iOS Kernel Heap Armageddon

Stefan Esser
stefan.esser@sektioneins.de

VERSION 1.0

Introduction

When you look at the public research covering iOS kernel heap exploitation it all comes down to the kernel heap zone allocator that was first discussed by nemo[1]. In short this allocator separates kernel memory in zones containing memory blocks of the same size. It comes with heap meta data that when overwritten can be exploited to inject arbitrary memory areas into the freelist of the zone.

In this paper we will first recapitulate this information about the kernel heap zone allocator as previously discussed by nemo and Esser[2][3]. We will then take a look into other kernel heap managers or memory allocation wrapper functions that can be found in the Mac OSX and iOS kernel. After a brief overview over these wrappers we will look into recent changes in these allocators in latest iOS 5 versions. The paper continues by a look into kernel level application data overwrites in contrast to attacks against the zone allocator's freelist. The paper closes with the introduction of a generic technique that allows to perform kernel heap spraying and to control the layout of the kernel heap layout for kernel heap exploits.

Kernel Heap Zone Allocator

Mac OSX and also jailbroken iPhones come with a tool called `zprint` that allows to have a look into the kernel memory zones that are registered with the kernel heap allocator. It can be used like in the following example:

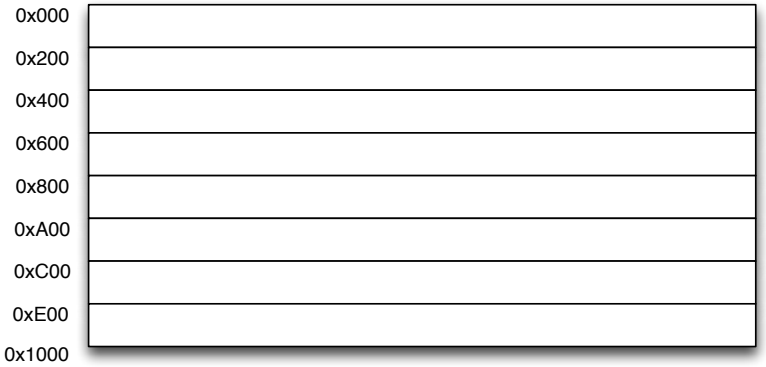
```
$ zprint kalloc
                          elem    cur    max    cur    max   cur alloc alloc
   zone name              size   size   size  #elts  #elts inuse  size count
   ------------------------------------------------------------------------------
   zones                   460    84K    90K    187    200   167   20K    44
   vm.objects              148   487K   512K   3375   3542  3103    4K    27 C
   vm.object.hash.entries   20    19K   512K   1020  26214   704    4K   204 C
   maps                    168    11K    40K     72    243    61    4K    24
   VM.map.entries           48   203K  1024K   4335  21845  3859    4K    85 C
   Reserved.VM.map.entries  48    27K  1536K    597  32768   191    4K    85
   VM.map.copies            48     3K    16K     85    341     0    4K    85 C
   pmap                   2192   134K   548K     63    256    52   20K     9 C
   ...
   tcp_bwmeas_zone          32     0K     4K      0    128     0    4K   128 C
   igmp_ifinfo             112     3K     8K     36     73     3    4K    36 C
   ripzone                 268     3K  1072K     15   4096     0    4K    15 C
   in_multi                136     3K    12K     30     90     2    4K    30 C
   ip_msource               28     0K     4K      0    146     0    4K   146 C
   in_msource               20     0K     4K      0    204     0    4K   204 C
   in_ifaddr               156     3K    12K     26     78     1    4K    26 C
   ip_moptions              52     3K     4K     78     78     1    4K    78 C
   llinfo_arp               36     0K    12K      0    341     0    4K   113 C
   unpzone                 152    27K  1132K    182   7626   129    4K    26 C
```

```
fs-event-buf                64     64K     64K    1024    1024      0     4K     64
bridge_rtnode               40      0K     40K       0    1024      0     4K    102 C
vnode.pager.structures      20     19K    196K    1020   10035    655     4K    204 C
kernel_stacks            16384   1232K   1232K      77      77     33    16K      1 C
page_tables               4096   6688K    ----    1672    ----   1672     4K      1 C
kalloc.large             64898   2218K   8961K      35     141     35    63K      1
```

This information is based on the kernel API functions `host_zone_info` and `mach_zone_info`. Both these API functions are very useful when it comes to constructing kernel heap exploits, because they allow retrieving detailed information about each kernel zone, like the amount of allocated blocks and the amount of free memory blocks. The later is very useful for controlling the kernel heap via heap feng shui[4] techniques as discussed by Sotirov. However with the introduction of iOS 6 Apple has closed down this path by protecting the kernel API functions against being used on factory iPhones. Nowadays it calls the `PE_i_can_haz_debugger` function that will only return true on jailbroken iPhones or on special Apple internal debugging devices or devices being booted by special debugging ramdisks that Apple most probably has. Anyway future kernel heap exploits can no longer rely on these functions.

