Smashing Heap by Free Simulation

Sandip Chaudhari
sandipchaudhari@gmail.com

Acknowledgements
Thanks to everyone in my Security Team for their support and encouragement, especially to Jonathan Leonard, Jeremy Jethro and Nick Seidenman.

Smashing Heap by Free Simulation

Abstract

*This paper describes exploitation of heap overflows. Exploitation can be achieved in just one or two (one for **AIX** and two for **Solaris**) calls to **malloc** after the overflow **without** the need of any call to the **free()** function. The overflowed memory is given a value such that a previous call to free() is simulated, causing the next malloc call to misinterpret that the memory was free'd before. We can call this technique*

*- **Free Simulation**. We overflow the heap and manipulate in-band heap information to simulate free address space, and eventually gain control of the process execution. Though the Free Simulation technique demonstrated in this paper has been tried successfully on AIX and Solaris, it should be applicable on all systems.*

Introduction

A recent paper [1] has very nicely outlined the generic Third Generation [2] Heap Based Overflow exploitation techniques. A formal discussion of heap based overflows and its run-time detection and protection schemes has been presented in yet another paper referenced as [3] in the References section. Almost all the papers referenced in the References section [1] through [7], discuss heap overflows, that seem to talk or provide sample code snippet where ***free()*** is being called. What if free() is never called and the process takes in user input that can lead to a heap overflow? Is it still possible to exploit such a process that never calls *free*()? The answer to this is yes, and thats what this paper is all about.

Core Ideas

As mentioned earlier, the papers in the References section describe the interesting heap overflow technique where the free() is being called and is usually referred to as a **4-byte overwrite**. The core idea is "*to attack the memory management algorithm, first (publicly?) demonstrated by Solar Designer for a heap overflow found in the Netscape browser"* [3], [1]. This class of attacks on memory management algorithms *necessarily* involves pointer assignment instructions.

Let's refer to the primitive logical constructs involving these pointer assignments that get executed on a call to *free()* [4 – Section: Anatomy of a Heap Overflow Exploit] & [5].

Fig. 1

unlink

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

frontlink

```
#define frontlink(A, P, S, IDX, BK, FD)
{
  [1] FD = start_of_bin(IDX);
  [2] while ( FD != BK && S < chunksize
              (FD) )
    {
      [3] FD = FD->fd;
    }
  [4] BK = FD->bk;
  [5] FD->bk = BK->fd = P;
}
```

Note: Both of the above macros are a set of logical statements that explain pointer assignments. Either or both of these may be executed on call to *free()*. "P" is the pointer that has been passed to be free'd.

What Exactly is Free Simulation?

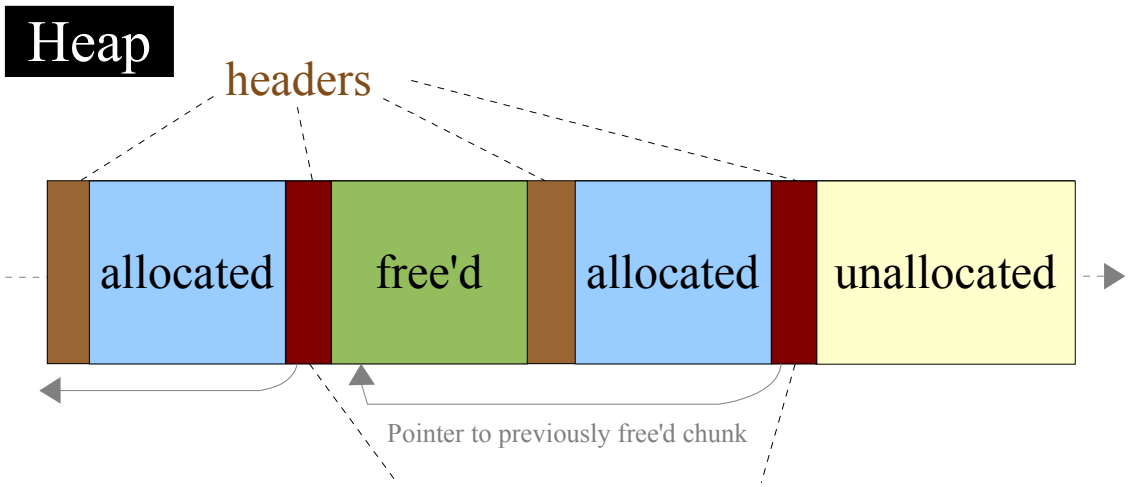
Let's take into consideration the case of a process where there is a heap-based overflow and free() is never called. We will see how the heap-based overflows can be very easily exploited without any call to *free*(). We instead trigger the exploit on a call to *malloc*(). It should be pointed out that the memory overwrite still takes place by the same (or similar) primitive logical statements of pointer assignments [Fig. 1]. We will further dig deeper and establish conditions that trigger such assignments with our control over the pointer addresses.

