SektionEins
http://www.sektioneins.de

Tales from iOS 6 Exploitation

and iOS 7 Security Changes

Stefan Esser <stefan.esser@sektioneins.de>

Who am I?

Stefan Esser

- from Cologne / Germany
- in information security since 1998
- PHP core developer since 2001
- Month of PHP Bugs and Suhosin
- recently focused on iPhone security (ASLR, kernel, jailbreak)
- Head of Research and Development at SektionEins GmbH

What is this talk about?

- the posix_spawn() vulnerability
- and how it turned out to be more than an information leak
- various iOS 7 changes with an influence on security

Part I

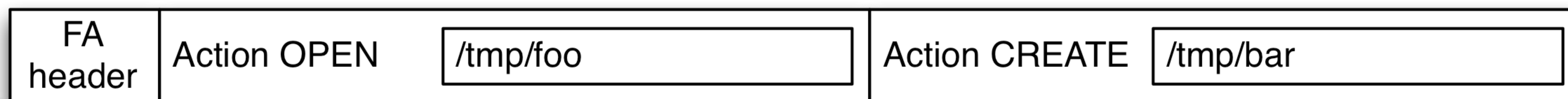
posix_spawn() - The info leak that was more…

posix_spawn() and the SyScan Garage Sale

- bunch of vulnerabilities were dropped at SyScan Singapore 2013
- the posix_spawn() vulnerability was one of them
- posix_spawn() is a more powerful way to spawn/execute processes
- vulnerability was declared a kernel heap information leak

posix_spawn() File Actions

- file actions allow parent to open, close or clone file descriptors for the child
- each action is defined in a structure about 1040 bytes in size
- prefixed by a small header




```c
typedef struct _psfa_action {
    psfa_t  psfaa_type;             /* file action type */
    int psfaa_filedes;              /* fd to operate on */
    struct _psfaa_open {
        int psfao_oflag;            /* open flags to use */
        mode_t  psfao_mode;         /* mode for open */
        char    psfao_path[PATH_MAX];   /* path to open */
    } psfaa_openargs;
} _psfa_action_t;
```

```c
typedef enum {
    PSFA_OPEN = 0,
    PSFA_CLOSE = 1,
    PSFA_DUP2 = 2,
    PSFA_INHERIT = 3
} psfa_t;
```


SektionEins

posix_spawn() File Actions

- data describing the actions is copied into the kernel after user supplied size is checked against upper and lower bounds

```
if (px_args.file_actions_size != 0) {
  /* Limit file_actions to allowed number of open files */
  int maxfa = (p->p_limit ? p->p_rlimit[RLIMIT_NOFILE].rlim_cur : NOFILE);
  if (px_args.file_actions_size < PSF_ACTIONS_SIZE(1) ||
    px_args.file_actions_size > PSF_ACTIONS_SIZE(maxfa)) {
    error = EINVAL;
    goto bad;
  }
  MALLOC(px_sfap, _posix_spawn_file_actions_t, px_args.file_actions_size, M_TEMP, M_WAITOK)
  if (px_sfap == NULL) {
    error = ENOMEM;
    goto bad;
  }
  imgp->ip_px_sfa = px_sfap;

  if ((error = copyin(px_args.file_actions, px_sfap,
          px_args.file_actions_size)) != 0)
    goto bad;
}
```

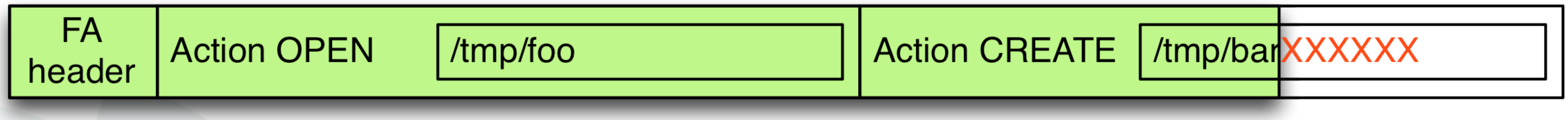
posix_spawn() File Actions Incomplete Verification

- check against upper and lower bound is insufficient
- because of a file action count inside the data that is trusted
- it is never validated that the supplied data is enough for the count
- loop over data can therefore read outside the buffer which might crash

```
static int
exec_handle_file_actions(struct image_params *imgp, short psa_flags)
{
  int error = 0;
  int action;
  proc_t p = vfs_context_proc(imgp->ip_vfs_context);
  _posix_spawn_file_actions_t px_sfap = imgp->ip_px_sfa;
  int ival[2];    /* dummy retval for system calls) */

  for (action = 0; action < px_sfap->psfa_act_count; action++) {
    _psfa_action_t *psfa = &px_sfap->psfa_act_acts[ action];

    switch(psfa->psfaa_type) {
        case PSFA_OPEN: {
```

posix_spawn() File Actions Information Leak

- by carefully crafting the data (and its size) it is possible to leak bytes from the kernel heap with a **PSFA_OPEN** file action
- choose size in a way that the beginning of the filename is from within the buffer and the end of the filename is taken from the kernel heap after it



- with **fcntl(F_GETPATH)** it is then possible to retrieve the leaked bytes

Only an Information Leak?

Only an information leak?

- questions came up on Twitter if **posix_spawn** is more than an information leak
- to be more than an information leak we need a write outside the buffer
- we need to check if there is any write in **exec_handle_file_actions()** function
- and if we can abuse it
- let's read more carefully ...

Structure of exec_handle_file_actions

- function consists of two loops
- with an error condition exit in-between
- both loops implement a switch statement for the cases
  - **PSFA_OPEN**
  - **PSFA_DUP2**
  - **PSFA_CLOSE**
  - **PSFA_INHERIT**
- let's check all cases ...

PSFA_OPEN (I)

- no write in first part of PSFA_OPEN in first loop

```
case PSFA_OPEN: {
    /*
     * Open is different, in that it requires the use of
     * a path argument, which is normally copied in from
     * user space; because of this, we have to support an
     * open from kernel space that passes an address space
     * context of UIO_SYSSPACE, and casts the address
     * argument to a user_addr_t.
     */
    struct vnode_attr va;
    struct nameidata nd;
    int mode = psfa->psfaa_openargs.psfao_mode;
    struct dup2_args dup2a;
    struct close_nocancel_args ca;
    int origfd;

    VATTR_INIT(&va);
    /* Mask off all but regular access permissions */
    mode = ((mode &~ p->p_fd->fd_cmask) & ALLPERMS) & ~S_ISTXT;
    VATTR_SET(&va, va_mode, mode & ACCESSPERMS);

    NDINIT(&nd, LOOKUP, OP_OPEN, FOLLOW | AUDITVNPATH1, UIO_SYSSPACE,
           CAST_USER_ADDR_T(psfa->psfaa_openargs.psfao_path),
           imgp->ip_vfs_context);

    error = open1(imgp->ip_vfs_context,
            &nd,
            psfa->psfaa_openargs.psfao_oflag,
            &va,
            ival);

    }
```

PSFA_OPEN (II)

- no write in second part of PSFA_OPEN in first loop

```
if (error || ival[0] == psfa->psfaa_filedes)
     break;

 origfd = ival[0];
 /*
  * If we didn't fall out from an error, we ended up
  * with the wrong fd; so now we've got to try to dup2
  * it to the right one.
  */
 dup2a.from = origfd;
 dup2a.to = psfa->psfaa_filedes;

 /*
  * The dup2() system call implementation sets
  * ival to newfd in the success case, but we
  * can ignore that, since if we didn't get the
  * fd we wanted, the error will stop us.
  */
 error = dup2(p, &dup2a, ival);
 if (error)
     break;

 /*
  * Finally, close the original fd.
  */
 ca.fd = origfd;

 error = close_nocancel(p, &ca, ival);
 }
 break;
```

PSFA_DUP2 (III)

- no write in PSFA_DUP2 in first loop

```
case PSFA_DUP2: {
     struct dup2_args dup2a;

     dup2a.from = psfa->psfaa_filedes;
     dup2a.to = psfa->psfaa_openargs.psfao_oflag;

     /*
      * The dup2() system call implementation sets
      * ival to newfd in the success case, but we
      * can ignore that, since if we didn't get the
      * fd we wanted, the error will stop us.
      */
     error = dup2(p, &dup2a, ival);
     }
     break;
```

PSFA_CLOSE

- no write in PSFA_CLOSE in first loop

```
case PSFA_CLOSE: {
    struct close_nocancel_args ca;

    ca.fd = psfa->psfaa_filedes;

    error = close_nocancel(p, &ca, ival);
    }
    break;
```

PSFA_INHERIT

- we found a write in PSFA_INHERIT
- **but can we make it write outside of our or another buffer?**

```
case PSFA_INHERIT: {
    struct fileproc *fp;
    int fd = psfa->psfaa_filedes;

    /*
     * Check to see if the descriptor exists, and
     * ensure it's -not- marked as close-on-exec.
     * [Less code than the equivalent F_GETFD/F_SETFD.]
     */
    proc_fdlock(p);
    if ((error = fp_lookup(p, fd, &fp, 1)) == 0) {
        *fdflags(p, fd) &= ~UF_EXCLOSE;
        (void) fp_drop(p, fd, fp, 1);
    }
    proc_fdunlock(p);
    }
    break;
```

This is a write in form of a binary AND

What is the macro fdflags()?

