Exploiting the iOS Kernel

Stefan Esser Cologne, Germany stefan@nopiracy.de

Version 1.0

July 13th, 2011

Abstract

In this paper we introduce the reader to an integer overflow vulnerability in the iOS kernel that is located in the setsockopt handler of the ndrv protocol. We discuss how this integer overflow leads to an exploitable heap buffer overflow and explain how the iOS kernel zone allocator can be exploited. Finally we will discuss the kernel heap feng shui technique used by the exploit.

Keywords: iOS kernel exploits, iOS, iphone, kernel exploitation, kernel heap feng shui

1 Introduction

Papers about iPhone exploitation have concentrated on the generation of sophisticated user land payloads that can be used to attack jailbroken and factory iPhones. Although the kernel locks down the user land very tightly previous research has not focused on exploiting the iOS kernel. Instead researchers have relied on the jailbreaking scene to provide them with working kernel exploits. The only material available focuses on Mac OS X kernel exploitation, which is similar because the iOS and Mac OS X kernel share a big amount of code, namely the XNU source tree. Mac OS X exploitation was discussed before by nemo in his Phrack papers[1] and within the kernel exploitation book by Perla and Oldani[2]. Therefore this paper discusses the exploitation of iOS kernel level vulnerabilities.

The remainder of this paper is organized as follows. In section 2 the integer multiplication overflow that powered the iOS 4.3.1-4.3.3 unterhered jailbreak is introduced. Section 3 then describes the properties of the kernel's heap zone allocator and how it can be exploited to achieve arbitrary code execution. In section 4 the heap feng shui exploitation technique is discussed and how it can be applied to zone allocator heap overflows. This paper ends with our conclusions in section 5.

This paper should be considered as work in progress. Therefore the latest version of this paper can be downloaded from http://www.suspekt.org/ios-kernel-exploitation-paper.

2 The Vulnerability

Before we discuss the exploitation of the kernel heap zone allocator we will first take a look at the vulnerability that is used to unterther the iOS 4.3.1-4.3.3 jailbreak. Because of the involvement of an heap buffer overflow the unterthering exploit is the most complex one used so far.

2.1 ndrv_setspec() Integer Multiplication Overflow Vulnerability

The following code is an extract of the ndrv_setspec() function that is defined in the file /bsd/net/ndrv.c. The function is called to handle socket options of type NDRV_SETDMXSPEC. Looking at the code in detail reveals that there are two integer overflows in it. One in lines 5-6 and one in lines 11-12. In both cases the user controlled demux_count is multiplied against the structure size in order to determine the amount of memory required. Of course very big values of demux_count will lead to an integer overflow and therefore only small amounts of memory are allocated.

```
bzero(&proto_param, sizeof(proto_param));
proto_param.demux_count = ndrvSpec.demux_count;
/* Allocate storage for demux array */
MALLOC(ndrvDemux, struct ndrv_demux_desc*, proto_param.demux_count *
           sizeof(struct ndrv_demux_desc), M_TEMP, M_WAITOK);
if (ndrvDemux ==
                      )
        return ENOMEM;
/* Allocate enough ifnet_demux_descs */
MALLOC(proto_param.demux_array, struct ifnet_demux_desc*
           sizeof(*proto_param.demux_array) * ndrvSpec.demux_count,
            M_TEMP, M_WAITOK);
if (proto_param.demux_array == NULL)
    error = ENOMEM;
if (error == 0)
ſ
         /* Copy the ndrv demux array from userland */
         error = copyin(user_addr, ndrvDemux,
                 ndrvSpec.demux_count * sizeof(struct ndrv_demux_desc));
         ndrvSpec.demux_list = ndrvDemux;
}
if (error == 0)
ſ
         /* At this point, we've at least got enough bytes to start looking around */
                         demuxOn = 0;
         u_int32_t
        proto_param.demux_count = ndrvSpec.demux_count;
        proto_param.input = ndrv_input;
proto_param.event = ndrv_event;
         for (demuxOn = 0; demuxOn < ndrvSpec.demux_count; demuxOn++)</pre>
                  /* Convert an ndrv_demux_desc to a ifnet_demux_desc */
                 error = ndrv_to_ifnet_demux(&ndrvSpec.demux_list[demuxOn],
                                                                             &proto_param.demux_array[demuxOn]);
                 if (error)
                          break:
         }
}
```

When looking at line 20-21 one can see that the call to copyin() contains the same integer overflow. Therefore it will not overflow the previously allocated buffer. The heap overflow we are going to exploit is hidden in the for loop between lines 34 and 41. The function ndrv_to_ifnet_demux() will convert elements of type ndrv_demux_desc into type ifnet_demux_desc as seen in the code below.

```
int
ndrv_to_ifnet_demux(struct ndrv_demux_desc* ndrv, struct ifnet_demux_desc* ifdemux)
{
    bzero(ifdemux, sizeof(*ifdemux));
    if (ndrv->type < DLIL_DESC_ETYPE2)
    {
        /* using old "type", not supported */
        return ENOTSUP;
    }
</pre>
```

 $\frac{3}{4}$

5 6 7

9 10

1

```
12
           if (ndrv->length > 28)
13
14
                return EINVAL;
15
           }
16
           ifdemux->type = ndrv->type;
ifdemux->data = ndrv->data.other;
17
18
19
           ifdemux->datalen = ndrv->length;
20
21
           return 0:
22
     }
```

We can see that the new ifnet_demux_desc struct is first zeroed out and then the fields type, data and datalen are set from the user supplied ndrv_demux_desc struct. This means we can partially control the written data and get a pointer to user supplied data written into the new data structure for free. This will be very handy when the exploit is written. However the tricky part of the exploit is not causing the overflow. This is not really hard, but in order to control the exploit we need to trigger the function two times and make it reuse the same allocated memory. Otherwise we cannot control the data we are overflowing with and stop the exploit from crashing. However before we can go into the details of exploiting heap overflows in the iOS kernel we have to first look at the _MALLOC() function which is what hides behind the MALLOC() macro.

3 Kernel Heap Zone Allocator

In the following sections the kernel zone heap allocator functions that are involved in the exploitation of this vulnerability are introduced.

3.1 _MALLOC()

In order to understand how heap exploitation works it is required to dig into the memory allocation functions used and to analyze how memory is handled internally. In case of the XNU kernel _MALLOC() is just a wrapper around the kalloc() function.

```
void *
_MALLOC(
            size_t
                                    size,
            int
                                    type,
                                    flags)
            int
{
            MDECL (size)
                                    *mem:
                                    nemsize
                                               = sizeof (*mem):
            size_t
            if (type >= M_LAST)
                        panic("_malloc TYPE");
            if (size == 0)
                        return (NULL):
            if (flags & M_NOWAIT) {
                        mem = (void *)kalloc_noblock(memsize);
            } else {
                        mem = (void *)kalloc(memsize);
                        if (mem == NULL) {
                                     * We get here when the caller told us to block waiting for memory, but
* kalloc said there's no memory left to get. Generally, this means there's a
                                       leak or the caller asked for an impossibly large amount of memory. Since there's
nothing left to wait for and the caller isn't expecting a NULL return code, we
just panic. This is less than ideal, but returning NULL doesn't help since the
                                         majority of callers don't check the return value and will just dereference the pointer and trap anyway. We may as well get a more descriptive message out while we can.
                                        trap anyway.
                                    panic("_MALLOC: kalloc returned NULL (potential leak), size %llu", (uint64_t) size);
                       }
           }
            if (!mem)
                        return (0);
```

 $\frac{1}{2}$

3

 $\frac{4}{5}$

6 7 8

9 10

11

12

 $13 \\ 14$

15

16

17

18

```
mem->hdr.mlen = memsize:
            if (flags & M_ZERO)
                     bzero(mem->hdr.dat, size);
             return (mem->hdr.dat):
45
    }
```

39

40 $\frac{41}{42}$

4344

2

3

10

11 1213

14 15 16

1718

19 20

 21 22

23

 $\frac{24}{25}$

26

 $\frac{27}{28}$

29

30

31

32

_MALLOC() more or less ignores the type parameter. Only unexpected values above M_LAST will result in a kernel panic. However this should never happen on a normal system unless there is broken kernel code. The second interesting fact about this function is that a request for zero bytes of memory will result in a NULL pointer being returned. Depending on the flags the call will be forwarded to the blocking or non-blocking version of kalloc() and if requested the data is zeroed out. The only notable thing about this function is that it will actually request a few bytes more than requested because it prefixes the data with a small header that remembers the original requested size. Next in line is the kalloc() function.

3.2kalloc()

The kalloc() function will check the requested size and in case it is large it will forward the call to kmem_alloc() otherwise the size will be used to detect the kernel memory zone it should be allocated in and then the call is forwarded to the zone allocator zalloc(). The kmem_alloc() function and its internals are not covered in this paper and might be discussed in a future paper. For now we only look into the zone allocator zalloc_canblock().

```
void *
kalloc_canblock(
                   vm size t
                                      size
                   boolean_t
                                      canblock)
ł
         register int zindex;
         register vm_size_t allocsize;
vm_map_t alloc_map = VM_MAP_NULL;
          * If size is too large for a zone, then use kmem_alloc.
* (We use kmem_alloc instead of kmem_alloc_kobject so that
           * krealloc can use kmem_realloc.)
           */
         if (size >= kalloc_max_prerounded) {
         3
          /* compute the size of the block that we will actually allocate */
         allocsize = KALLOC MINSIZE:
         zindex = first_k_zone;
         while (allocsize < size) {</pre>
                   allocsize <<= 1;
                   zindex++:
         }
          /* allocate from the appropriate zone */
          assert(allocsize < kalloc_max);
         return(zalloc_canblock(k_zone[zindex], canblock));
}
```

3.3**Zone Allocator Internals**

The kernel zone allocator organizes kernel memory in so called zones. A list of available zones can be viewed with the /usr/bin/zprint utility that ships with Mac OS X. Memory blocks within a zone are all of the same size. For most kernel objects there is a separate zone like socket, vnodes and kernel_stacks. But some of the zones are shared for multiple purposes. For example kalloc() defines multiple zones for different powers of two. They are called kalloc.16-kalloc.8192. Internally the information about all available zones is also stored within a zone. A single zone is described by

the following struct.

 $\frac{1}{2}$

1	struct	zone {		
2		int	count;	/* Number of elements used now */
3		vm_offset_t	free_elements;	
4		decl_lck_mtx_da	ta(,lock)	/* zone lock */
5		lck_mtx_ext_t	lock_ext;	/* placeholder for indirect mutex */
6		lck_attr_t	lock_attr;	/* zone lock attribute */
7		lck_grp_t	lock_grp;	/* zone lock group */
8		lck_grp_attr_t	lock_grp_attr;	/* zone lock group attribute */
9		vm_size_t	cur_size;	/* current memory utilization */
10		vm_size_t	max_size;	/* how large can this zone grow */
11		vm_size_t	elem_size;	/* size of an element */
12		vm_size_t	alloc_size;	/* size used for more memory */
13		unsigned int		
14		/* boolean_t */	exhaustible :1,	, /* (F) merely return if empty? */
15		/* boolean_t */	collectable :1,	, /* (F) garbage collect empty pages */
16		/* boolean_t */	expandable :1,	/* (T) expand zone (with message)? */
17		/* boolean_t */	allows_foreign	:1,/* (F) allow non-zalloc space */
18		/* boolean_t */	doing_alloc :1,	, /* is zone expanding now? */
19		/* boolean_t */	waiting :1,	/* is thread waiting for expansion? */
20		/* boolean_t */	async_pending :	:1, /* asynchronous allocation pending? */
21		/* boolean_t */	doing_gc :1,	/* garbage collect in progress? */
22		/* boolean_t */	noencrypt :1;	
23		<pre>struct zone *</pre>	next_zone;	/* Link for all-zones list */
24		call_entry_data	_t call_as	sync_alloc; /* callout for asynchronous alloc */
25		const char	*zone_name;	/* a name for the zone */
26	#if	ZONE_DEBUG		
27		queue_head_t	active_zones;	/* active elements */
28	#endif	/* ZONE_DEBUG *	/	
29	}:			

Within this struct it is visible that the zone struct contains information about the element size. the current number of used elements and the current size and maximum size of the overall zone. One can see that all the zones are kept in a linked list. It is also visible that zones can support garbage collection or can expand dynamically. It is also guessable from the structure that all the free elements of a zone are kept in a linked list starting at free_elements. The first bytes of each free memory block contain a pointer to the next free element. When memory is allocated the first element from the linked list will be used and only if the freelist is empty the zone will be enlarged. When a page is added to the freelist or when the zone is initially created the new memory blocks are put on the freelist one after each other. Therefore the first allocated memory block will be at the end of the page, not in the beginning.

When zalloc() takes an element from the freelist this is done by using the REMOVE_FROM_ZONE macro. The macro reads the pointer to the next element in the freelist and marks it as the new head of the freelist.



The macro contains code that checks the freelist for kernel heap corruption. However this code is only active if it is activated by supplying boot arguments when booting the kernel. So on a default iPhone this code is never active and we do not have to care about it. Therefore overflowing an already freed memory block allows overwriting the pointer to the next element of the freelist. This means if zalloc() reallocated the overwritten memory block the head of the freelist will point anywhere of attackers choice. The next allocation within the zone will therefore return a pointer controlled by the attacker. Data written to that pointer will therefore allow overwriting arbitrary kernel memory with more or less arbitrary data. In our exploit we used this technique to overwrite an unused part of the system call table. For the iOS jailbreak the syscall 207 is redirected to a BX R1 instruction in the kernel which will pass execution to the memory containing the syscall arguments. Because all kernel memory is executable this allows arbitrary code execution.

4 The Art of Kernel Heap Feng Shui

Heap Feng Shui is a term invented by Sotirov[3] that describes the art of carefully manipulating the heap of an application in order to control the layout of the heap. The goal of this technique is to get the heap into a known state so that the exploitation of heap buffer overflow vulnerabilities is possible. In order to apply this technique it is necessary to find heap manipulation primitives that allow the allocation and deallocation of arbitrary amounts of kernel memory. In the case of the iOS kernel zone allocator it is even possible to request information about the current state of the internal zones from the kernel. While this is not a requirement for the kernel heap feng shui technique it allows to create more reliable exploits.

4.1 Allocation Primitive

Within the iOS kernel there are many functions that allow to allocate memory from the kmalloc kernel zones, but in most cases the memory is allocated in the beginning of a function and released at the end of the function. For the exploitation of the ndrv_setspec() integer overflow it is however possible to find the necessary allocation primitive in the ndrv_connect() function as seen below. This function is called when a socket of the ndrv protocol is connected.

```
static int
    ndrv_connect(struct socket *so, struct sockaddr *nam, __unused struct proc *p)
2
 3
 4
            struct ndrv_cb *np = sotondrvcb(so);
 5
 6
            if (np == 0)
 7\\8
                   return EINVAL;
 9
            if (np->nd_faddr)
10
                   return EISCONN;
11
12
            /* Allocate memory to store the remote address */
            13
14
            if (np->nd_faddr ==
15
                   return ENOMEM:
16
17
18
            bcopy((caddr_t) nam, (caddr_t) np->nd_faddr, nam->sa_len);
19
            soisconnected(so):
20
            return 0;
21
    }
```

By adjusting the sa_len field in the sockaddr structure that is used for connecting the ndrv socket it is possible to allocate arbitrary memory blocks of up to 255 bytes. It is also important to keep in mind that the number of blocks we can allocate is limited by the number of possible file descriptors per process. For our purposes this is good enough, because we can raise that limit.

4.2 Deallocation Primitive

The corresponding deallocation primitive is hidden in the ndrv_do_disconnect() function that is called when the socket is disconnected again. Therefore in order to free a previously allocated block of memory it is enough to simply call close().

```
static int
2
     ndrv_do_disconnect(struct ndrv_cb *np)
3
     {
 4
              struct socket * so = np->nd_socket;
\frac{5}{6}
     #if NDRV DEBUG
              kprintf("NDRV disconnect: %x\n", np);
     #endif
7
8
9
              if (np->nd_faddr)
10
              FREE(np->nd_faddr, M_IFADDR);
11
                       np - > nd_faddr = 0;
12
              if (so->so_state & SS_NOFDREF)
13
              ndrv_do_detach(np);
soisdisconnected(so);
14
15
              return(0);
16
     }
17
```

4.3 Detecting the State of the Kernel Heap

The mach subsystem of the iOS kernel comes with a pretty interesting and useful function called host_zone_info() that can be used to query the current state of all the zones allocated by the kernel's zone allocator. This function is e.g. used internally by the /usr/bin/zprint utility that ships with Mac OS X. For every zone allocated by the zone allocator a zone_info struct is returned that contains the following information.

```
typedef struct zone info {
 1
 2
                integer_t
                                    zi_count;
                                                         /* Number of elements used now */
                                                         /* current memory utilization */
/* how large can this zone grow */
/* size of an element */
 3
                                    zi_cur_size;
                vm_size_t
 4
                vm size t
                                    zi max size:
 5
                vm_size_t
                                     zi_elem_size;
                vm_size_t
 6
                                    zi_alloc_size;
                                                         /* size used for more memory */
                                                         /* zone pageable? */
 7
                integer t
                                    zi pageable:
 8
                integer_t
                                     zi_sleepable;
                                                         /* sleep if empty? */
                                    zi_exhaustible; /* merely return if empty? */
zi_collectable; /* garbage collect elements? */
 9
                integer_t
10
                integer_t
11
     } zone_info_t;
```

The information that can be retrieved from the kernel does not leak any kernel memory addresses, but allows an insight into the current state of the kernel heap. The field **zi_count** contains the number of currently allocated memory blocks in the zone. By subtracting this number from the maximum number of elements at the moment **zi_cur_size** / **zi_elem_size** it is possible to see how many empty blocks are somewhere within the zone. When the current number of elements is exhausted the zone will be grown by **zi_alloc_size** bytes. These numbers help a lot when it comes to control the state of the kernel heap.

4.4 Heap Feng Shui

The technique called heap feng shui is a multi-step process that first tries to solve the problem of kernel heap fragmentation. In order to do that the heap allocation primitive is used to fill the holes in the heap until all of them are closed. Because the state of the heap at time of exploitation is usually unknown and partly random it is normally not possible to know how many holes there are and when they are all closed. Therefore the number of required allocations are usually unknown. In the case of the iOS kernel this is however different, because the host_zone_info() function can be used to determine the current size of a kernel zone and the number of blocks currently allocated inside that

zone. Because of this the exploit can calculate how many allocations are required to fill all the holes. Because it is possible that other kernel operations allocated or freed memory in the meanwhile, the state of the heap should be queried again and in case of unexpected values the process should be repeated.

The second step of heap feng shui is to allocate a number of adjacent memory blocks. This step is normally combined with the first step, because a high number of memory blocks is already allocated in order to be sure to fill all the holes in the kernel heap. In the case of iOS it is however a bit easier because repeatedly calling host_zone_info() allows us to know exactly when a number of allocations have been allocated adjacent of each other. We also know exactly where a certain allocation is within a memory page due to that function.

The last step of heap feng shui uses the deallocation primitive to free memory blocks of our choice. This allows to force the next allocation of a specific size to be in a specific controlled place. In case of a heap buffer overflow it is therefore possible to control the block the overflow runs into.

5 Conclusion

In this paper we revealed the integer overflow vulnerability in the iOS kernel that drives the iOS 4.3.1-4.3.3 unterhered jailbreak. This vulnerability is already closed in current beta versions of iOS 5.0 and is expected to be closed in the soon arriving iOS 4.3.4. Therefore it was publicly discussed in this paper for the first time in detail. The remaining chapters of this paper discussed the iOS zone allocator and its internal data structures. It is shown how overwriting the header of already freed memory blocks can be exploited and result in arbitrary code execution. The paper closed with an explanation of the heap feng shui exploitation technique and how it is applied to the iOS kernel zone allocator overflow we are exploiting.

Future research into iOS kernel exploitation should take other kernel heap memory allocators like the kmem allocator into consideration.

References

- [1] Nemo, "Phrack 64 Mac OS X wars a XNU Hope", 2008, http://phrack.org/issues.html?issue=64id=11article
- [2] E. Perla, M. Oldani, "A Guide To Kernel Exploitation Attacking the Core", 2010, http://www.attackingthecore.com/
- [3] A. Sotirov, "Heap Feng Shui in Javascript, BlackHat Europe", 2007 http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirovapr19.pdf