Simply explained, Free Simulation is nothing but the allocation of memory on **simulated** free region in the memory with our choice of length, and located anywhere we choose in the process' address space. Free Simulation can be differentiated broadly into 2 cases:

- **Arbitrary buffer allocation –** The heap datastructure pointers are manipulated such that the simulated free buffer, when allocated may exist arbitrarily anywhere in process memory address space (Free Simulation on AIX, Free Simulation on Solaris (<40 bytes buffer))
- **Arbitrary address over-write (4-byte i.e word-size overwrite)** – The heap datastructure pointers are manipulated such that the pointer assignments causes an address to be overwritten arbitrarily anywhere in the process memory address space (Free Simulation on Solaris (>=40 bytes buffer))

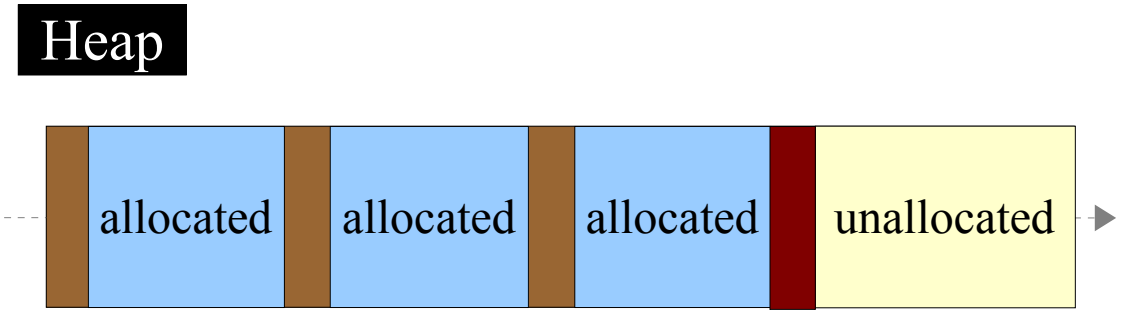
A free() called on the last malloc'ed chunk triggers coalesce. Free() simply free's the chunk and consolidates it with the yet un-malloc'ed heap region. Now, if we consider the case of free'ing a chunk which lies in the middle of the malloc'ed chunks, then the book-keeping information of this free'd chunk has to be stored somewhere, and of course it is maintained on the heap. Whenever the next malloc() is called, it first tries to allocate the chunk from the existing available free'd() memory, if any, with the requested chunksize being equal or less than the available free'd space. To exploit the malloc'ed heap overflow, the chunk header is manipulated such that on the next call to malloc(), the heap management algorithm is confused to take this simulated free chunk into consideration as an available free space for chunk allocation!

The following is the visual representation of the heap state at the moment of time when a number of chunks have been malloc'ed and a few have been free'd.

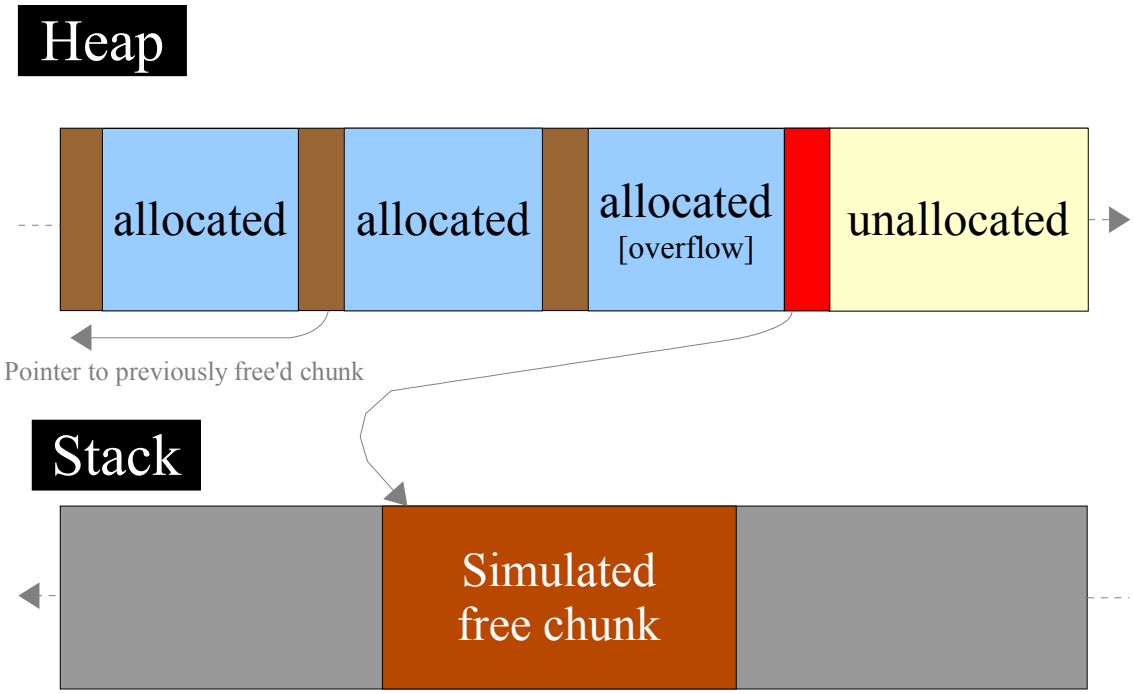


Heap in-band header with pointers to previously free'd chunks

Here's an example of the usual state of heap with a few allocated chunks.



