HEAP OVERFLOWS AND THE IOS KERNEL HEAP

September 5, 2019

Last week, Google published a series of blog posts detailing five iOS exploit chains being used in the wild that were found by Google's Threat Analysis Group (TAG) team back in February. The first part of my write-up was an overview of the different stages in the first exploit chain. In this part, we will look at the iOS kernel heap and how the exploit developer uses the heap overflow in this exploit.

Before talking about the specific heap overflow used in the exploit, let's first do a quick recap on heaps in general, the iOS kernel heap in particular, and how heap overflows actually work.

Heaps, in the context of programs written in languages like C and C++, allow functions to allocate a temporary exclusive region of memory to store variables and structured data. Once allocated, these memory regions are permanently reserved until the memory is manually released by the programmer back to the heap manager, at which point the heap manager can "recycle" it for use by another bit of code.

In order for programs to operate correctly, it is critically important that programmers using the heap follow a few simple rules. For example, they must avoid writing to addresses outside of the allocated region, and they must be careful to never use or write to data in an allocation that has been released back to the heap.

Failure to follow these rules invalidates the heap's *exclusivity* guarantee. This can lead to different variables ending up unintentionally aliased to the same underlying memory addresses. Since those variables might have very different security properties, like one containing data to be processed and another containing security critical properties like function

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

pointers, this can very often be exploited by exploit developers to take control of the process — in this case, the process taken over is the iOS operating system kernel itself.

In this iOS exploit, the vulnerability around which the exploit is built is in a graphics driver, and occurs on an object allocated via the *IOMalloc* function. This allocator uses the *kalloc allocator* under the hood, and will therefore a heap overflow of this object will spill data onto adjacent kalloc-allocated objects.

The *kalloc* heap allocator works in a very different way to the *glibc* heap allocator, which I've discussed in two previous posts, but its broad purpose is still basically the same: it allows kernel driver developers to allocate and deallocate memory for variables during the normal course of managing the system.

One of the best primers on how the iOS kernel's *kalloc* heap allocator works is by Stefan Esser, and can can be found here, but there are a few things we need to highlight. First of all, *kalloc* allocations are private to the operating system itself. The location and contents of these allocations are not visible to apps running on the device.

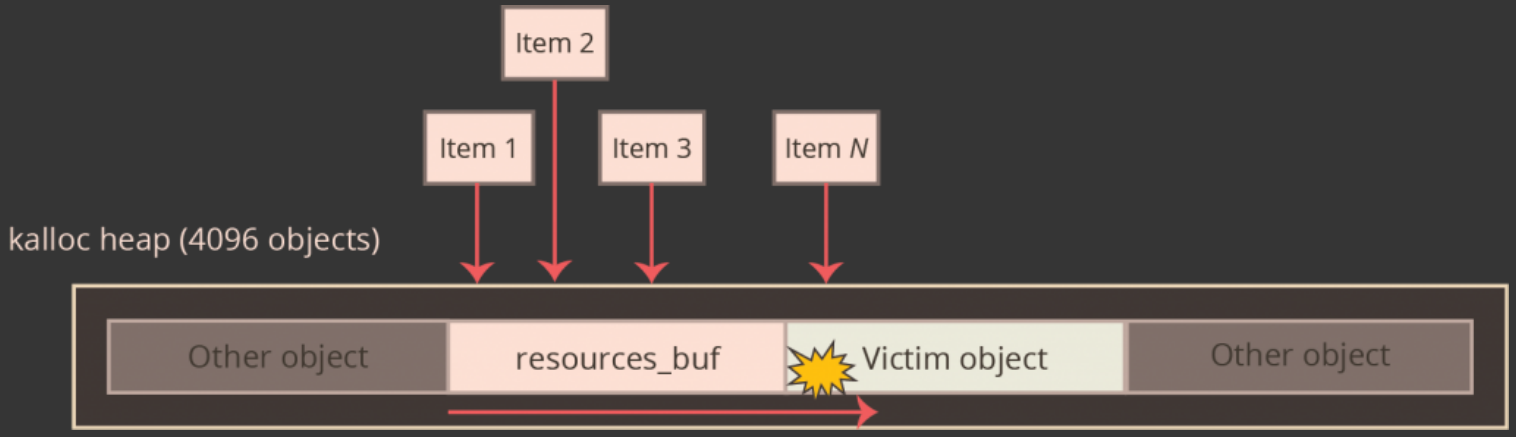
Second, objects allocated with *kalloc* traditionally didn't use heap metadata between live objects to track things like allocation chunk lengths. When a *kalloc* heap allocation is released, the programmer must remind *kfree* of the original allocation length so that *kfree* can recycle it for other allocations. If this length was misremembered, this would lead to a vulnerability category called a *zone-transfer*, leading to a heap overflow, although iOS since iOS 9 now has mitigations in place to defend against this attack category. For this reason, iOS kernel developers often use *kalloc* indirectly through easier-to-use APIs such as _MALLOC, even though the allocation is still handled internally by *kalloc*.