- **fdflags** addresses an element in the current processes' **fd_ofileflags** structure
- write position depends on supplied file descriptor **fd**
- we need to check what and how big **fd_ofileflags** is
- then we can see if we can make it write outside that buffer

```
#define     fdflags(p, fd)                      \
            (&(p)->p_fd->fd_ofileflags[(fd)])
```

The filedesc struct

- **fd_ofileflags** is actually a byte array
- now we check where it points to our how it is allocated

```
struct filedesc {
    struct  fileproc **fd_ofiles;   /* file structures for open files */
    char    *fd_ofileflags;     /* per-process open file flags */
    struct  vnode *fd_cdir;     /* current directory */
    struct  vnode *fd_rdir;     /* root directory */
    int fd_nfiles;      /* number of open files allocated */
    int fd_lastfile;        /* high-water mark of fd_ofiles */
    int fd_freefile;        /* approx. next free file */
    u_short fd_cmask;       /* mask for file creation */
    uint32_t    fd_refcnt;      /* reference count */

    int     fd_knlistsize;          /* size of knlist */
    struct  klist *fd_knlist;       /* list of attached knotes */
    u_long  fd_knhashmask;          /* size of knhash */
    struct  klist *fd_knhash;       /* hash table for attached knotes */
        int fd_flags;
};
```

Where does fd_ofileflags come from?

- **fd_ofileflags** is actually not the start of an allocated memory block
- first allocation of **fd_ofiles** as 5 bytes times current max file descriptor
- then **fd_ofileflags** set to point to the last „current max file descriptor" bytes

```
MALLOC_ZONE(newofiles, struct fileproc **,
        numfiles * OFILESIZE, M_OFILETABL, M_WAITOK);
proc_fdlock(p);
if (newofiles == NULL) {
    return (ENOMEM);
}
if (fdp->fd_nfiles >= numfiles) {
    FREE_ZONE(newofiles, numfiles * OFILESIZE, M_OFILETABL);
    continue;
}
newofileflags = (char *) &newofiles[numfiles];

...

ofiles = fdp->fd_ofiles;
fdp->fd_ofiles = newofiles;
fdp->fd_ofileflags = newofileflags;
fdp->fd_nfiles = numfiles;
FREE_ZONE(ofiles, oldnfiles * OFILESIZE, M_OFILETABL);
```

What do we know so far?

- **fd_ofileflags** is not start of a buffer but points into the middle of one
- buffer it points to is allocated with **MALLOC_ZONE()**
- in case of dynamic buffers **MALLOC_ZONE()** is identical to **kalloc()**
- and finally the length of **fd_ofileflags** is „current max filedescriptors“ bytes

- to write outside of that buffer we need to pass illegal file descriptor to **fdflags**

PSFA_INHERIT and illegal file descriptors?

- in **PSFA_INHERIT** passed **fd** is verified by **fp_loopkup**
- so we cannot pass an illegal **fd** to **fdflags** here

```
case PSFA_INHERIT: {
    struct fileproc *fp;
    int fd = psfa->psfaa_filedes;

    /*
     * Check to see if the descriptor exists, and
     * ensure it's -not- marked as close-on-exec.
     * [Less code than the equivalent F_GETFD/F_SETFD.]
     */
    proc_fdlock(p);
    if ((error = fp_lookup(p, fd, &fp, 1)) == 0) {
        *fdflags(p, fd) &= ~UF_EXCLOSE;
        (void) fp_drop(p, fd, fp, 1);
    }
    proc_fdunlock(p);
    }
    break;
```

fp_lookup will ensure only valid fd pass

Is there a write in the second loop?

- second loop also contains an **fdflags** write (binary OR)
- and **fd** is either filled from **psfaa_filedes** or **psfaa_openargs.psfao_oflag**
- **both these variables are checked to only contain valid fd in first loop**

```
proc_fdlock(p);
for (action = 0; action < px_sfap->psfa_act_count; action++) {
    _psfa_action_t *psfa = &px_sfap->psfa_act_acts[action];
    int fd = psfa->psfaa_filedes;

    switch (psfa->psfaa_type) {
    case PSFA_DUP2:
        fd = psfa->psfaa_openargs.psfao_oflag;
        /*FALLTHROUGH*/
    case PSFA_OPEN:
    case PSFA_INHERIT:
        *fdflags(p, fd) |= UF_INHERIT;
        break;

    case PSFA_CLOSE:
        break;
    }
}
proc_fdunlock(p);
```

both validated in loop 1

another potential write

Vulnerable or Not?

- so is this code vulnerable or not?
- in both cases the file descriptors passed to **fdflags** are verified

- ... but can you spot an important difference in both verifications?

Write One

- for write one the **fd** is read from memory
- then verified
- and then used for the write

```
case PSFA_INHERIT: {
    struct fileproc *fp;
    int fd = psfa->psfaa_filedes;

    /*
     * Check to see if the descriptor exists, and
     * ensure it's -not- marked as close-on-exec.
     * [Less code than the equivalent F_GETFD/F_SETFD.]
     */
    proc_fdlock(p);
    if ((error = fp_lookup(p, fd, &fp, 1)) == 0) {
        *fdflags(p, fd) &= ~UF_EXCLOSE;
        (void) fp_drop(p, fd, fp, 1);
    }
    proc_fdunlock(p);
    }
    break;
```

read from memory

verification

write

Write Two

- in the second loop the used **fd** is read from memory
- and then used
- no check in second loop because it relies on check of first loop

```
proc_fdlock(p);
for (action = 0; action < px_sfap->psfa_act_count; action++) {
    _psfa_action_t *psfa = &px_sfap->psfa_act_acts[action];
    int fd = psfa->psfaa_filedes;

    switch (psfa->psfaa_type) {
    case PSFA_DUP2:
        fd = psfa->psfaa_openargs.psfao_oflag;
        /*FALLTHROUGH*/
    case PSFA_OPEN:
    case PSFA_INHERIT:
        *fdflags(p, fd) |= UF_INHERIT;
        break;

    case PSFA_CLOSE:
        break;
    }
}
proc_fdunlock(p);
```

read from memory

write

Difference in Writes: TOCTOU

- the obvious difference between the writes is the **TOCTOU (Time Of Check Time To Use)**
- for write two the final re-read is happening **AFTER verification**
- for write one the read is happening **BEFORE verification**

Write One	**Write Two**
READ FROM MEMORY	READ FROM MEMORY
VERIFICATION	VERIFICATION
WRITE	...
	RE-READ FROM MEMORY
	WRITE

Is difference in TOCTOU a vulnerability here?

- **Re-phrasing:**
  Is it possible for the memory containing the **fd** to change between **TOCTOU**?

- **Under normal circumstances:**
  The **fd** is read from memory only this kernel thread has access to.
  It does not change the value in-between so no **TOCTOU** problem.

- **But we are not in a normal situation:**
  We have a vuln that allows file actions to be read from outside the buffer.
  Anything outside buffer can be modified at any time by another kernel thread.

  => this is a TOCTOU / race condition vulnerability

Winning the Race?

Winning the Race?

