So if we choose to trigger the outer if statement, we will call unlink, which is defined as macro, too:

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

The unlink is called with a pointer to a free chunk and two temporary pointer variables, called bck and fwd. It does this to the 'next' chunk header:

```
  *(next->fd + 12) = next->bk
  *(next->bk + 8) = next->fd
```

They are not swapped, but the 'fd' and 'bk' pointers point to other chunks. This two chunks being pointed to are linked, zapping the current chunk from the table.

So to exploit a malloc based buffer overflow, we have to write a bogus header in the following chunk and then wait for our chunk getting free'd.

```
        [buffer .... ] | [ prev_size ] [ size ] [ fd ] [ bk ]

'|' is the chunk boundary.

The values we set for 'prev_size' and 'size' do not matter, but two
conditions have to be met, in case it should work:

  a) the least significant bit of 'size' has to be zero
  b) both, 'prev_size' and 'size' should be add-safe to a pointer that is
     read from. So either use very small values up to a few thousand, or -
     to avoid NUL bytes - use big values such as 0xfffffffc.
  c) you have to ensure that at (chunk_boundary + size + 4) the lowest bit
     is zeroed out (0xfffffffc will work just fine)

'fd' and 'bk' can be set this way (as used in Solar Designers Netscape
Exploit):

  fd = retloc - 12
  bk = retaddr

But beware, that (retaddr + 8) is being written to and the content there
will be destroyed. You can circumvent this by a simple '\xeb\x0c' at
retaddr, which will jump twelve bytes ahead, over the destroyed content.

Well, however, exploitation is pretty straight forward now:

<jmp-ahead, 2> <6> <4 bogus> <nop> <shellcode> |
        \xff\xff\xff\xfc \xff\xff\xff\xfc <retloc - 12> <retaddr>

Where '|' is the chunk boundary (from that point we overflow). Now, the
next two negative numbers are just to survive a few checks in free() and to
avoid NUL bytes. Then we store (retloc - 12) properly encoded and then the
return address, which will return to the 'jmp-ahead'. The buffer before the
'|' is the same as with any x86 exploit, except for the first 12 bytes,
because we have to take care of the extra write operation by the unlink
macro.


Off-by-one / Off-by-five
------------------------

It is possible to overwrite arbitrary pointers, even in cases where you can
overwrite only five bytes, or - in special cases - only one byte. When
overwriting five bytes the memory layout has to look like:

        [chunk a] [chunk b]

Where chunk a is under your control and overflowable. Chunk b is already
allocated as the overflow happens. By overwriting the first five bytes of
chunk b, we trash the 'prev_size' element of the chunks header and the
least significant byte of the 'size' element. Now, as chunk b is free()'d,
backward consolidation pops in, since 'size' has the PREV_INUSE flag
cleared (see below). If we supply a small value for 'prev_size', which is
smaller than the size of chunk a, we create a fake chunk structure:
```

```
        [chunk a ... fakechunk ...] [chunk b]
                     |
                     p
```

Where prev_size of chunk b points relativly negative to the fake chunk.
The code which is exploitable through this setting was already discussed:

```
  if (!(hd & PREV_INUSE))                    /* consolidate backward */
  {
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -(long)prevsz);
    sz += prevsz;

    if (p->fd == last_remainder(ar_ptr))     /* keep as last_remainder */
      islr = 1;
    else
      unlink(p, bck, fwd);
  }
```

'hd' is the size element of chunk b. When we overwrite it, we clear out the lower two bits, so PREV_INUSE is cleared and the if condition is matched (NUL will do it in fact). In the next few instructions 'p', which was a pointer to chunk b originally, is relocated to our fakechunk. Then the unlink macro is called and we can overwrite the pointers as usual. We use backward consolidation now, while in the previous description we have used forward consolidation. Is this all confusing? Well, when exploiting malloc overflows, do not worry about the details, they will become clearer as you understand the malloc functions from a broader scope.

For a really well done overview and description of the malloc implementation in the GNU C Library, take a look at the GNU C Library reference manual [3]. It makes a good read for non-malloc related things, too.

Possible obstacles and how to get over with them

As with any new exploitation technique people will show up which have the 'perfect' solution to the problem in their head or in form of a patch to the malloc functions. Those people - often ones who have never written an exploit themselves - are misleading into a wrong sense of security and I want to leave a few words about those approaches and why they seldomly work.

There are three host based stages where you can stop a buffer overflow resulting in a compromise:

1. The bug/overflow stage

   This is the place where the real overflow happens, where data is overwritten. If this place is known, the origin of the problem can be fixed (at source level). However, most approaches argue that this place is not known and therefore the problem cannot be fixed yet.

2. The activation stage

After the overflow happened parts of the data of the application are corrupted. It does not matter what kind of data, whether it is a stack frame, a malloc management record or static data behind a buffer. The process is still running its own path of code, the overwritten data is still passive. This stage is everything after the overflow itself and before the seize of execution control. This is where the natural, non-artificially introduced hurdles for the attacker lies, code which must be passed in a certain way.

3. The seized stage

This is everything after control has been redirected from its original path of execution. This is the stage where nopspace and shellcode is executed, where no real exploitation hurdles are left.

Now for the protection systems. Most of the "non-exec stack" and "non-exec heap" patches try to catch the switch from stage two to three, where execution is seized, while some proprietary systems check for the origin of a system call from within kernel space. They do not forbid you to run code this way, they try to limit what code can be run.

Those systems which allow you to redirect execution in the first place are fundamentally flawed. They try to limit the exploitation in a black-listing way, by trying to plug the places you may want to go to. But if you can execute legal code within the process space its almost everytime enough to compromise the process as a whole.

Now for the more challenging protections, which try to gripe you in stage two. Those include - among others - libsafe, StackGuard, FormatGuard, and any compiler or library based patches. They usually require a recompilation or relinking of your existing code, to insert their security 'measures' into your code. This includes canary values, barriers of check bytes or reordering and extensive checking of sanity before doing things which might be bad. While sanity checking in general is a good policy for security, it cannot fix stuff that was broken before. Every of those protections is assuming a certain situation of a bug which might appear in your program and try to predict the results of an attacker abusing the bug. They setup traps which they assume you will or have to trigger to exploit the bug. This is done before your control is active, so you cannot influence it much except by choosing the input data. Those are, of course much more tight than protection systems which only monitor stage three, but still there are ways around them. A couple of ways have been discussed in the past, so I will not go into depth here. Rather, I will briefly address on a protection which I already see on the horizon under a name like 'MallocGuard'.

Such a protection would not change the mechanism of malloc management chunks much, since the current code has proved to be effective. The malloc function plays a key role in overall system performance, so you cannot tweak freely here. Such a protection can only introduce a few extra checks, it cannot verify the entire consistency everytime malloc() is called. And this is where it is flawed: Once you seize control over one malloc chunk

information, you can seize control over other chunks too. Because chunks are 'walked' by using either stored pointers (SysV) or stored lengths (GlibC), it is possible to 'create' new chunks. Since a sanity check would have to assume inconsistency of all chunks in the worst case, it would have to check all chunks by walking them. But this would eat up too much performance, so its impossible to check for malloc overflows easily while still keep a good performance. So, there will be no 'MallocGuard', or it will be a useless guard, in the tradition of useless pseudo protections. As a friend puts it - 'for every protection there is an anti-protection'.

Thanks
======

I would like to thank all proofreaders and correctors. For some really needed corrections I thank MaXX, who wrote the more detailed article about GNU C Library malloc in this issue of Phrack, kudos to him ! :)

References
==========

[1] Solar Designer,
    http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
[2] DD Sleator, RE Tarjan, "Self-Adjusting Binary Trees", 1985,
    http://www.acm.org/pubs/citations/journals/jacm/1985-32-3/p652-sleator/
    http://www.math.tau.ac.il/~haimk/adv-ds-2000/sleator-tarjan-splay.pdf
[3] The GNU C Library
    http://www.gnu.org/manual/glibc-2.2.3/html_node/libc_toc.html
[4] Solaris 8 Foundation Source Program
    http://www.sun.com/software/solaris/source/

|=[EOF]=---=|