The *kalloc* allocator internally uses a *zone allocation* strategy (implemented via *zalloc*), which has the effect that allocations cluster together based on their size. For example, an allocation of size 4096 bytes will be allocated in the *kalloc.4096* zone alongside other 4096-byte allocations, whereas an allocation of size 2048 will be allocated far away in the *kalloc.2048* zone. This means that when a heap overflow on a 2048 byte *kalloc*-allocated object occurs, the adjacent victim object that gets corrupted will be of a similar size in the *kalloc.2048* zone.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

In this particular exploit, the vulnerability occurs on an object whose length the exploit developer can influence. The exploit developer therefore can choose which *kalloc* zone the vulnerability will be triggered in. The exploit developer chooses to arrange for all of the interesting objects to be allocated in the *kalloc.4096* zone. This zone is a bit quieter than the smaller zones, since programs tend to allocate and deallocate small allocations much more frequently than larger allocations. Choosing this zone therefore makes the exploit developer's life a little bit easier.

The specific vulnerability in this exploit is nothing particularly special; the author of a graphics kernel driver made a bad assumption about a complex input data structure and performs a heap allocation whose size is based on this faulty assumption. By providing an input that violates this assumption, the kernel is tricked into writing more entries into that heap allocation than it had planned for, and the final few entries can be made to spill over onto an unsuspecting object adjoining it on the kernel's heap.



EXPLOITING THE BUG

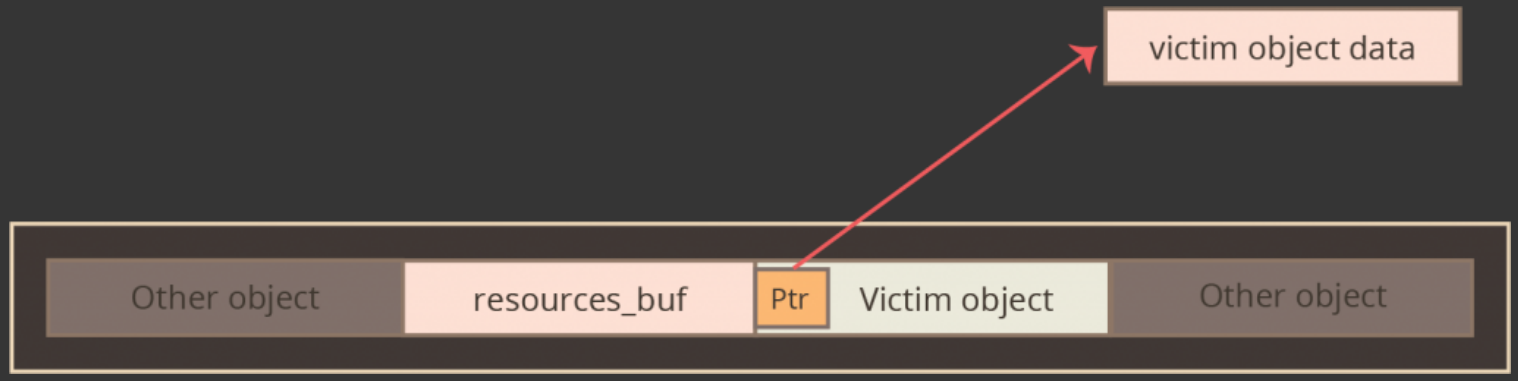
Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Although the bug itself is nothing special, what is interesting about this heap overflow vulnerability is the narrow field of control that the exploit writer has over the data that spills onto the neighboring object during the heap overflow. Linear heap overflow vulnerabilities often allow an exploit developer to spill *attacker-controlled* data beyond a memory allocation's defined limits, but here this is not the case. The overflowed data is necessarily always a pointer to an *IOAccelResource2* object. The attacker has no control over the value of that pointer, and also has limited control over the data that it points to.