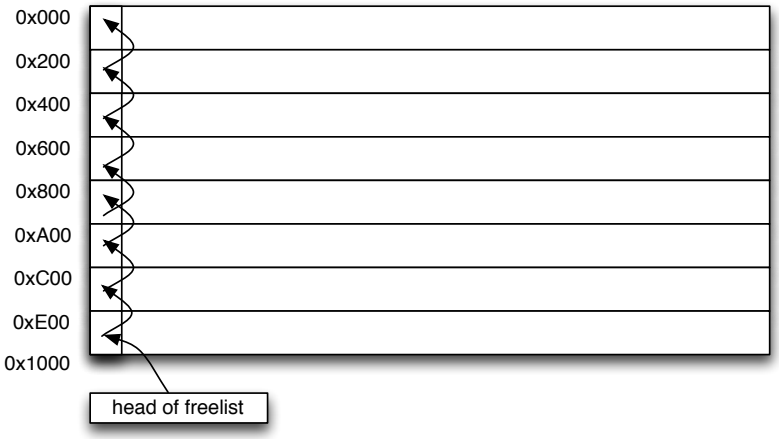
To understand how the kernel heap zone allocator works and can be exploited the following figures will document the inner working step by step. The allocator divides the kernel memory into zones that contain memory blocks of the same size. It starts by assigning a first chunk of memory (usually a single memory page) to the zone.

All the memory inside the zone is then divided into blocks of the same size. In our example each memory block is exactly 512 bytes in size.

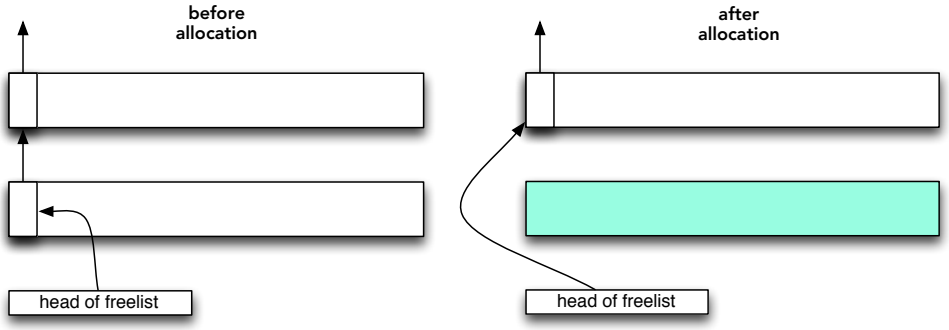


The memory manager uses the first 4 bytes of each free memory blocks as a pointer to another memory block. This is showed in the next figure.

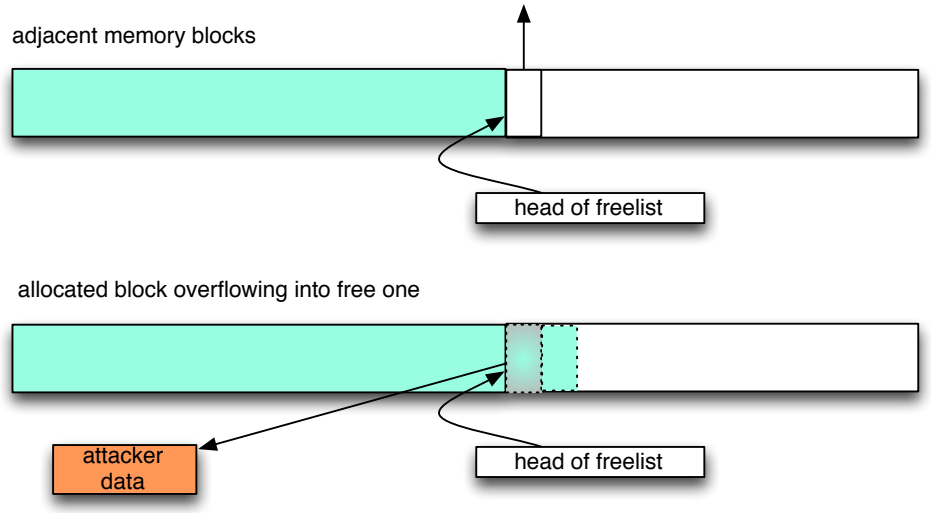
The zone allocator creates a linked list of free memory blocks, the so called freelist. The freelist is a LIFO list, with each element pointing to the next element in the freelist. Because the first free memory block inside a new memory page is added first, the free memory will be used in a reverse order, as you can see in the next figure.