- the race condition can only be exploited if we manage to change the memory between verification and re-read
- so we need a second thread to do the modification at the right moment
- we need to have good syncing and be fast enough to change between check in loop 1 and usage in loop 2
- whenever possible we try to slow down the vulnerable kernel thread to enlarge the window of opportunity

Write Two

READ FROM MEMORY

VERIFICATION

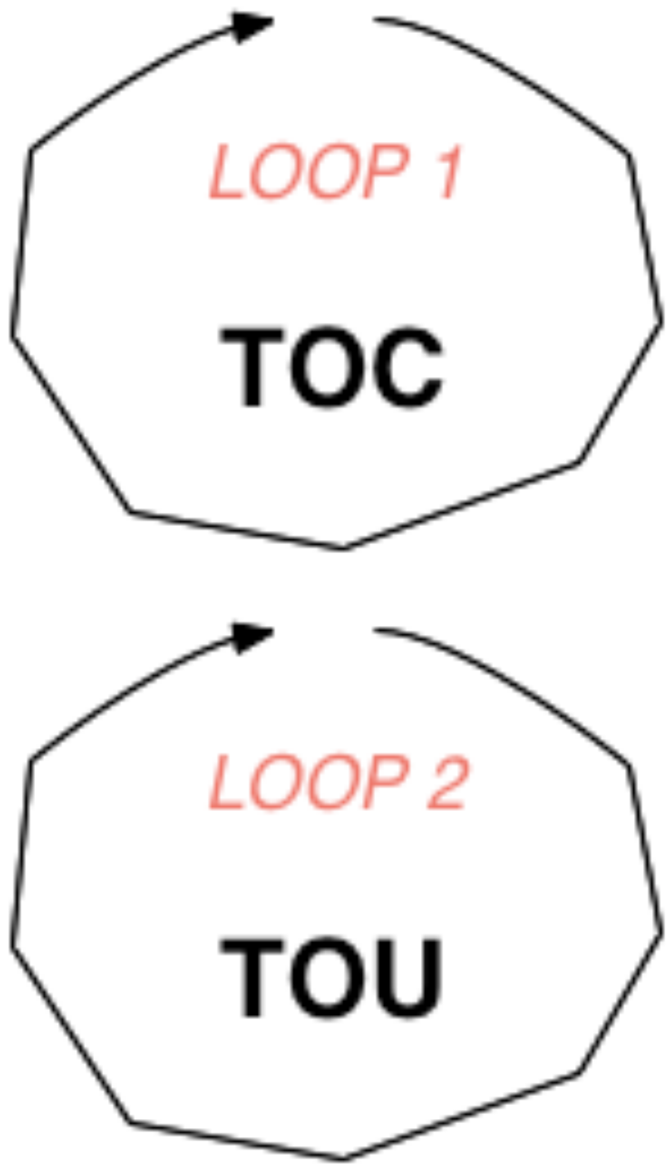
...

RE-READ FROM MEMORY

WRITE

Slowing down exec_handle_file_actions()?

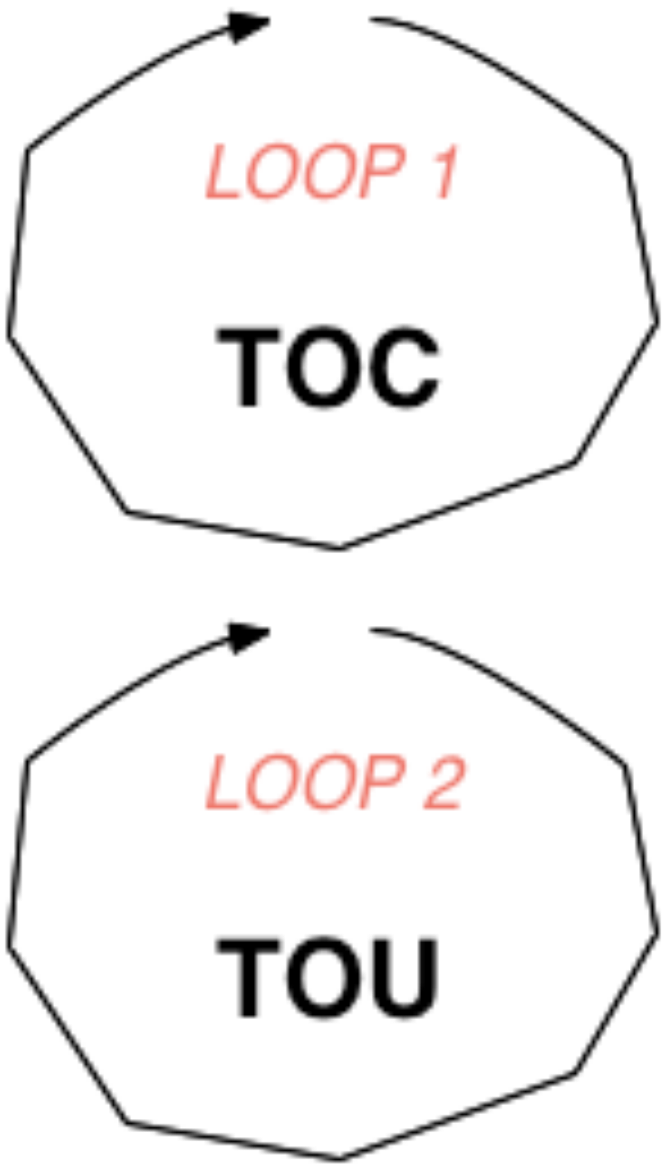
- slowing down a loop can be done by either
    - increasing the iterations of the loop
      = increasing number of file actions
    - slowing down operations inside the loop
      = slowing down **open() / dup2() / close()**



Increasing number of file actions?

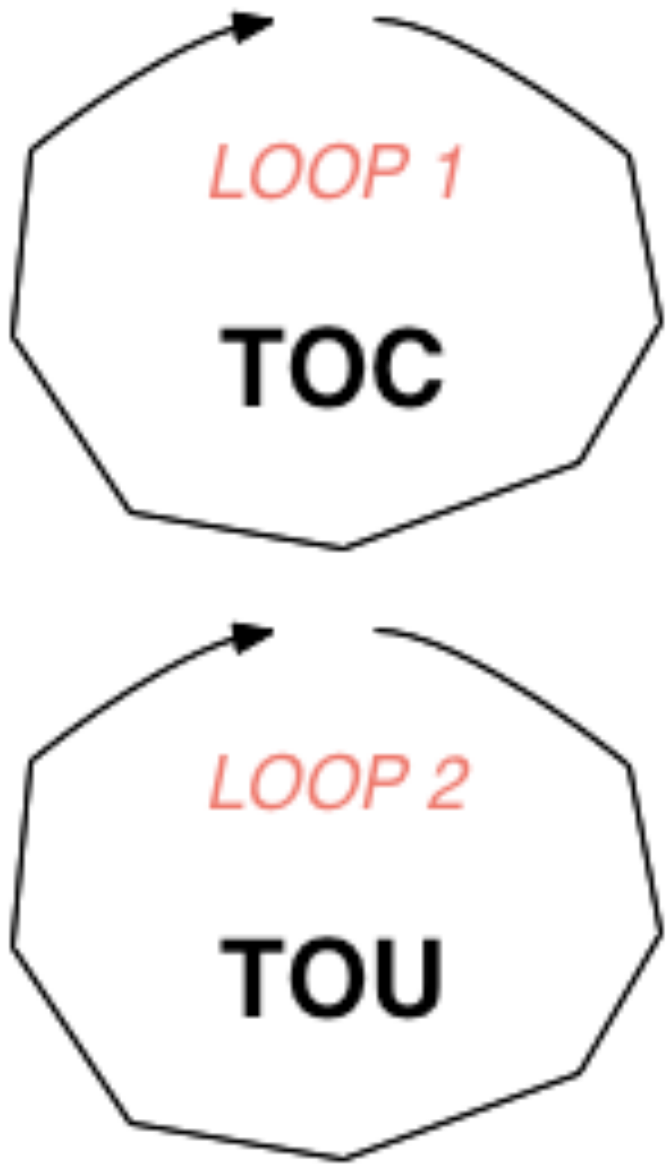
- each file action is **1040** bytes
- file actions are allocated with **kalloc()**
- so we have either **4kb** or **12kb** memory
- only space for **3** to **11** file actions

- **NOT ENOUGH FOR NOTABLE SLOW DOWN**



Slowing down file actions?

- we cannot slow down **dup2()**
- we cannot slow down **close()**
- **but what about open() ???**



Manpage of open()

OPEN(2) BSD System Calls Manual OPEN(2)

NAME
    open -- open or create a file for reading or writing

SYNOPSIS
    #include <fcntl.h>

    int
    open(const char *path, int oflag, ...);

DESCRIPTION
    The file name specified by path is opened for reading and/or writing, as specified by the argument oflag; the file descriptor is returned to the calling process.

    The oflag argument may indicate that the file is to be created if it does not exist (by specifying the O_CREAT flag). In this case, open requires a third argument mode_t mode; the file is created with mode mode as described in chmod(2) and modified by the process' umask value (see umask(2)).

    The flags specified are formed by or'ing the following values:

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open or for data to become available
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if O_CREAT and the file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an exclusive lock
O_NOFOLLOW	do not follow symlinks
O_SYMLINK	allow open of symlinks
O_EVTONLY	descriptor requested for event notifications only
O_CLOEXEC	mark as close-on-exec

open supports
file locking

if we open already
locked file
posix_spawn will
sleep until lock is released

Winning the Race !!!

- turns out that the race condition is easy to win 100% of the time
- just need to sync with a secondary thread via file locking

Write Two

READ FROM MEMORY

VERIFICATION

...
OPEN LOCKED FILE
...

RE-READ FROM MEMORY

WRITE

File Locking Sync

Thread 1

OPEN FILE A (O_EXLOCK)

POSIX_SPAWN

File Action 1
SOME ACTION

File Action 2
CLOSE FILE A (LOCK RELEASE)

... wait for unlock of file B ...
... wait for unlock of file B ...
... wait for unlock of file B ...

File Action 3
OPEN FILE B (O_EXLOCK)

Thread 2

OPEN FILE B (O_EXLOCK)

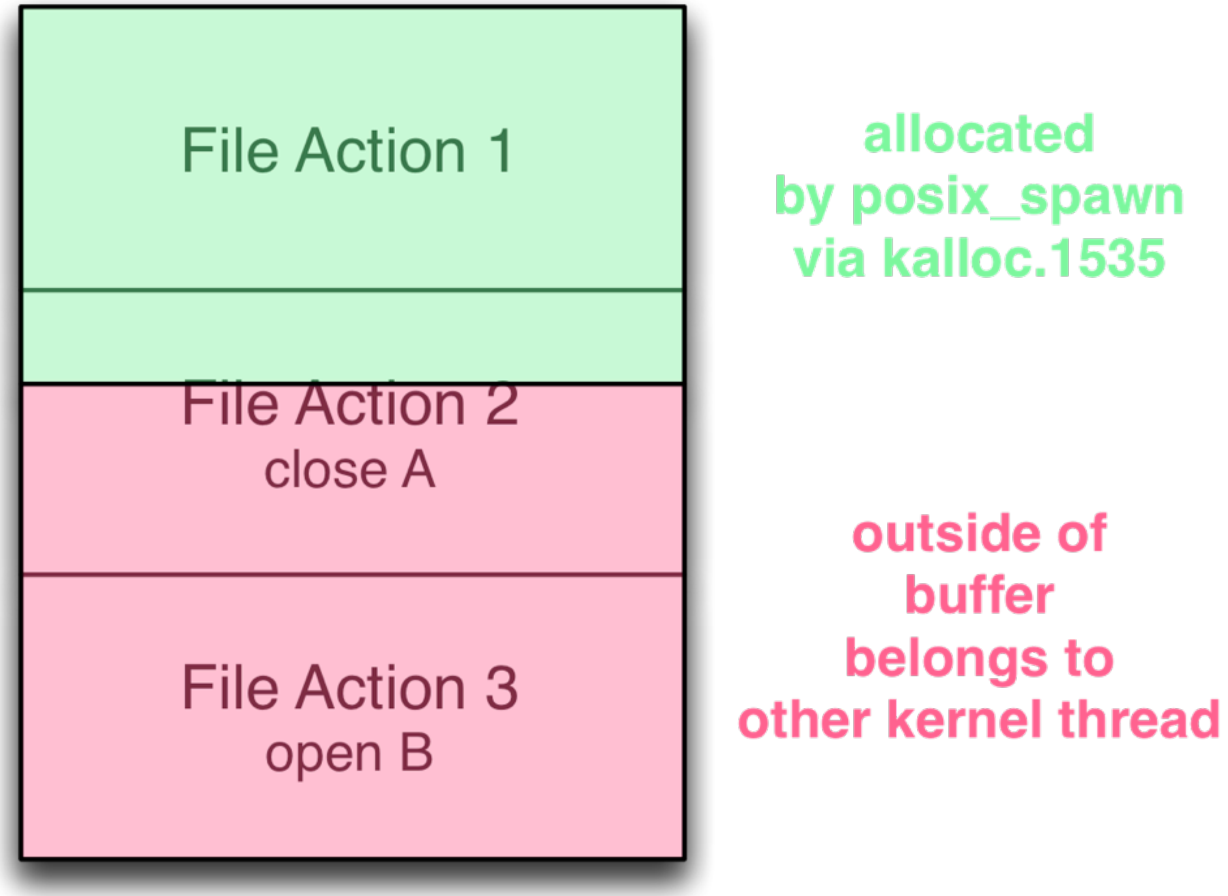
OPEN FILE A (O_EXLOCK)
... wait for unlock of file A ...
... wait for unlock of file A ...
... wait for unlock of file A ...
... wait for unlock of file A ...

MODIFICATION OF MEMORY OF FILE ACTION 2

CLOSE FILE B (LOCK RELEASE)

At this point we have the following

- winning the race is easy with
  3 file actions, 2 file locks
  and 2 threads
- we need to deal with **kalloc.1536** or bigger
- most of file action 2 and whole file action 3 outside of buffer
- requires already Heap-Feng-Shui to achieve this



SektionEins

How to control the write?

How to control the write?

```
*fdflags(p, fd) |= UF_INHERIT;
```

- the write is a **BINARY OR** against **UF_INHERIT** = 0x20
- we can only set bit **5** in some byte anywhere in memory
- write is relative to **fd_ofileflags**

- **PROBLEM**: where is **fd_ofileflags**?

Where is fd_ofileflags?

- **fd_ofileflags** is allocated after process is started
- and we have no idea where it is
- to find out the address of **fd_ofileflags** we require some information leak

- we have no information leak that gives us its address :-(
- so we have to abuse the relative write to create a man-made information leak

Force fd_ofileflags relocation (I)

- **fd_ofileflags** is allocated in an unknown position
- to abuse the relative write we need to be **at least able to relocate** it
- reallocation happens in **fdalloc()** when all file descriptors are exhausted
- by default we start with a limit of **256** allowed file descriptors

```
int fdalloc(proc_t p, int want, int *result)
{
    ...
    lim = min((int)p->p_rlimit[RLIMIT_NOFILE].rlim_cur, maxfiles);
    for (;;) {
        ...

        /*
         * No space in current array.  Expand?
         */
        if (fdp->fd_nfiles >= lim)
            return (EMFILE);
        if (fdp->fd_nfiles < NDEXTENT)
            numfiles = NDEXTENT;
        else
            numfiles = 2 * fdp->fd_nfiles;
        /* Enforce lim */
        if (numfiles > lim)
            numfiles = lim;
        proc_fdunlock(p);
        MALLOC_ZONE(newofiles, struct fileproc **,
                numfiles * OFILESIZE, M_OFILETABL, M_WAITOK);
        proc_fdlock(p);
        if (newofiles == NULL) {
            return (ENOMEM);
        }
        ...
        newofileflags = (char *) &newofiles[numfiles];
```

Force fd_ofileflags relocation (II)

- forcing a **fd_ofileflags** reallocation comes down to
  - raising the limit for openable files with **setrlimit(RLIMIT_NOFILE)** to **257**
  - using **dup2()** to force use of highest allowed file descriptor

- memory allocation will be for **5 * 257 = 1285**
- reallocated **fd_ofileflags** ends up in the **kalloc.1536** zone

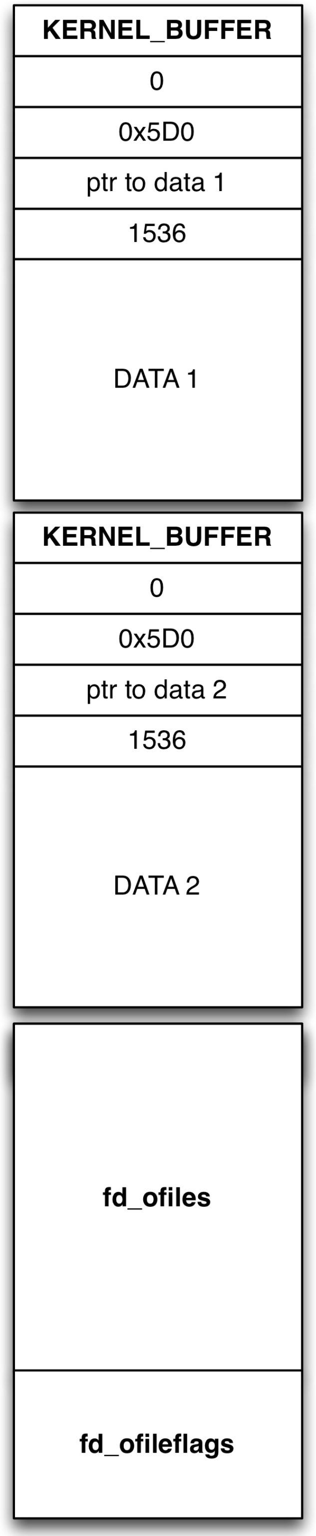
Relocated... What now?

- re-allocation allows to put **fd_ofileflags** into a relative position to other blocks
- heap-feng-shui in **kalloc.1536** zone required
- so what can we do with our relative **binary-or** of **0x20**?

- use **Azimuth's vm_map_copy_t** self locating technique

Self-Locating with vm_map_copy_t

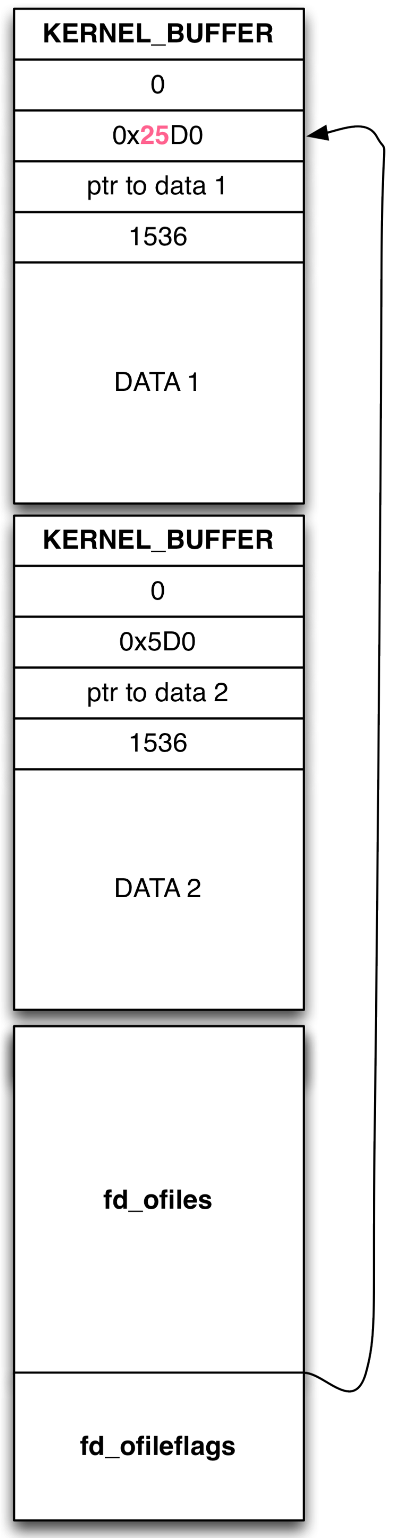
- need to relocate **fd_ofileflags** to be behind two **vm_map_copy_t** structures
- use relative write to increase 2nd byte of size field of first **vm_map_copy_t**
- now receive the first message to information leak the content behind
- discloses the 2nd **vm_map_copy_t** including its address
- and also the content of the **fd_ofileflags** structure



SektionEins

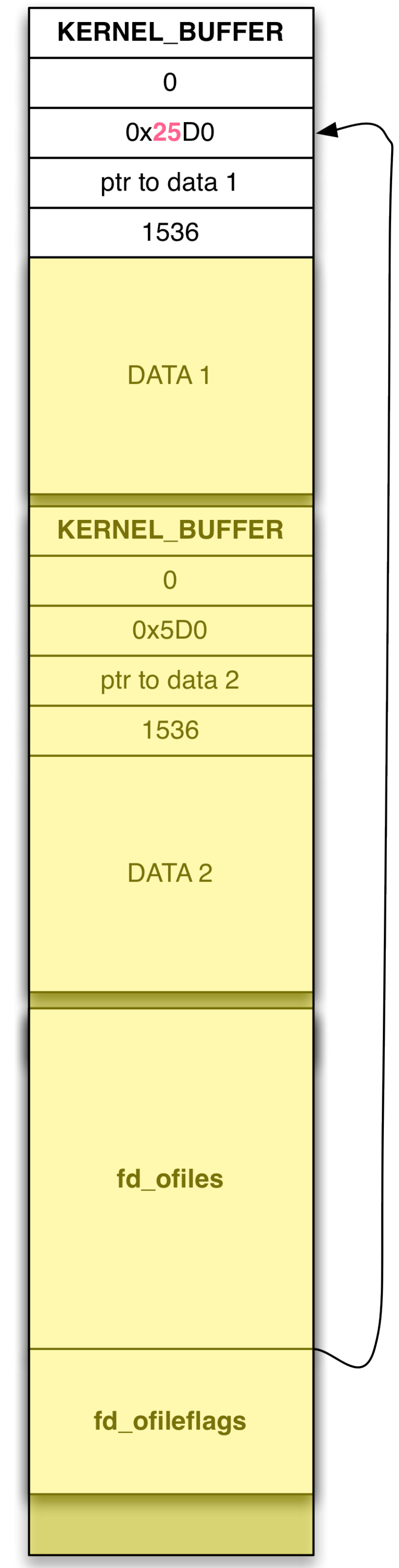
Self-Locating with vm_map_copy_t

- need to relocate **fd_ofileflags** to be behind two **vm_map_copy_t** structures
- use relative write to increase 2nd byte of size field of first **vm_map_copy_t**
- now receive the first message to information leak the content behind
- discloses the 2nd **vm_map_copy_t** including its address
- and also the content of the **fd_ofileflags** structure



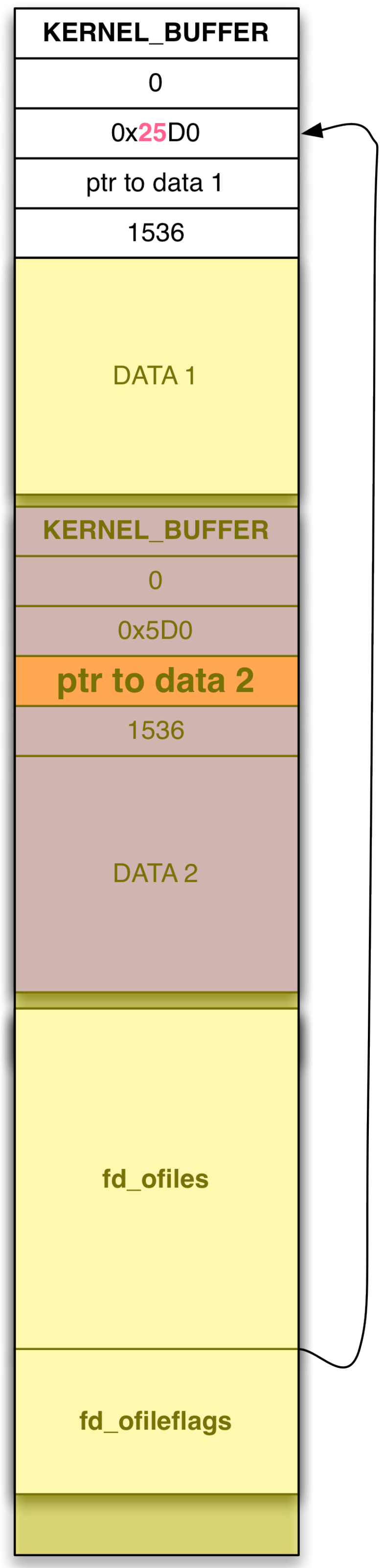
Self-Locating with vm_map_copy_t

- need to relocate **fd_ofileflags** to be behind two **vm_map_copy_t** structures
- use relative write to increase 2nd byte of size field of first **vm_map_copy_t**
- now receive the first message to information leak the content behind
- discloses the 2nd **vm_map_copy_t** including its address
- and also the content of the **fd_ofileflags** structure



Self-Locating with vm_map_copy_t

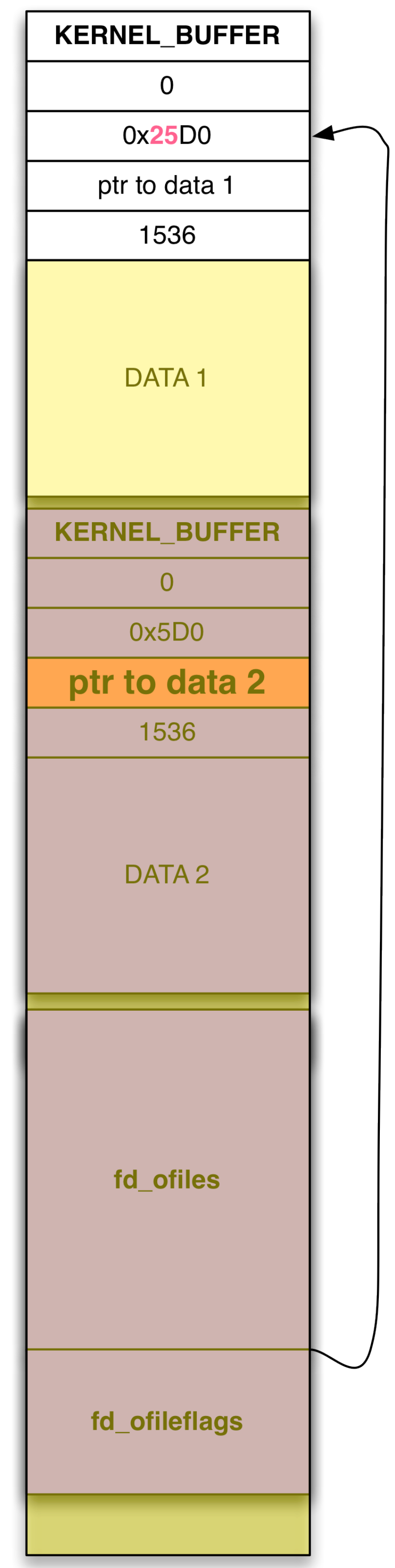
- need to relocate **fd_ofileflags** to be behind two **vm_map_copy_t** structures
- use relative write to increase 2nd byte of size field of first **vm_map_copy_t**
- now receive the first message to information leak the content behind
- discloses the 2nd **vm_map_copy_t** including its address
- and also the content of the **fd_ofileflags** structure



SektionEins

Self-Locating with vm_map_copy_t

- need to relocate **fd_ofileflags** to be behind two **vm_map_copy_t** structures
- use relative write to increase 2nd byte of size field of first **vm_map_copy_t**
- now receive the first message to information leak the content behind
- discloses the 2nd **vm_map_copy_t** including its address
- and also the content of the **fd_ofileflags** structure



What we have so far ...

- fill the **kalloc.1536** zone via **vm_map_copy_t (OOL mach_msg)**
- peek a hole and trigger **fd_ofileflags** relocation into it (**setrlimit + dup2**)
- poke two more holes (**H1 followed by H2**) and
  re-fill **H2** with our initial file actions 2+3 (close A+open B) **(OOL mach msg)**
- do **posix_spawn**
- when it releases file A and waits for file B let other thread modify memory
- ...

What we have so far ...

- ...
- second thread pokes a hole at **H2** and re-fill it with new file actions
  - file action 2 is changed from **PSFA_CLOSE** to **PSFA_DUP2**
  - **fd** of file action 2 is set to relative position of size field of the first **vm_map_copy_t** structure
- second thread closes file B to wake-up **posix_spawn**
- after **posix_spawn** has returned with an error receive the first mach message

=> from leaked data we now know the address of **fd_ofileflags**

Now write where?

Now write where?

- we now have the address of **fd_ofileflags**
- further writes can be anywhere in memory
- what to overwrite to control code execution?

=> many possibilities

=> we go after the size field of a data object to create a buffer overflow

From Data Objects to Overflows...

- we have to solve the following problems
  - how to create a data object to overwrite
  - how to get its address so that we know where to write
  - and finally destroying the data object to trigger kfree into wrong zone

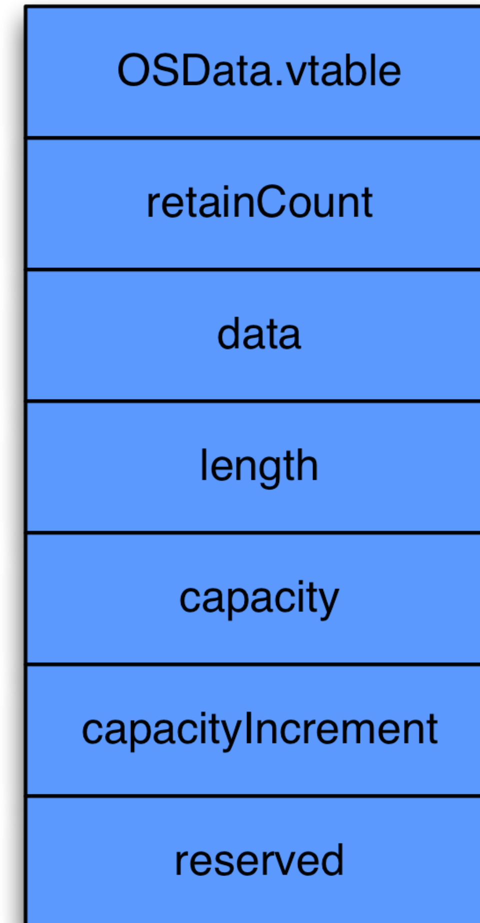
Creating Data and Leaking its Address

- creating data objects is easy with **OSUnserializeXML()**
- we can do this via **io_service_open_extended()** and properties

- leaking is also easy in our situation
- we put the data object and 256 references to it into an array
- array bucket will be allocated into the **kalloc.1536** zone
- we can do this in parallel to the **vm_map_copy_t** self-locating and leak the content of the array bucket at the same time

  => this gives us the data object address

Overwriting and Destroying the Data Object



- we now have to do the **posix_spawn()** attack again with the data object's **capacity** field as target
- we can then free the data object by closing the driver connection again

=> this will free the data buffer into the wrong zone
=> next allocation in that zone will give back a too short buffer
=> we can send a **OOL mach_msg** to trigger that overflow

SektionEins

What to overflow into…

- now we can create a heap buffer overflow out of posix_spawn()
- we need a target to overflow into
- again we have a multitude of options
- some examples:
    - overflow an **IOUserClient** created by a driver connection for code exec
    - overflow into a **vm_map_copy_t** for arbitrary information leaks
    - …

Overflowing into vm_map_copy_t

- by overflowing into a **vm_map_copy_t** structure we can
  - read "any amount" of bytes from anywhere in kernel into user space
  - just need to setup a fake **vm_map_copy_t** header
  - and then receive the message

Overflowing into a driver connection

- by overflowing into a **IOUserClient** object instance we can
    - replace the **vtable** with a list of our own methods
    - set the **retainCount** to a high value to not cause problems

  => but what to overwrite the **vtable** with?

Vtable where are thou?

- our fake **vtable** is a list of pointers that we just need to put into memory
- we can put it into kernel memory by sending a **mach_msg**
- we best use the **kalloc.1536** target zone
    - cause enough space for a long **vtable**
    - and we already know address of blocks in a relative position to it

From Vtable to Pwnage

From Vtable to Pwnage (I)

- at this point we have to select the addresses our **vtable** should point to
- for this we need to know the current address of the kernel
- and the content of the kernel

- we can use any KASLR information leak for getting the kernel base address or just leak the **vtable** of an object via the **vm_map_copy_t** technique
- the second we can also get by overflowing into **vm_map_copy_t** instead of a user client object

From Vtable to Pwnage (II)

- from here it is easiest to go after **IOUserClient** external traps
- they can be called from **mach_trap 100 iokit_user_client_trap**
- allows to call arbitrary functions with arbitrary parameters in the kernel

```
kern_return_t iokit_user_client_trap(struct iokit_user_client_trap_args *args)
{
    kern_return_t result = kIOReturnBadArgument;
    IOUserClient *userClient;

    if ((userClient = OSDynamicCast(IOUserClient,
            iokit_lookup_connect_ref_current_task((OSObject *)(args->userClientRef))))) {
        IOExternalTrap *trap;
        IOService *target = NULL;

        trap = userClient->getTargetAndTrapForIndex(&target, args->index);

        if (trap && target) {
            IOTrap func;

            func = trap->func;

            if (func) {
                result = (target->*func)(args->p1, args->p2, args->p3, args->p4, args->p5, args->p6);
            }
        }
        userClient->release();
    }
    return result;
}
```

fake vtable
needs to
implement this

From Vtable to Pwnage (III)

- default implementation in **IOUserClient** does call **getExternalTrapForIndex()**
- its default is returning NULL
- we should only overwrite **getExternalTrapForIndex()**

```
IOExternalTrap * IOUserClient::
getExternalTrapForIndex(UInt32 index)
{
    return NULL;
}

IOExternalTrap * IOUserClient::
getTargetAndTrapForIndex(IOService ** targetP, UInt32 index)
{
      IOExternalTrap *trap = getExternalTrapForIndex(index);

      if (trap) {
              *targetP = trap->object;
      }

      return trap;
}
```

From Vtable to Pwnage (IV)

- in our **vtable** we set **getTargetAndTrapForIndex** to the original **IOUserClient::getTargetAndTrapForIndex**
- and we set **getExternalTrapForIndex()** to a gadget that performs the below **(e.g. MOV R0, R1; BX LR)**

```
IOExternalTrap * IOUserClient::
OUR_FAKE_getExternalTrapForIndex(void *index)
{
    return index;
}

IOExternalTrap * IOUserClient::
getTargetAndTrapForIndex(IOService ** targetP, UInt32 index)
{
      IOExternalTrap *trap = getExternalTrapForIndex(index);

      if (trap) {
              *targetP = trap->object;
      }

      return trap;
}
```

index from
user space
will be used
as kernel pointer
to IOExternalTrap

From Vtable to Pwnage (V)

- by setting the „index" argument of **iokit_user_client_trap** to our buffer
- we can call any function in the kernel with up to 7 parameters

```
kern_return_t iokit_user_client_trap(struct iokit_user_client_trap_args *args)
{
    kern_return_t result = kIOReturnBadArgument;
    IOUserClient *userClient;

    if ((userClient = OSDynamicCast(IOUserClient,
            iokit_lookup_connect_ref_current_task((OSObject *)(args->userClientRef))))) {
        IOExternalTrap *trap;
        IOService *target = NULL;

        trap = userClient->getTargetAndTrapForIndex(&target, args->index);

        if (trap && target) {
            IOTrap func;

            func = trap->func;

            if (func) {
                result = (target->*func)(args->p1, args->p2, args->p3, args->p4, args->p5, args->p6);
            }
        }
        userClient->release();
    }
    return result;
}
```

we can
call everything

Part II

iOS 7 Security Changes

System Call Table Hardening (Structure)

- in previous versions of iOS Apple has protected the table by
  - removing symbols
  - moving variables like the system call number around
- this was done to protect against easy detection in memory / in the binary
- in iOS 7 they went a step further and changed the actual structure of the system call table entries

➡ unknown if Apple did this a security protection but it makes all public detectors fail

System Call Table Hardening (Access)

- in iOS 6 Apple has moved system call table into **__DATA::__const**
- this section is read-only at runtime
- protects system call table from overwrites
- but the code would access table via a writable pointer in **__nl_symbol_ptr**

- iOS 7 fixes this by using **PC** relative addressing when accessing **_sysent**

System Call Table Hardening (Variables)

- potential attack has always been tampering with the **nsys** variable
- overwriting this allowed referencing memory outside the table
- executing illegal syscalls would have resulted in execution hijack

- iOS 7 fixes this by removing access to the **nsys** variable
- maximum number of system calls is now hardcoded into the code

Sandbox Hardening

- requires more research
- but filesystem access has been locked down once more
- application containers can access fewer files in the filesystem
    - example iOS 7 disallows access to **`/bin`** and **`/sbin`**
    - applications can no longer steal e.g. **`launchd`** from **`/sbin/launchd`**

Read-Only Root Filesystem Enforcement

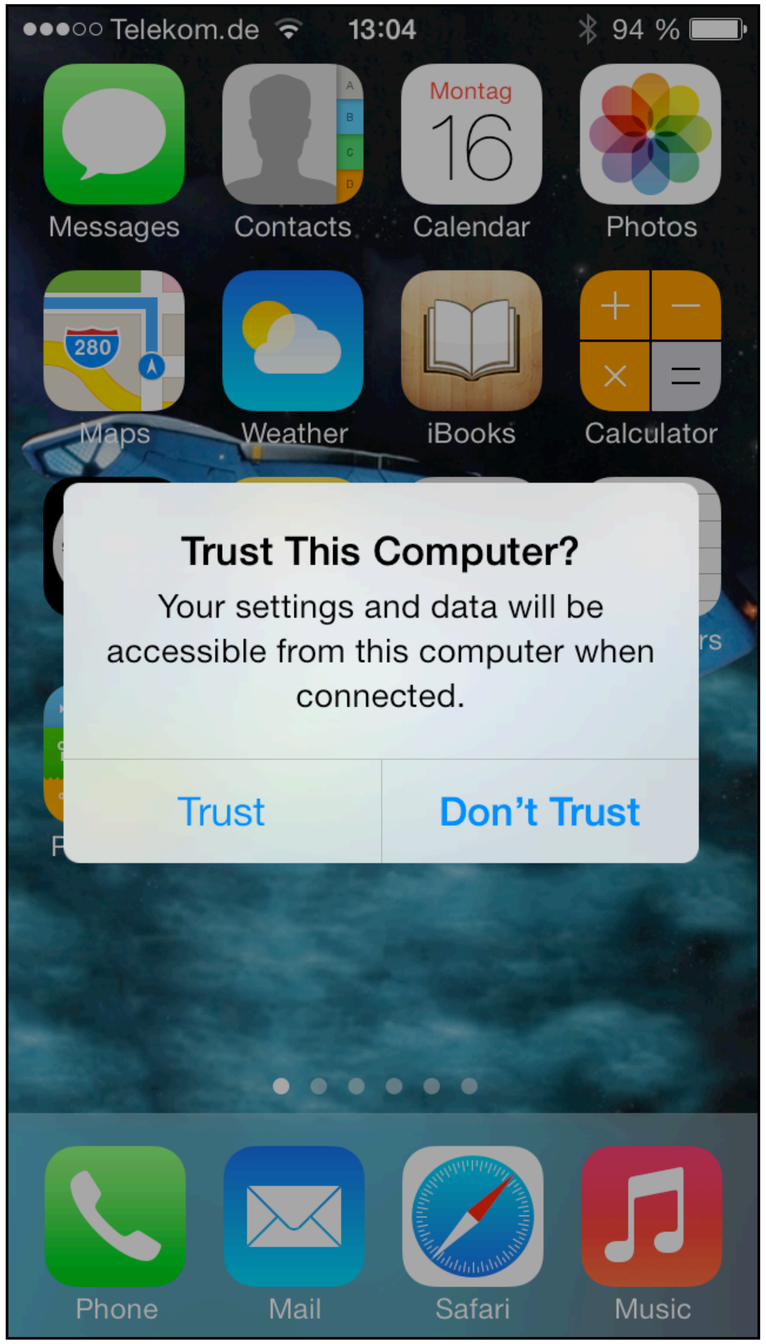
- iOS 7 introduces a "security" check into the **`mount()`** systemcall
- attempt to load the root filesystem as readable-writable results in **`EPERM`**
- mounting the root fs as readable-writable now requires kernel trickery
- **`/etc/fstab`** trickery no longer enough

```
    if ((vp->v_flag & VROOT) &&
        (vp->v_mount->mnt_flag & MNT_ROOTFS)) {

        flags &= ~MNT_UPDATE;
        if ( !(flags & MNT_UNION) ) {
            flags |= MNT_UPDATE;
        }

        if ( !(flags & MNT_RDONLY) ) {
            error = EPERM;
            goto out;
        }
    }

    error = mount_common(fstypename, pvp, vp, &nd.ni_cnd, uap->data, flags, 0,
                         labelstr, FALSE, ctx);
out:
```

read only mount
results in EPERM

Juice Jacking

- attack vector known for years
- iOS devices vulnerable to malicious USB ports (e.g. charger)
- malicious USB port can pair with device and use features like backup, file transfer or activate developer mode
- in developer mode malware upload is trivial
- largely ignored until BlackHat + US media hyped it
- iOS 7 adds a popup menu as countermeasure



LaunchDaemon Security

- Apple added code signing for launch daemons in iOS 6.1
- but Apple forgot / or ignored `/etc/launchd.conf`
- `/etc/launchd.conf` defines commands **launchctl** executes on start
- jailbreaks like evasi0n abused this to execute arbitrary existing commands
- in iOS 7 Apple removed usage of this file

```
bsexec .. /sbin/mount -u -o rw,suid,dev /
setenv DYLD_INSERT_LIBRARIES /private/var/evasi0n/amfi.dylib
load /System/Library/LaunchDaemons/com.apple.MobileFileIntegrity.plist
bsexec .. /private/var/evasi0n/evasi0n
unsetenv DYLD_INSERT_LIBRARIES
bsexec .. /bin/rm -f /private/var/evasi0n/sock
bsexec .. /bin/ln -f /var/tmp/launchd/sock /private/var/evasi0n/sock
```

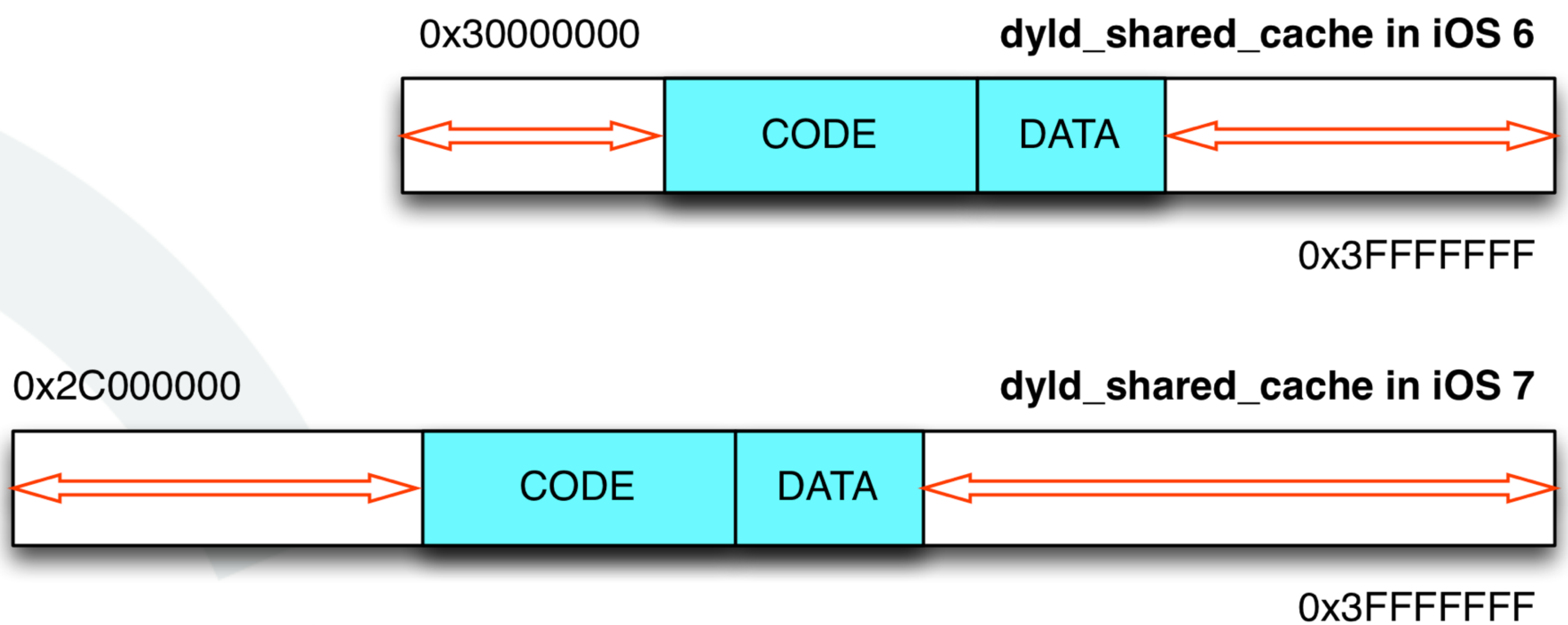
Partial Code Signing Hardening

- many jailbreaks used partial code signing vulnerabilities for persistence
- basically all those exploited the dynamic linker **dyld**
- with iOS 7 Apple has added a new function called **crashIfInvalidCodeSignature**
- function touches all segments to cause crashes if invalid signature is provided

```
int __fastcall ImageLoaderMachO::crashIfInvalidCodeSignature(int a1)
{
  int v1; // r4@1
  int result; // r0@1
  unsigned int v3; // r5@2

  v1 = a1;
  result = 0;
  if ( *(_BYTE *)(v1 + 72) )
  {
    v3 = 0;
    while ( (*(int (__fastcall **)(int, unsigned int))(*(_DWORD *)v1 + 208))(v1, v3)
         || !(*(int (__fastcall **)(int, unsigned int))(*(_DWORD *)v1 + 200))(v1, v3) )
    {
      ++v3;
      result = 0;
      if ( v3 >= *(_BYTE *)(v1 + 72) )
        return result;
    }
    result = *(_DWORD *)(*(int (__fastcall **)(int, unsigned int))(*(_DWORD *)v1 + 236))(v1, v3);
  }
  return result;
}
```

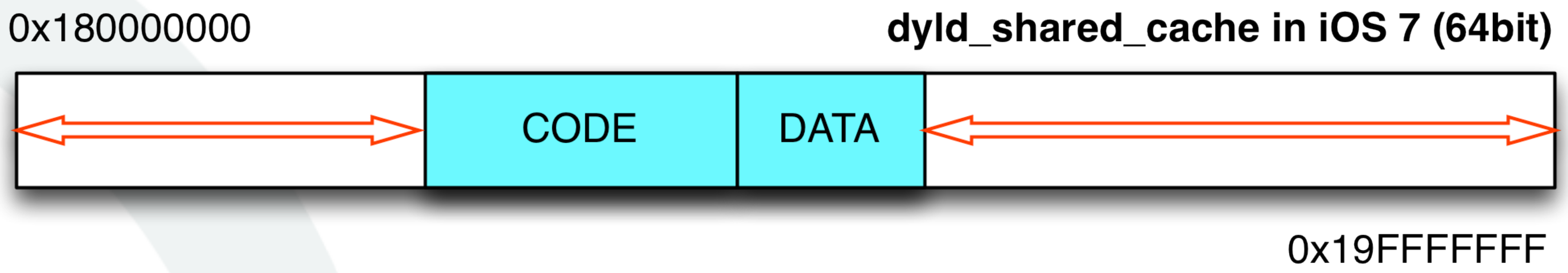
Library Randomization

- iOS 6 slid the dynamic shared cache between 0x30000000 - 0x3FFFFFFF
- in this 256MB window 21500 different base addresses possible (iPod 4G)
- new devices = more code = less random
- iOS 7 now slides between 0x2C000000 - 0x3FFFFFFF adds 2^13 entropy



Library Randomization (64 bit)

- iPhone 5S and its 64 bit address space allows for better randomization
- separate 64 bit shared cache file
  /System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64
- dynamic shared cache loaded between 0x180000000 - 0x19FFFFFFF
- finally fixes the cache overlap vulnerability



Questions

?