Narrow field-of-control vulnerabilities such as this one can often feel unexploitable, but a creative approach can often be used to convert such vulnerabilities into a more useful category. In this case, the exploit developer uses the heap-overflow to construct an artificial *use-after-free* vulnerability, and this serves as the platform for building a more controllable exploit.

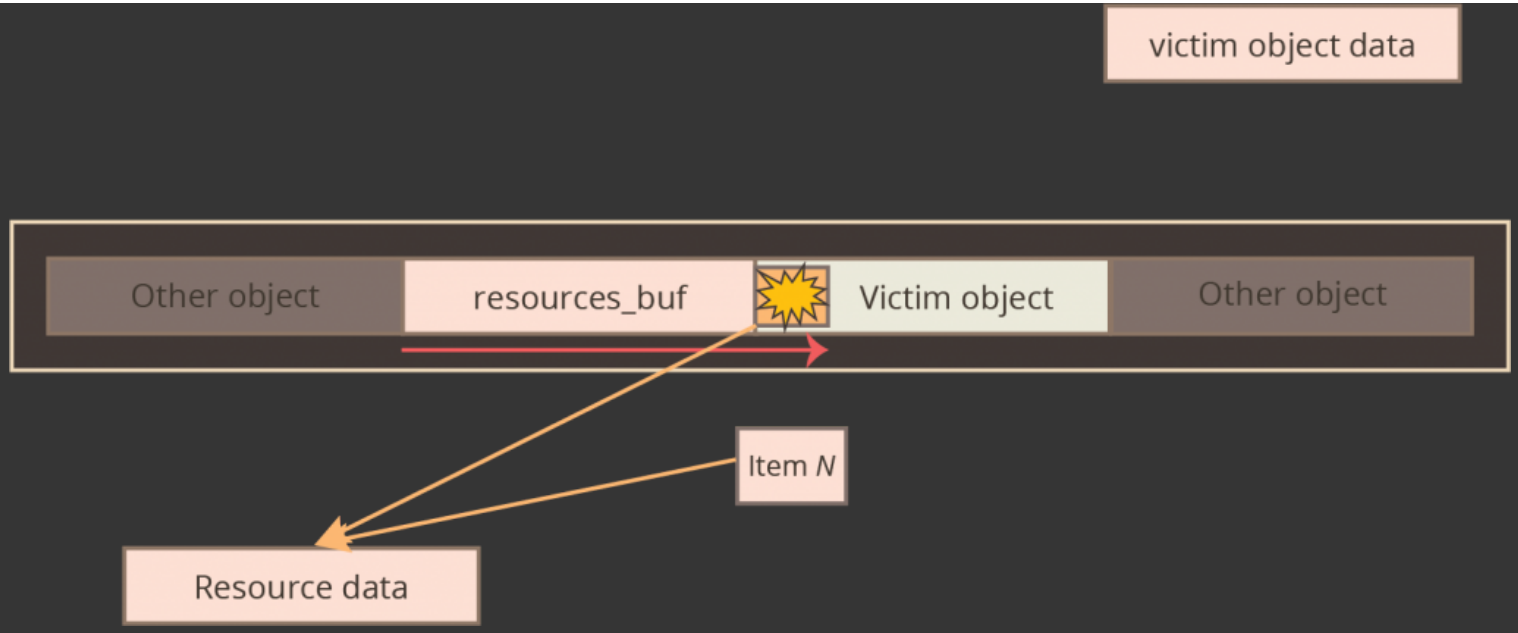
This is a little bit complicated, and is perhaps best explained slowly, step-by-step. The strategy is as follows:

First, the exploit developer will need to find a suitable *victim object* whose first field is a pointer to a buffer under the victim object's control. This buffer needs to be freed at a controllable time, probably when the victim object is itself released.



