Adobe Reader's Custom Memory Management:

a Heap of Trouble

Version: 1.0

Research and Analysis:	**Haifei Li**	hfli@fortinet.com
Contributor and Editor:	**Guillaume Lovet**	glovet@fortinet.com

Abstract

PDF vulnerabilities are hot. Several AV and security companies, in their 2010 predictions, cited an increase in PDF vulnerabilities volume, possibly driven by demand from Cybercriminals, eager to leverage them in focused and large-scale attacks alike.

But how serious could it really be, and what's the share of casual marketing FUD spreading here? After all, many PDF vulnerabilities out there are structure (i.e. file format) based ones, and essentially result in heap corruption situations. And everybody knows that leveraging a heap corruption bug into actual exploitation, with execution of attacker-supplied code, is no piece of cake. Indeed, MS Windows' heap is hardly predictable, and is armoured with protection mechanisms such as safe-unlinking.

Yet, the main PDF reader software outthere, called Adobe Reader, has a specificity that may lead us to revise our beliefs: for performance purpose, it implements its own heap management system, on top of the Operating System's one. And it turns out that, performance sometimes (often? nah...) being the enemy of security, this custom heap management system makes it significantly easier to exploit heap corruption flaws in a solid and reliable way. Coupled with the recent developments in DEP protection bypass[1], this makes heap corruption exploitation potentially consistent across a very large amount of setups (a very interesting characteristic for the Cybercriminal, either for "blind-shooting" at a targeted system, or for compromising a large amount of systems at once).

This paper introduces Adobe Reader's custom heap management system, dissects its mechanisms, and points out its weaknesses (with examples showing how to obtain control of the execution flow in different heap corruption situations, to illustrate the point) in order to shed light and awareness on the PDF vulnerabilities issue. In addition, limitations will be discussed and possible mitigation leads briefly evoked.

Table of Contents

Introduction

In today's vulnerability landscape, PDF vulnerabilities play an important role. Some may say that they are the link between the vulnerability research community and the malware world. Indeed, 80% of exploits in the wild in Q4 2009 being PDF ones[2], no other class of vulnerabilities has been leveraged by cybercriminals as much as those to silently plant Trojan Horses, keyloggers and other backdoors in their victims' OS.

The cause for such an infamous popularity may be twofold: the ubiquity of PDF Reading Software, making it a target of choice, and above all the false belief among most users that opening a PDF document is close to be the safest move you could do on your computer. Indeed, those are no executable files, and can't even get you a macro virus, as MS Office documents might.

As a matter of course, users cannot be blamed for such beliefs. Actually, the many user education campaigns begging users not to click on executable attachments in emails have probably led them to think that viruses can only take this form. Almost righteously: the latter was actually true until recently, the booby-trapped PDF documents trend having really picked up in 2008 only[3].

Of course, running an up-to-date, patched PDF Reader does not eliminate the risk fully: 0-day, unpatched PDF vulnerabilities frequently circulate in the Wild every now and then. One of such sounded particularly interesting to us, not so much for the vulnerability itself, but rather for the way it was being leveraged. Indeed, in late 2009 a new "high-risk PDF Zero-Day vulnerability" (CVE-2009-3459) was reported on Adobe's blog as being exploited in the Wild, in the frame of a targeted attack. A thorough analysis[4] revealed an original way to leverage this mere heap-corruption flaw into a full-blown, functional exploit -- which is rather uncommon with heap-corruption vulnerabilities, due to the hardened nature of MS Windows' Heap. But here, precisely, the custom Adobe Reader's Heap management was the one to be abused, and it is not any close to being hardened... It lead us to dig into it deeper, and to identify a number of new effective and reliable strategies to turn heap-corruption flaws in PDF Reader into open doors toward shellcode execution.

Overview: Custom Heap Management on Adobe Reader

In "traditional" computer programs, allocation (and conversely, de-allocation, aka "freeing") of memory storage during the program runtime is outsourced to the Operating System: Through a system call, programs request memory storage to the OS, which is then in charge of finding an unused memory block of sufficient size (on the "Heap"), and to manage all the issues that may arise in the process (memory fragmentation, referencing, security, ...).

Probably for performance (in the sense of "speed") reasons, Adobe chose to implement its own Heap memory management system in Adobe Reader, rather than resorting to the built-in OS features. This custom Heap management system defines and makes use of three essential memory structures: **Acro Blocks**, **Acro Cache Blocks**, and **Bib Blocks**.

Acro Blocks are the "top" containers: they may be used "directly" or serve as containers for Acro Cache Blocks or Bib Blocks. The latter two may only exist within an Acro Block -- but cannot be mixed: the same Acro Block can contain either Acro Cache Blocks **or** Bib Blocks, not a mix of both.

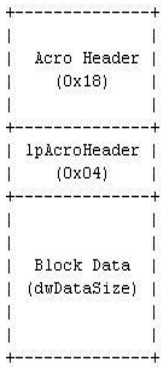
In Adobe Reader, when memory is requested or freed, the custom Heap management system manipulates those structures according to various algorithms, in order to satisfy the request. This is what is described in this paper, along with the subsequent security issues it introduces: we will see how it can enable and solidify exploitation of heap-corruption flaws.

I. Acro Blocks

As evoked above, Acro Blocks are the "base bricks", which the heap management system relies on.

Data Structures

In memory, an Acro Block has the following form:



The lpAcroHeader pointer points to the Acro Header of the block itself (that is to say, to the beginning of the block), which, as we will observe later on, is needed when the custom "free" function is called on a block to unallocate it.

The Acro Header structure is defined as follows:

```
struct acro_header
{
    acro_managing_pool* lpAcroPool; // Points to the Acro Managing Pool
    DWORD               reserved    // Set to 0
    DWORD               flag;       // Type of Block. Set to 2 for Acro Blocks
    acro_header*        Blink;      // Previous Acro Header in the list
    acro_header*        Flink;      // Next Acro Header in the list
    DWORD               dwDataSize  // Size of the Block Data
}
```

The Blink and Flink pointers immediately suggest that the Acro Blocks are organized as a doubly-linked list in memory. The "Head" of this list has its Blink pointer set to NULL.

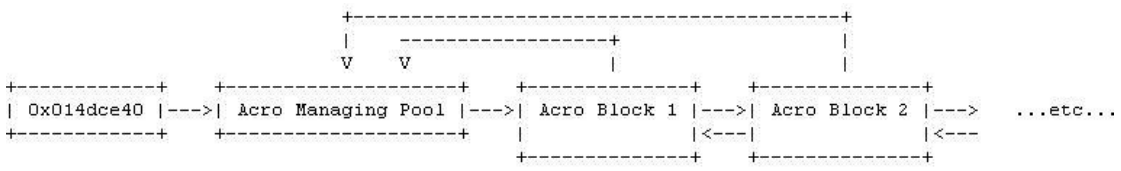
The Acro Managing Pool is defined as follows:

```
struct acro_managing_pool
{
    DWORD           reserved[3];
    cache_managing* lpCacheManaging[32]; // Managing structures for the Acro Cache
    DWORD           reserved;
    acro_header*    lp_head_acro_header; // Header of the first Acro Block in the list
}
```

lp_head_acro_header thus stores the head of the doubly-linked list of Acro Blocks. It is located at the offset 0x90 of this structure.

Organization

We already know that Acro Blocks are organized as a doubly-linked list in memory, the head of which being a field of the Acro Managing Pool structure. The final piece of the puzzle is the address of this structure in memory, which sits at a hardcoded location (For Adobe Reader 9.3.1: 0x014dce40). Thus, if we recollect everything, we obtain the following figure:



The address showed above, 0x014DCE40, is a hard-coded fixed address and the value only depends on the version of the AcroRd32.dll (this value corresponds to Adobe Reader 9.3.1).

Knowing the Adobe Reader version, it is therefore possible to access all the Acro Blocks and managing structures, by following the various pointers.

The process of allocation and un-allocation of acro block will be discussed in the following section, along with Acro Cache Blocks.

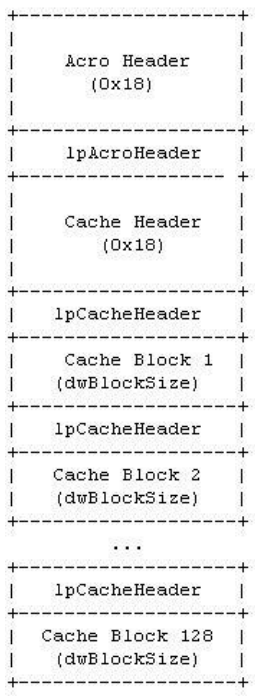
II. Acro Cache

An Acro Block might be returned to the application requesting storage memory to the Heap management system (more precisely, it's a pointer to its data block that is returned), in which case it is said to be used "directly", and its data block will then be used to store application data.