When memory is allocated for a specific zone the last element added to the freelist, also named the head of the freelist is returned as allocated memory block. However before the newly allocated memory is returned, the pointer to the next element in the freelist is read from the first 4 bytes of the memory block. The pointer read is then made the new head of the freelist. The memory block it points to will therefore be the next returned. This principle is demonstrated in the next figures.



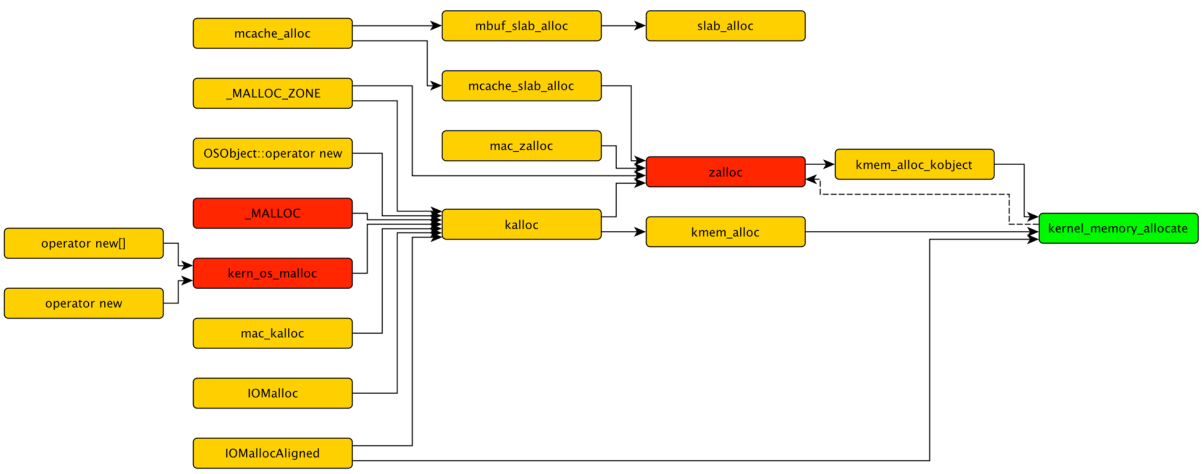
Now that we know the basic functionality of the heap zone allocator we can have a look into the exploitation of this memory allocator. When we look at two adjacent memory blocks, the first being an allocated buffer and the second being a free memory block, a buffer overflow will overwrite the heap meta data.



In case an attacker controls the data the buffer is overflown with, he can completely control the pointer to the next element in the freelist. As discussed before the allocation that will return the overwritten memory block will make the attacker controlled pointer the head of the freelist. And the allocation following that will return an attacker controlled memory block. In public iOS kernel heap exploits this technique was used to return a piece of memory that is in the middle of the system call table. By forcing the kernel to allocate that piece of memory and overwriting it with attacker controlled data it was possible to replace arbitrary system call handlers and achieve arbitrary kernel code execution.
It is reported that current betas of iOS 6 add some kind of memory tagging to the kernel heap allocator that while it does not stop attacks against the freelist in general, it stops the publically used attacks, because it only allows to inject memory blocks into the kernel's freelist that are already under full control of the attacker.

Other Kernel Heap Memory Managers and Wrappers

The Mac OSX and iOS kernels contain a number of other kernel heap memory managers and wrappers. The following figure shows a number of these wrappers and memory managers, but is most probably still incomplete.



In this section we will have a look into several of the mentioned allocators and wrappers and discuss their properties when it comes to exploitation.

kalloc()