The exploit developer now triggers the linear heap overflow. This overflow will cause the victim object's *pointer* value to be invalidly replaced with a pointer to an *IOAccelResource2* object. This has the effect of orphaning the victim object's data buffer, but we don't particularly care about this memory leak for the purposes of this exploit.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



The next step is to trigger a *free* of the resource data itself. This causes the *victim object*'s pointer to now be unexpectedly free. Nobody in the kernel expects this pointer to be freed yet, because this is a completely unnatural event. All further interactions with the victim object's pointer are now use-after-free events.

Since a use-after-free vulnerability will give the attacker a much wider field of control compared with this heap overflow, this is a solid strategy.

The question then arises: what *victim object* should the attacker choose?

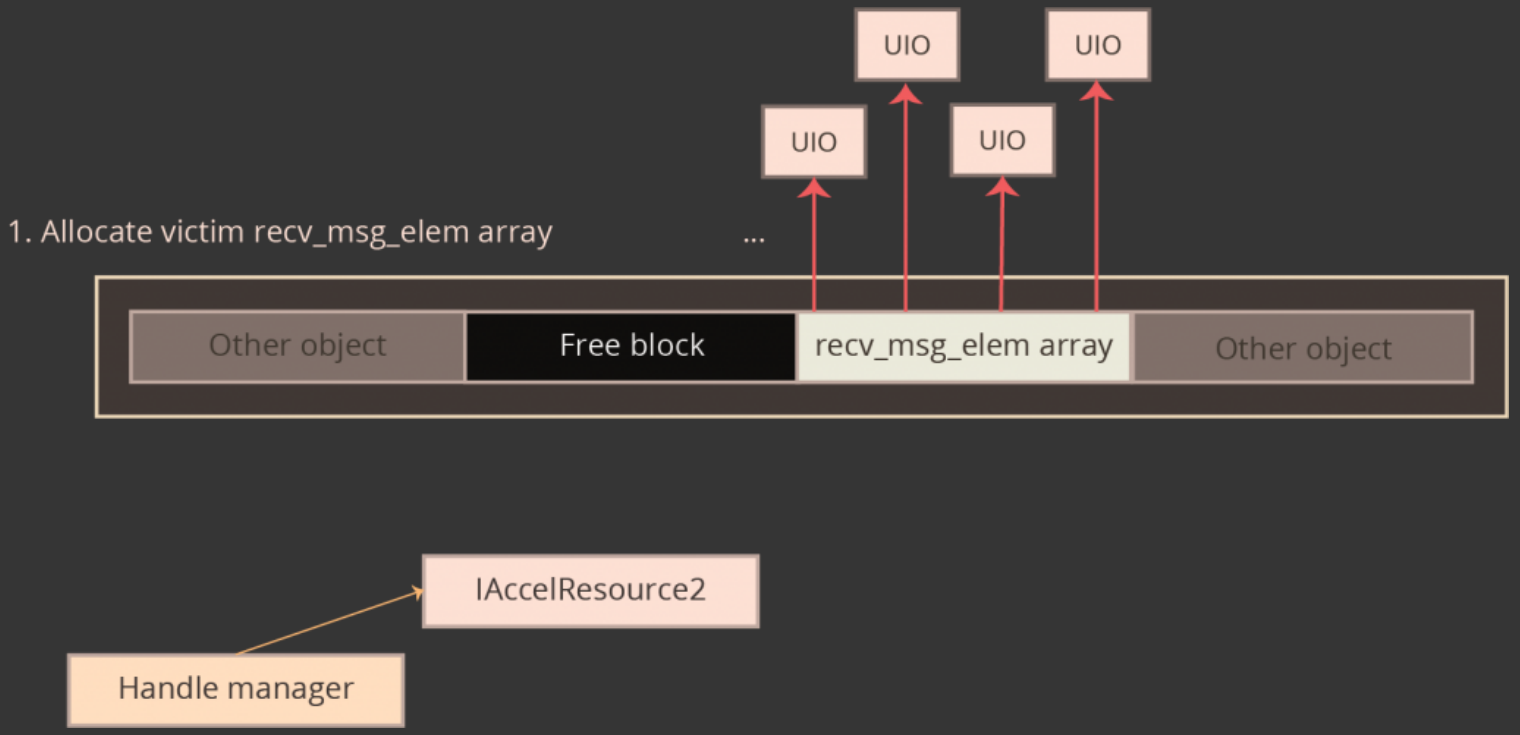
The attacker has a few constraints on what objects to choose from. The victim object should be something that can lie adjacent to our overflowing object, and thus the exploit developer needs to be able to make it live in the same *kalloc.4096* zone near the *resources_buf*. It needs a *kalloc*-allocated pointer near the beginning of it, and we need to be able to cause it to release that pointer at a controllable time.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