But as evoked above, it might as well contain Cache Blocks. In that case, the Acro Block is used as a Heap cache, that we'll refer to as "Acro Cache".

Data structures

An Acro Cache looks like the following:



In other words, an Acro Cache is made of a Cache Header (0x18 bytes) and 128 Cache Blocks of constant size (dwBlockSize bytes), each preceded by a pointer to the Cache Header (used by the management system for unallocation purpose).

The possible Cache Block size in Adobe's custom heap management system (dwBlockSize above) range from 8 bytes to 128 bytes, and is necessarily a DWORD multiple. Therefore, there are 31 possible Cache Block sizes: 8, 12, 16, ...etc... until 128 bytes.
Accordingly, there are 31 "types" of Acro Cache, since a given Cache contains only Cache Blocks of the same size.

That said, the Cache Header structure is defined as follows:

```
struct cache_header
{
    cache_managing* lpCacheManaging;  // Pointer to the Cache Manager structure
    DWORD           dwAllocatedBlocks;// Number of allocated Cache Blocks among the 128 ones in this Acro Cache
    DWORD           flag;             // Type of object. Set to 0 for Acro Cache
    cache_header*   Blink;            // Previous Acro Cache
    cache_header*   Flink;            // Next Acro Cache
    DWORD           dwBlockSize;      // Size of contained Cache Blocks
}
```

The presence of Blink and Flink pointers suggest that Acro Caches of the same kind (i.e. containing Cache Blocks of the same size dwBlockSize) are organized as a doubly-linked list. The existence of a Cache Managing structure for that doubly-linked list is suggested by the lpCacheManaging pointer. This structure is defined as follows:

```
struct cache_managing
{
    acro_managing_pool* lpAcroPool;           // Pointer to the Acro Managing Pool (see I.)
    free_cache_block*   lp_head_free_block;   // Head of Free Cache Blocks list
    cache_header*       lp_head_cache_header; // Head of Acro Caches list
    DWORD               dwBlockSize;          // Managed Cache Blocks size
}
```

As a matter of course, there is one instance of this structure per kind of Acro Cache (thus 31 instances). Each instance of this structure points to the list of Acro Caches of a kind, and is *pointed to* by the Acro Managing Pool that was defined in section I. above. Indeed, we recall that the Acro Managing Pool structure has the following field:

```
cache_managing* lpCacheManaging[32];   // Managing structures for the Acro Cache
```

This array obviously has an entry for each of the 31 Cache Managing instances, leading to the 31 lists of Acro Caches (plus one unused entry).

Free Cache Blocks

Now, it also appears that the Cache Managing structure has a pointer to a list of "Free Cache Blocks". As a matter of fact, Free Cache Blocks are simply Cache Blocks (thus contained in an Acro Cache of the list pointed to by the Cache Managing instance), that are cast into the Free Cache Blocks structure, defined as follows:

```
struct free_cache_block
{
    free_cache_block* Blink; // Previous free cache block
    free_cache_block* Flink; // Next free cache block
    ...
}
```

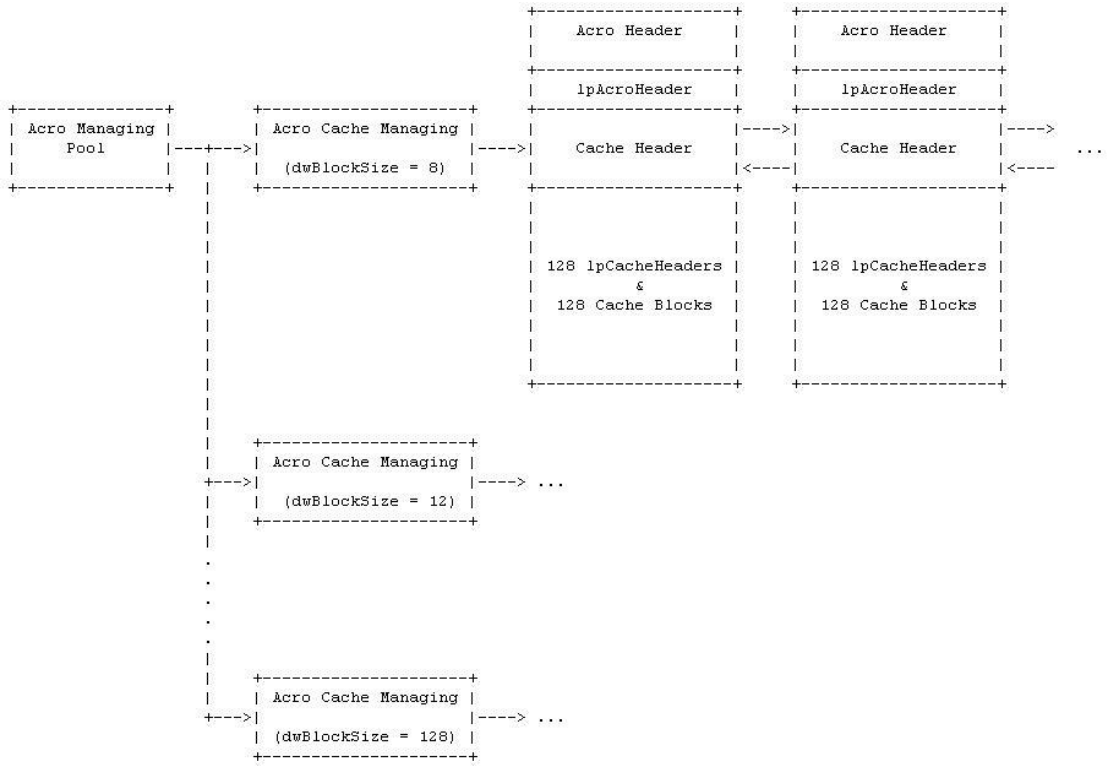
Free Cache Blocks are thus doubly-linked to one another, the head of the list being pointed to by the Cache Managing instance. At this point it is important to understand that once a Cache Block has been allocated - and therefore its address passed to the application requesting storage

memory - this Blink/Flink data will be overwritten by whatever data the application will store in the block (to be more precise, the functions wrapping the acro_allocate call do a memset 0 on the block before passing it to the application, so the latter cannot retrieve the pointers, for whatever more or less malicious intent it may have...). MS Windows' heap management system resorts to a similar process.

An astute reader might have now understood why the minimum size possible for a Cache Block is 8 bytes (rather than 4 bytes, as the 4 bytes increment for possible size would suggest): it's to make room for these 2 pointers when a block is not allocated.

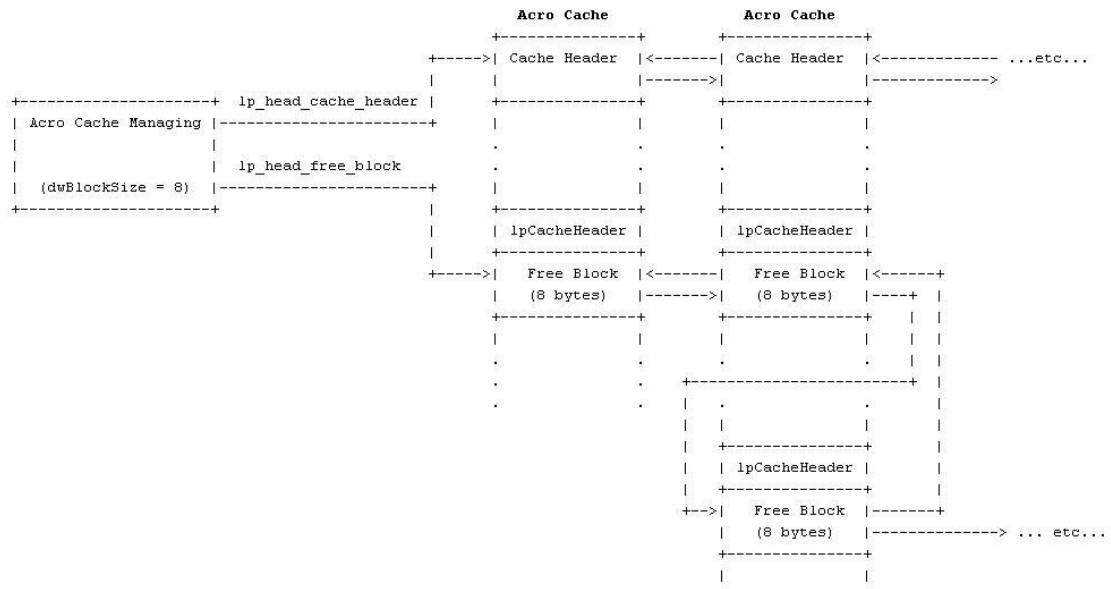
Organization

If we assemble all the pieces of the puzzle, the Acro Cache system is organized as the following:



It could be said that all the Acro Caches together and their managing structures effectively form "the Cache" in memory.

Now "Zooming" on the Cache Blocks will highlight the Free Cache Blocks list. The following example illustrates this:



Allocation

When the application queries Adobe's heap management system for storage memory space (say, of size dwSize), and that certain conditions are met, the function acro_allocate is called. In general, allocation requests processed by acro_allocate - thus using the Acro Cache system - are those concerning basic functions (eg: stream decoding, processing top objects in PDF such as "/Pages", "/Page", etc...)

The general logic of this function is to distinguish between two cases:

1. If **dwSize > 128 bytes**, then the system allocates a "direct" Acro Block, whose Data Block has the requested dwSize. Then it returns a pointer to that Data Block to the application. The allocation for the Acro Block is done by asking the OS for some heap space. This is the slower scenario, since it operates as an interface between the application and the traditional OS Heap management system, adding overhead where the OS could be queried directly (by the application).

2. If **dwSize <= 128 bytes**, then the system looks for a Free Cache Block whose size suffices to contain the requested dwSize bytes, and returns a pointer to it (after having updated the relevant structures to unlink it from the Free Cache Blocks list). This is the faster scenario, where storage memory blocks are not requested to the underlying OS, but fetched directly from the Cache.

The pseudo-code of the acro_allocate() function is the following:

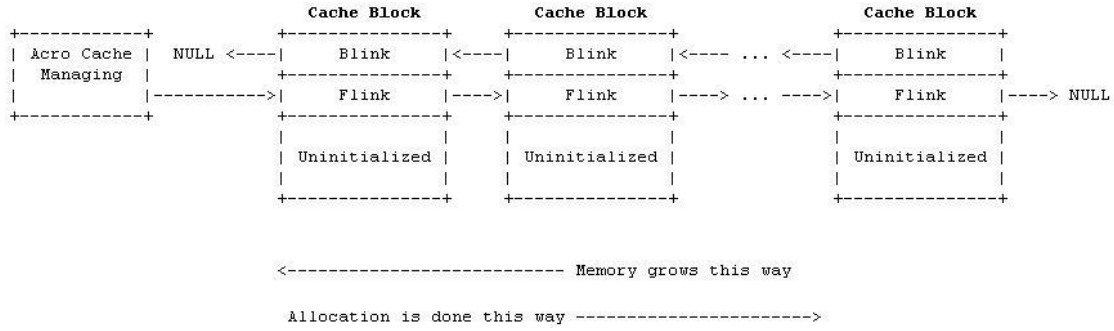
```
void* acro_allocate(DWORD dwSize)
{
    if (dwSize > 128)
    {
        // Allocate an acro block
        malloc(dwSize + sizeof(acro_header) + sizeof(lpAcroHeader))

        ... [Acro management to adjust the control structures] ...

    }
    else // size <= 128
    {
        // Obtain the proper "cache_managing" structure
        cache_managing lpCacheManaging = acro_managing_pool->lpCacheManaging[m_Magic[dwSize]];

        // First free cache block
        free_cache_block p_block = lpCacheManaging->lp_head_free_block;

        if (p_block == NULL) // No Free Cache Block of this size available
        {

           ... [ call acro_allocate(( dwSize + 4 ) * 128 + sizeof(cache_header)) to allocate
                 a large acro block, where the payload is initialized to contain 128 cache blocks ] ...

        }
        else // a Free Cache Block was available
        {
           // Updates the list of Free Cache Blocks
           free_cache_block* p_next = p_block->Flink;
           if (p_next != NULL)
           {
               p_next->Blink = NULL;
           }

           lpCacheManaging->lp_head_free_block = p_next;

           cache_header *lpCacheHeader = (cache_header *)*(DWORD *)((unsigned char*)p_block - 4);
           lpCacheHeader->dwAllocatedBlocks++;

           // Returns the found Free Cache Block to the caller
           return p_block;
        }
    }
}

// Small trick to quickly transform the requested dwSize into the index of the Acro Cache Managing structure
// that corresponds (i.e. which dwBlockSize is just above dwSize)
unsigned char m_Magic[128]
{
    0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x02,0x02,0x02,0x02...0x1F,0x1F,0x1F,0x1F
}
```

The m_Magic array, used to convert the requested size into the index of a cache_managing instance might be seen as a minimalistic Hash function, allowing random access (as opposed to sequential access) to the Cache Managing structure we need (i.e. the one which size is immediately superior to the requested storage memory space, dwBlockSize).

Then, the system looks at the head of the Free Cache Blocks list pointed to by the obtained Cache Managing structure. If it's not null, it pulls it out and returns its address. If it's null, it must allocate a new Acro Cache of the same kind (inside an Acro Block) and initialize it, so it can purvey fresh Free Cache Blocks.

Initialization

Once the Acro Cache has been allocated - which requires a system call to the traditional OS heap management system, and setting the relevant pointers in the headers and managing structures - the fresh Free Cache Blocks are initialized: their Blink and Flink pointers are set, in the following fashion:



As shown above, after initialization, the last block by memory address is the first Free Cache Block on the Free Cache Block doubly linked list. When the acro_allocate() function finally returns, it provides the caller with a pointer to the Free Cache Block list head (and then unlinks it, as described in the Allocation section above), which is the one with the highest memory address. In other words: Allocation of Cache Blocks to the caller is done from higher addresses to lower addresses. Something that might be useful to remember in exploitation conditions.、

Unallocation

The unallocation process is rather straightforward. The caller application that wishes to free a block of storage memory once it's done with it passes the address of the block to the acro_free() function. Freeing the block then essentially involves 4 phases:

1. The function locates the header. It's easy: no matter what block it is (Acro Block or Cache Block), a pointer to the block header (or the cache header) sits just above the block (lpHeader or lpCache Header in previous sections above).

2. The function identifies the type of block, thanks to the flag in the header

3. If the block is a Cache Block, it simply adds it to the head of the Free Cache Block list of its kind (this of course involves setting pointers in the Cache Managing structure and the previous head of the list).

4. If the block is an Acro Block, it unlinks it from the Acro Block list, and asks the OS to free it (traditional free).

For reference, the pseudo-code of “acro_free()” is given below:

```
void acro_free(void *lpHeap)
{
  // The header sits one DWORD above the freed block
  void* lpHeader = *(void *)(lpHeap - 0x04);

  // The flag is at offset 0x08 of the header
  DWORD flag = *(DWORD *)(lpHeader + 0x08);

  if (flag == 0)
  {
    // Flag 0 indicates a Cache Block

    cache_header*     lpCacheHeader   = (cache_header *)lpHeader;
    cache_managing*   lpCacheManaging = lpCacheHeader->lpCacheManaging;
    free_cache_block* lpBlock         = (cache_block *)lpHeap;

    // The freed block becomes the head of the Free Cache Blocks list
    lpBlock->Blink   = NULL;
    cache_block* tmp = lpCacheManaging->lp_head_free_block;
    tmp->Blink       = lpBlock;
    lpBlock->Flink   = tmp;
    lpCacheManaging->lp_head_free_block = lpBlock;

    //update the number of allocated blocks
    lpCacheHeader->dwAllocatedBlocks--;
  }
  else if (flag == 1)
  {
    //unused currently
  }
  else
  {
    // Flag 2 indicates an Acro Block
    acro_header* lpAcroHeader = (acro_header *)lpHeader;

    //update the Acro Blocks list
    if (lpAcroHeader->Flink != NULL)
    {
      lpAcroHeader->Flink->Blink = lpAcroHeader->Blink;
    }

    if (lpAcroHeader->Blink != NULL)
    {
      lpAcroHeader->Blink->Flink = lpAcroHeader->Flink;
    }
    else
    {
      lpAcroHeader->lpAcroPool->lp_head_acro_header = lpAcroHeader->Flink;
    }

      ... [ Then free the acro block with a system call ] ...
  }
}
```

It can be noted that since the freed Cache Blocks are added at the head of the Free Cache Blocks list (i.e. the Acro Cache Managing structure points directly to it), the allocation/unallocation strategy is "Last Freed First Used". Again, this might be useful in exploitation context.

III. Exploiting the Acro Cache

The Acro Cache is undeniably an interesting concept, and likely serves its purpose well: it speeds up allocation of "small" storage memory blocks by eliminating the need to resort to system calls. If we consider that this operation in traditional applications is frequently a speed execution bottleneck, then it'd certainly speed up the whole application.

However, what it gains in speed, it gives it up in security -- as compared to a traditional Heap Management System outsourced to the OS. We will point why in this section.

Exploitation Strategies

Traditionally, there are two main ways to exploit Heap corruption flaws (Heap overflow, use after free, integer overflow, etc...): Either the attacker tries to leverage the flaw to **overwrite some application-provided** data sitting in the heap (of course this data must pertain to the execution flow, for the attacker to gain control), or she leverages it by **corrupting the internal structures** used by the Heap management system (eg: block headers, etc...) so as to make the system itself overwrite "interesting" (i.e. pertaining to the execution flow) data, for instance during blocks unlinking operations, where several pointers are updated.

Today, both strategies have a limited efficiency with OS Heap management systems. Indeed, over time, those have implemented security measures to address the issue. For instance, Windows XP SP2 introduced "safe unlinking", which prevents exploitation by corrupting internal structures. As for application heap data overwrite, it is poorly effective because the Heap state at any given point in execution is hard to predict: running twice the same application in a row will lead to two different heap configurations, hence a non-solid relative position of the targeted data and the overflow start point.

The situation with the Acro Cache system for heap blocks management is however different, and both strategies are here relevant.

Overwriting Application Data - Practical Example

For the sake of research, let's assume we are able to overwrite the Heap from a given "vulnerable" Cache Block, via whatever relevant flaw in the application (i.e. Adobe Reader). Two essential questions are to be answered:

1. Is there any data within a Cache Block in the same Acro Cache (but sitting after our vunerable Block) that pertains to the execution flow?
2. If yes, is the distance between this targeted Cache Block and our vulnerable Block predictable enough to overwrite this data piece with precision (i.e. without smashing the whole heap)?

The Key Pointer

In short, the answer to the first question above is: Yes. Plenty.

The prime targets for such data may be function pointers. Indeed, once overwritten to point to attacker-controlled data (i.e. a shellcode), the next call that dereferences them (typically: call [lpFunction] in assembly) happily transfers control to the attacker.

Also interesting are the v-pointers, generated by C++ compilers to implement the language polymorphism. Sitting in instances of a virtual class, a v-pointer points to this class' v-table, and is therefore used to call the class member functions from the considered instance, via double dereferencing. Objects being typically allocated on the heap via calls to new(), it is not extremely challenging to find v-pointers in heap blocks of the Acro Cache system.

For the sake of the experiment, we will consider the so-called "Key Pointer" (lp_key below), which is exactly that: a v-pointer pointing to a v-table sitting at the fixed address 0x0124f878 on Adobe Reader 9.3.1. It is just an example, but its somewhat frequent presence in several Cache Blocks seems to make it a... Key element of Adobe Reader's implementation. Thus an excellent candidate for "execution flow relevant" data to overwrite in an exploitation scenario.

Predictability

Answering the second question above may be a little bit more challenging. Mostly, the answer is yes -- at least more so than with OS-allocated Heap blocks. Indeed, the allocation and unallocation of Cache Blocks following a known algorithm exposed in section II above (Last Freed First Used), logically the distance between two given blocks (for what matters to us, the vulnerable one and the targeted one) is predictable at any point in execution -- and constant over two different executions in time. However, in a multi-threaded environment, running on heterogeneous systems, things tend to get not so clear-cut.

Again, for the sake of the experiment, let's consider a very basic PDF document, whose source is given below for reference:

```
%PDF-1.1

1 0 obj
<< /Type /Catalog /Pages 2 0 R >>
endobj

2 0 obj
<< /Count 1 /Kids [3 0 R] /Type /Pages >>
endobj
```

```
3 0 obj
<< /Contents [4 0 R] /Type /Page /Parent 2 0 R>>
endobj

xref
0 4
0000000000 65535 f
0000000012 00000 n
0000000071 00000 n
0000000140 00000 n
trailer
<<
/Size 4
/Root 1 0 R
>>
startxref
217
%%EOF
```

When opening it with Adobe Reader, memory will be allocated - partly in the form of Cache Blocks. Knowing the organisation of the Acro Cache (see section II above), we use a debugger to dump the number of Cache Blocks of each of the 31 kinds that were allocated during 2 successive openings of this document, on the same system:

First Opening	Second Opening
allocated 000003AF [0x08-length] blocks	allocated 000003B2 [0x08-length] blocks
allocated 000007CA [0x0C-length] blocks	allocated 000007CE [0x0C-length] blocks
allocated 00000800 [0x10-length] blocks	allocated 00000805 [0x10-length] blocks
allocated 00000669 [0x14-length] blocks	allocated 0000066C [0x14-length] blocks
allocated 00000BCF [0x18-length] blocks	allocated 00000BDE [0x18-length] blocks
allocated 0000027B [0x1C-length] blocks	allocated 0000027C [0x1C-length] blocks
allocated 000003A3 [0x20-length] blocks	allocated 000003A4 [0x20-length] blocks
allocated **00000093** [0x24-length] blocks	allocated **00000093** [0x24-length] blocks
allocated **00000310** [0x28-length] blocks	allocated **00000310** [0x28-length] blocks
allocated **00000195** [0x2C-length] blocks	allocated **00000195** [0x2C-length] blocks
allocated **00000185** [0x30-length] blocks	allocated **00000185** [0x30-length] blocks
allocated **00000037** [0x34-length] blocks	allocated **00000037** [0x34-length] blocks
allocated 000000E0 [0x38-length] blocks	allocated 000000E1 [0x38-length] blocks
allocated **00000027** [0x3C-length] blocks	allocated **00000027** [0x3C-length] blocks
allocated **000000A7** [0x40-length] blocks	allocated **000000A7** [0x40-length] blocks
allocated **0000006E** [0x44-length] blocks	allocated **0000006E** [0x44-length] blocks
allocated 00000218 [0x48-length] blocks	allocated 00000216 [0x48-length] blocks
allocated 0000002A [0x4C-length] blocks	allocated 0000002B [0x4C-length] blocks
allocated **00000015** [0x50-length] blocks	allocated **00000015** [0x50-length] blocks
allocated 00000113 [0x54-length] blocks	allocated 00000112 [0x54-length] blocks
allocated **0000002B** [0x58-length] blocks	allocated **0000002B** [0x58-length] blocks
allocated **00000014** [0x5C-length] blocks	allocated **00000014** [0x5C-length] blocks
allocated **00000029** [0x60-length] blocks	allocated **00000029** [0x60-length] blocks
allocated **00000006** [0x64-length] blocks	allocated **00000006** [0x64-length] blocks
allocated **00000009** [0x68-length] blocks	allocated **00000009** [0x68-length] blocks
allocated **00000005** [0x6C-length] blocks	allocated **00000005** [0x6C-length] blocks
allocated **0000000A** [0x70-length] blocks	allocated **0000000A** [0x70-length] blocks
allocated **00000008** [0x74-length] blocks	allocated **00000008** [0x74-length] blocks
allocated **00000009** [0x78-length] blocks	allocated **00000009** [0x78-length] blocks
allocated **00000019** [0x7C-length] blocks	allocated **00000019** [0x7C-length] blocks
allocated **0000000C** [0x80-length] blocks	allocated **0000000C** [0x80-length] blocks

Highlighted in bold are the amounts of allocated blocks that did not vary from one opening to the other. Bigger blocks (which are allocated less often) tend to follow a more stable allocation figure, therefore distance between 2 big cache blocks will be more stable and predictable than between 2 small blocks. In other words: the bigger the vulnerable Cache Block, the more chances to craft a reliable exploit of the "application-data overwrite" kind.

Connecting the dots

To follow up on our experimental case, let's assume our vulnerable block is the next 128 bytes Cache Block to be allocated (it could be another big one, it is not extremely important) at a given point in execution, and let's dump its contents from the debugger used above, attached to an Adobe Reader instance displaying our basic PDF document.

We know that the Acro Managing Pool is pointed to by a pointer at 0x014DCE40, and that it contains at offset 0x0C an array of pointers to the Cache Managing structures (see section I). We're looking for the one for 128 bytes blocks (i.e. the 32nd one in the array), which at offset 0x04 has a pointer to the head of the Free Cache Blocks list. According to the allocation process described in section II, this is our block. To obtain its contents, we therefore issue the following command:

```
0:008> dd poi(poi(poi(0x014dce40)+0x0c+0x1f*4)+4)

0222db8c  00000000 0222db08 862a0906 0df78648
0222db9c  05010101 004b0300 41024830 697de600
0222dbac  76e1bea7 b80af241 88eb03d9 09f49099
0222dbbc  61759b5d 1e30caf1 a8ec15e9 4f047b2a
0222dbcc  18760000 f6443d72 9ad121f9 a299e3bd
0222dbdc  c416fb7b 9b75c1d8 40dff9cf 021f1b37
0222dbec  01000103 00000040 00001876 00000000
0222dbfc  00000000 00000000 00000000 00000000
```

The next block in the Acro Cache is situated 132 bytes further (128 bytes for our block, plus the 4 bytes of the lpCacheHeader pointer preceding the next block).

Dumping it gives nothing interesting, thus we dump the "next-next" one in the Acro Cache:

```
0:008> dd poi(poi(poi(0x014dce40)+0x0c+0x1f*4)+4)+0x80+4+0x80+4

0222dc94  0124c080 021c9088 0124f878 000005dc
0222dca4  00184540 ffffffff 00000000 00000000
0222dcb4  00000000 00000000 00000000 00000000
0222dcc4  00000186 00000000 00000000 00000000
0222dcd4  00000001 00000000 00000000 00000000
0222dce4  00000000 0124c070 00000000 00000000
0222dcf4  0233ecf0 00000001 0124c078 00000000
0222dd04  00000000 0233ed08 00000000 00000000
```

We immediately recognize the value of the lp_key pointer at the third DWORD, highlighted above. To validate our hypothesis, we overwrite it from our vulnerable block, simulating a heap overflow. It gives us:

```
0:008> dd poi(poi(poi(0x014dce40)+0x0c+0x1f*4)+4)

0222db8c   44444444 44444444 44444444 44444444
0222db9c   44444444 44444444 44444444 44444444
...
0222dc94   44444444 44444444 55555555
```

Then we command the debugger to resume execution, which triggers the following exception:

```
(380.298): Access violation - code c0000005 (first chance)

009da8ff    833858          cmp         dword ptr [eax],58h    ds:0023:55555555=????????
```

Register eax is loaded with the value we used for overwriting the lp_key v-pointer value (55555555). This does not correspond to a valid address, hence the exception. However, let's dump the code following the instruction that triggered the exception, to see what would happen if we used a valid address:

```
0:000> u eip

009da8ff    833858          cmp         dword ptr [eax],58h
009da902    7610            jbe         AcroRd32!AVAcroALM_IsFeatureEnabled+0x4596f (009da914)
009da904    8b4058          mov         eax,dword ptr [eax+58h]
009da907    85c0            test        eax,eax
009da909    7409            je          AcroRd32!AVAcroALM_IsFeatureEnabled+0x4596f (009da914)
009da90b    8b490c          mov         ecx,dword ptr [ecx+0Ch]
009da90e    894c2404        mov         dword ptr [esp+4],ecx
009da912    ffe0            jmp         eax
```

The highlighted instruction loads eax with the address of the function at offset 0x58 in the v-table pointed to by our lp_key. Then control is transferred to that function by the final jmp. Therefore, if instead of 55555555, we had put the address of a v-table crafted by us, we would effectively gain control of the execution flow.

As a side note, although the default on recent versions of Adobe Reader is to have the DEP protections enabled, as long as we can set the EIP to what we want those protections can be bypassed. The whys and hows are out of the scope of this paper, but are have been exposed in the literature[5][6]. In our example above, it is for instance possible to use the ecx register (loaded with the dword at offset 0x0C in the targeted block - thus data we control) to flip the heap to the stack, and "run" a return oriented shellcode there. This is left as an exercise for the reader.

Corrupting the Structures

Corrupting the internal structures used by the heap management system is the another main strategy relevant to exploiting heap-based vulnerabilities. It is by large similar to what was done on Windows (and other OS) heap management system, prior safe-unlinking appearance in the SP2 in 2004[7].

In a Blink/Flink of an eye

As exposed in section III, when a block is given to acro_free() for freeing, the latter function:
1. Retrieves the header pointer of the block (DWORD right above the block)
2. Checks the type flag in the header
3. If the type is Acro Block (flag >= 2), unlinks the block out of the doubly-linked list of Acro Blocks it belongs to.

Of course, pulling an element out off a doubly-linked list (i.e. "unlinking" it) involves updating two pointers, to keep the list consistently chained: the FLINK pointer of the previous block, and the BLINK pointer of the next block. In the code of acro_free, this is reflected by the two instructions:

```
lpAcroHeader->Flink->Blink = lpAcroHeader->Blink;
lpAcroHeader->Blink->Flink = lpAcroHeader->Flink;
```

Where lpAcroHeader is the header pointer of the block to unlink. Knowing that the Blink and Flink fields of the acro_cache structure (defined in section I) are at respective offsets 0x0C and 0x10 in the structure, this could also be written in the more "machine-oriented" way:

```
[[lpAcroHeader + 0x10] + 0x0C] = [lpAcroHeader + 0x0C]
[[lpAcroHeader + 0x0C] + 0x10] = [lpAcroHeader + 0x10]
```

Where brackets are the dereference operator.

Now, in an exploitation scenario (say, a heap overflow situation), we could very well overwrite the lpAcroHeader or the lpCacheHeader of a block with the address of a fake header we crafted, conforming to the acro_header structure described in section I:

Assuming that CCCCCCCC (type flag) >= 2, when the block whose lpAcroHeader (or lpCacheHeader) was hijacked will be unallocated, the system will perform unlinking according to data in our forged header. This in effect results in the two operations:

[EEEEEEEE + 0x0C] = DDDDDDDD
[DDDDDDDD + 0x10] = EEEEEEEE

This is equivalent to:

[X] = Y
[Y + 0x10] = X - 0x0C

Where X = EEEEEEEE + 0x0C and Y = DDDDDDDD

In other words, we can get the system to write whatever value Y we want, at whatever address X we want, which can easily be used to gain control of the execution flow. **But** there is a side effect: the value X - 0x0C will be written at address Y + 0x10. This may prevent exploitation, depending on the conditions (see below).

Heap Spraying on Adobe Reader

In the scenario described above, overwriting the lpAcroHeader (or the lpCacheHeader) of an Acro Block (or a Cache Block, respectively) might actually be the easy part. The difficult part being to overwrite it with the correct value, pointing to our forged acro_header sitting on the heap. Indeed, this implies that the address of the latter is consistently predictable.

The best approach here is in fact to fill the heap with our forged header, to maximize the chances of the corrupted pointer pointing to an instance of it. The heap spraying technology resorting to javascript, described by Alexander Sotirov[9][10], fits this task. Importing his Javascript spraying code in Adobe Reader yields following allocations:

```
alloc(0xfffe0) = 0x02E70020
alloc(0xfffe0) = 0x03D80000
alloc(0xfffe0) = 0x03E80000
alloc(0xfffe0) = 0x04160000 <-- contiguous allocations after this point
alloc(0xfffe0) = 0x04260000
alloc(0xfffe0) = 0x04360000
alloc(0xfffe0) = 0x04460000
alloc(0xfffe0) = 0x04560000
alloc(0xfffe0) = 0x04660000
alloc(0xfffe0) = 0x04760000
alloc(0xfffe0) = 0x04860000
alloc(0xfffe0) = 0x04960000
alloc(0xfffe0) = 0x04A60000
alloc(0xfffe0) = 0x04B60000
alloc(0xfffe0) = 0x04C60000
alloc(0xfffe0) = 0x04D60000
alloc(0xfffe0) = 0x04E60000
...
```

If we spray our fake header in each 1MB block, we ensure that the middle blocks in contiguous blocks are very likely to contain our fake header at certain fixed addresses.

Non-DEP conditions

Now, if **D**ata **E**xecution **P**revention is not enabled, we can take control of the execution flow by simply rewriting a function pointer stored at a fixed memory address X to a shellcode we put (the term "spray" might be better adapted...) on the heap at address Y. As a side effect, this will overwrite the dword at Y + 0x10, which does not matter, as it will likely be on the heap also, and will not trigger an access violation.

The code snippet below exposes a function pointer at the fixed address 0x014C56A4, which makes a very good candidate; indeed, it yields a call to "msvcr80!free()", in order to free the block at the OS level. It will therefore be used right after the unlinking operation. In other words: the execution flow will jump to our shellcode right after the function pointer to it was set.

```
0095BD80    8B4424 04           mov     eax, dword ptr [esp+4]
0095BD84    8B0D 5CD24D01       mov     ecx, dword ptr [14DD25C]
0095BD8A    50                  push    eax
0095BD8B    51                  push    ecx
0095BD8C    FF15 A4564C01       call    dword ptr [14C56A4]         ; AcroRd_1.0095BDA0
```

Again, this is just an example.

DEP conditions

If Adobe Reader's DEP is enabled, the above will obviously fail, since we will not be able to execute code on the heap. Having the function pointer above point to "interesting" code living in an existing function for DEP bypassing purpose (as also described in existing literature[5][6]) would be an interesting approach, but would also fail: The side effect would result in attempting to write a dword at offset 0x10 from the code entry point, thereby triggering an access violation error (functions of course sit in non-writeable pages).

Now, if instead of messing with a function pointer as above, we choose to overwrite a v-pointer, and have it point to a v-table we crafted on the heap, that limitation disappears: the side effect will write a dword at offset 0x10 of the forged v-table, which is not a problem (as long as the function offset in the v-table selected by the caller is not 0x10, of course).

Adobe Reader actually makes use of many v-pointers; of course, those frequently used in execution flows involving no user interaction are obviously preferred for exploitation. The following highlights such a v-pointer, found in Annots.api (for Adobe Reader 9.3.1):

```
.text:2210F1F0          mov     eax, dword_223DB4C8
.text:2210F1F5          push    esi
.text:2210F1F6          push    [ebp+var_4]
.text:2210F1F9          call    dword ptr [eax+4]
```

The strategies discussed in this section are of course applicable to various vulnerability scenarios beyond simple heap overflow. It was demonstrated with Cache Blocks, but it can be likewise used with acro blocks in special cases, such as heap underflow, as well as with the BIB blocks described in section IV.

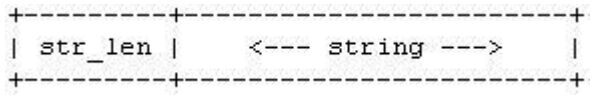
Adjusting the Memory State

In certain exploit scenarios, such as those leveraging use-after-free vulnerabilities, effective exploitation requires adjusting the memory state, in order to have a Cache Block filled with data we control at a relatively precise place (usually, around where the freed object about to be used was standing...).

There are many ways to achieve this in the Acro Cache system. One of them is to use the "Show text" operator ("Tj") in the stream of the "Contents" object of the crafted PDF document[8].

```
BT              // text object start
/F1 12 Tf       // set font
<string> Tj     // show a string
ET              // text object end
```

The "Tj" operator will cause the allocation of an Acro Block or Cache Block with the following contents:



str_len is a USHORT, therefore, a 0x7E-long string will yield the allocation of a Cache Block of size 0x80.

This simple trick allows for relatively effective memory adjusting.

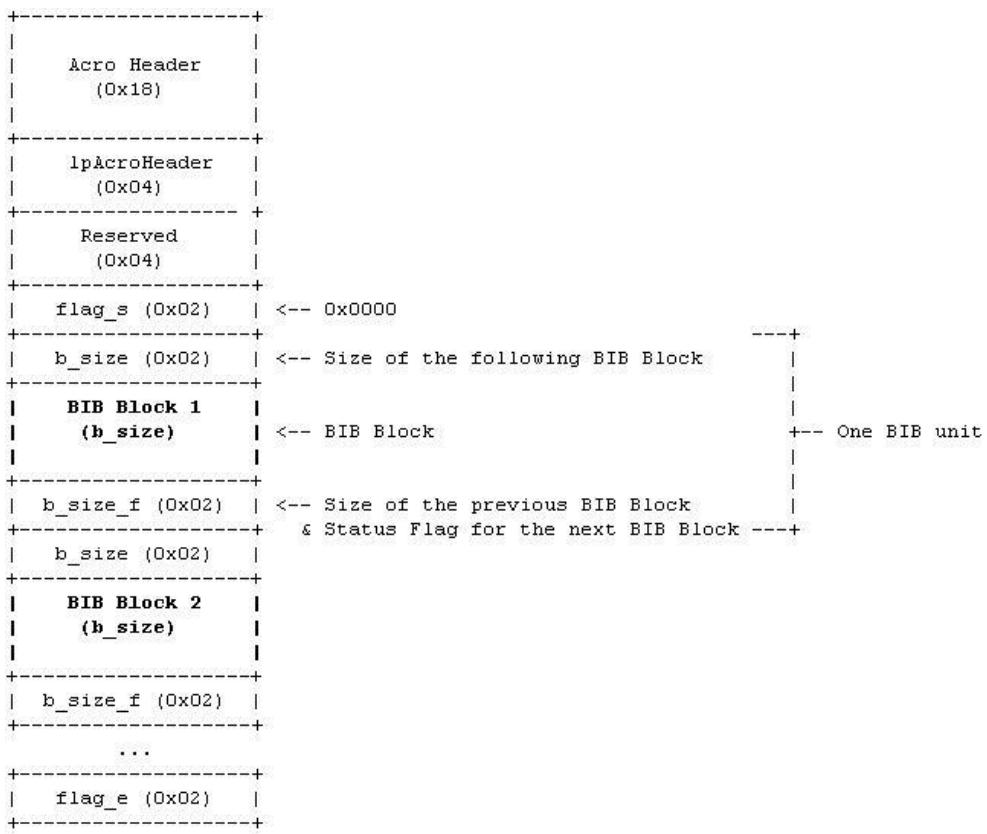
IV. BIB Cache

As mentioned in the Overview section, the Acro Blocks may contain another type of blocks, called "BIB Blocks". Much like the Acro Cache Blocks studied in previous sections, the BIB Blocks form a heap blocks cache, with its own management system and its own allocation/unallocation algorithms, different from the Acro Cache one. We'll refer to this specific cache as "the BIB Cache".

The BIB Cache is mainly used when resources within the PDF are being processed by the application, such as, say, when parsing a font stream. Thus, whenever a heap corruption fault occurs in the handling of a given PDF stream, the subsequent vulnerability may be a BIB block related one.

Data Structures

An Acro Block containing BIB Blocks looks like the following:



As can be seen above, a BIB Block has a variable size, specified by the USHORT (2 bytes) value sitting before **and** after it: *b_size* and *b_size_f* respectively.

In addition of containing the size of the preceding BIB Block, *b_size_f* also carry the Status boolean flag of the next BIB Block (1: free, 0: allocated). This flag sits in the bit of lesser weight of

the USHORT value. For instance, if the value of a given *b_size_f* is 0x19, it means that the size of the preceding BIB Block is 0x18, **and** that the following BIB Block is free.

An immediate consequence of such a system is that the size of BIB Blocks can only be even.

That said, the two important points here are:

1. The payload size of the Acro Block ("dwPayLoad" of the acro_header structure -- see Section I) is always 65036 bytes (0xFE0C in hexadecimal notation).
2. The BIB Blocks size being variable (specified by the USHORTs before and after each block), the number of total BIB Blocks in an Acro Block is variable as well.

Free BIB Blocks

Similarly to what we have observed for Acro Cache Blocks earlier, unallocated BIB Blocks are cast to a structure called *free_bib_block*. It has the following definition:

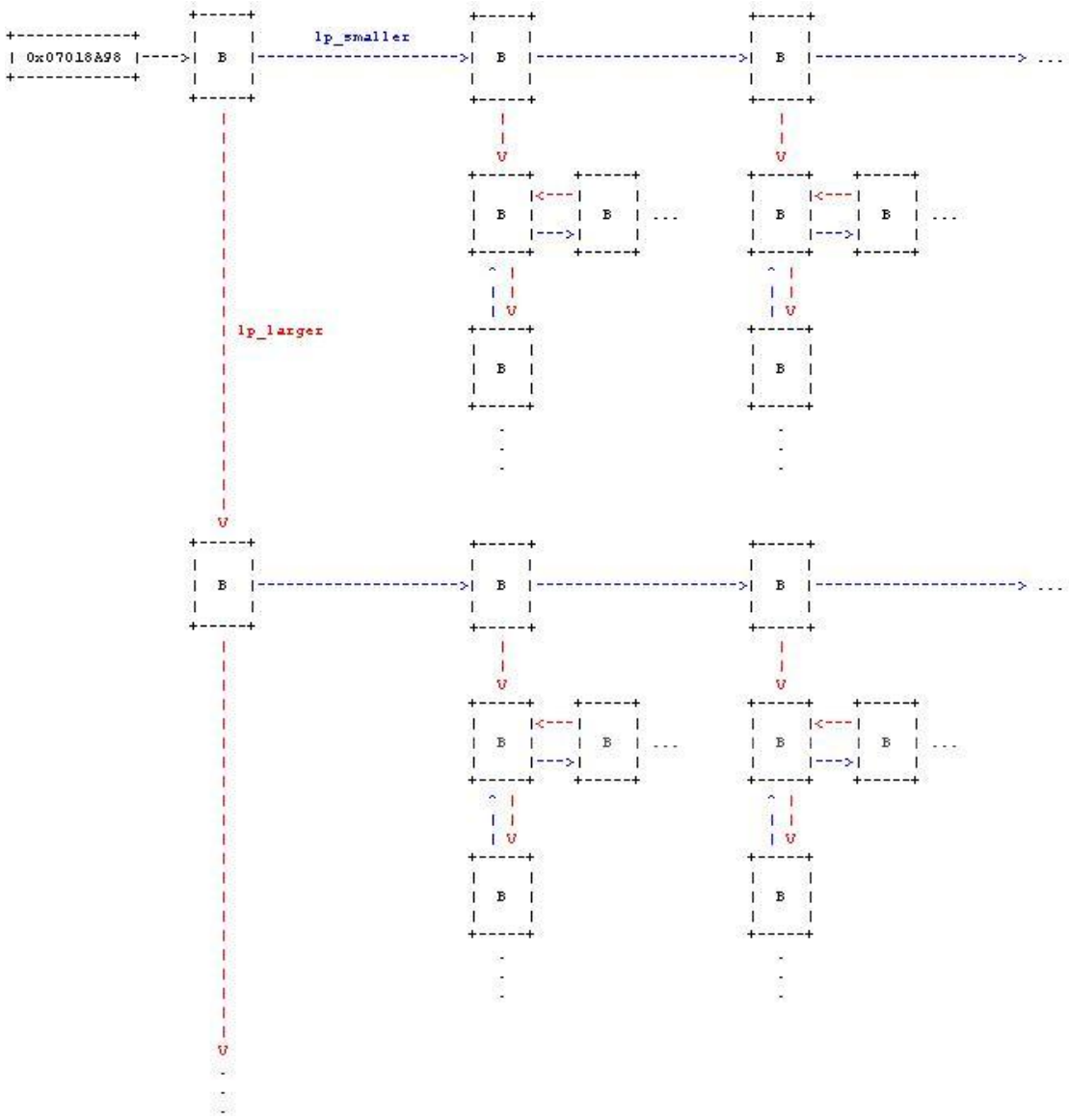
```
struct free_bib_block
{
    free_bib_block*  lp_smaller;        // Pointer to a smaller free bib block in the cache
    free_bib_block*  lp_larger;         // Pointer to a larger free bib block in the cache
    DWORD            reserved;
    DWORD            reserved;
    free_bib_block*  lp_pre_same_size;  // Blink pointer to a free bib block of same size
    free_bib_block*  lp_next_same_size; // Flink pointer to a free bib block of same size
}
```

Therefore, each free BIB Block points to a larger block, a smaller block, and two blocks of the same size (unless such don't exist, of course).

Organization

As we've seen in section II, the Acro Cache is organized as a hash table: there is an array of elements (the Cache Managing structures), each pointing to the first available block of a given possible size, and a (basic and fast) hash function converts the requested size into an index in this array, thereby fulfilling the typical request: *"give me the smallest node whose size is bigger than the requested size".*

It is likely the fastest possible method to search and return an element matching this request in a data set. But for some reason -- that probably has to deal with the fact that there are many more possible different sizes for a BIB Block than for an Acro Block -- the BIB Cache resorts to a different strategy (and thus, a different data structure): it is organized as a binary search tree (with the block size being the node key), with some of its leave nodes pointing to a doubly-linked ordered (by size) list. It looks like the following:



Where all the arrows pointing downwards and leftwards represent lp_larger pointers, and all arrows pointing rightwards and upwards represent lp_smaller pointers, and where the doubly-linked list of "same-size" nodes was not represented.

This can effectively be viewed as a Binary Search Tree, where each smaller child (in bold in the figure above) of any given node points to a doubly-linked ordered list, in the place of its bigger child.

As a result, the method to access a node matching the classical request (*"give me the smallest node whose size is bigger than the requested size"*) will itself be hybrid in nature: Partly a binary tree search to a certain point (i.e. the entry in the doubly-linked list), then if applicable, ordered sequential. Discussing the possible reasons for such a design choice is beyond the scope of this paper.

Allocation

BIB Block allocation is ensured by the function *bib_allocate* in *bib.dll.* Its high-level logic is the following:

1. If the requested size of storage memory space is greater than 65024 (0xFE00) bytes, a classical Acro Block is allocated and returned.
2. Pulls *the smallest node whose size is bigger than the requested size* from the hybrid cache. If there are more than one, the first same-size block (for the "same-size" linked list) is taken.
3. If that node is bigger than the requested size by an amount of 28 bytes (0x1C), the node is divided in two, the first part (of requested size) being returned to the requester, and the second part being placed in the cache at the appropriate place (which is unique, due to the organization of the cache). Otherwise, the whole node is returned to the requester for memory storage.

In the event that in step 2, the cache doesn't contain a big enough node, a new Acro Block container (of size 65036, as always) is created, with two BIB Blocks inside: one of the requested size, immediately returned to the requester, and one of the remaining size, immediately inserted in the cache at the appropriate place.

This is illustrated by the following pseudo-code of the function:

```
void* bib_allocate(DWORD dwSize)
{
	if (dwSize > 0xFE00)
	{
		//No "caching" on large bib allocation
		//return acro_allocate(dwSize + 8) + 8;
	}
	else
	{
		//DWORD align
		dwSize = ((dwSize + 3) / 4) * 4;


		//Pulls the smallest node whose size is bigger than dwSize
		unsigned char *lpRetBlock = bib_find_minimal_appropriate_block(dwSize);


		if (lpRetBlock == NULL)
		{
			//apply a new 0xFE0C-length acro block
			unsigned char*		lp_new = acro_allocation(0xFE0C);

			//initialize it


			//bib block starts from offset 8 (reserve + flag_s + b_size )
			lpRetBlock = (unsigned char *)lp_new + 0x08;

		}
		else
		{
			//check if a same-size block on "same-size-list"
```

```
                free_bib_block*        lp_current = (bib_block *)lpRetBlock;
                free_bib_block*        lp_next = lp_current->lp_next_same_size;

                if (lp_next != NULL)
                {
                        //return the next one
                        lpRetBlock = (unsigned char *)lp_next;

                        //fix same-size-list
                        bib_block*       lp_next_next = lp_next->lp_next_same_size;
                        lp_current->lp_next_same_size = lp_next_next;

                        if (lp_next_next != NULL)
                        {
                                lp_next_next->lp_pre_same_size = lp_current;
                        }

                }
                else
                {
                        //if no same-size block, use itself and update all lists.
                }

        }

        DWORD        dwRetBlockSize = (DWORD)*(USHORT *)(lpRetBlock-2);

        if ( dwRetBlockSize > (dwSize+0x1C))
        {
                /*****************************************************************
                if returning block is big enough, divide the block into two blocks
                return the first block to the caller

                |lpRetBlock| ==> |block1|len1_flag2|len2|block2|
                *****************************************************************/

                 USHORT        len2 = dwRetBlockSize - dwSize - 4;

                 *(USHORT *)(lpRetBlock + dwRetBlockSize) = len2;
                 *(USHROT *)(lpRetBlock + dwSize + 2) = len2;

                 *(USHORT *)(lpRetBlock + dwSize) = dwSize;
                 *(USHORT *)(lpRetBlock - 2) = dwSize;

                 unsigned char *lp_new_produced_block = lpRetBlock + dwSize + 4;

                 // Then merge possible neighbor blocks, update block flags,
                 // update the management linked-lists,
                 // And insert the new block in the data structure

        }
        return lpRetBlock;
}
```

V. Exploiting the BIB Cache

Similarly to what we have seen in Acro Cache exploitation, knowing the underlying data structure and the algorithms employed by the BIB Cache certainly gives a definite advantage in exploitation scenarios that involve BIB Blocks (heap overflow, heap underflow, use-after-free, etc...) -- and the limitations thereof.

Predictability of the Memory State

Generally speaking, leveraging a heap-corruption flaw into an exploit makes sense only if the exploitation method is "reliable", that is to say if it can be reproduced successfully over time and space (i.e. on different systems). Such a reliability is of course tightly dependent on the predictability of the memory state.

The good news is that thanks to the analysis exposed in the previous section, we can predict exactly which BIB Block is going to be allocated upon a storage memory request of a given size, given a configuration of the BIB Cache (we just have to emulate the allocation algorithm on the hybrid binary tree).
Now the bad news is that the BIB Cache configuration doesn't happen to be deterministic: opening twice the same document, in apparently the same conditions, will lead to significantly different BIB Cache lay outs, hampering predictability of the "vulnerable" and "targeted" blocks locations in say, a v-pointer overwrite attempt (see section III).

This issue is illustrated by the following, showing two dumps of free BIB Blocks locations (obtained by walking down the cache tree) upon two consecutive opening of our minimal PDF file:

BIB Blocks Size	Locations (Dump 1)	Locations (Dump 2)
0018	02D82318, 02D119A0	02D13410, 02D4BA78
001C	02D5E3E4	02D4A108, 02D49EE0
003C	02D82640	
004C		02D4CBC0
0050	02D828F8	02D3F798
00E8	02D5CE88	
010C	02D5F0F8	
0128	02D5F2D0	
0130		02D3F8A4
0134	02D10B50	
0144		02D4DA48, 02D4A420
0154	02008C18	
0164	02D5D628	
0174	02D5E178	02D036D4

01BC		02D11304
01E4	02D5E984	02D4B7F4
024C	02D033C4	
0274		02D4C458
029C	02D10554	
03CC		02D4D4CC
0990		02D83304
223C	02D5FCB8	
2CA0		02D84400
8744		02D8A718
A2D0	02D52B48	02D3FB48
A768	02D8760C	

If we observe the free blocks configuration in the first execution instance (Dump 1), we can very easily see that knowing the allocation strategy, if we want to "take the hand" over memory at 0x02D5CE88 (position of the 0xE8 long block in the table above), all we have to do is to get the application (by crafting a stream in the PDF file) to request a memory block whose size is in the range 0x51-0xE8. Indeed the system will then return the free 0xE8 block, and unlink it from the free blocks hybrid tree.

However, quick comparison with the lay out in the second execution instance (Dump 2) informs us that basing our exploit on that prediction will be highly unreliable, to say the least. Indeed, the free blocks lay out seems completely different, and the 0xE8-long block doesn't even exist anymore. This is due to the fact that BIB Blocks are frequently allocated/unallocated and to the very nature of the BIB Cache management strategy, where blocks are divided and merged upon allocations and unallocations. In such a context, slight initial differences soon give rise to very different lay outs.

Adjusting the Memory State

There are however ways to somehow simplify the free blocks lay out. The method we introduce here makes use of Font Resource objects[8], to fill the cache with small BIB blocks. This requires to add a significant number of such objects in the PDF file. For the sake of the demonstration, let's extend our minimal PDF with 800 Font Resources objects:

4 0 obj

<<

/Type /Page

/Parent 3 0 R

/MediaBox [0 0 612 792]

/Resources

<<

/ProcSet [/PDF /Text]

/Font << /F10 10 0 R /F11 11 0 R ... /F799 799 0 R >>

>>

>>

endobj

10 0 obj

<<

/Type /Font

/Subtype /Type1

/Name /F10

/BaseFont /HaifeiLiAAAAAAAA

/Encoding /MacRomanEncoding

>>

Endobj

...

799 0 obj

<<

/Type /Font

/Subtype /Type1

/Name /F799

/BaseFont /HaifeiLiAAAAAAAA

/Encoding /MacRomanEncoding

>>

Endobj

Dumping the free BIB Blocks locations now only returns:

0BADF00D **********Free Bib Block Dumping...**********
0BADF00D length A740 at:
0BADF00D 02DB39AC

As shown, only one 0xA740-length block is remaining now. Essentially, what happened is that all the small blocks were allocated for the Font Resources objects, as well as most big blocks, after being divided.

Corrupting the Structures

The BIB Cache "free blocks" data structure being a hybrid tree/list (thus heavily relying on pointers) rather than a hash table as in the Acro Cache case, unlinking attacks opportunities are numerous: each time a corrupted element of the data structure is unlinked (for allocation, but not only - as we will see afterwards), we get a free write (of an arbitrary dword at an arbitrary address) and its side effect, allowing us to gain control of the execution flow (see section III). Some possible scenarios are evoked next.

Overwriting lpAcroHeader

As described before, a BIB Block is necessarily contained in an Acro Block. Thus, in some special cases such as heap underflow vulnerabilities, it is possible to overwrite the "lpAcroHeader" pointer. This directly branches us to the exploitation scenario described in the "Corrupting the structure" part of section III.

In the perhaps more frequent cases of heap overflow or use-after-free vulnerabilities, only data inside the BIB Blocks themselves may easily be overwritten. A free BIB Block however contains numerous pointers (*lp_smaller, lp_larger, lp_pre_same_size, lp_next_same_size*) that can be abused to gain control of the execution flow.

Overwriting lp_next_same_size - The Universal Method

Let's consider the *lp_next_same_size* pointer. As a reminder, it is used during allocation, once the free block with the requested size was found, to see if it has same-size little brothers. In which case, the first block of the list is indeed considered. If its size allows, the latter is then divided in two blocks; the first one is returned to the requester, and the second one is inserted into the free blocks data structure, like any good newly created block should be. Now, interestingly, the insertion procedure contains the following instructions:

```
DWORD block_size = (DWORD)*(USHORT *)(lpBibBlock - 2);

//if the bib block size is 0xFE01, handle it as an acro block
if ((block_size == 0xFE01) && (lpBibBlock != NULL))
{
  //locate the acro block pointer
  unsigned char *lpAcroBlock = lpBibBlock - 8;
  //obtain the value of "reserve"
  v_reserve = *(DWORD *)(lpBibBlock - 8);
  if (v_reserve >= 0x00020000)
  {
    //free the acro block
    acro_free(lpAcroBlock);
  }
}
```

In short, if the free BIB Block to insert in the structure has a size of 0xFE01 bytes, meaning it occupies a full Acro Block (we remind that Acro Blocks containing BIB Blocks have a fixed size: 0xFE0C bytes), then the system frees it with *acro_free.* Of course, a legitimate block resulting from a division cannot have such a size; but the insertion procedure is generic, it is applied not only to new blocks resulting from a split, but also from new blocks resulting from a merge... in which case it does make sense.

Therefore, if we overwrite the *lp_next_same_size* pointer of a free BIB Block to point to a fake block of us, we can force the call to *acro_free*, by carefully setting its size (the resulting size of the newly created block after the division must be 0xFE01). Of course, the pointer sitting 12 bytes above our fake block (corresponding to *lpAcroHeader*) is also controlled by us and points to a forged Acro Header... Which places us in the same situation as in section III for the Acro Block unlinking attack.

Obviously, because the size we set for our fake block cannot exceed 0xFFFF, the request yielding allocation of our corrupted block (and thus triggering the exploit) must be for a size inferior to or equal to 0x1FA (0xFFFF-0x4-0xFE01). Since there are many allocation requests for small size blocks in Adobe Reader, it is relatively easy to find one with a suitable size (i.e. <= 0x1FA) closely following the corruption operation.

Overwriting lp_smaller or lp_larger

The above could actually be viewed as an universal approach for the BIB Cache exploitation. Indeed, overwriting *lp_smaller* and *lp_larger,* if done astutely, can result in the same exploitation scenario.

For this, we just have to forge a block sitting at the address pointed to by the corrupted *lp_smaller* (or *lp_larger*) pointer, via the heap spraying technology (described in previous part). Since we do control the *lp_next_same_size* of our forged block, the previous scenario can effectively be reproduced.

As a matter of fact, the astute reader might have noticed that in a a heap overflow scenario, one cannot overwrite the *lp_next_same_size* pointer without blasting the *lp_larger* and *lp_smaller* ones. Since those are frequently used by the system to walk through the data structure, it is advisable to not fill them with random values, that could generate faults in the BIB Cache management system and hamper exploitation. The forged block technique described here is therefore essential to successful exploitation.

Conclusion

We have seen throughout this paper that Adobe Reader's Custom heap management employ various strategies and data structures to achieve memory management in a more efficient and fast way than the OS does. Whether or not it succeeds (and how much it succeeds) in this goal is of course out of the scope of this paper, but one thing is sure: it lacks all the security mechanisms that modern OS memory management systems have. This, in effect, empowers attackers with the capacity to exploit heap corruption vulnerabilities, which were once impossible to leverage. In a context where malicious PDF files have become one of the prime infection vectors for Cybercriminals when conducting large scale campaigns or focused attacks (eg: GhostNet), this must be addressed. The good news, however, is that the protection mechanisms do exist (safe unlinking, heap metadata cookies, etc...[9]), and as noted above, have been successfully implemented in other OS. The vendor thus has plenty of options to harden its custom heap.

References

[1] Interpreter Exploitation: Pointer Inference and JIT Spraying, Dion Blazakis
[2] Annual Global Threat Report 2009, Cisco ScanSafe
[3] 2008 Report, Secunia
[4] Smashing Adobe's Heap Memory Management Systems for Profit, Hafei Li
[5] Bypassing Windows Hardware-enforced Data Execution Prevention, skape and Skywing
[6] Return-Oriented Programming, Hovav Shacham
[7] JPEG COM Marker Processing Vulnerability in Netscape Browsers, Solar Designer
[8] PDF Reference Version 1.7 (6th Edition), Adobe Systems Incorporated
[9] Bypassing Browser Memory Protections, Alexander Sotirov
[10] Heap Feng Shui in JavaScript, Alexander Sotirov