`kalloc()` is a wrapper around `zalloc()` and `kmem_alloc()`. It uses `zalloc()` for all the smaller allocations and `kmem_alloc()` for the larger memory requests. It does this without keeping any extra heap meta data. It is therefore up to the caller to remember the size of the memory allocated, because the exact same size value is required when the memory is later freed with `kfree()`.
For storing data in kernel zones the memory manager registers a number of zones by the names `kalloc.xxx` where `xxx` is one of the kalloc zone sizes. When you use the `zprint` utility on iOS 5 it allows you to extract the following zones.

```
$ zprint kalloc
                          elem    cur    max    cur    max   cur alloc alloc
   zone name              size   size   size  #elts  #elts inuse  size count
   ------------------------------------------------------------------------------
   kalloc.8                 8    68K    91K   8704  11664  8187    4K   512 C
   kalloc.16               16    96K   121K   6144   7776  5479    4K   256 C
   kalloc.24               24   370K   410K  15810  17496 15567    4K   170 C
   kalloc.32               32   136K   192K   4352   6144  4087    4K   128 C
   kalloc.40               40   290K   360K   7446   9216  7224    4K   102 C
   kalloc.48               48    95K   192K   2040   4096  1475    4K    85 C
   kalloc.64               64   144K   256K   2304   4096  2017    4K    64 C
   kalloc.88               88   241K   352K   2806   4096  2268    4K    46 C
   kalloc.112             112   118K   448K   1080   4096   767    4K    36 C
   kalloc.128             128   176K   512K   1408   4096  1049    4K    32 C
   kalloc.192             192   102K   768K    546   4096   507    4K    21 C
   kalloc.256             256   196K  1024K    784   4096   740    4K    16 C
   kalloc.384             384   596K  1536K   1590   4096  1421    4K    10 C
   kalloc.512             512    48K   512K     96   1024    46    4K     8 C
   kalloc.768             768    97K   768K    130   1024   115    4K     5 C
   kalloc.1024           1024   128K  1024K    128   1024    80    4K     4 C
   kalloc.1536           1536   108K  1536K     72   1024    59   12K     8 C
   kalloc.2048           2048    88K  2048K     44   1024    39    4K     2 C
   kalloc.3072           3072   672K  3072K    224   1024    59   12K     4 C
   kalloc.4096           4096   120K  4096K     30   1024    28    4K     1 C
   kalloc.6144           6144   420K   576K     70     96    38   12K     2 C
   kalloc.8192           8192   176K 32768K     22   4096    20    8K     1 C
```

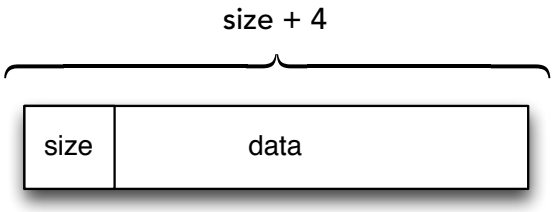
As you can see there is one kernel zone for each power of 2 between 8 and 8192 plus a number of additional zones for numbers dividable by 8 in between the powers of 2. These are the zones `kalloc.24`, `kalloc.40`, `kalloc.48`, `kalloc.88`, `kalloc.112`, `kalloc.192`, `kalloc.384`, `kalloc.768`, `kalloc.1536`, `kalloc.3072` and `kalloc.6144`. Prior to iOS 5 these kalloc zones were not available and the smallest zone was of the size 16. It is likely that this change was made to add zones that are better fitted to the most often performed allocations so that less memory is wasted.

kfree()

Before moving on to the next wrapper there is something notable about the `kfree()` function. As previously mentioned the caller needs to remember the size of the block it frees, because otherwise `kfree()` cannot know if `zfree()` or `kmem_free()` is supposed to be called and what zone it should return the memory to. In addition to that the memory manager keeps track of the larges allocated memory block and an attempt to free a block that is larger than this remembered value will just be ignored. This is a simple protection against double frees.

_MALLOC()

`_MALLOC()` is a wrapper around the `kalloc()` function. It prepends a short header to the allocated memory block that stores the size of the allocation. That way memory allocated by `_MALLOC()` can be freed without the kernel code needing to keep track of the size of the block. This e.g. enables memory allocations across system calls.



A special case is an allocation of 0 bytes. `_MALLOC()` simply refuses to allow such an allocation and returns a `NULL` pointer. It is unknown why Apple does not simply return a smallest size allocation, because allocating 0 bytes can happen in legal situations. The addition of the size header has two downsides, first it requires an integer addition in order to determine the size to be allocated and secondly it represents extra heap meta data that when overwritten can lead to exploitable conditions.

The danger of the integer addition in `_MALLOC()` becomes obvious when you look into its source code in the XNU source code tree, which is the version used for iOS 4. As you can see in the code below, Apple did not catch the integer overflow in iOS 4 and Mac OSX Lion, which results in a number of possible kernel heap corruptions.

```
void *_MALLOC(size_t size, int type, int flags)
{
    struct _mhead   *hdr;
    size_t memsize = sizeof (*hdr) + size;

    if (type >= M_LAST)
        panic("_malloc TYPE");

    if (size == 0)
        return (NULL);

    if (flags & M_NOWAIT) {
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        hdr = (void *)kalloc(memsize);
        ...
    }
    ...
    hdr->mlen = memsize;

    return  (hdr->dat);
}
```

