Pwning Adobe Reader

Abusing the Reader's embedded XFA engine for reliable Exploitation

Sebastian Apelt
sebastian.apelt@siberas.de

2016/03/24

Agenda

- whoami
- Motivation
- (Short!) Introduction to XFA
- XFA Internals
  - XFA Objects
  - jfCacheManager
- Exploiting the Reader
- Conclusion
- Q&A

whoami

- Sebastian Apelt (@bitshifter123)
- Co-Founder of siberas in 2009
  - IT-Security Consulting (Pentests, Code Audits, etc.)
  - Research
- Low-level addict
  - Reverse Engineering, Bughunting, Exploitation
    - > 100 CVEs in all kinds of Products
    - Pwn2Own 2014 (IE11 on Win8.1 x64)

Motivation

Motivation

- Fuzzing at siberas
  - Let‘s pwn the Reader @ Pwn2Own 2016!!
    - Unfortunately, no love for Reader this time ☹
  - In 2015: XFA fuzzing on 128 cores
  - Fuzz run yielded thousands of crashes
  - So far ~ 20 Bugs identified as unique (upcoming)
  - Analysis took ages…
  - Let‘s take a look at a typical Reader crash!

Motivation

```
(72fc.72ec): Access violation - code c0000005 (!!! second chance !!!)
eax=69572c30 ebx=00000002 ecx=07b2f3cc edx=05658af8 esi=0549e538 edi=07b2f3cc
eip=20a29654 esp=0031d8c4 ebp=00000003 iopl=0         nv up ei pl nz na
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b         efl=00210206

AcroForm!DllUnregisterServer+0x2f73ce:
20a29654   mov edx,dword ptr [eax]  ds:002b:69572c30=????????
```

Awesome, we have a crash!

But no useful function name (DllUnregisterServer??)

```
0:000> !heap -p -a ecx
    address 07b2f3cc found in
    _HEAP @ 11a0000
      HEAP_ENTRY Size Prev Flags    UserPtr  UserSize  -  state
        07b24eb0 199c 0000  [00]   07b24eb8   0ccd8  - (busy)
```

Offset 0xa514 !?

The object holding the bad reference is located in the middle of a huge buffer => Page Heap useless

```
0:000> kc
AcroForm!DllUnregisterServer+0x2f73ce
AcroForm!DllUnregisterServer+0x2f7212
AcroForm!DllUnregisterServer+0x2f7504
AcroForm!DllUnregisterServer+0x35f3ae
AcroForm!DllUnregisterServer+0x358f50
```

Stacktrace also not helpful

Motivation

- Adobe Reader => No symbols / RTTI infos!
  - No function names
  - No object / vtable information
  - No meaningful stacktraces
  - Page Heap useless
- Root cause analysis is very hard without context
- Complicates crash triaging during fuzz runs

Motivation

- How do we ANALYZE crashes in XFA?
- How do we EXPLOIT these crashes?

- Obvious: We need context! We need symbols!

- No *in-depth* research about XFA internals so far:
  - Most useful: Writeups about XFA exploit from 2013 (David and Enrique of Immunity Inc, Matthieu Bonetti of Portcullis Labs)
  - Good technical analysis, but only scratching the surface

Motivation

- Write tools to recover contextual information
  - Lower the bar for other researchers!
  - Check https://github.com/siberas in the next days
- Facilitate:
  - Vulnerability discovery and root cause analysis
  - Crash triaging during fuzz runs
- Deliver XFA-specific background for exploitation

(Short!) Introduction to XFA

(Short!) Introduction to XFA

- XFA: „XML Forms Architecture“
  - Specification developed by JetForm, later Accelio (acquired by Adobe in 2002) – not a standard
  - Latest version: 3.3 (01/2012): Easy read of 1584 pages.
  - Brings *dynamic* behavior to the *static* PDF world: Forms that can dynamically change their layout!
  - Dynamic nature of XFA is powered by Javascript (Spidermonkey 24 since AR DC)
  - XFA not supported by many PDF Readers, yet (Chrome/Chromium, Firefox, Windows,...)

(Short!) Introduction to XFA

- XFA form data itself is an XML-structure embedded in the PDF, a so-called *XDP*-Packet
- Javascript embedded in this XDP
  - Executed upon events (e.g. document is fully loaded, user clicks on button, etc.)
- A practical example...

(Short!) Introduction to XFA

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
  <config xmlns:xfa="http://www.xfa.org/schema/xci/3.0/">
    [...]
  </config>
  <template xmlns:xfa="http://www.xfa.org/schema/xfa-template/3.0/">
    <subform layout="tb" name="form1">
      <pageSet>
        <pageArea id="PageArea1" name="PageArea1">
          <contentArea w="612pt" h="792pt" x="20pt" y="20pt"/>
        </pageArea>
      </pageSet>
     <field name="button1" w="41.275mm" h="9.525mm">
       <ui>
         <button highlight="inverted"/>
       </ui>
       [...]
       <event activity="click" name="event__click">
         <script contentType="application/x-javascript">
             app.alert(1337);
          </script>
       </event>
    [...]
</xdp:xdp>
```

XDP Packet is XML embedded in the PDF
The root tag is always „xdp“

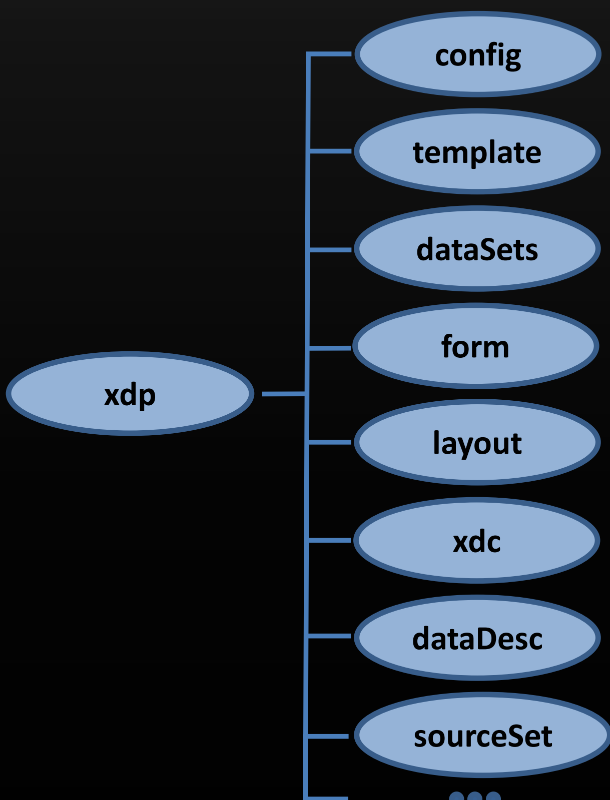
Config DOM contains configuration options for XFA processing

Template DOM is structured in subforms, containing objects like „field“, „text“, etc.

Objects can contain event objects that fire on certain actions (e.g. „click“)

(Short!) Introduction to XFA

- XFA spec defines multiple DOMs
  - HUGE attack surface (> 200 objects accessible via JS)



Configuration Options

Tpl DOM: Objects which will be visible in the PDF

XML-Data that can be used to populate fields in the PDF

Template and Data are merged into Form DOM

Layout DOM makes layout information accessible

Device-specific information

dataDescription DOM: Data schema

DOM for DB- / WebService-Connections

XFA Internals

XFA Internals - General Approach

- Tweet by @nils

> **nils**
> @nils
>
> @_frego_ You can get the symbols from the Solaris build of Reader ;) The allocator is called jfCacheManager (functions:getMemoryCache, etc.)

  - Nice! Some Solaris build seems to have symbols!
  - Newest version which still has symbols: Solaris v9.4.1
- We need a *reliable* heuristic to port symbols in AcroForm.api (module which implements XFA functionality) to newer AR versions

XFA Internals - General Approach

- Problems:
  - Code is rather old (2012) -> Many Code changes from v9.X to AR DC…
    - Function count: Solaris ~48 K, AR DC ~ 95 K
  - Functions differ even if code stays the same (compiler optimizations like heavy inlining in v9.4.1 screw it up)
    - Tried diffing with Diaphora – Too many false positives
  - Structures, objects and vtable sizes differ (slightly, but enough to make it very hard to create reliable heuristics)
  - etc.

XFA Internals - General Approach

- Approach: Trying to *understand* Reader v9.4.1 as much as possible with the help of symbols
- Find bulletproof ways to recover the *most important* symbols, i.e.
  - Heap Mgmt functions for the custom allocator
  - Object information

XFA Internals - Objects

- What do we need to know about objects?
  - How to identify an object in memory
  - Vtable offsets
  - Methods and properties exposed to JavaScript
  - Offsets of the entrypoints for methods / property-getters and -setters
  - Function names of vtable entries

XFA Internals - Objects: Identification

- First attempt: XFANode::getClassTag

```
; _DWORD __cdecl XFANode::getClassTag(XFANode *this)
public _ZNK7XFANode11getClassTagEv
_ZNK7XFANode11getClassTagEv proc near

this= dword ptr  4

mov     eax, [esp+this]
mov     eax, [eax]
mov     eax, [eax+10h]
retn
_ZNK7XFANode11getClassTagEv endp
```

classTag attribute can be found @ <XFAobj> + 0x10

```
mov     eax, ds:(_ZTV12XFAFieldImpl_ptr - 0D91324h)[ebx]
add     eax, 8
mov     [esi], eax
mov     eax, ds:(aXFA_FIELD_ptr - 0D91324h)[ebx]
mov     eax, [eax]
mov     [esi+0Ch], eax
mov     dword ptr [esi+10h], 86h
add     esp, 10h
pop     ebx
pop     esi
leave
retn
```

From Field constructor method: classTag for Field-Object in **Adobe Reader 9.4.1: 0x86**