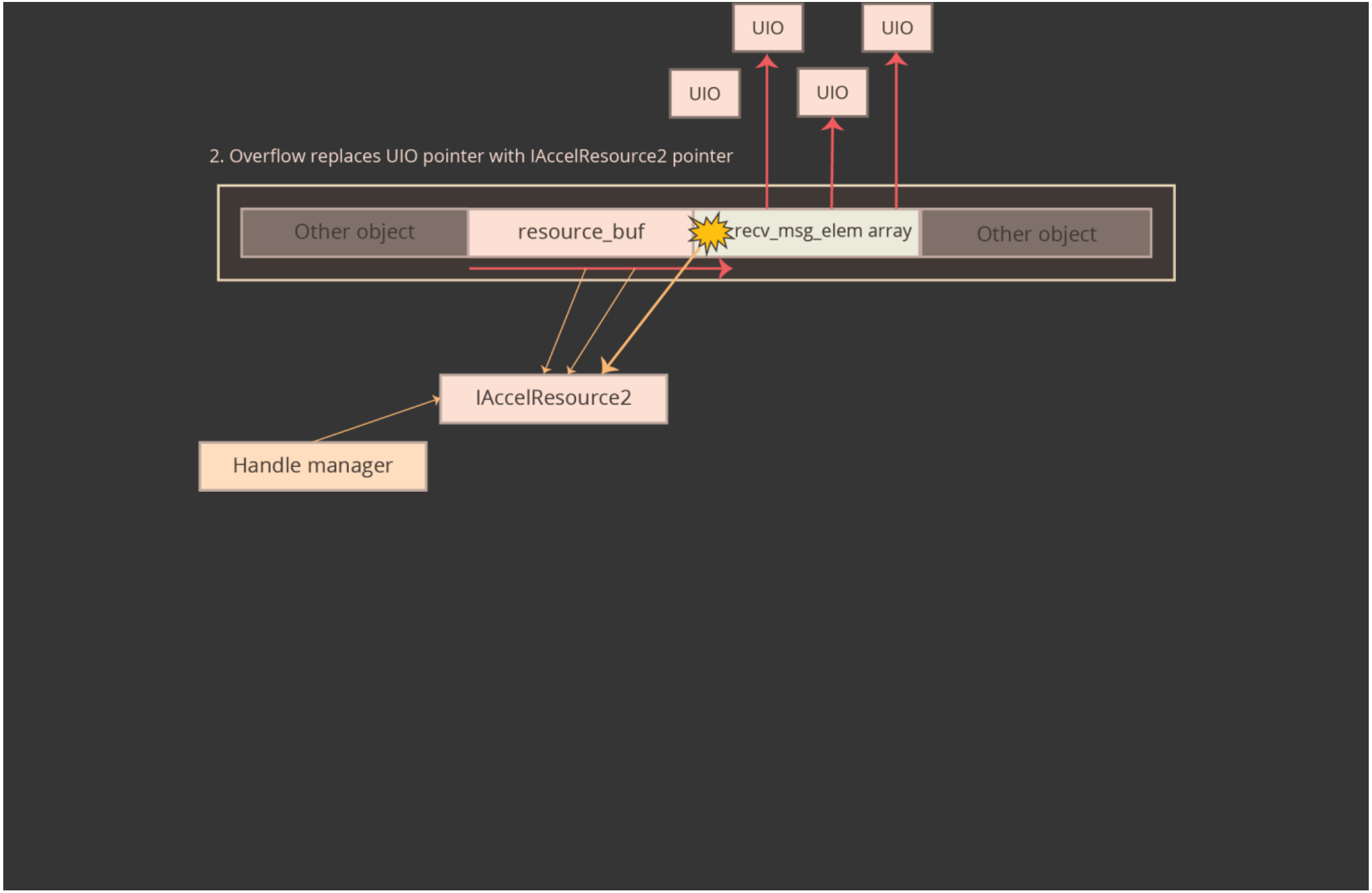
In this particular exploit, the exploit developer eventually settled on using a *recv_msg_elem* array as the victim object. This technique is not novel, but let's look at how it works.

An individual *recv_msg_elem* object is only 32-bytes long, but consecutive groups of them can be allocated simultaneously at the start of a *recvmsg_x* system call. By manipulating the arguments to the *recvmsg_x* system call, we can arrange for several of these to be allocated simultaneously, and therefore for the allocation as a whole to land in the *kalloc.4096* zone.

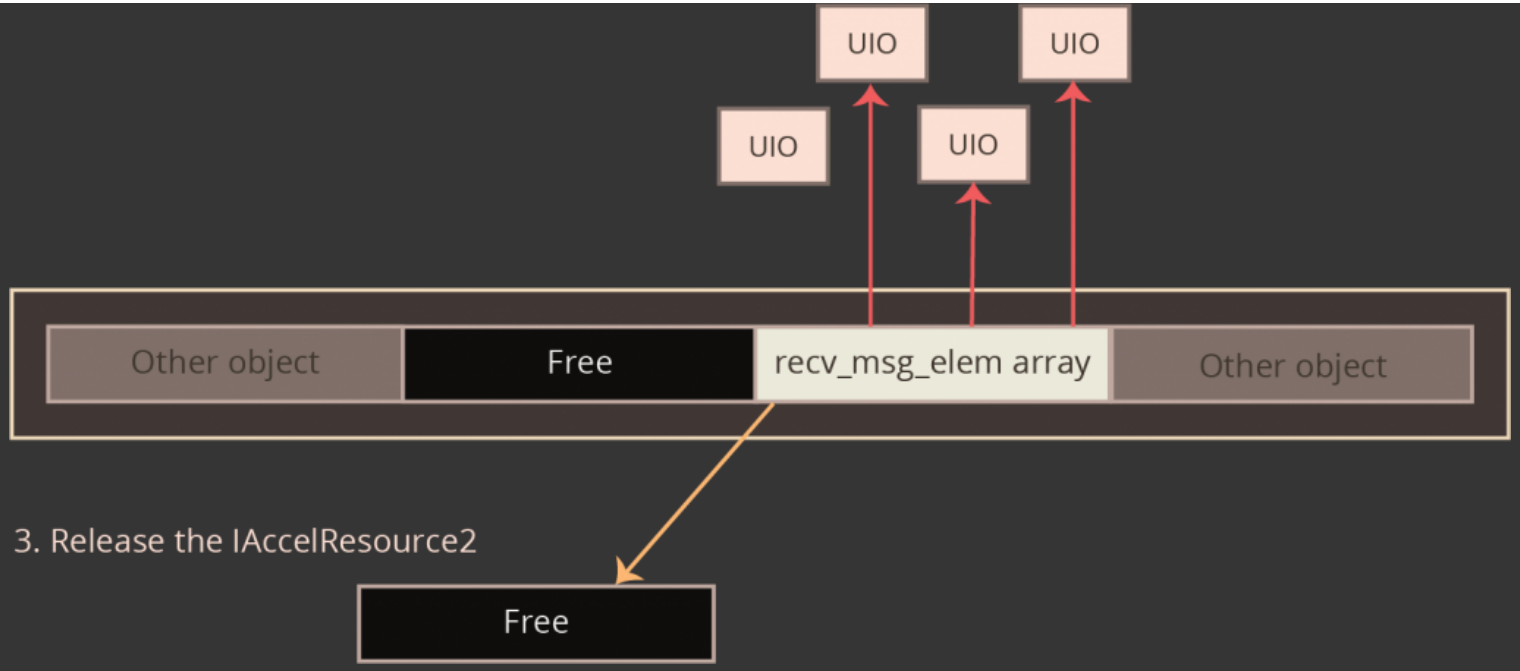
Each of the *recv_msg_elem*s within this array begin with a pointer to a *uio* structure. The structure itself isn't particularly interesting, except to say that it is allocated via *kalloc* based on a controllable length field, and eventually released by *kfree* when the *recv_msg_elem* array is deallocated at the end of the system call.



Create PDF in your applications with the Pdfcrowd HTML to PDF API
PDFCROWD



Create PDF in your applications with the Pdfcrowd HTML to PDF API
PDFCROWD



By choosing this *recv_msg_elems* array as our victim object, we satisfy all of the requirements that we needed and can turn the heap overflow into a use-after-free. Our sequence of operations is therefore as follows: first we arrange for a *recv_msg_elems* victim array to lie next to our overflowing *resources_buf* object on the *kalloc.4096* heap. Next we trigger the overflow **of** the *resources_buf*, which unexpectedly overwrites the first *recv_msg_elems* array item's *uio* pointer with a pointer to the *IOAccelResource* object.

Next, we free the IOAccelResource2 object and the resources buffer, but the pointer overwrite of the recv_msg_elem array lives on, continuing to point to where the IOAccelResource2 object was, which is now free. The victim ***recv_msg_elems*** object has no reason to believe its UIO object has been freed, but any use of it now leads to a *use-after-free*.

There is one final complexity in using an *recv_msg_elem* array as our victim object that the exploit developer needs to work around. This allocation occurs at the beginning of a *recvmsg_x* system call, but that system call blocks the thread waiting for messages to arrive. When the request completes or is cancelled, the *recvmsg_x* releases its internal victim object. This

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

means that while this is a great victim candidate, the system call will need to occur on a subordinate thread, since it will block the calling thread.

This is why, if you read Google's write-up, you will see that the exploit developer starts a series of threads, each one calling *recvmsg_x* with crafted parameters. The threads are not about race-conditions or performance; they are just about arranging for a single *recv_msg_elem* array to be allocated in the *kalloc.4096* zone in a way that gives the exploit developer control over the allocation's lifetime. Once allocated, the attacker's primary thread triggers the overflow, overwriting the *uio* pointer of the first *recv_msg_elem* array entry. The attacker can then release the IOAccelResource2 object, causing the first recv_msg_elem to suddenly and unexpectedly point to free data, and further use of that pointer will lead to a *use-after-free*.

In this post we discussed the strategy for how the exploit writer planned to turn the heap overflow into a use-after-free. In the next post I will discuss the *heap groom* used by the exploit developer to turn the theory of this strategy into practice. After that we can look at how the exploit developer makes use of the artificial use-after-free to build exploit primitives and take full control over the iOS kernel.

ARM Exploit Development

Writing ARM Shellcode

TCP Bind Shell (ARM 32-bit)

TCP Reverse Shell (ARM 32-bit)

Process Memory and Memory Corruption

Stack Overflow Challenges

Process Continuation Shellcode

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Introduction to Glibc Heap (malloc)

Introduction to Glibc Heap (free, bins)

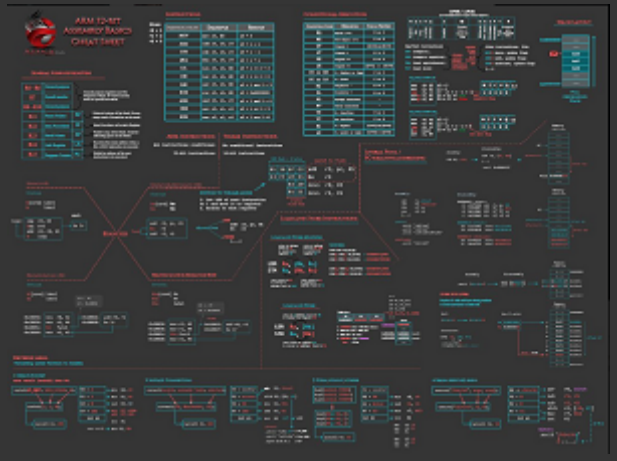
Part 1: Heap Exploit Development

Part 2 Heap Overflows and iOS Kernel

Twitter: @Fox0x01 and @azeria_labs

ARM Assembly Cheat Sheet

POSTER DIGITAL



Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

© 2017 Azeria-Labs

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD