SektionEins
http://www.sektioneins.de

death of the vmsize=0 dyld trick

(one more way to persist on your iPhone killed)
SyScan 2015 Bonus Slides

Stefan Esser <stefan.esser@sektioneins.de>

Who am I?

Stefan Esser

- from Cologne / Germany
- in information security since 1998
- invested in PHP security from 2001 to 20xx
- since 2010 focused on iPhone security (ASLR/jailbreak)
- founder of SektionEins GmbH

Introduction

- at SyScan 2015 I made a talk about
  - how Apple failed to fix vulnerabilities used in iOS 678 jailbreaks over and over again
  - how and why Chinese jailbreak teams took over the jb scene in 2014

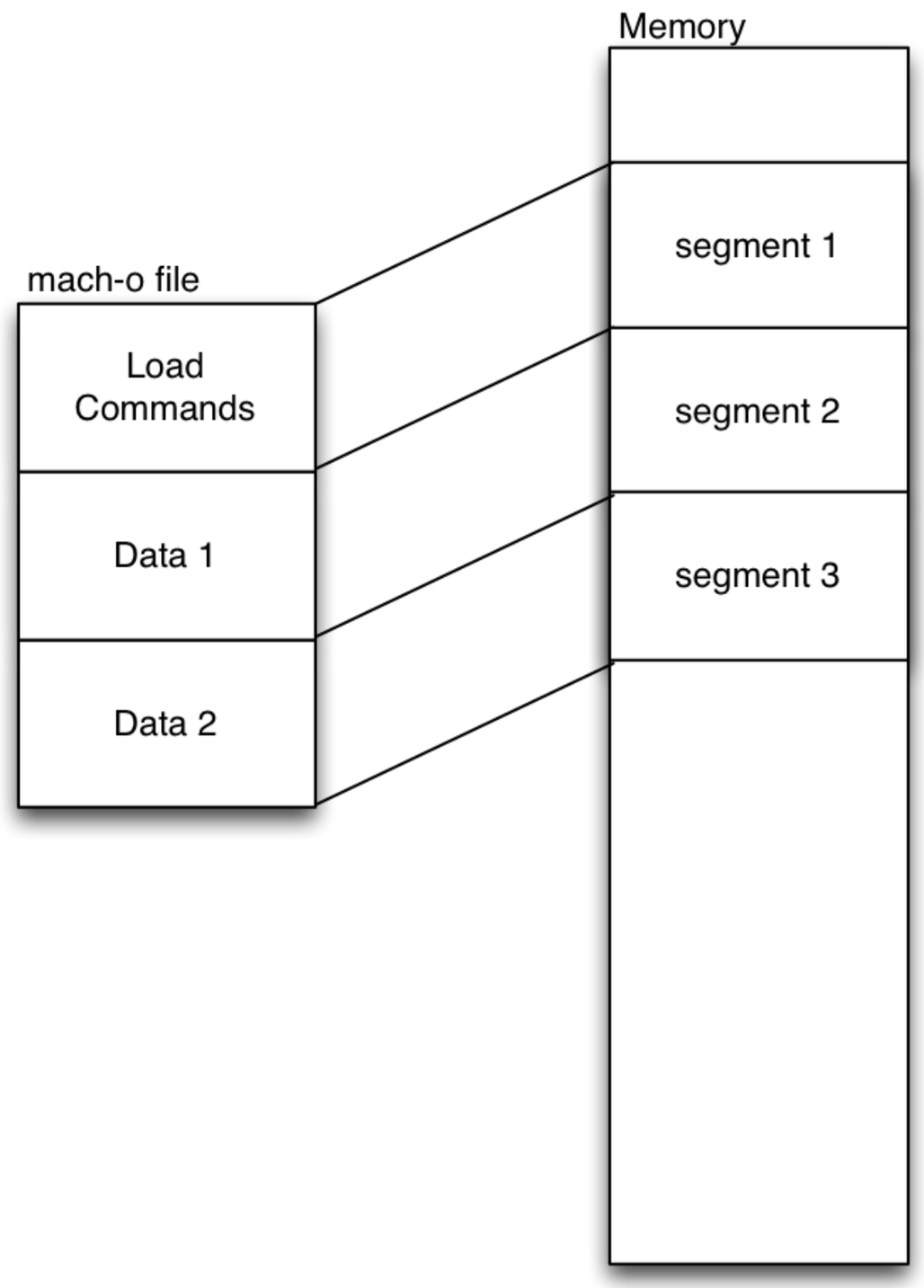
- during the talk I discussed ***"Patient ALPHA"*** an incomplete code signing bug that Apple failed to analyse correctly and therefore had to issue 4 security updates for
- during the talk I also promised to disclose another incomplete code signing vulnerability that Apple closed by accident with their patches for **"Patient ALPHA"**
- we should be fair in information security and not only discuss fail, but also show how they killed a bug (even if they most probably did not know about it)