```
0:011> dd 0x6883d74
06883d74  6822bd94 00000001 00000000 683412fc
06883d84  0000008e 00000002 00011952 00000000
06883d94  00000000 00000000 068765ec 00000002
06883da4  04c39bcc 04d950a0 00000000 00000000
```

classTag for Field-Object in **Acrobat Reader DC: 0x8e**

- Fail! classTags not constant across versions!

XFA Internals - Objects: Identification

- <XFAObj>::Type method to the rescue
- Located @ vtable+8 of each XFA-Object

Adobe Reader 9.4.1

```
; _DWORD XFAFieldImpl::Type(XFAFieldImpl *__hidden this)
public _ZNK12XFAFieldImpl4TypeEv ; weak
_ZNK12XFAFieldImpl4TypeEv proc near
mov     eax, 7C46h
retn
_ZNK12XFAFieldImpl4TypeEv endp
```

Acrobat Reader DC

```
0:011> uf poi(poi(0x6883d74)+8)
AcroForm!DllUnregisterServer+0x34020a:
67fc2490 b8467c0000      mov     eax,7C46h
67fc2495 c3              ret
```

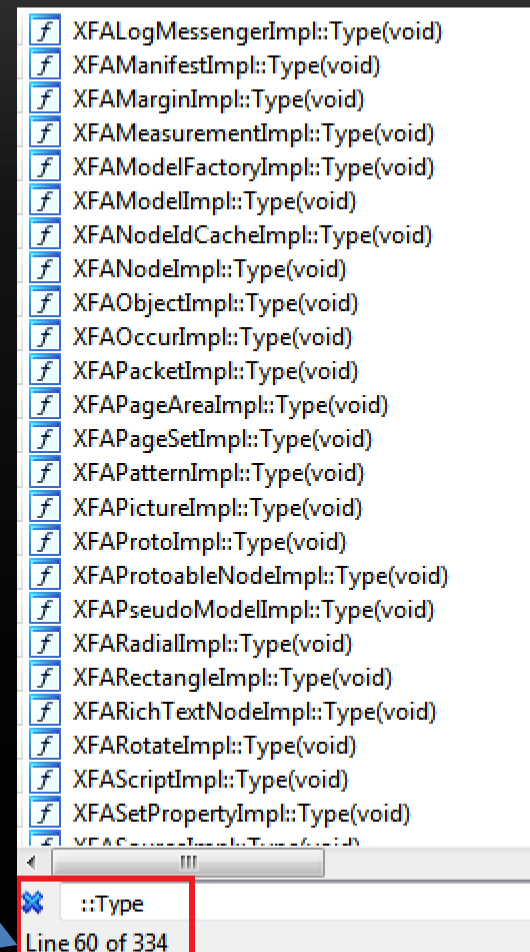
Type is 0x7C46 for both v9.4.1 AND Acrobat Reader DC! ☺

- Type-IDs are static across versions!

XFA Internals - Objects: Identification

- Possible to identify every object by a binary pattern in newer versions of AcroForm.api
  - mov eax, 7C46h
    retn
    ⇔ B8 46 7C 00 00 C3
- Xref to the Type method gives us the vtable offset (RVA) to each object!

We can safely identify 334 objects! Not too bad!



XFA Internals - Objects

- What do we need to know about objects?
  - How to identify an object in memory ✓
  - Vtable offsets ✓
  - Methods and properties exposed to JavaScript
  - Offsets of the entrypoints for methods / property-getters and -setters
  - Function names of vtable entries

XFA Internals - Objects

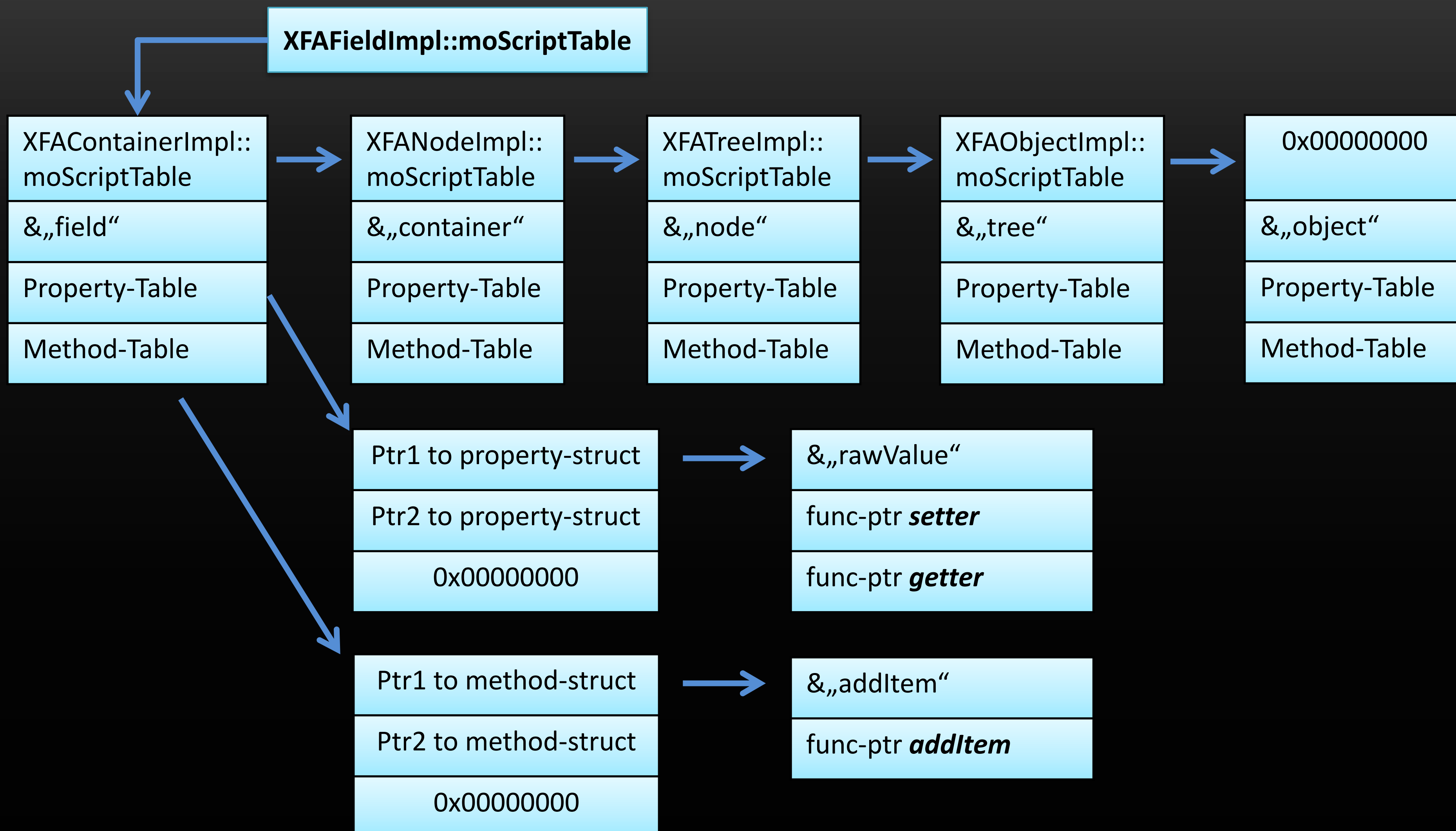
- How about methods and properties?
- <XFAObj>::getScriptTable() @ vtable offset 0x34

```
; _DWORD XFAFieldImpl::getScriptTable(XFAFieldImpl *__hidden this)
public _ZNK12XFAFieldImpl14getScriptTableEv
_ZNK12XFAFieldImpl14getScriptTableEv proc near
call    $+5
pop     ecx
add     ecx, 65F419h
mov     eax, ds:(_ZN12XFAFieldImpl13moScriptTableE_ptr - 0D91324h)[ecx]
retn
_ZNK12XFAFieldImpl14getScriptTableEv endp
```

XFAFieldImpl::moScriptTable

- References *moScriptTable* structure
  - Structure contains information about method and property names, function pointers, etc.

XFA Internals - Objects



XFA Internals - Objects

- What do we need to know about objects?
  - How to identify an object in memory ✓
  - Vtable offsets ✓
  - Methods and properties exposed to JavaScript ✓
  - Offsets of the entrypoints for methods / property-getters and -setters ✓
  - Function names of vtable entries

TODO...
Not trivial... ;-(

XFA Internals - jfCacheManager

- Most allocations in AcroForm.api are managed by a custom allocator called *jfCacheManager*
- LIFO-style heap manager
- Data buffers („blocks“) stored in big heap „chunks“
- Introduced most likely for performance reasons
- No security features…
  - No Heap Isolation (see IE, Flash, etc.)
  - No Anti-UAF like MemProtect/MemGC
  - …

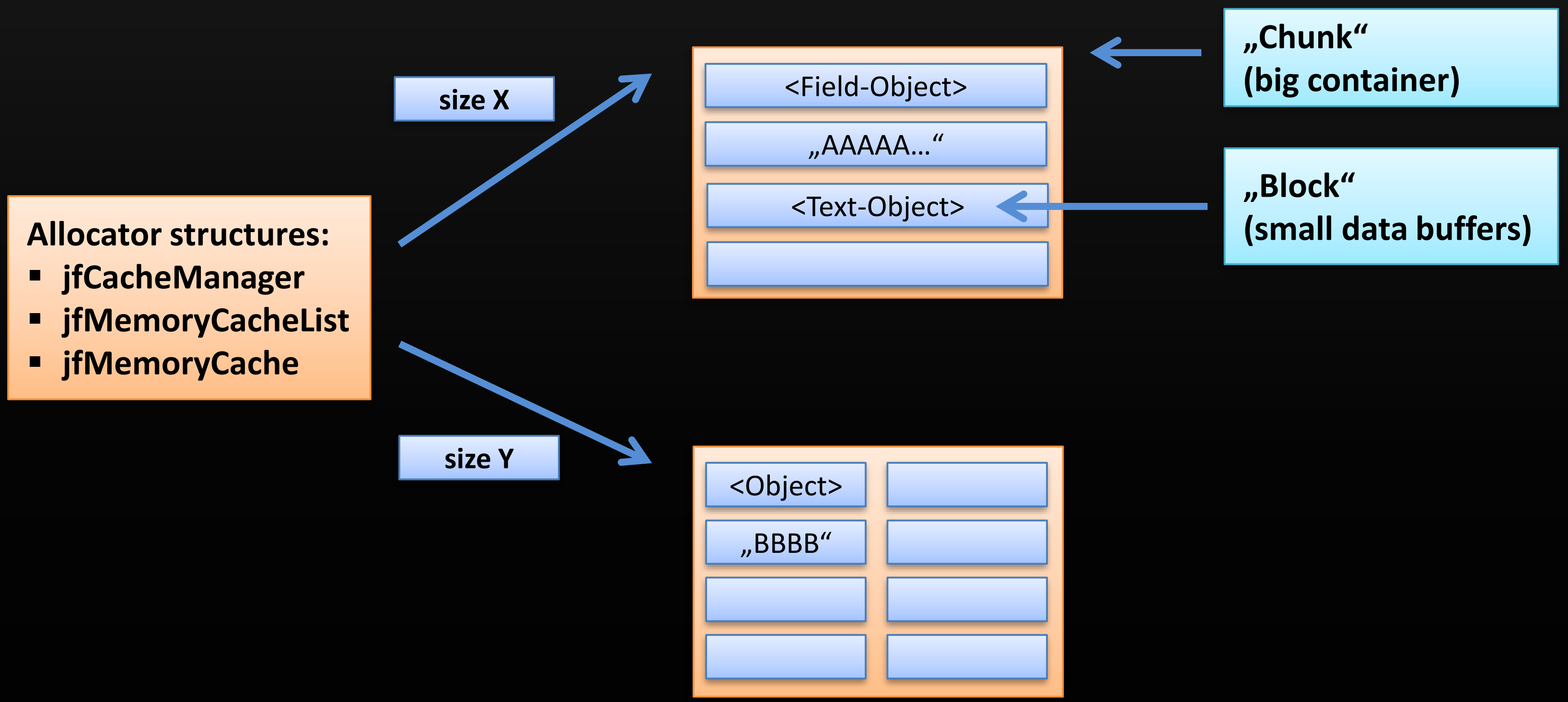
XFA Internals - jfCacheManager

Disclaimer: Next slides will only cover the *relevant* details of the memory manager in terms of *exploitation*!

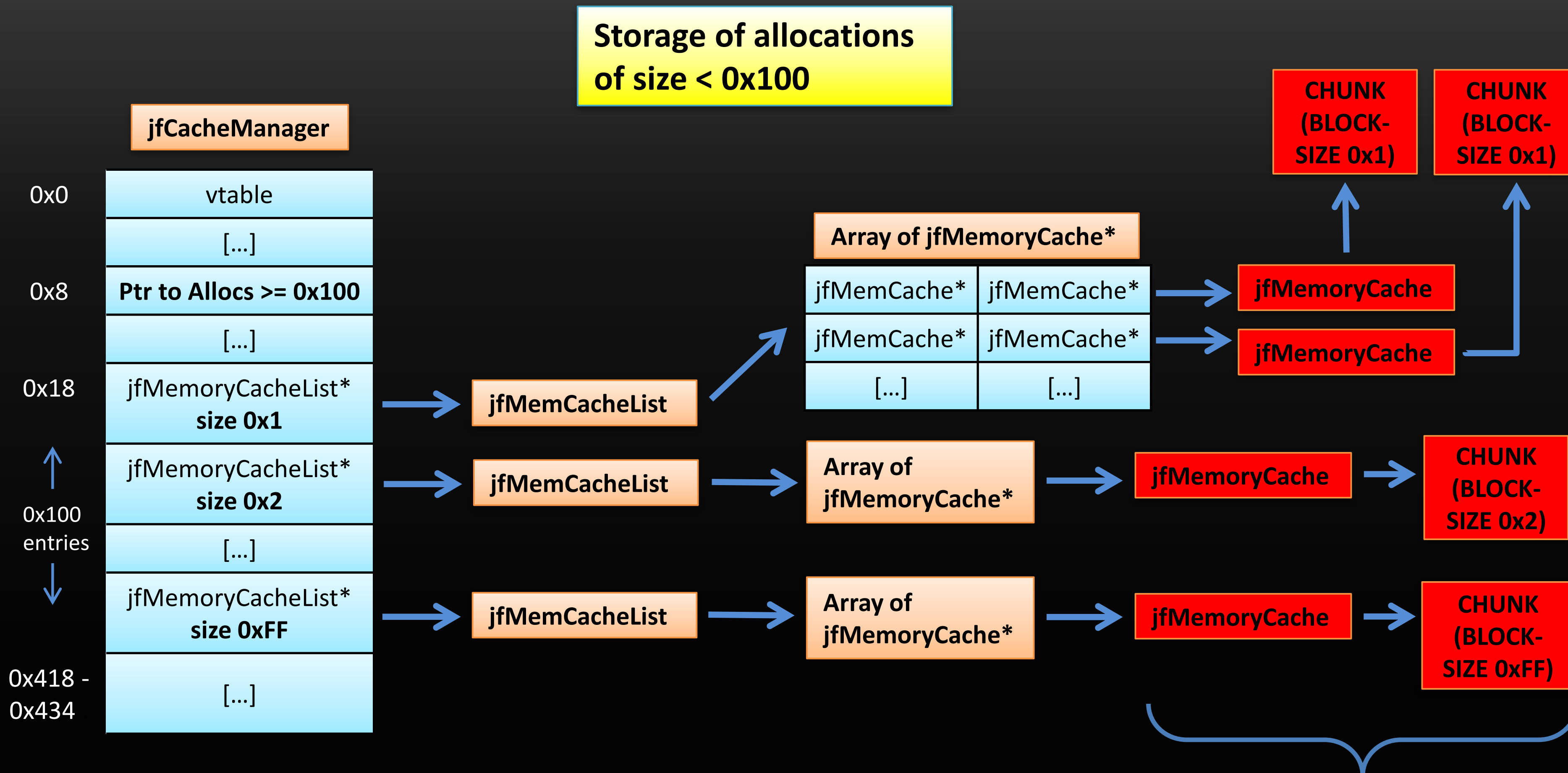
(More in-depth analysis will be covered by a paper which will be released soon)

XFA Internals - jfCacheManager

- Very simplified version of the jfCacheManager:



XFA Internals - jfCacheManager



jfMemoryCache and the *chunks* will be relevant for exploitation!

XFA Internals - jfCacheManager

- *sizeof(chunk)* derived from block size:

```
base_size = 0xc350 // 50.000
chunksize = ((((size + 3) / 4 ) + 1 ) * ((base_size + size - 1) / size)) * 4
```

Example: allocation size = 0x64

=> chunksize = 26 * (0xc3b3 / 0x64) * 4 = 0xcb20

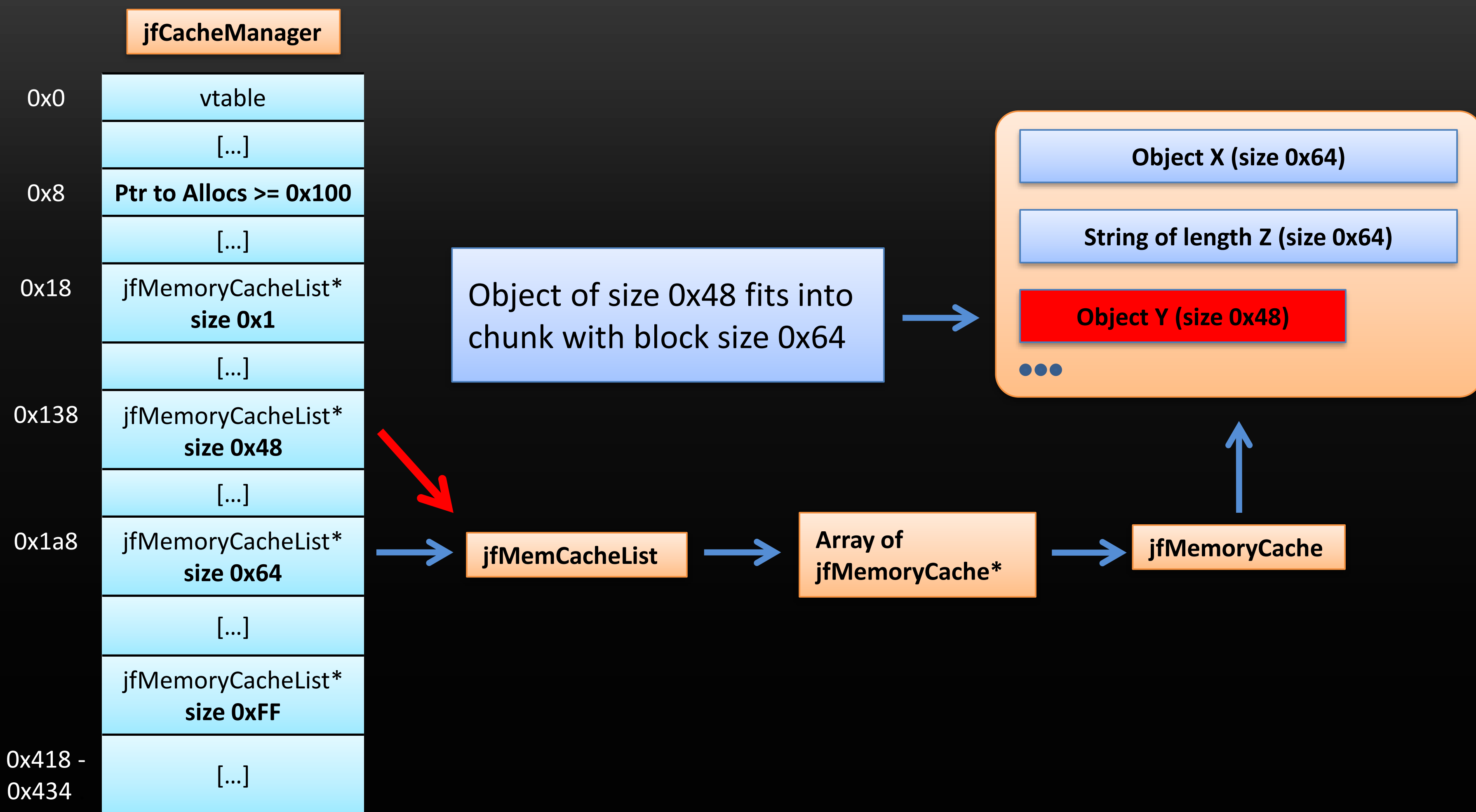
- „So, if I get a crash and I see my object located in a chunk of size 0xcb20, then sizeof(obj) == 0x64?"
  - Unfortunately not...

XFA Internals - jfCacheManager

- jfMemoryCacheLists can manage blocks of *multiple* sizes => blocks of sizes X and Y can both end up in chunk Z!
- alloc(X) will be placed in same chunk as alloc(Y) if
  - an allocation for a size Y > X has occured before and
  - size X is in the same „range“ as size Y
    - Ranges reach from 2^n to $(2^{n+1}-1)$ (e.g. 0x20 - 0x3f, 0x40 - 0x7f)
- In short:
  - Does the new block fit into some chunk that we already have?
  - If yes, use that chunk instead of allocating a new one!

© siberas 2016

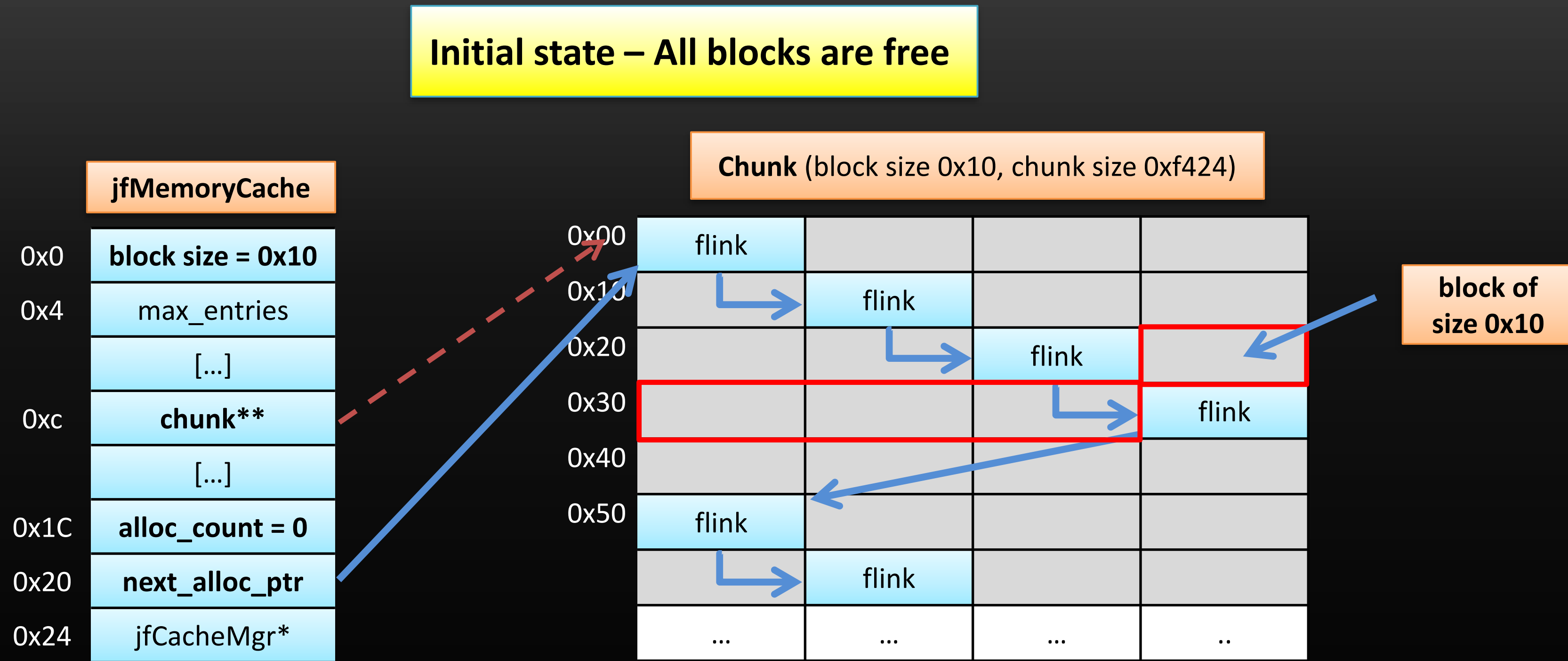
XFA Internals - jfCacheManager



XFA Internals - jfCacheManager

- Let‘s take a look at the structures within the chunks and what happens during alloc / free operations…

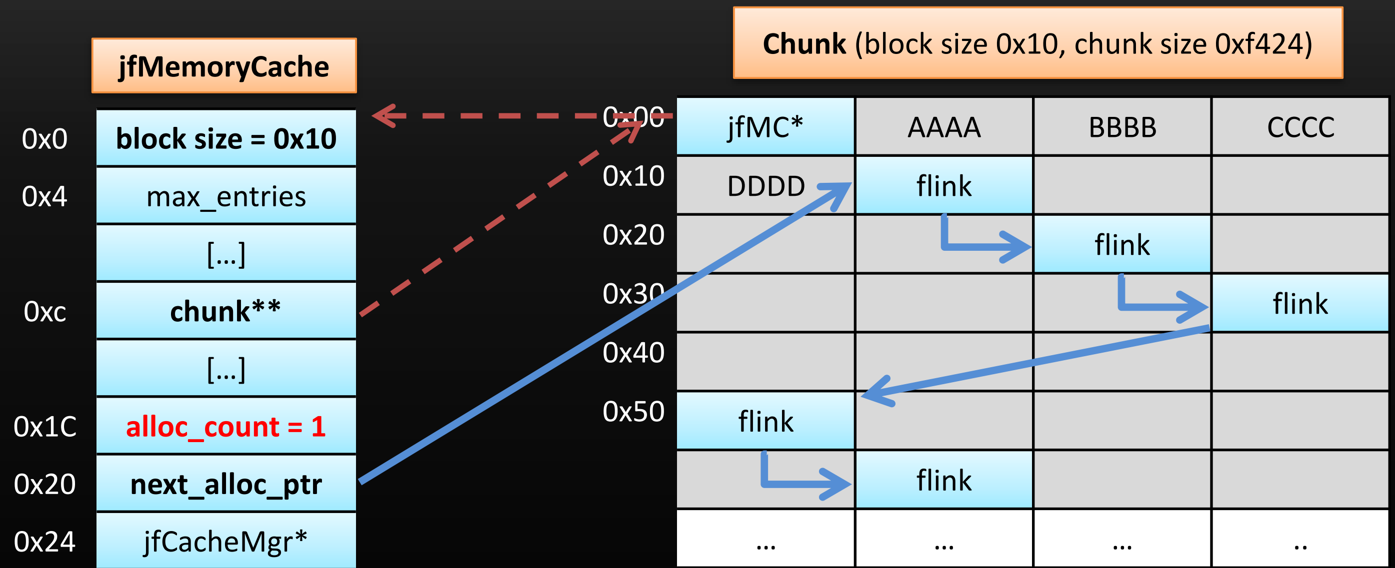
XFA Internals - jfCacheManager



- next_alloc_ptr points to the block which will be returned with the next allocation
- flinks form a single linked list separating the data blocks

XFA Internals - jfCacheManager

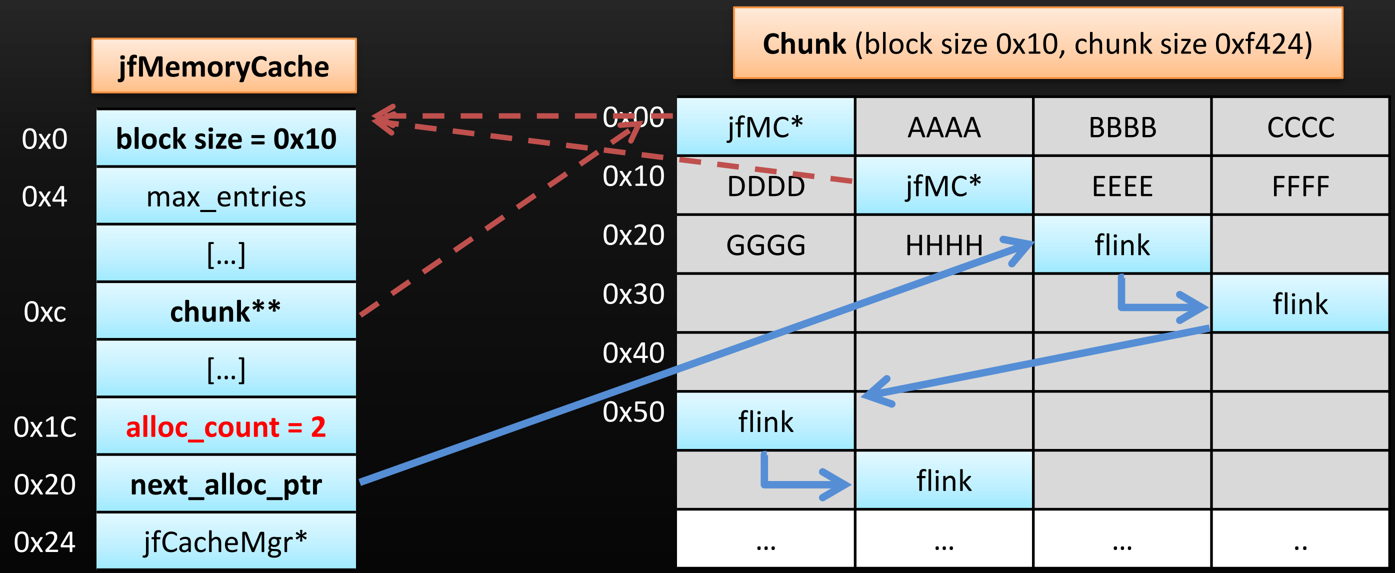
After first allocation



- *next_alloc_ptr* is overwritten with flink
- *flink* is overwritten with pointer back to jfMemoryCache
- *allocs_counter* is incremented to 1

XFA Internals - jfCacheManager

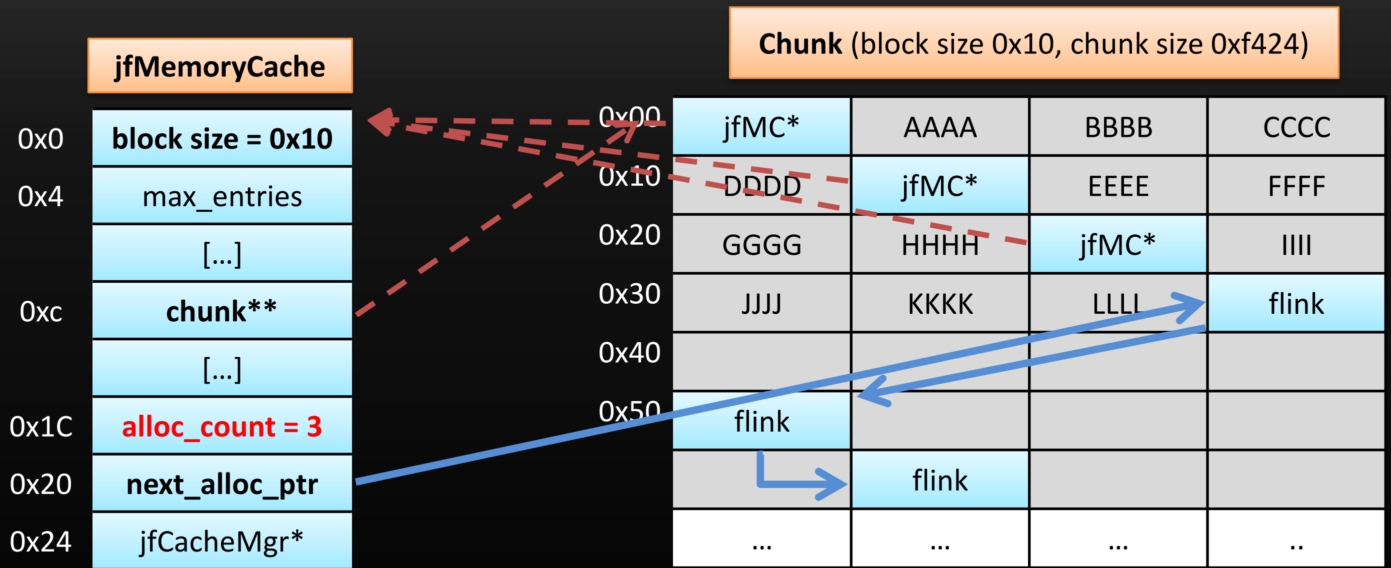
After second allocation



- *next_alloc_ptr* is overwritten with flink
- *flink* is overwritten with pointer back to jfMemoryCache
- *allocs_counter* is incremented to 2

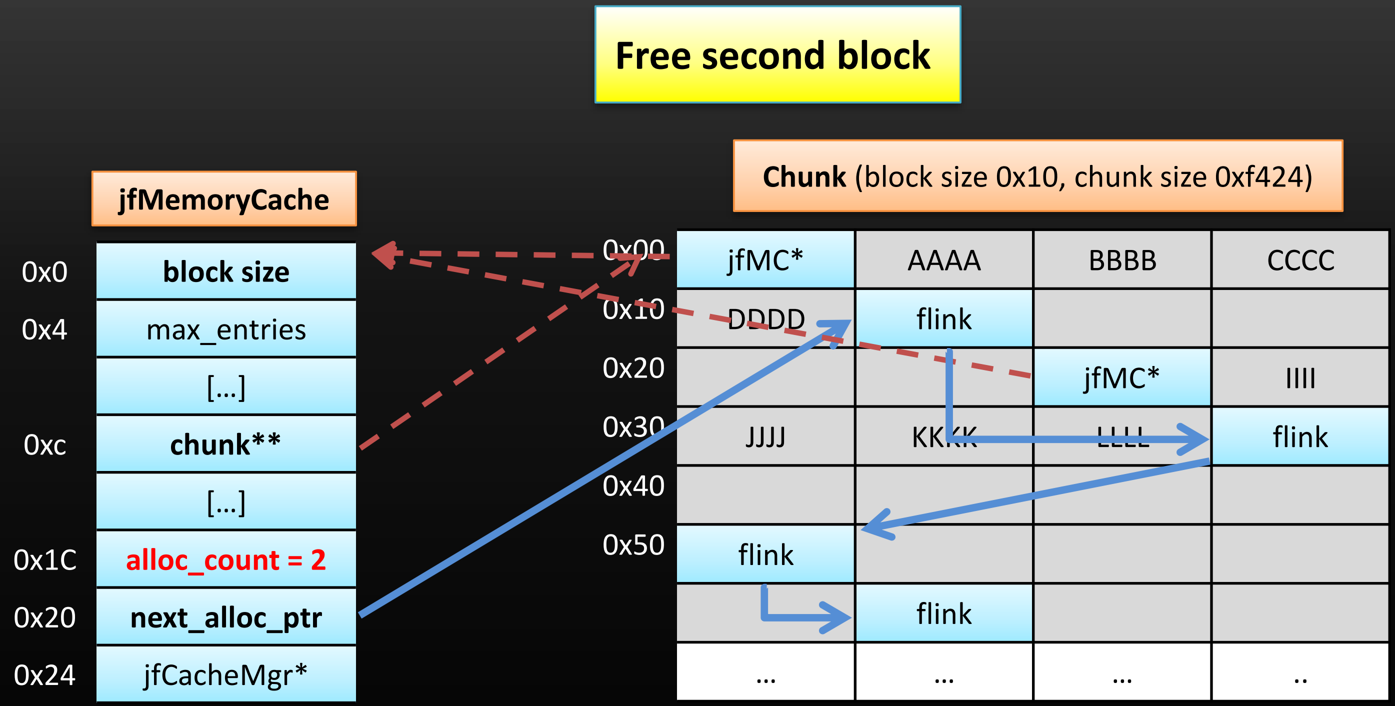
XFA Internals - jfCacheManager

After third allocation



- *next_alloc_ptr* is overwritten with flink
- *flink* is overwritten with pointer back to jfMemoryCache
- *allocs_counter* is incremented to 3

XFA Internals - jfCacheManager



- *next_alloc_ptr* is overwritten with pointer to free block - 4
- *jfMC** is overwritten with *next_alloc_ptr* (becomes flink again)
- *allocs_counter* is decremented to 2

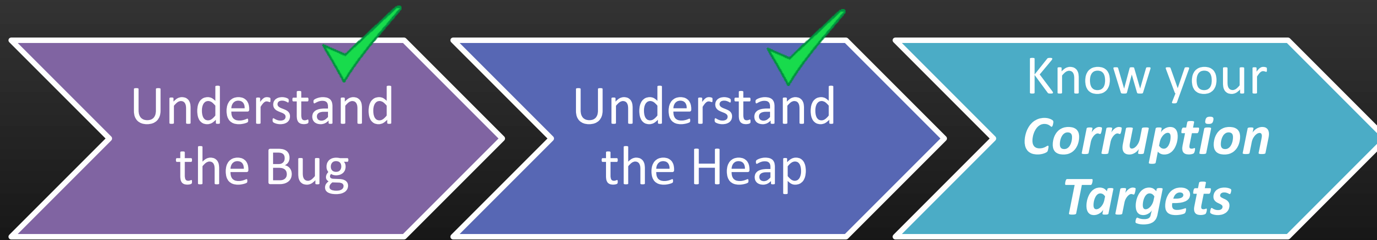
XFA Internals - jfCacheManager

- Still don‘t like the jfCacheManager?
- Still missing Page Heap?

- Get offset „jfCacheManager_active“ with XFAnalyze_funcs.py
- Change byte from 1 to 0 in binary
- Replace original AcroForm.api
- You just switched off the jfCacheManager :P

Exploiting the Reader

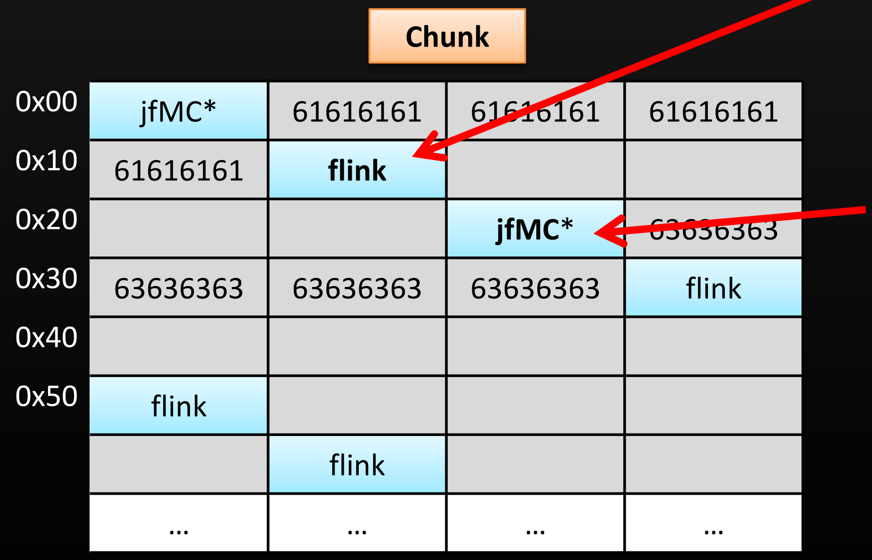
Exploiting the Reader



- Goals
  - Bypass ASLR by corrupting specific byte(s) to cause a memory leak
  - Find „flexible" overwrite target
    - No need for a write-what-where (e.g. 0-DWORD write or a partial overwrite to a controlled address should suffice!)
  - Find technique which is fast, reliable and most importantly independant from OS and AR version

Exploiting the Reader

- Let‘s target the metadata contained within the chunks!
- Two possibilities:

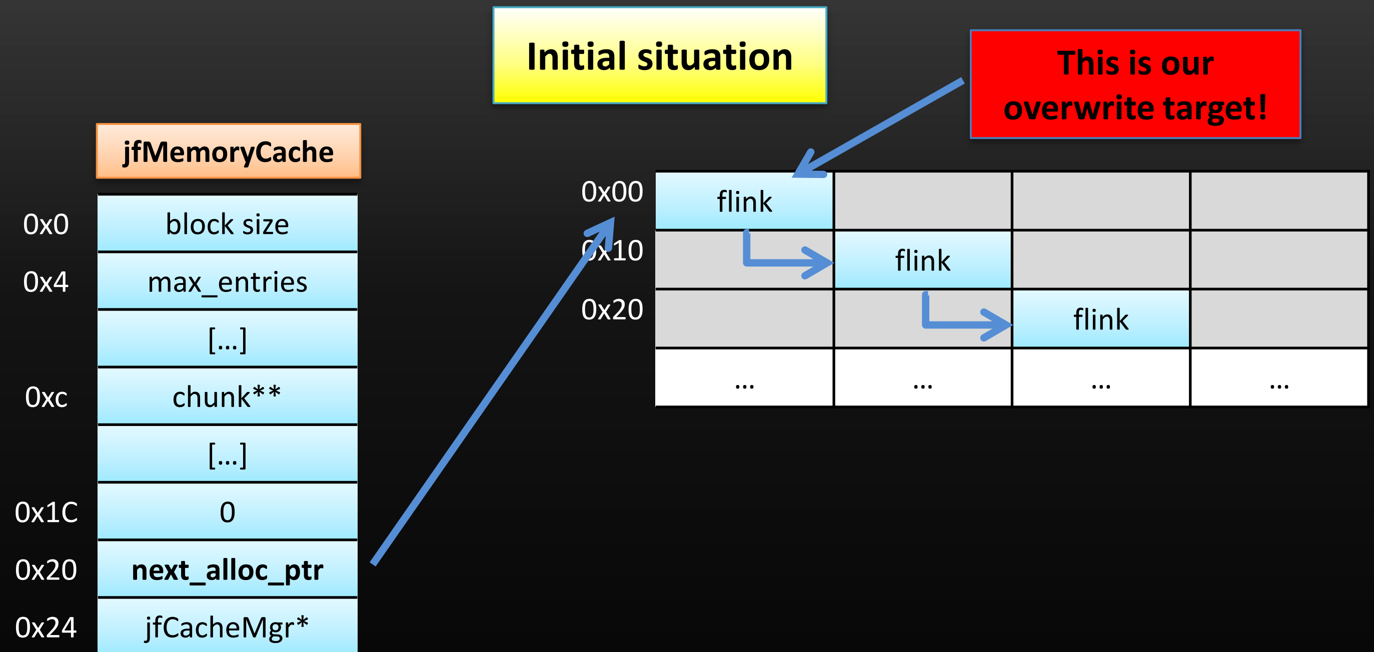


Hit a flink
⇒ Block is *free*
⇒ Triggers when block is allocated

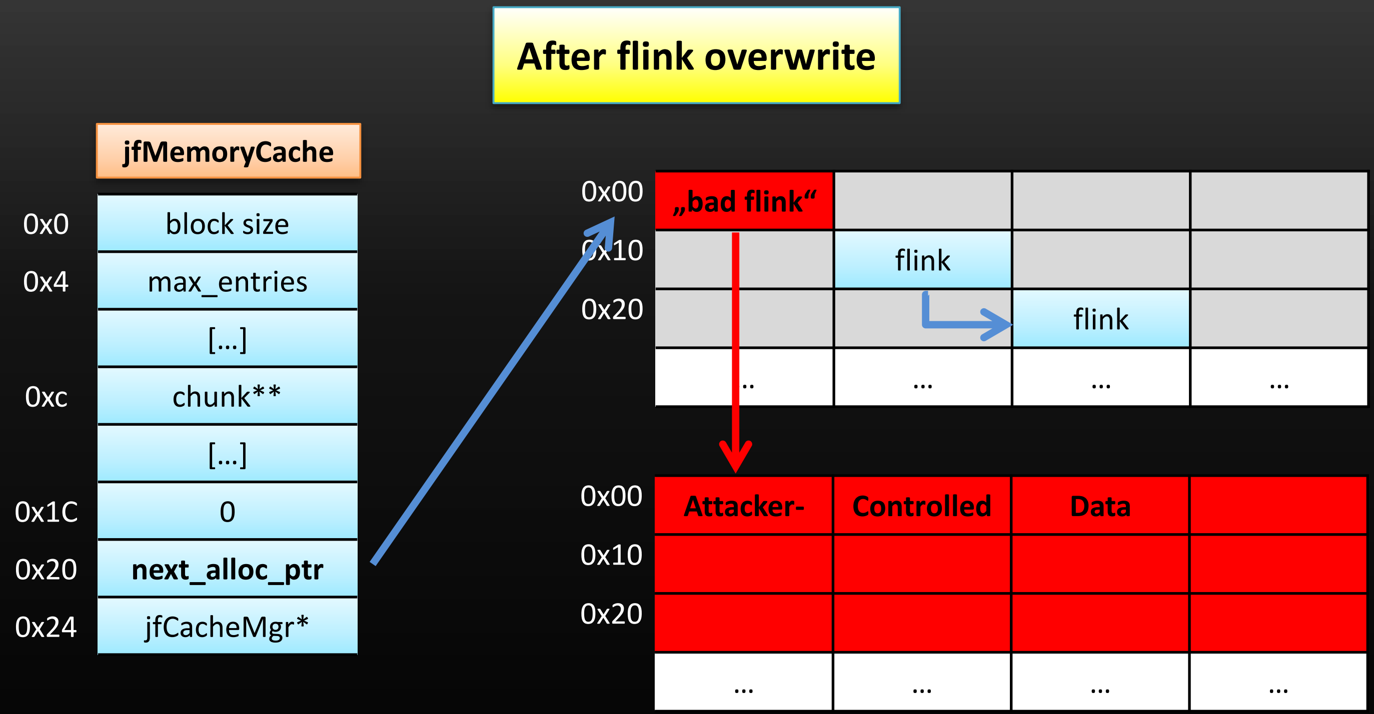
Hit the jfMemoryCache*
⇒ Block is *allocated*
⇒ Triggers when block is freed

- Both methods can be abused create a memory leak! But hitting the *flink* is the easiest way to go

Exploiting the Reader - Hit the flink!

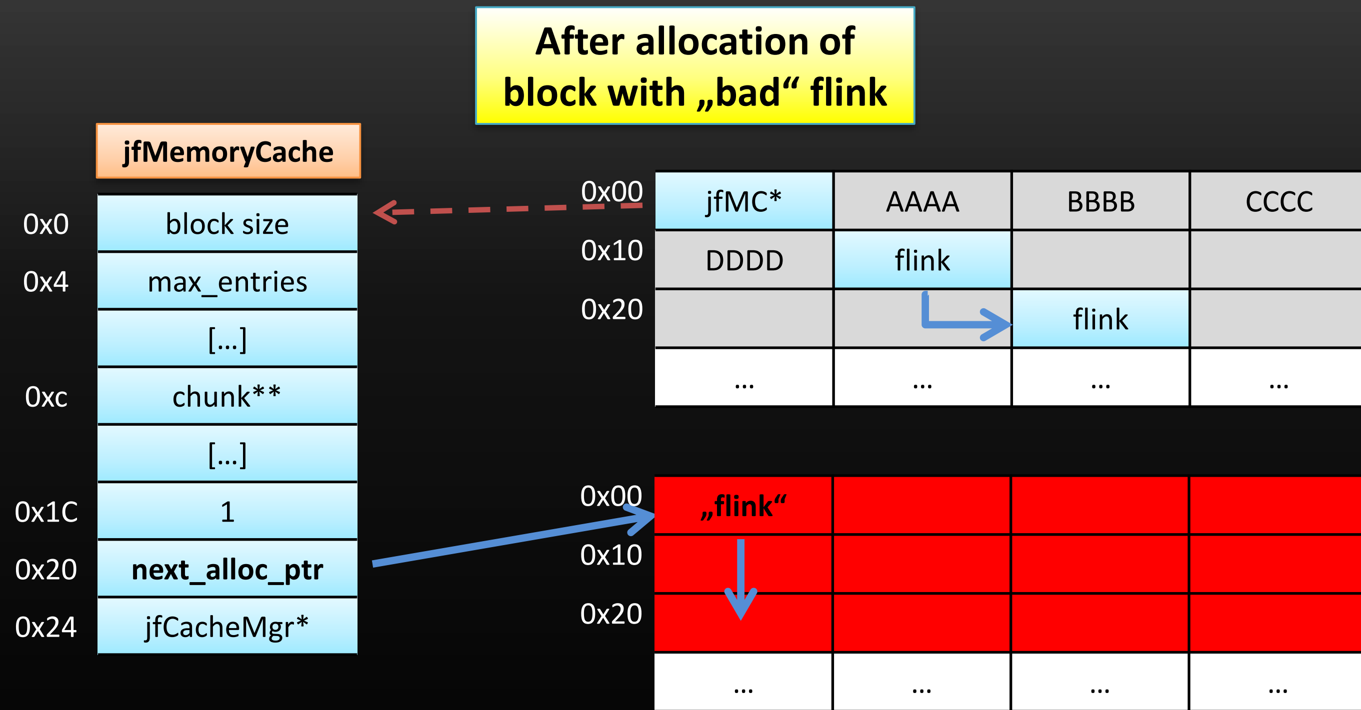


Exploiting the Reader - Hit the flink!



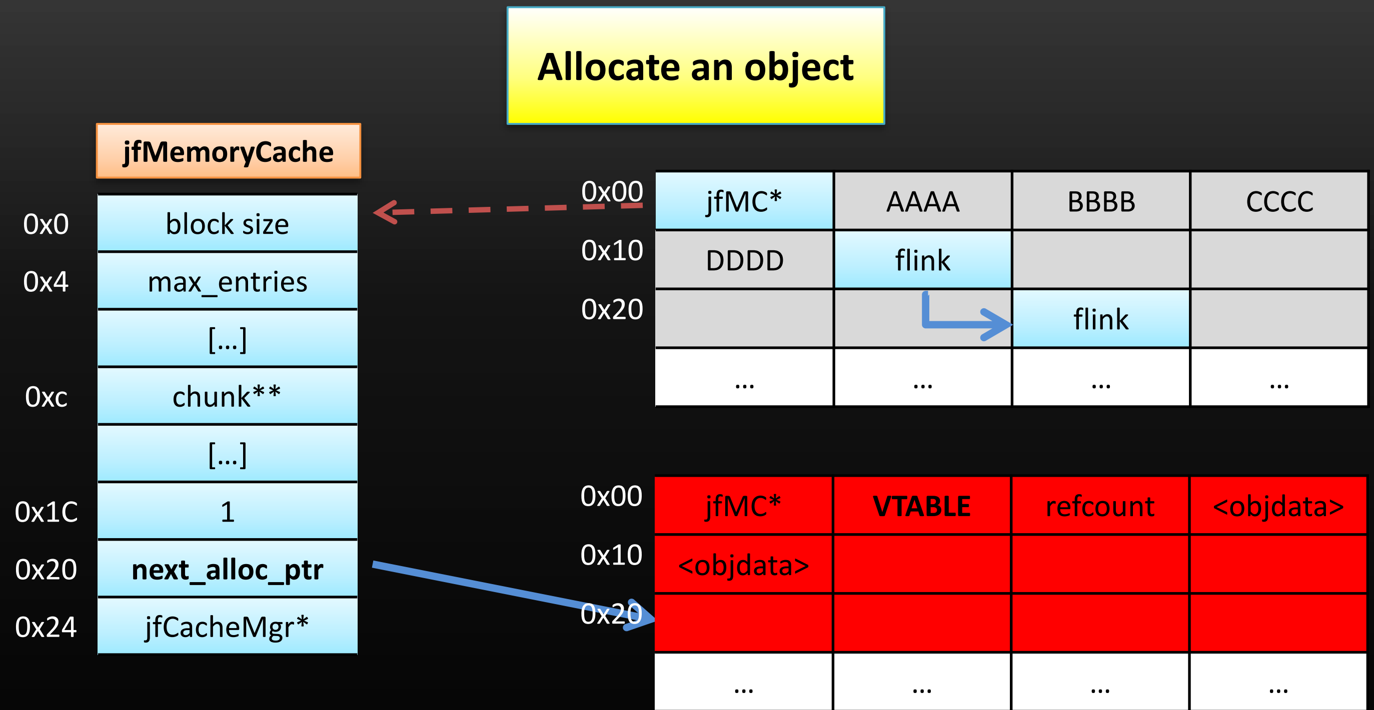
- Requirement: flink must point to controlled data after overwrite
- Still very flexible: Doable with nearly any kind of mem corruption!
- Let‘s see what happens when we allocate the „bad“ block

Exploiting the Reader - Hit the flink!



- *next_alloc_ptr* is overwritten with the „bad“ flink
- *flink* is overwritten with pointer back to jfMemoryCache
- Now what happens when we allocate an object of size 0x10…?

Exploiting the Reader - Hit the flink!



- Next allocation will return the data buffer after the „flink“
- The object will be placed in the middle of our controlled data => We get a vtable in controlled data!!

Exploiting the Reader - Hit the flink!

- As soon as the vtable is in a controlled area you can just read it out
- The controlled data area can be sprayed with strings or even float arrays as „landing zone“
- Set the overwritten float or replace the string with data which will point to your ROP pivot gadget
- For floats: You can compute their binary represenation after spec IEEE754:
  - 4.18356164518379836860971488084E-216 will be 0x13371337deadc0de on the heap
- GAME OVER!

Let‘s have a look at a practical example…

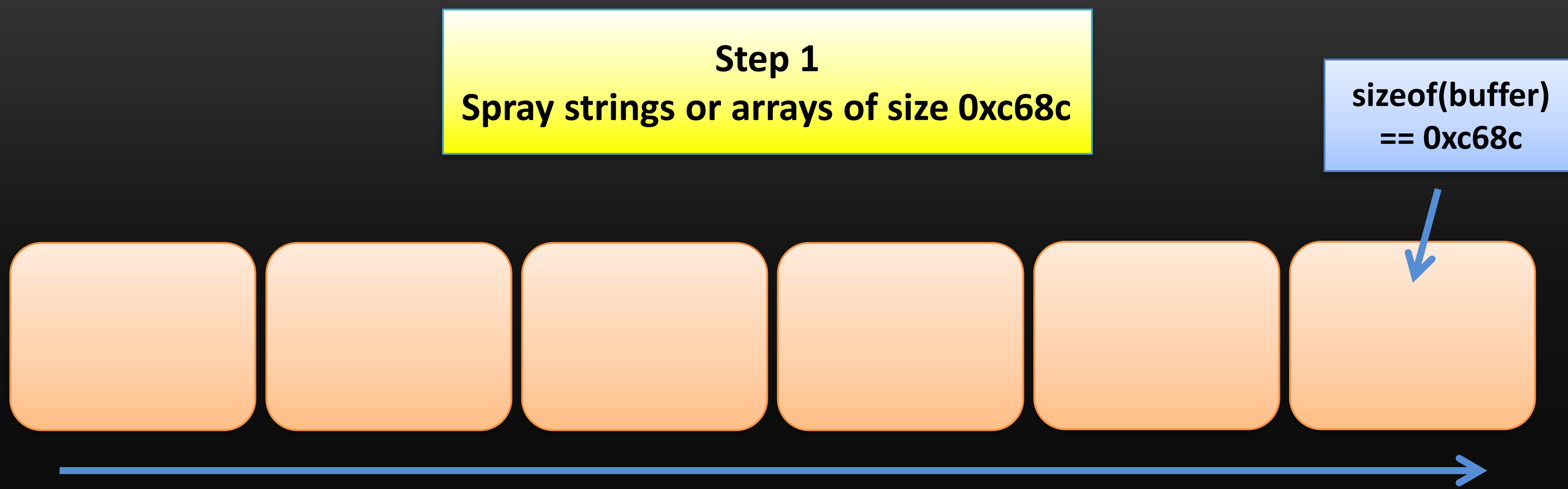
Setting:

A *0-DWORD write primitive* to an arbitrary address

Exploiting the Reader - Practical example

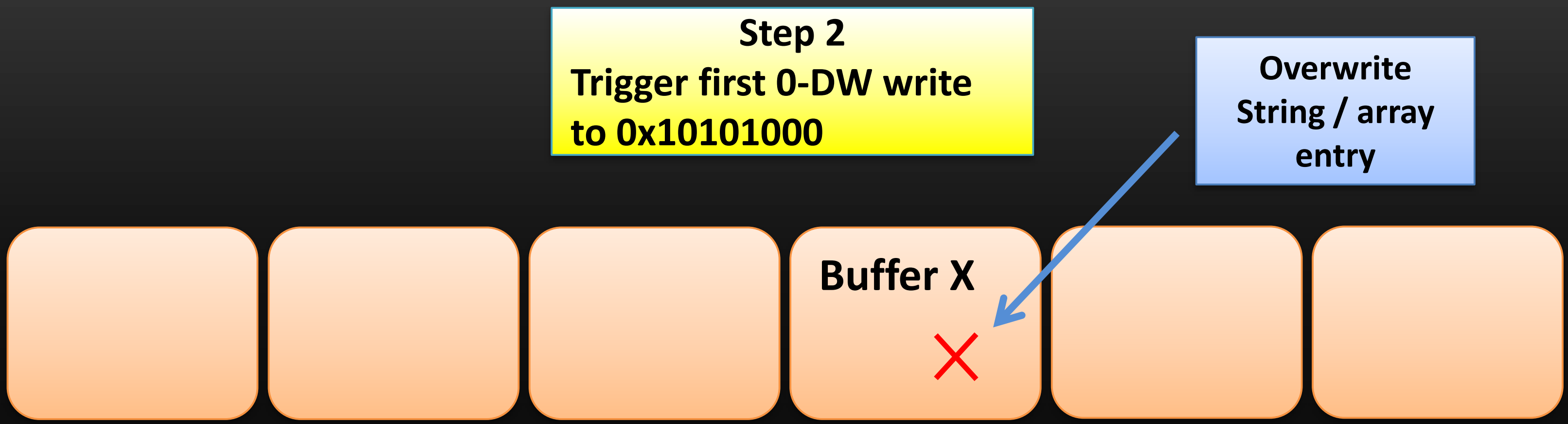
- Plan: Attack a flink in a chunk with block size 0x180 => corresponding target chunk size will be 0xc68c
- 0x180??
  - 0x180 == sizeof(jfDocumentImpl Object)
  - First object of this size which is created on jfCache
  - Range mechanism => Every object of size 0x100 - 0x180 will be placed in same chunk
  - Biggest object we can create: The template object
    - sizeof(Template object) == 0x140
  - Due to the rather unusual size it is „quiet“ in this chunk
    - Perfect for exploit reliability

Exploiting the Reader - Practical example



- Spray ~ 5000 * 0xc68c-sized buffers (~ 250MB)
- Address 0x10101000 will be mapped
  - This will be our target address for the „first shot“

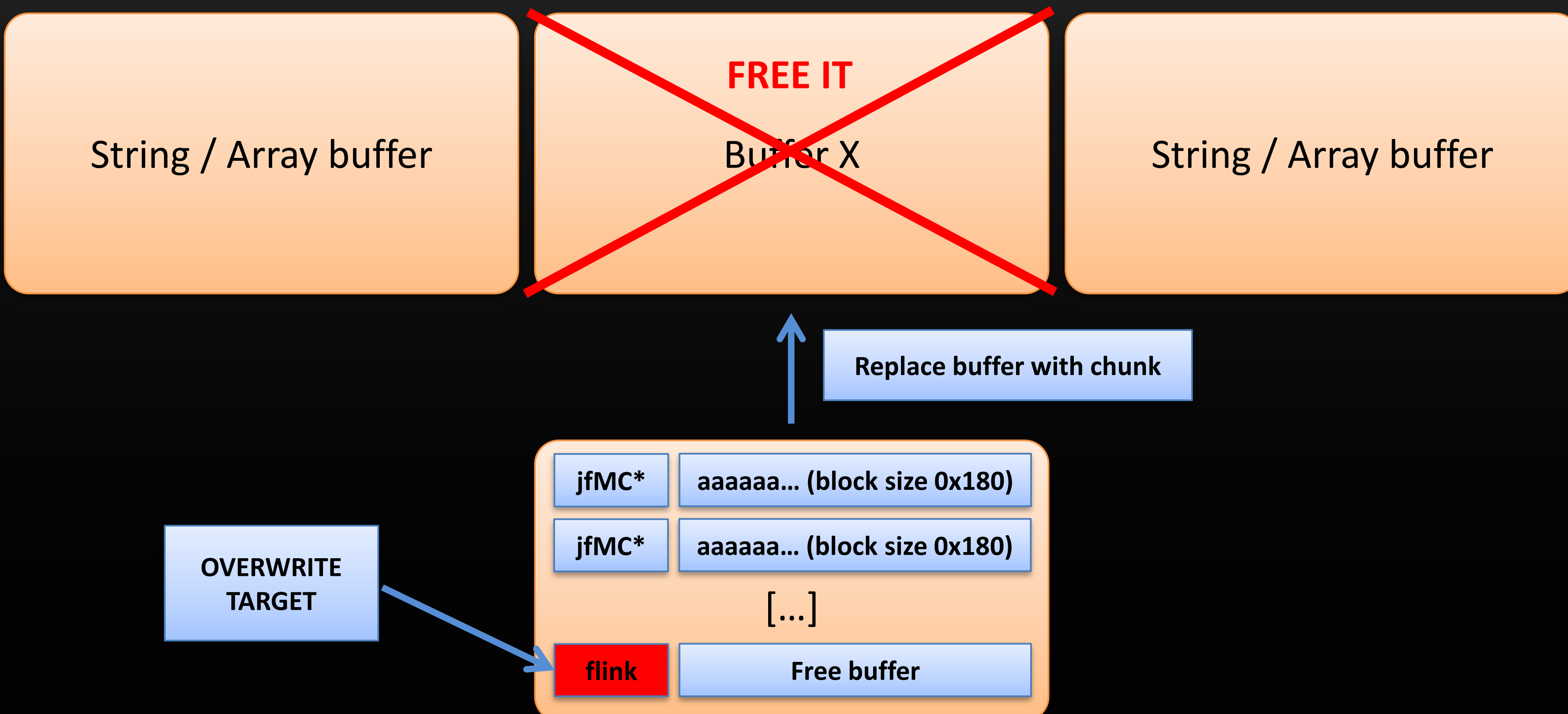
Exploiting the Reader - Practical example



- After writing to 0x10101000 search through strings or array entries for the overwritten data
- From the overwrite offset you can compute the *base address of buffer X*!

Exploiting the Reader - Practical example

Step 3
Free buffer X and replace it with a chunk



Exploiting the Reader - Practical example

- Before freeing the 0xc68c-sized buffer: Defragment size 0x180 on jfCache to fill „holes“ in the heap
- After freeing the 0xc68c-sized buffer: Allocate exactly 132 template objects:

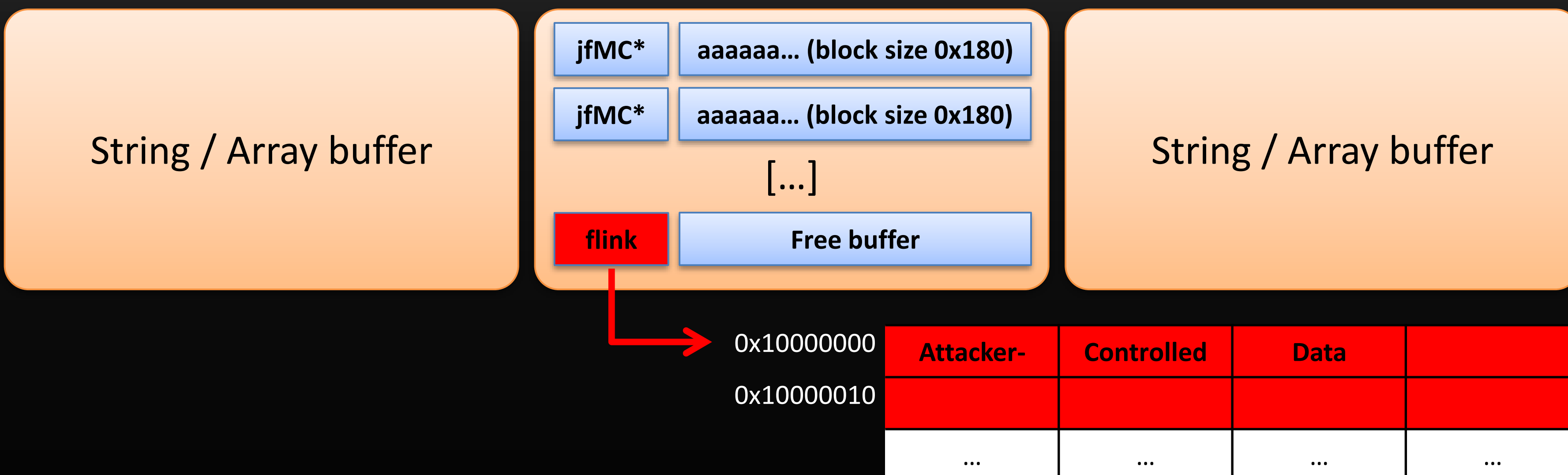
  132 * (0x180 + 4) = 0xc810

  => At least one chunk of size 0xc68c *must* be allocated

  => This chunk will replace the freed buffer
- The newly allocated chunk is NOT filled completely with allocated items – the last block in the chunk will be *free* with near 100% reliability

Exploiting the Reader - Practical example

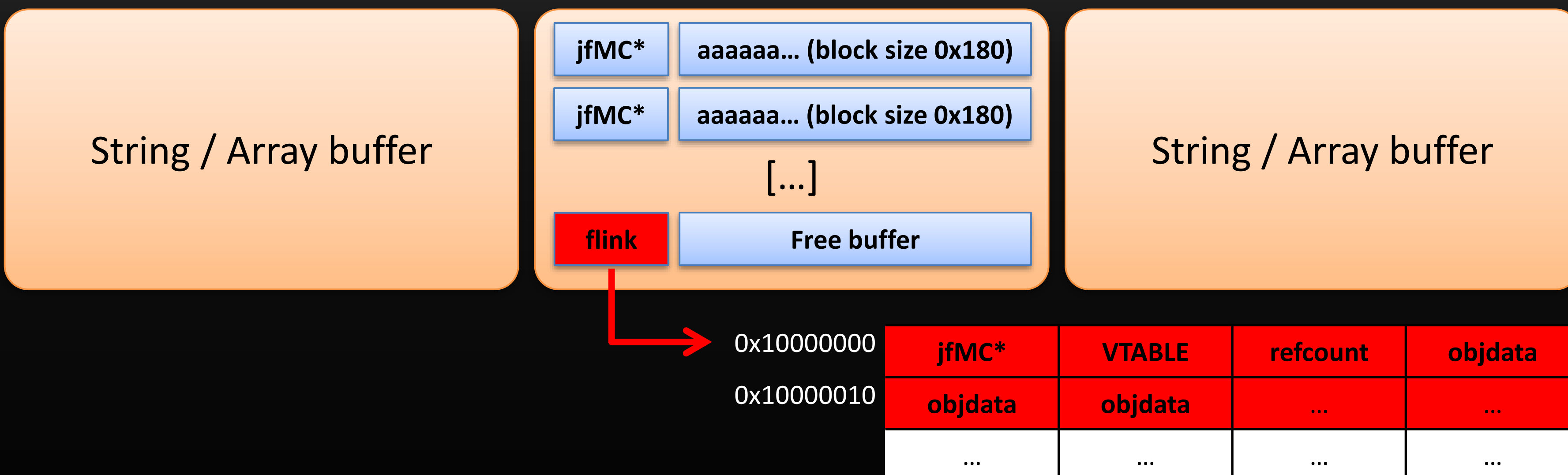
Step 4
Partially overwrite flink (set last 3 bytes to 0)



- We know the address of flink: chunkaddr + 131*(0x180+4)
- Partial overwrite: 0x10XXYYZZ => 0x10000000 (controlled!)
- Now allocate template objects of size 0x140…….!

Exploiting the Reader - Practical example

Step 4
Partially overwrite flink (set last 3 bytes to 0)



- The template-object will be placed into our data
- Search for changed bytes in our strings / arrays again
- Find vtable => ASLR bypassed => PWND! (EIP/ROP trivial…)

Conclusion

Conclusion

- Very easy, but highly effective technique to leak data
- No global RW primitive, but enough to pwn AR
- Version-independant
- OS-independant
- Very fast: From start to pwn ~ 1 sec if you use strings
  - Arrays are more elegant but searching them is sloooow…
- Flexible technique which can be used with almost every kind of overwrite
- Custom allocator proves once again to be a perfect target in memory corruption scenarios

Thank you for your attention! ☺

- Q&A