However before the release of iOS 5 Apple learned about the possible integer overflow and closed it. The code was changed to catch the integer overflow and in case of an

overflow a `NULL` pointer is returned in the non blocking case. In the blocking case however a kernel panic is triggered as you can see.

```
void *_MALLOC(size_t size, int type, int flags)
{
    struct _mhead   *hdr;
    size_t memsize = sizeof (*hdr) + size;
    int overflow = memsize < size ? 1 : 0;

    ...
    if (flags & M_NOWAIT) {
        if (overflow)
            return (NULL);
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        if (overflow)
        panic("_MALLOC: overflow detected, size %llu", size);
        hdr = (void *)kalloc(memsize);
        ...
    }
    ...
    hdr->mlen = memsize;

    return  (hdr->dat);
}
```

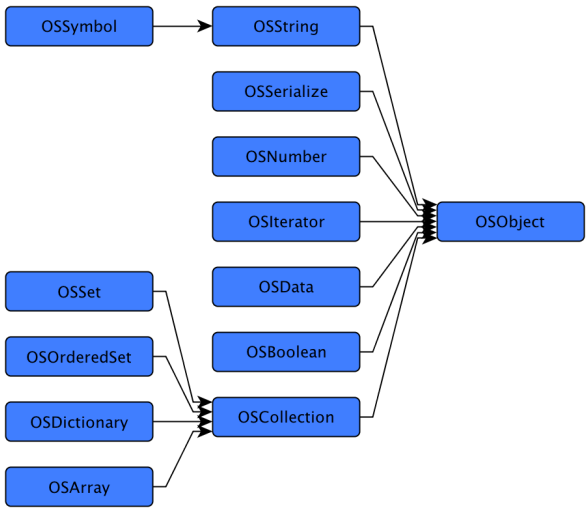
The memory block header containing the extra size field is a very interesting target for overwrites, because by overwriting it, the memory manager can be tricked to release the memory block into the freelist of the wrong zone. If the size is overwritten with a smaller size the block will be added to the freelist of smaller sized blocks. This does not result in a memory corruption but will result in a memory leak, because the end of the longer block will never be overwritten. If instead a bigger size is written into the header, the block will be added to a freelist of larger sized blocks. This will lead to a memory corruption, because on allocation the kernel believes the block to be larger than it is which will overwrite the adjacent memory when it is filled.

Kernel Heap Application Data Overwrites

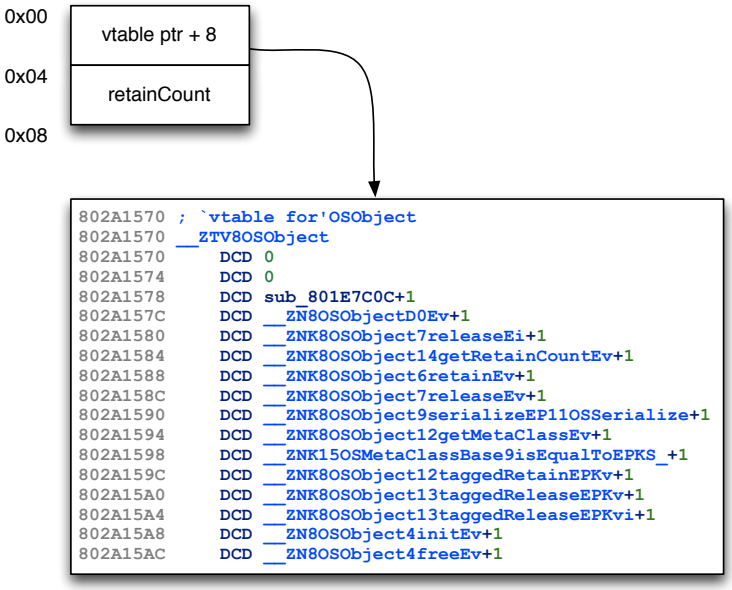
Considering that Apple is hardening the zone allocator right now and that some memory allocators do not have inbound heap meta data that can be overwritten, we will now have a look at attacking interesting kernel application data that is stored in the heap. For the rest of this section, we will use kernel level C++ objects as an example of such interesting application data that can be abused.

The `libkern` inside the iOS kernel implements a subset of a C++ runtime. It allows kernel drivers to be written in C++, which is heavily used in especially IOKit drivers. This is interesting, because it brings C++ vulnerability classes into the iOS kernel. However for our purposes only the in memory layout of these objects is of interest.

The following figure shows an overview of the base objects that are supported iOS kernel's C++ runtime and their inheritance:



As you can see all these objects are derived from the base object `OSObject`. Next we want to look into the memory layout of such objects. As you can see an `OSObject` consists merely of a `vtable ptr` and a reference counter.