Now we represent the state of the heap after the overflow and Free Simulation.



For all of the above manipulations to be possible certain pre-conditions have to be satisfied. Let's now focus on these pre-conditions or trigger-points that lead to Free Simulation. To simply put it, these are the conditional statements in the malloc call that check if there is some previous or last free'd memory available (of appropriate size) that can be used to allocate the new chunk.

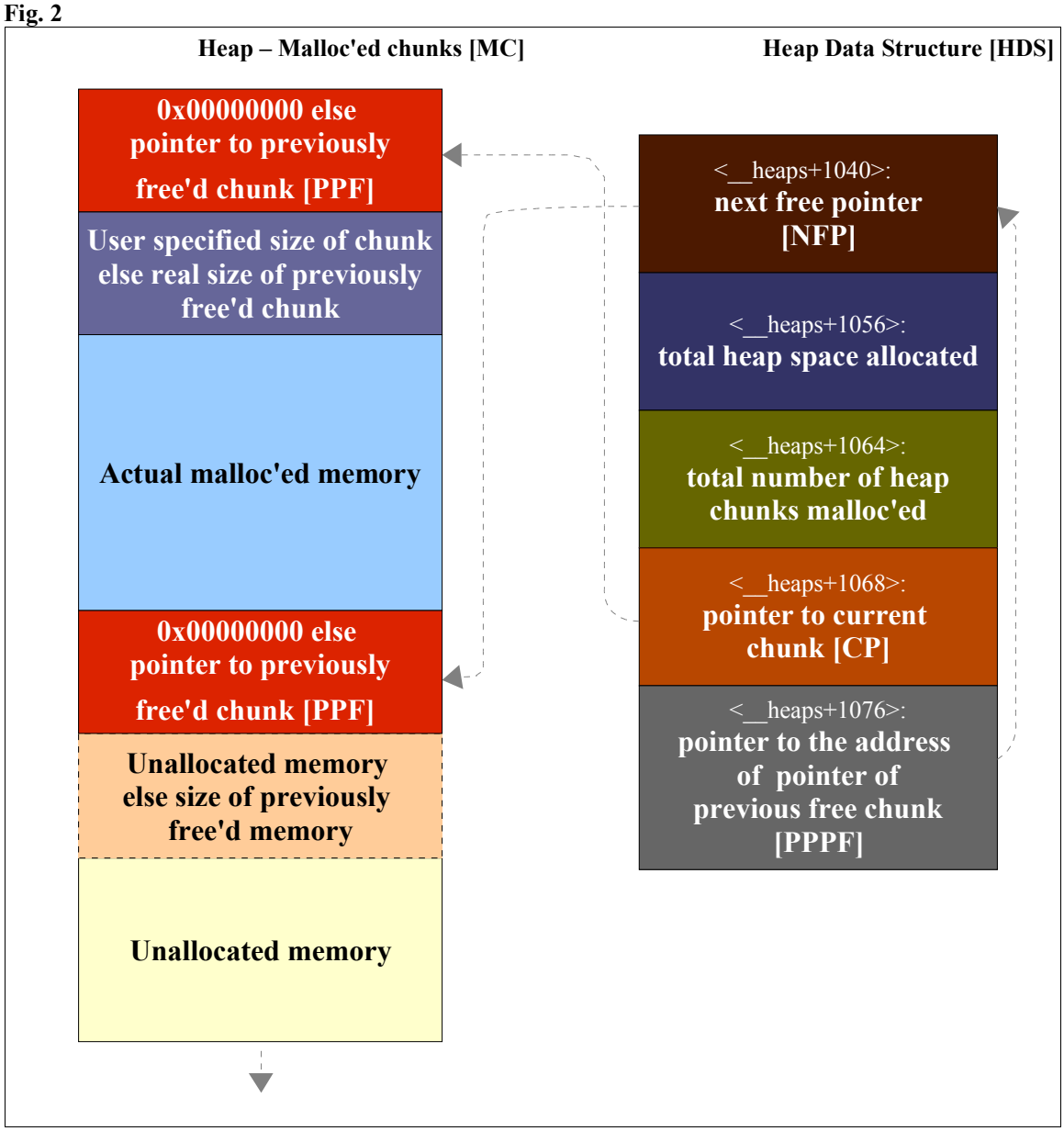
if (previous free'd chunk available && requested size <= available free'd size) then ...

The above logical free conditional primitive lies at the heart of successful Free Simulation. The conditional primitive triggers the execution of further pointer assignment instructions. Though logically it is just one conditional statement, but it varies across systems based on the implementations of malloc(). As a proof of concept, let's further study the Free Simulation trigger points on the **AIX** and **Solaris** systems.

Free Simulation on AIX

Please refer to **Fig. 2**. What follows below, is an attempt to represent the general working of the malloc() implementation on the AIX heap.

Fig. 2



Usually, the PPF (Pointer to Previously Free'd chunk) is NULL and NFP (Next Free Pointer) points to the last PPF (NULL), indicating availability of free memory. When a *malloc* is called, it checks if the value pointed by NFP is NULL. If NFP is NULL, it writes the requested size as the next word and returns the address of the word next to the size as the return value of *malloc()* call. Now we will try to understand what happens if PPF is **not** NULL, which is very important for the success of exploitation by Free Simulation.

Normally the heap keeps growing beyond the address space pointed to by NFP in the heap region of memory. The PPF which is usually NULL, is now assigned the value of the address of the previously free'd chunk! The core idea is to overflow the malloc() allocated chunk by 2 words having the first word as an address (we will soon see what address) so that it will be now interpreted as PPF and the second word as some arbitrary size. What should be the address value of the first word i.e. PPF, such that it may be to the attacker's advantage? How about pointing PPF to the stack? Is this possible? Yes! In a way, we are smashing the heap, simulating free(), and then smashing the stack!

Thus the simulated free space can be located **anywhere** in the process writable memory address space.The reason this is possible is because the malloc() function **trusts** the address retrieved, as a valid heap address for memory allocation. As soon as malloc returns, the user will now be returned a pointer to the address that is pointing to the stack instead of an address on the heap. At this point we control the stack. Any string or memory copy operation on this malloc'ed chunk would actually copy to the stack. Now the attacker has complete control to over-write the return address and point it somewhere on the stack or to the code that has been injected on the heap. Even a non-executable stack would fail to prevent the success of this exploit if the attacker opts to point the return address to the heap.

Free Simulation Conditional Trigger for AIX

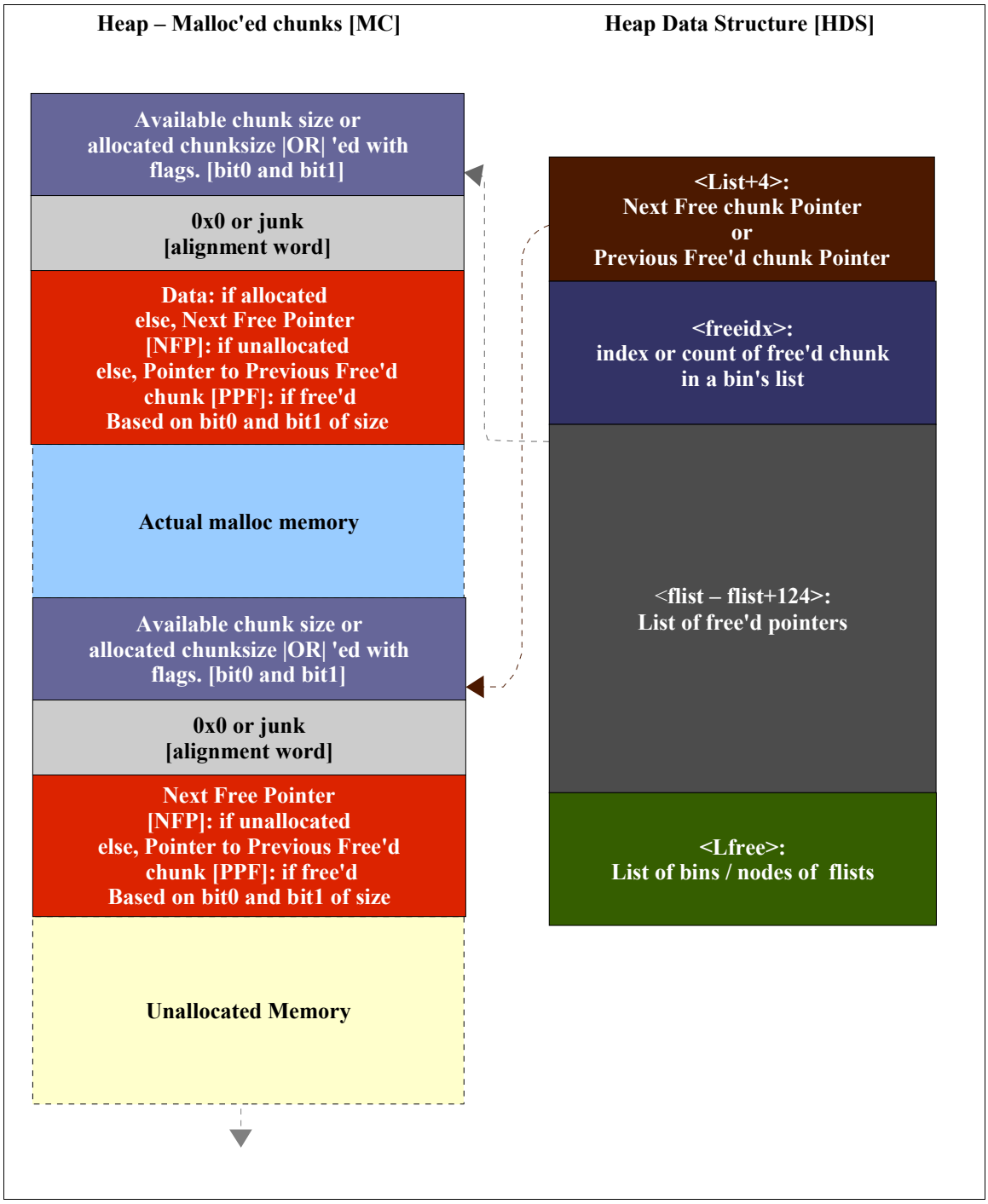
For AIX we can sum-up the logic for triggering Free Simulation when malloc() tries to allocate a new chunk as follows:

```
if ( Pointer to Previously Free'd chunk [PPF] != NULL && requested_size <= chunk_size )   ... [A]
{
        consider the value of PPF as address of previously
        free'd chunk and try to allocate memory on this free'd
        chunk
}
```

The above [A] is a mere logical summary of conditional instructions or statements. The “if()” may have been actually implemented as “while()”. The conditional statements make it very clear that triggering Free Simulation on AIX is quite easy.

Free Simulation on Solaris [size < 40 bytes]

Similar to what we did for AIX, we will simply follow the same procedure for Solaris. We will try to represent all important and relevant parts of memory during malloc/free operations. On Solaris, 2 types of data-structures are involved in heap management, based on the chunk size requested for. If the requested chunk **size is less than 40** then **linked-lists** are involved and different sections of code get executed, while for **sizes greater than 40**, a **binary search tree** structure is maintained. In this first part we will focus on the exploitation where sizes lesser than 40-bytes are involved.



Exploiting heap-based overflows is slightly tricky in case of Solaris. Though linked list heap management on Solaris is similar to AIX, we have to take into consideration some more elements that come into picture along with malloc'ed chunks and heap data-structures [Lists] for Solaris. They can be referred to as flists or Free-lists, which hold addresses of free'd chunks and the Lfree pointer, that points to the bins of lists of particular sizes, available in the memory space.

The decision to allocate or consider the previously free'd chunk of memory for allocation is based primarily on the **bit0** and **bit1** of the size word on Solaris. The size is specified in bytes and we get the last 2 bits free.

- **bit0: 1 if chunk is allocated else 0.**
- **bit1: 1 if a previous block has been free'd in local list of the bin else 0.**

When malloc allocates memory at the address pointed to by the Next Free chunk pointer, it checks for the state of flags in the size-word's bit0 and bit1. If both are zero, it allocates the chunk of the requested size with OR'ed bit0=1 flag i.e. literally requested_size+1. I would like to point out that whenever the size of a chunk is requested, it may be modified for alignment. After it is aligned, it is ensured that last two bits are 0,0, for further updates based on the chunk's state. The address of start-of-chunk+8 is returned as the return-value for malloc where data may be written.

The freelists are structured like lists in bins of various sizes. There may be multiple bins, each with its list that can be of the same size. In case malloc finds that **bit1**'s state is "**1"** it considers that there is previous free'd memory chunk. The word next to the immediate next word is considered as the address pointing to the previously free'd chunk. Malloc checks the size of this previously free'd chunk against the requested size. If the requested size is less, malloc allocates this previously free'd chunk, and returns the address for data write with the bits set appropriately in the size-word in the chunks' header.

We can simulate free() and exploit a heap-based overflow on Solaris on a call to malloc in similar way as we did for AIX. In the case of Solaris, we overflow the allocated chunks' boundary such that the **bit1** of the size in the header of next chunk is set to **"1"** and the word next to the immediate-next word can be given the address where we would like to overwrite on the next write operation. Two calls to malloc() are needed before the address can be overwritten on the stack. In the first call there is allocation, then a buffer copy operation leading to the overflow. On second call to malloc(), the Next Pointer is accessed from the HDS and the overflown address is read into the Head Data Structure [HDS] as the future Next Pointer. This results owing to satisfactory pre-conditions for Free Simulation.The second copy operation executes safely. But on any future malloc() call, the NEXT operation would return the address of our simulated free buffer. This buffer can be located anywhere as we please, on stack or heap or any writable region of process' memory address space. Memory is allocated with reference to this Next Pointer address retrieved from HDS. Hence, if this address is pointing to stack, memory will be allocated on the stack and any write operation in future would be directly taking place on this allocated memory on the stack.

As before in AIX, again, the simulated free space can be located **anywhere** in the process writable memory address space.The reason this is possible is because the malloc() function **trusts** the address retrieved, as a valid heap address for memory allocation.

Free Simulation Conditional Trigger for Solaris [size < 40 bytes]

For Solaris we can sum-up the logic for triggering Free Simulation, when a malloc tries to allocate new chunk as follows:

if (size.bit1 equals 1) **.... [B]**

{

After size checks, consider address next to immediate-next word as previously free'd chunk and assign it to the Next Pointer of Heap Data Structure. Again, after size checks this simulated free space will be used to allocate memory on any call to malloc() in the future.

}

As stated before in the case of AIX, the above [B] is a mere logical summary of conditional instructions for Solaris. The conditional "if" is logical and it may be a conditional "while" in actual implementation.

Free Simulation on Solaris [size > 40 bytes]

As pointed out in Solaris [size<40 bytes], for sizes greater than or equal to 40 bytes on Solaris, a binary search tree [splay-tree] has been implemented to manage heap. Such heap-based overflows for Solaris, involving tree-data-structures have been described in **"Once upon a free()"** paper [8] published in Phrack magazine. In this paper, exploitation has been demonstrated by calls only to malloc() that further calls realfree(). The focus is on creation of the fake-chunk that leads to 4-byte overwrite when the heap-management data is manipulated.

The paper also clearly mentions that -- "**Overflowed chunk must not be the last chunk"**. Until this point in the paper, we have seen how the last malloc'ed chunk can be overflown and the exploit can be triggered. Even in this case, we would focus on the same technique for sizes greater than or equal to 40-bytes on Solaris. We follow the same procedure as sescribed in [8] except certain minor differences. We will simulate free() by overflowing the last malloc'ed chunk and trigger further exploit operations using the 4-byte overwrite technique. As in [8], we will focus on pointer assignments that take place in free(). In the case of Solaris, free() or realfree() is called internally in malloc(), due to delayed-free policy. We take advantage of the delayed free call and achieve 4-byte overwrite in the realfree()'s **coalesce** operation. This is similar to the exploit mentioned in [8] but differing by overflowing the last malloc'ed chunk.

As mentioned in Solaris [size<40 bytes], we still use the same **bit0** and **bit1** to trigger our exploit. Though many other ways are possible to exploit heap-based overflows, we have chosen this one because it seems to be one of the easiest ways and bypasses the "chunk before last chunk" limitation mentioned in [8]. The exploit can be achieved in just 2 calls to malloc. The first call obtains a chunk, then some operation leads to an overflow. In the overflow, we fake the header of next fabricated chunk and with the size such that bit0 is "0". On the second call to malloc() the 4-byte overwrite is triggered. On Solaris, the next pointer is calculated based on the size in the header. We take advantage of this and make the size greater than **-4** but lesser than **0**. Further more we create our fake chunk starting from the next word after the size word. The NEXT(p) macro gets confused due to manipulated size and returns the next word as the header of the next chunk! In this next fake chunk, we make sure size-bit0 is "0" and bit1 is "1" indicating that there is a previously free'd chunk in contiguous memory. When realfree reaches the last overflown malloc'ed chunk, it finds the last chunk has been previously free'd (Free Simulation) and then further finds the next-chunk is free too. This triggers the **coalesce** operation. We make sure, that the chunks involved are interpreted as list-chunks and not tree-nodes by assigning left-node pointer '-1' as its value. Manipulation of the NEXT(p) macro further helps us to by-pass the **bottom chunk** check.

Digging Deeper

We will refer to the Open-Solaris site [9] for source code to better understand the exploitation. Let's start by looking at the important tree data-structures involved.

Source: mallint.h

```
     80 /* the proto-word; size must be ALIGN bytes */
     81 typedef union _w_ {
     82         size_t          w_i;            /* an unsigned int */
     83         struct _t_      *w_p[2];        /* two pointers */
     84 } WORD;
     85
     86 /* structure of a node in the free tree */
     87 typedef struct _t_ {
     88         WORD    t_s;    /* size of this element */
     89         WORD    t_p;    /* parent node */
     90         WORD    t_l;    /* left child */
     91         WORD    t_r;    /* right child */
     92         WORD    t_n;    /* next in link list */
     93         WORD    t_d;    /* dummy to reserve space for self-
pointer */
     94 } TREE;
```

Few important macros.
Source: mallint.h

```
 ................
    98 #define      RSIZE(b)         (((b)->t_s).w_i & ~BITS01)
 ................
    112 /* set/test indicator if a block is in the tree or in a list */
    113 #define SETNOTREE(b)     (LEFT(b) = (TREE *)(-1))
    114 #define ISNOTREE(b)      (LEFT(b) == (TREE *)(-1))


 ................
    121 #define NEXT(b)            ((TREE *)(((char *)(b)) + RSIZE(b) +
WORDSIZE))
```

Sections of functions relevant to our exploit
Source: malloc.c – realfree()

```
    511         /* see if coalescing with next block is warranted */
    512         np = NEXT(tp);
    513         if (!ISBIT0(SIZE(np))) {
    514                 if (np != Bottom)
    515                         t_delete(np);
    516                 SIZE(tp) += SIZE(np) + WORDSIZE;
    517         }
```

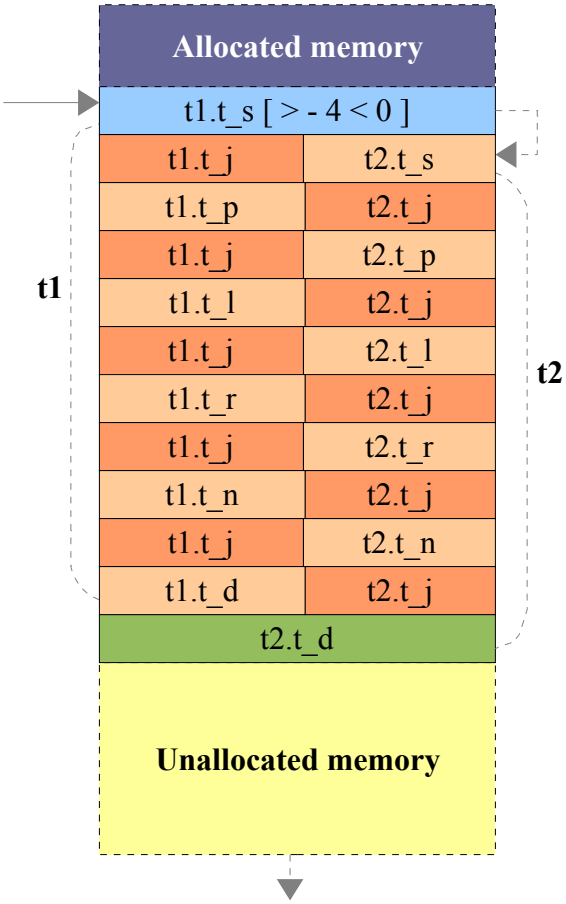
Source: malloc.c – t_delete()

```
    756         /* if this is a non-tree node */
    757         if (ISNOTREE(op)) {
    758                 tp = LINKBAK(op);
    759                 if ((sp = LINKFOR(op)) != NULL)
    760                         LINKBAK(sp) = tp;
    761                 LINKFOR(tp) = sp;
    762                 return;
    763         }
```

Note the above highlighted assignments in orange (747 and 748) are the two word assignments, where user controlled data (8-byte overwrite in this case, but we will still refer it as 4-byte) can be injected. We can summarize above operation in instructions as follows:
0xff2c7808 <t_delete+52>: st %o0, [%o1 + 8]
0xff2c780c <t_delete+56>: st %o1, [%o0 + 0x20]

Now, let's represent the overwrite involving heap-data manipulation diagrammatically.

Malloc'ed heap chunk and overflow



In the figure shown above we have 2 structures involved: **t1.t_* and t2.t_***. Following are the details:

t_s : size. We assign t1.t_s to - 2 so that **np = NEXT(p)** will return np pointing to t1.t_j and bit0 is '0' for both t1.t_s and t2.t_s.

t_j : as every pointer in this structure occupies 2 words owing to alignment logic, we can consider all t_j as junk.

t_p : pointer to previous node, can be junk for t1.t_p, and t2.t_p can be the address with which the return address on the stack is to be replaced.

t_l : can be junk for t1.t_l but must be "**-1**" for t2.t_l, thus guarantee that malloc() would not interpret the node as a tree node but would interpret it as a list node.

t_r : can be completely ignored and hence can be junk.

t_n : t1.t_n can be junk but t2.t_n will be the address we would like to overwrite – 8.

t_d : may be ignored and can be junk for both t1.t_d and t2.t_d.

As each pointer associates an extra word with it, all those ignored words come in handy for practical use in our exploit. We simply overlay our exploit data of next-node-structure over these ignored junk words! It enables the feasibility of a nice and clean exploit. The same size may be manipulated such that the overlay of our fake-chunk header can be at some different location.

Free Simulation Conditional Trigger for Solaris [size > 40 bytes]

We can sum-up the logic for triggering Free Simulation when a malloc() tries to allocate a new chunk and calls realfree and further tries to coalesce. The logical steps involved are as follows:

1. if (size.bit0 equals 0) ....[C]

{

consider this as a free chunk, check if next chunk is also free and if coalesce is possible.

}

2. if (next chunk size.bit0 equals 0)

{

Next chunk in contiguous memory block is free proceed with coalesce.

}

3. size should be such that NEXT(p) calculation will return our fake-chunk as next chunk.

4. The returned fake chunk should bypass is-bottom check [np != Bottom]. Would be automatically taken care of.

5. The value of left-node pointer t_l of fake chunk must be '-1' for interpretation as list node rather than tree node.

6. If (value of left-node equals -1) ....[D]

{

interpret it as list-node and proceed further with coalesce operation involving pointer assignments.

}

As stated before for AIX, the above [C] and [D] are mere logical summaries of the conditional instructions for Solaris. Step [C] indicates Free Simulation. The remaining steps including [D] indicate the trigger to coalesce, the fake-chunk creation, and the coalesce operation that involves pointer assignments for the linked-list.

Advantages of Free Simulation

1. Relatively easy to exploit.
2. Provides a consistent and generic model to pursue the heap overflow-based exploits.
3. For processes / applications where free() is never called, Free Simulation may be the best exploitation technique.
4. Usually data-write follows after a chunk from malloc has been obtained, favoring Free Simulation exploitation.
5. Some heap algorithms do not actually free the memory at the free() call. This delayed/lazy free() behaviour is feasible due to certain supportive free-structures like free-list / flist (Solaris). Whenever a malloc() is called, it internally calls free() or rather the realfree() (especially on Solaris) that actually free's the memory. Hence the focus on malloc() calls might provide an easier approach and save time.
6. Usually, malloc() and realloc() calls are called more frequently compared to free().
7. Exploitation can be triggered at a considerably earlier stage in a process's life cycle because of the fact that the malloc() (memory allocation) always precedes free().
8. Enables arbitrary overwrites anywhere in process memory regions including stack, heap, function pointers, and Procedure Linkage Table.

Limitations of Free Simulation

1. Usually works well and easily when the overflow occurs in last malloc'ed chunk. For overflows in in-between malloc'ed chunks, depends on implementation of the memory allocation algorithm.
2. Needs guesswork to overwrite the addresses , especially for a fake chunk creation.

Preventive Measures

1. Best preventive measure is at the code-implementation level itself by altogether avoiding or by careful usage of function calls that may potentially lead to the memory overflows.
2. At the system level, protection policies like non-executable data regions (that includes the heap and the stack, on AIX – sedmgr) can make heap based overflow vulnerabilities more difficult to exploit.

Conclusion

The need of heap in-band information (metadata) is a necessary evil for practical and efficient malloc implementation. Let's summarize the logical conditional primitives for Free Simulation on both the AIX and the Solaris systems:

Free Simulation Conditional Triggers

AIX	**if (Pointer to Previously Free'd chunk [PPF] != NULL ...[A] && requested_size <= chunk_size) { consider the value of PPF as address of previously free'd chunk and try to allocate memory on this free'd chunk }**
Solaris [size < 40]	**if (size.bit1 equals 1) ....[B] { consider address next to immediate-next word as previously free'd pointer and try to allocate memory on this free'd chunk after size checks. }**
Solaris [size >= 40]	**1. if (size.bit0 equals 0) ....[C] { consider this as a free chunk, check if next chunk is also free and if coalesce is possible. } 2. if (next chunk size.bit0 equals 0) { Next chunk in contiguous memory block is free proceed with coalesce. } 3. size should be such that NEXT(p) calculation will return our fake-chunk as next chunk. 4. This returned fake chunk should bypass is-bottom() check. 5. The value of left-node pointer t_l of fake chunk must be '-1' for interpretation as list node rather than tree node. 6. If (value of left-node equals -1) ....[D] { interpret it as list-node and proceed further with coalesce operation involving pointer assignments. }**

References

1. http://md.hudora.de/presentations/summerschool/2005-09-21/vansprundel-ctt-heapoverflows.pdf - Generic Heap Overflow Exploitations.
2. http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt – Third Generation Exploitation.
3. http://www.openwall.com/advisories/OW-002-netscape-jpeg/ - Solar Designer
4. https://www.usenix.org/publications/library/proceedings/lisa03/tech/full_papers/robertson/robertson_html/ - Run-time Detection of Heap-based Overflows (Anatomy of a Heap Overflow Exploit, Logical Constructs).
5. http://doc.bughunter.net/buffer-overflow/heap-corruption.html
6. http://www.w00w00.org/files/articles/heaptut.txt
7. http://cansecwest.com/csw04/csw04-Oded+Connover.ppt
8. http://www.phrack.org/phrack/57/p57-0x09 – Once Upon a free()
9. http://cvs.opensolaris.org/source/ - Solaris source code on OpenSolaris website