Incomplete Codesigning

- **simplified:**
  - if a page is not executable a missing code sig is not a problem
  - if a page is executable there must be a code sig on first access
- prior to **iOS 5** therefore jailbreaks would use **ALL DATA dylibs** to exploit dyld via various meta-data structures
- around the **end of iOS 4** Apple added checks to dyld to **enforce** load commands are in an **executable segment**
- therefore while header parsing (first access) code sig is required

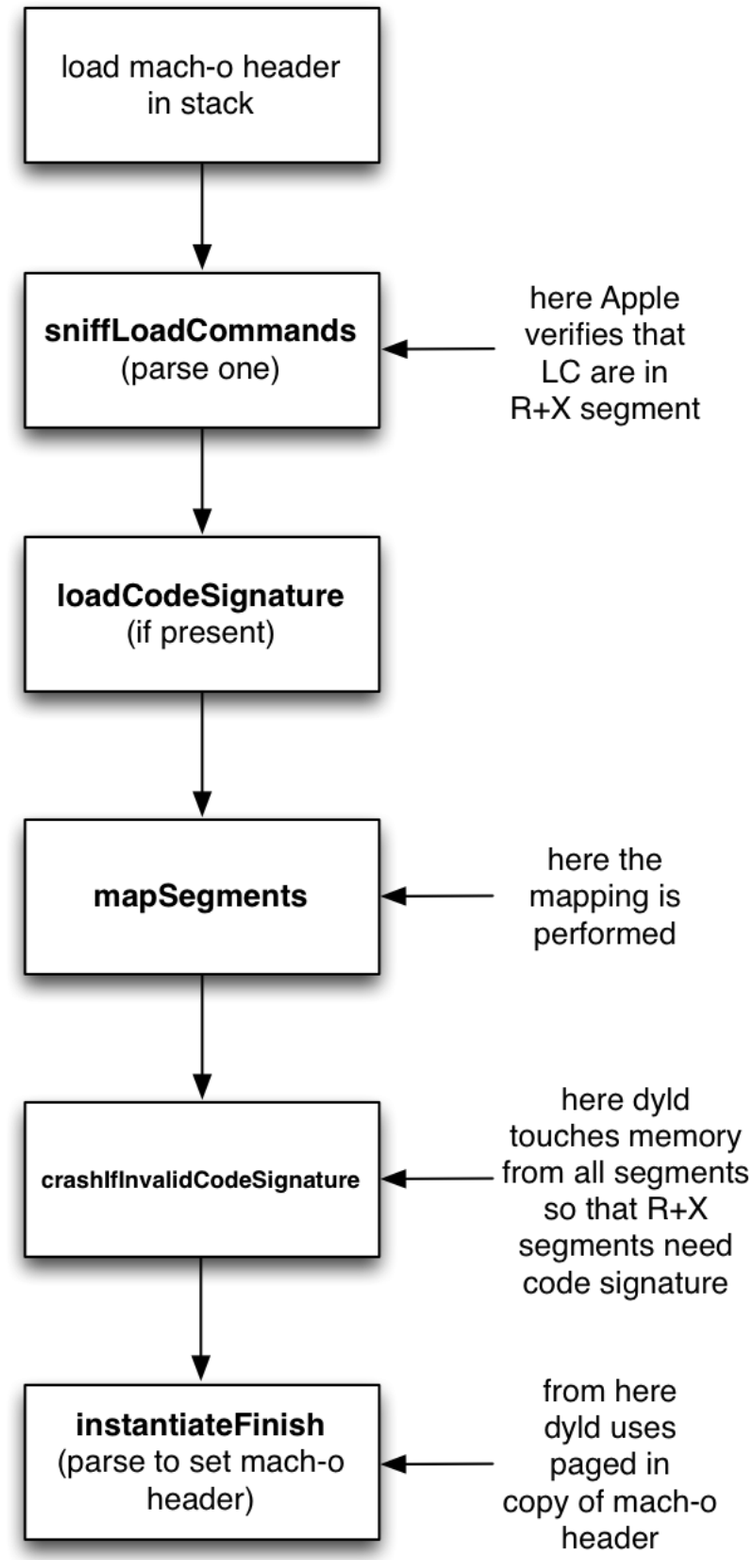
mach-o dynamic library loading

- **mach-o** dynamic libraries loaded by **dyld**
- **load commands** describe i.a. layout of segments in memory
  - **virtual address** and **virtual size** of segments
  - **file position** and **file size** of segment



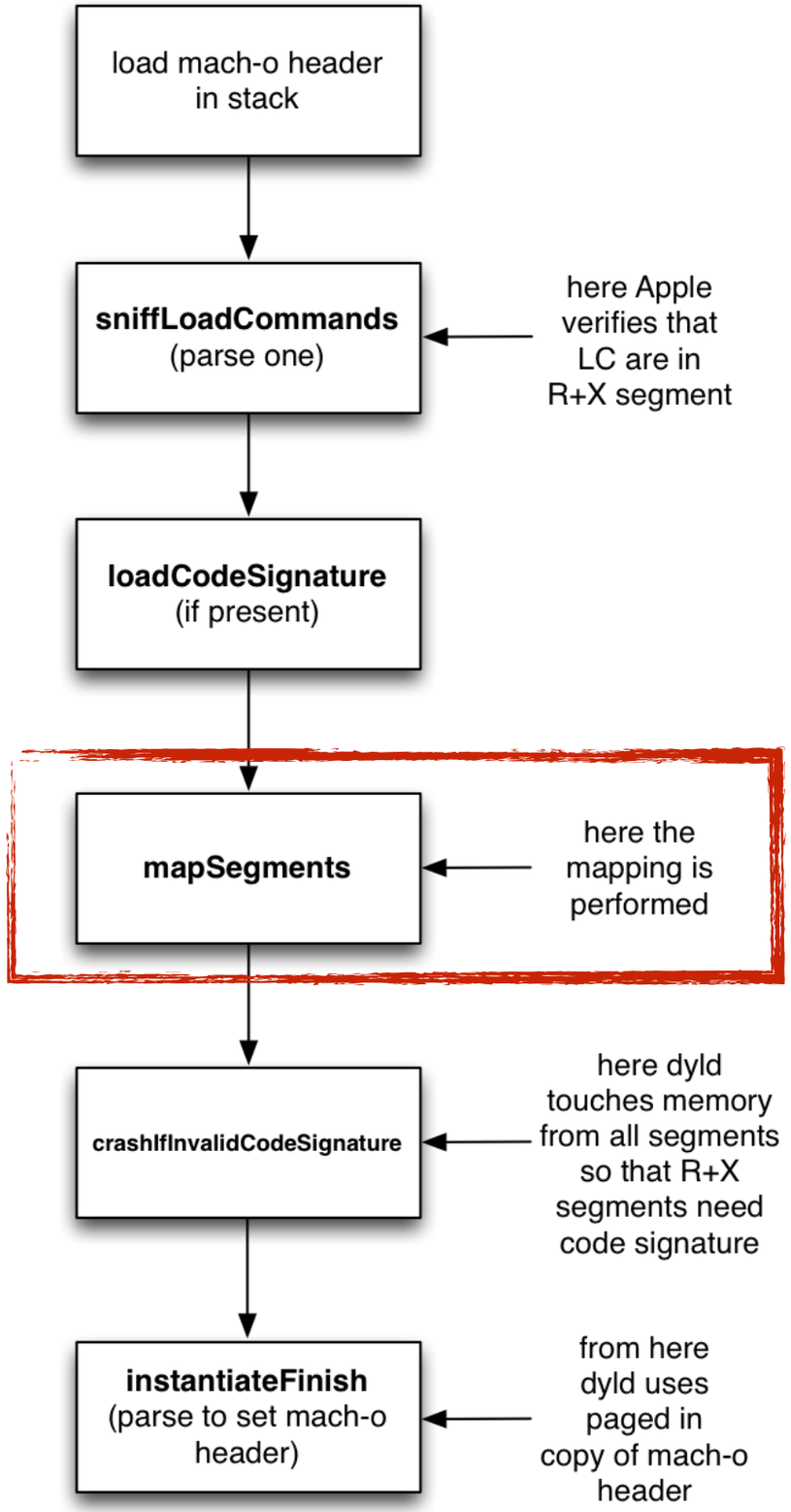
Wait a second ...