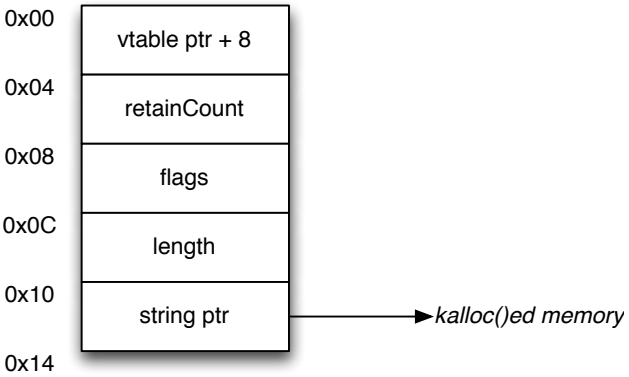


The `vtable ptr` points into the kernel's data segment, where the method table of the object is stored. The `retainCount` on the other hand is a bit more complicated. It is a 32bit value that stores a 16bit reference counter in its lower 16 bits. The upper 16 bits are used for a second reference counter that counts how often this object is part of a collection. It seems this was originally meant for debugging purposes, because the collection counter seems to be only used to verify that the normal reference counter does not drop below the collection counter. If that ever happens a kernel panic is triggered. A special thing about the reference counter is that is has an integer overflow protection built in. If the value of the reference counter ever reaches the number 65534 then the counter is frozen, which means that the reference counter will not be increased

and also not decreased anymore. Therefore the object cannot be destructed anymore and its memory never be freed.

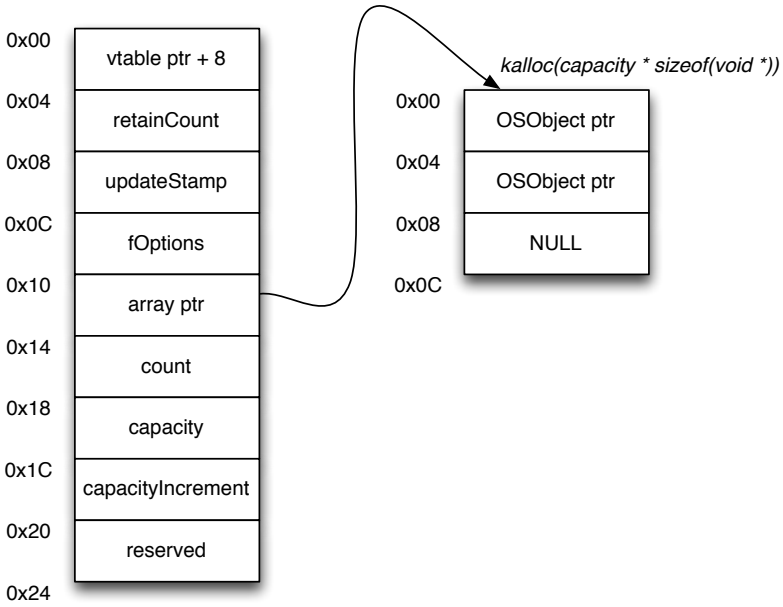
In order to understand how iOS kernel object overwrites are exploitable it is necessary to understand the impact from overwriting each field of an `OSObject` in memory. In case it is possible to overwrite the `vtable ptr` this allows to change the address of the table that is used to lookup methods of the object. Once this pointer is overwritten every action performed on the object will allow arbitrary kernel code execution. If the `retainCount` is overwritten this allows to set the reference counter to a value smaller than the actual number of references existing. This allows freeing the object early, which results in the typical use after free exploitation through the use of dangling references. Once freed the next allocation of the same size allows to completely replace the content of the object.

`OSObject` is the simplest of all the C++ objects in the iOS kernel. Other objects like `OSString` are more complicated and contain more properties or properties of different types. It is therefore interesting to look a bit further and analyze their memory layout. First lets have a look at the `OSString` object, its memory layout is visible in the next figure.



In addition to the properties known from `OSObject` the three properties `flags`, `length` and `string ptr` are new. The `flags` field just controls if the string pointer inside the object should be freed on its destruction or not. This is usually only interesting if the other fields can also be overwritten. More interesting is the `length` field. If the length of the string is changed to a value bigger than the original value this can either lead to a kernel heap information leak or to a memory corruption on destruction. The memory corruption is caused because a too large length will result in a short length memory block being added to the freelist of the wrong kernel heap zone. If the freed memory is later reallocated, the pointer returned will point to a memory block that is actually smaller than expected. When the kernel fills this smaller block with data the superfluous data will overwrite the adjacent memory. The last field that could be overwritten is the string pointer itself. If this pointer is overwritten it can result kernel heap information leakage or a memory corruption on destruction. In this case an attacker could inject an arbitrary memory address into the freelist of a specific zone, which results in memory corruption once that memory is reallocated and filled by the kernel.