- actually it is not that simple
- mach-o header is first loaded into stack
- initial LC parse is performed to collect info
- this info is used to map the file into memory
- segments are touched to enforce code sig
- another LC parse is performed to make dyld use the mach-o header from paged memory
- more and more parsing



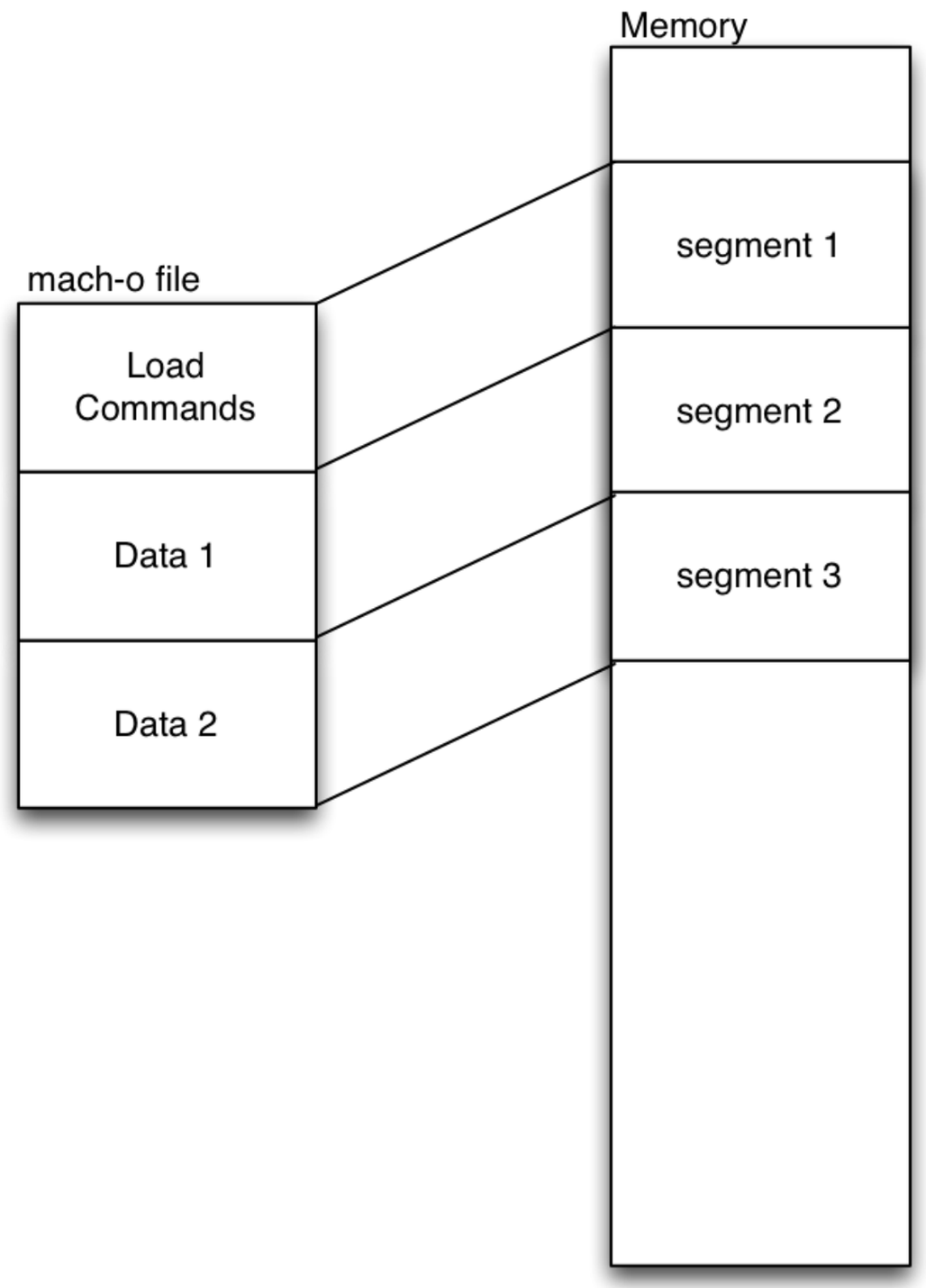
TOCTOU Problems

- this chain of events has <u>some</u> **TOCTOU** *(Time of Check Time of Use)* problems
- attacking the code flow between **sniffLoadCommands** and **crashIfInvalidCodesignature**
- tricking e.g. **mapSegments**
- evad3rs tricked that function first, but they went after replacing segment mappings or stripping the X flag
- however there was a 0-day that tricked this whole logic in a different way



mapSegments

- **mapSegments** goes through the list of known segments and maps them one by one
- count of segments is calculated inside the **sniffLoadCommands** functions
- **mapSegments** fully relies on that count
- apparently there will be problems if **sniffLoadCommands** is counting the segments wrong



sniffLoadCommands and Segment Counting

- code inside **sniffLoadCommands** counts **LC_SEGMENT_COMMAND** commands
- but it ignores segments if their **VMSIZE** is 0

```
switch (cmd->cmd) {
    case LC_DYLD_INFO:
    case LC_DYLD_INFO_ONLY:
        *compressed = true;
        break;
    case LC_SEGMENT_COMMAND:
        segCmd = (struct macho_segment_command*)cmd;
        // ignore zero-sized segments
        if ( segCmd->vmsize != 0 )
            *segCount += 1;
        // <rdar://problem/7942521> all load commands must be in an executable segment
        if ( context.codeSigningEnforced && (segCmd->fileoff < mh->sizeofcmds) && (segCmd->filesize != 0) ) {
            if ( (segCmd->fileoff != 0) || (segCmd->filesize < (mh->sizeofcmds+sizeof(macho_header))) )
                dyld::throwf("malformed mach-o image: segment %s does not span al…", segCmd->segname);
```

ImageLoaderMachO::ImageLoaderMachO (I)

- inside the constructor of **ImageLoaderMachO** the addresses of the **LC_SEGMENT_COMMAND** load commands are put into a cache
- this is done to easier traverse through the list of segments
- the code also ignores segments with a **VMSIZE**=0
- **IMPORTANT: this means that if the segment containing load commands has vmsize=0 it is not in list of segments and will not be traversed by later code**

```
// construct SegmentMachO object for each LC_SEGMENT cmd using "placement new" to put
// each SegmentMachO object in array at end of ImageLoaderMachO object
const uint32_t cmd_count = mh->ncmds;
const struct load_command* const cmds = (struct load_command*)&fMachOData[sizeof(macho_header)];
const struct load_command* cmd = cmds;
for (uint32_t i = 0, segIndex=0; i < cmd_count; ++i) {
    if ( cmd->cmd == LC_SEGMENT_COMMAND ) {
        const struct macho_segment_command* segCmd = (struct macho_segment_command*)cmd;
        // ignore zero-sized segments
        if ( segCmd->vmsize != 0 ) {
            // record offset of load command
            segOffsets[segIndex++] = (uint32_t)((uint8_t*)segCmd - fMachOData);
        }
    }
    cmd = (const struct load_command*)(((char*)cmd)+cmd->cmdsize);
}
```

ImageLoaderMachO::ImageLoaderMachO (II)

- **IMPORTANT:** internally all accesses to the mach-o header go through **fMachOData**
- so initially it is set to the stack
  (which is not executable and therefore requires no code signature)
- later on it is supposed to be pointing to the mapped executable segment containing the load commands

```
// construct SegmentMachO object for each LC_SEGMENT cmd using "placement new" to put
// each SegmentMachO object in array at end of ImageLoaderMachO object
const uint32_t cmd_count = mh->ncmds;
const struct load_command* const cmds = (struct load_command*)&fMachOData[sizeof(macho_header)];
const struct load_command* cmd = cmds;
for (uint32_t i = 0, segIndex=0; i < cmd_count; ++i) {
    if ( cmd->cmd == LC_SEGMENT_COMMAND ) {
        const struct macho_segment_command* segCmd = (struct macho_segment_command*)cmd;
        // ignore zero-sized segments
        if ( segCmd->vmsize != 0 ) {
            // record offset of load command
            segOffsets[segIndex++] = (uint32_t)((uint8_t*)segCmd - fMachOData);
        }
    }
    cmd = (const struct load_command*)(((char*)cmd)+cmd->cmdsize);
}
```

mapSegments mapping segments

- **mapSegments** traverses ONLY the cached list of **LC_SEGMENT_COMMANDS**
- any segment that had a **vmsize=0** will never be mapped (because they are not in there)

```
// map in all segments
for(unsigned int i=0, e=segmentCount(); i < e; ++i) {
    vm_offset_t fileOffset = segFileOffset(i) + offsetInFat;
    vm_size_t size = segFileSize(i);
    uintptr_t requestedLoadAddress = segPreferredLoadAddress(i) + slide;

    ...

        void* loadAddress = xmmap((void*)requestedLoadAddress, size, protection,
MAP_FIXED | MAP_PRIVATE, fd, fileOffset);
        if ( loadAddress == ((void*)(-1)) ) {
            dyld::throwf("mmap() error %d at address=0x%08lX, size=0x%08lX segment=%s in
Segment::map() mapping %s",
                errno, requestedLoadAddress, (uintptr_t)size, segName(i), getPath());
        }
    }
  …
}
```

crashIfInvalidCodeSignature

- because the executable segment with the load commands had **vmsize=0** it is never in the segment list and therefore never touched in here
- therefore there is no crash on invalid code signature

```
int ImageLoaderMachO::crashIfInvalidCodeSignature()
{
    // Now that segments are mapped in, try reading from first executable segment.
    // If code signing is enabled the kernel will validate the code signature
    // when paging in, and kill the process if invalid.
    for(unsigned int i=0; i < fSegmentsCount; ++i) {
        if ( (segFileOffset(i) == 0) && (segFileSize(i) != 0) ) {
            // return read value to ensure compiler does not optimize away load
            int* p = (int*)segActualLoadAddress(i);
            return *p;
        }
    }
    return 0;
}
```

parseLoadCmds

- is called by **instantiateFinish**
- it should set the **fMachOData** pointer to the mapped executable segment
- but because our LC segment with **vmsize=0** is never in that list this never happens
- this means **fMachOData** keeps pointing to the stack copy of mach-o header
- **there will never be access to an executable segment = code signing bypassed = WIN !!!**

```
void ImageLoaderMachO::parseLoadCmds()
{
    // now that segments are mapped in, get real fMachOData, fLinkEditBase, and fSlide
    for(unsigned int i=0; i < fSegmentsCount; ++i) {
        ...
        // some segment always starts at beginning of file and contains mach_header and ..
        if ( (segFileOffset(i) == 0) && (segFileSize(i) != 0) ) {
            fMachOData = (uint8_t*)(segActualLoadAddress(i));
        }
    }
```

Conclusion

- it was trivial to bypass the dynamic linkers security checks
- the trick was to give the load command segment a **vmsize=0**
- Apple accidentally killed that trick by enforcing that **vmsize > filesize**
- one less way attackers can use to persist on iDevices to surveil you
- while this fix is most probably an accident Apple did good here :)

Questions

?