Another interesting object for overwrites is `OSArray`. It contains more properties and therefore offers some new and interesting possibilities for overwrites. Lets have a look at in memory layout of an `OSArray` object:



The fields `updateStamp`, `reserved` and `fOptions` are not interesting for overwrites, because they cannot lead to some interesting exploitable scenario. However all the other fields allow for interesting exploitable scenarios. The fields `count`, `capacity` and `capacityIncrement` are all numbers involved in the allocation of memory with `kalloc()`. Overwriting these numbers will confuse the kernel and make him allocate or deallocate wrong amounts of memory. This can lead to information leaks or different memory corruptions. The last field the `array ptr` is the most interesting one to overwrite. This pointer points to an array of pointers to objects derived from `OSObject`. By overwriting it is possible to let the kernel access arbitrary constructed objects, which will result in arbitrary code execution inside the kernel. An alternate attack would be to overwrite the memory block that stores the array directly.

This concludes our little overview of C++ objects in the iOS kernel, their memory layout and the possibilities that arise from overwriting them. Keep all this information in mind for the next section where we use these objects to fill the iOS kernel heap and control its layout.

Controlling the iOS Kernel Heap

For successful exploitation of kernel heap memory corruptions it is required to bring the kernel heap from an unknown starting point to a predictable state in a controlled way. For this purpose there have been different techniques proposed. The simplest of these is called heap spraying, which just comes down to filling memory with specific data by repeatedly triggering the same allocations over and over again, until a large percentage of the memory is filled with this pattern (or another terminating condition is triggered). In order to implement heap spraying it is required to have an allocation primitive that is executed repeatedly. It is unknown who originally invented heap spraying, because this technique was already used around or before 2001.

A more complicated and better technique to control the state of a heap was called heap feng-shui by A. Sotirov in 2007. In his BlackHat talk he described how it is possible to get from a heap in an unknown state to one with a controlled memory layout. For this it is required to first fill all the holes in the heap by repeatedly allocating memory. Once all holes are closed further allocations will be adjacent to each other. Freeing memory blocks within these adjacent areas will poke holes in controlled positions so that following allocations end up in these holes. In that way it is possible to control the heap layout in a way that an overflowing buffer will overflow exactly the data we want it to overflow. Naturally implementing a heap feng-shui technique is more complicated than a heap spray and requires not only an allocation primitive, but also an deallocation primitive.

In previous public iOS kernel exploits the allocation and deallocation primitives were always very specific and depending on the actual exploited functions. We will therefore introduce you to a more generic solution that can control the kernel heap without vulnerability specific allocation and deallocation primitives.

The iOS kernel contains a very interesting function called `OSUnserializeXML()`. It is called by many of the IOKit API functions and is used to pass objects from user space to kernel pace. The functions takes an input in XML .plist format, that defines numbers, booleans, strings, data, dictionaries, arrays, sets and references. An example of such an XML plist can be seen below.

```
<plist version="1.0">
<dict>
        <key>IsThere</key>
        <string>one technique to rule them all?</string>
    <key>Answer</key>
        <true />
        <key>Audience</key>
        <string>meet OSUnserializeXML()</string>
</dict>
</plist>
```

By constructing such XML .plist data packages it is therefore possible to create arbitrary object collections in memory and therefore allocate arbitrary amounts of different types of objects in all sizes and shapes. We can use that to control the kernel heap in any way we like. The following table can be used as a cheat sheet listing the memory sizes of the base objects.

	in memory size	kalloc zone size	additional alloc
OSArray	36	40	+ capacity * 4
OSDictionary	36	40	+ capacity * 8
OSData	28	32	+ capacity
OSSet	24	24	+ sizeof(OSArray)
OSNumber	24	24	
OSString	20	24	+ strlen + 1
OSBoolean	12	16	cannot be generated by OSUnserializeXML()

We will now look into how the XML data must be constructed to allow for heap spraying and heap feng-shui.

Allocate repeatedly

The first thing we need to be able todo is to allocate arbitrary amounts of memory blocks in arbitrary sizes. Unfortunately it is not possible to loop inside an XML .plist data block. It is however not limited and we can therefore allocate as much data as we want.

```
<plist version="1.0">
<dict>
        <key>ThisIsOurArray</key>
        <array>
                <string>again and</string>
                <string>again and</string>
                <string>again and</string>
                <string>again and</string>
                <string>again and</string>
                <string>again and</string>
                <string>...</string>
        </array>
</dict>
</plist>
```

The previous example uses an `<array>` object and fills it with an arbitrary number of strings. In order to do a kernel heap spray we therefore just need to construct a very big XML data object and feed it to an appropriate IOKit API function.

Allocate attacker controlled data

The downside of using `<string>` data objects for spraying the iOS kernel heap is that NUL bytes cannot be contained. It is therefore not possible to use string objects for spreading the heap with completely arbitrary data structures. However the `<data>` object type comes to the rescue. It allows the creation of arbitrary data structures because the data is base64 encoded and therefore has no limit of character values. Alternatively the kernel also supports simple hex format. You can see this in the example below.

```
<plist version="1.0">
<dict>
   <key>ThisIsOurData</key>
   <array>

      <data>VGhpcyBJcyBPdXIgRGF0YSB3aXRoIGEgTlVMPgA8+ADw=</data>
      <data format="hex">00112233445566778899aabbccddeeff</data>
      <data>...</data>
   </array>
</dict>
</plist>
```

The `<data>` object type is also more convenient, because it reads in chunks of 4096 and therefore it does not allocate blocks in memory zones we are interested in while decoding the XML. By combining the power of `<data>` and `<array>` it is possible to perform a kernel level heap spraying. Heap feng-shui requires more control which we will solve next.

Fill arbitrary sized memory blocks with application data

For heap feng-shui we need to be able to not only allocate arbitrary sized memory blocks, but to allocate arbitrary sized memory blocks that are filled with data that when overwritten can result in arbitrary code execution. For this we once again make use of the `<array>` object type (alternatively we can use the `<dict>` object type, too).

```
<plist version="1.0">
<dict>
   <key>ThisArrayAllocates_4_Bytes</key>
   <array>
      <true />
   </array>
   <key>ThisArrayAllocates_12_Bytes</key>
   <array>
      <true /><true /><true />
   </array>
   <key>ThisArrayAllocates_28_Bytes</key>
   <array>
      <true /><true /><true ><true /><true /><true /><true />
   </array>
</dict>
</plist>
```

In this example we use the allocated arrays to fill the memory with pointers to boolean objects. The booleans will not be separate objects that allocate memory. Instead they will increase the reference counter of a global true object. If overwritten this will allow to supply crafted objects to the kernel that when used result in arbitrary code execution. Alternatively the `<dict>` object type can be used for this attack. The difference is that in one case the multiplier is 4 for the single object pointer and in the dictionary case the multiplier is 8, because a key and a value object pointer is stored.

Poking holes into allocated areas

The last thing that is required for total control of the kernel heap is the possibility to not only allocate arbitrary sized memory blocks, but also the ability to poke arbitrary sized holes into these allocations. To understand how poking holes in already allocated memory is possible, with the help of `<dict>` objects, have a look at the following example.

```
<plist version="1.0">
<dict>
  <key>AAAA</key>
  <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
  <key>BBBB</key>
  <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
  <key>CCCC</key>
  <data>ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ</data>
  <key>DDDD</key>
  <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
  <key>EEEE</key>
  <data>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</data>
  <key>CCCC</key>
  <true />
</dict>
</plist>
```

In this example you can see that the key `CCCC` is assigned two times. The first time it is inserted into the dictionary and the second time the key's value is updated and the previous value object is destructed. The destruction of this data object will therefore free the data object itself and also free the data, which consists of a base64 decoded repeated Z character in this case. We have therefore effectively poked a hole into the memory. With this last piece in the puzzle is should be no problem for you construct XML .plist documents that control the heap to your bidding.

Conclusion

Within this paper we have started with a recapitulation of the iOS kernel's heap zone allocator and its exploitation as previously discussed by various authors. We then had a look at other kernel heap allocators and the additional heap meta data they come with. We have discussed how overwriting this data can be abused and have also mentioned recent changes in these allocators. We have then taken a step away from the exploitation of kernel heap meta data structures and discussed iOS C++ kernel objects, their in memory structure layout and what can be gained by overwriting them in memory. Finally we have introduced a generic technique to perform kernel heap spraying and kernel heap feng-shui with the help of the `OSUnserializeXML()` function that is used by many IOKit API functions. This new technique allows to not only spray the heap with arbitrary data and control its layout completely, but it also fills the kernel heap with interesting kernel application data in form of kernel level C++ objects that once overwritten allow for easy code execution.

References

[1] E. Perla, M. Oldani, "A Guide To Kernel Exploitation - Attacking the Core", 2010, http://www.attackingthecore.com/

[2] S. Esser, "IOS Kernel Exploitation, BlackHat USA", 2011 https://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_WP.pdf

[3] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, R.-P. Weinmann, "IOS Hacker's Handbook", 2012, http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118204123,descCd-description.html

[4] A. Sotirov, "Heap Feng Shui in Javascript, BlackHat Europe", 2007 https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf