iOS Application Code Integrity

An application anti-tampering solution

Christos Koninis 2/3/2021

- Apps from the App Store are encrypted and re-signed by Apple
- When launching an App, iOS performs validations to ensure that the code is intact
- iOS guarantees **integrity** and **isolation** of your application
  - Sandboxed
  - Must use APIs to communicate with system/other apps
  - The code that runs is the code you provided

- But when the environment is **compromised** all the iOS protections are circumvented
- In **Jailbroken** or affected by **exploit** devices (Insomnia exploit*)
  - Your app has no protections
  - No one will validate that you application code/data has not been tampered with

* https://googleprojectzero.blogspot.com/2019/08/a-very-deep-dive-into-ios-exploit.html

Technical Impact

- If can attacker can tamper with your app and modify how you code works then
  - Add/Remove features (Remove adds, by-pass code that detect successfully in-app purchases, re-package and re-distribute app)
  - Steal User’s personal information

Business Impact

- Revenue loss due to piracy
- Damage to reputation

How can an attacher modify an App

- An attacker can **inject/modify** code with 3 ways
  - method hooking/swizzling
  - binary patching
  - dynamic memory modification

Solutions?

- But I will not allow my app to run on jailbroken devices!

```
if github_library.isJailbroken() {
  //crash!
}
```

But…

- The problem is if someone can tamper with our app’s code, he can **remove** the **protections**

So…

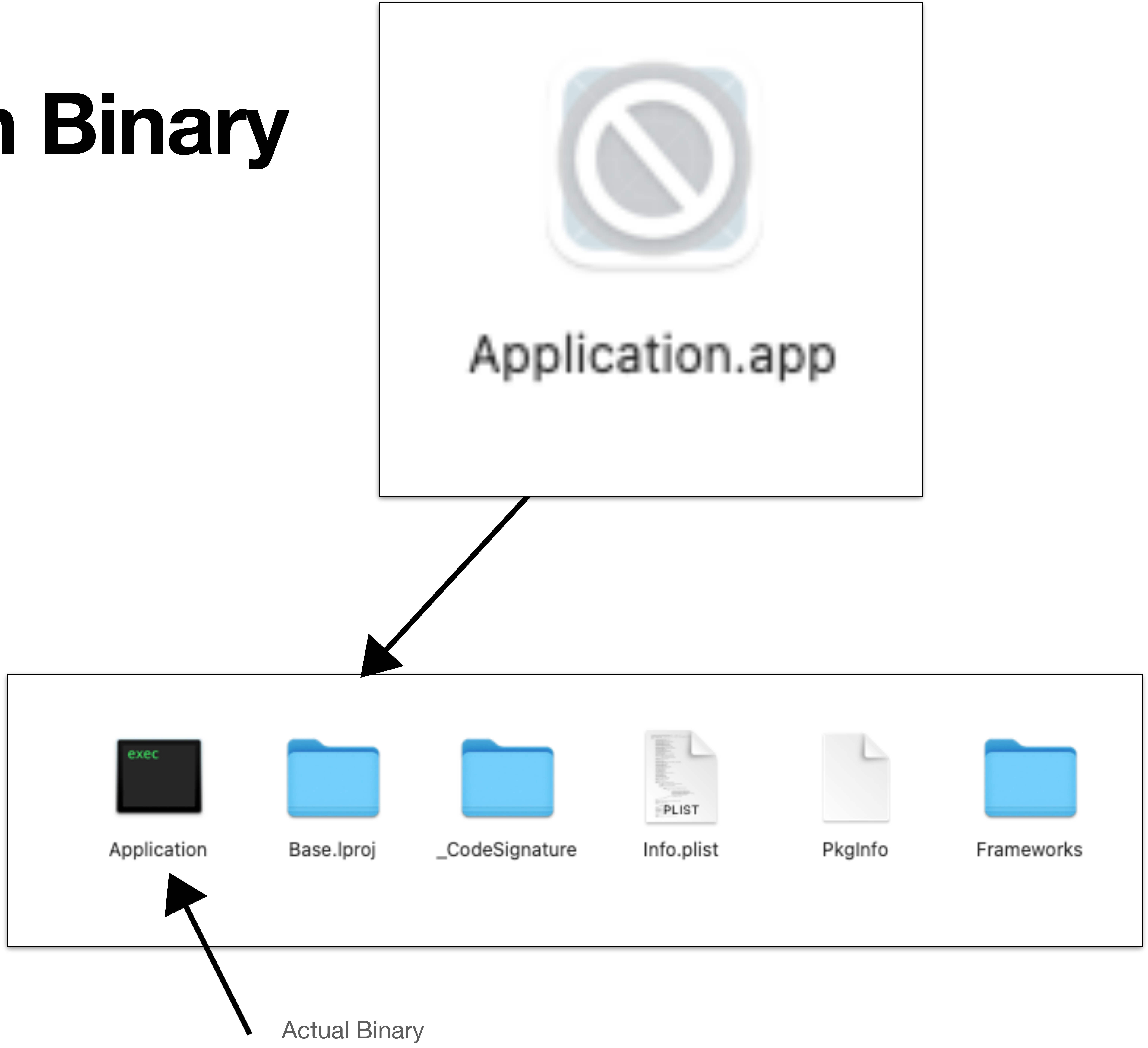
- Relying on off-the-shelf Open Source libraries for protection can give false sense of security
  - The more well-known & **popular** are the solutions the greater the possibility the attacker has made code to by-pass them
- That is why in this presentation I will focus on the basic build blocks and methodology in order to implement you own solution

A solution

- We need to **validate** that: the code that is executing is the one we compiled
  - i.e. **validate the application integrity**
- Create a build phase script that after the compilation (and before the signing)
  - Calculates the hash(sha1) of your code
  - Then adds this hash to your App’s binary
- Add method(s) that will be called during the app’s launch (and preferably at other random times)
  - It will calculate the hash of **current** running code and compare the original

The Application Binary

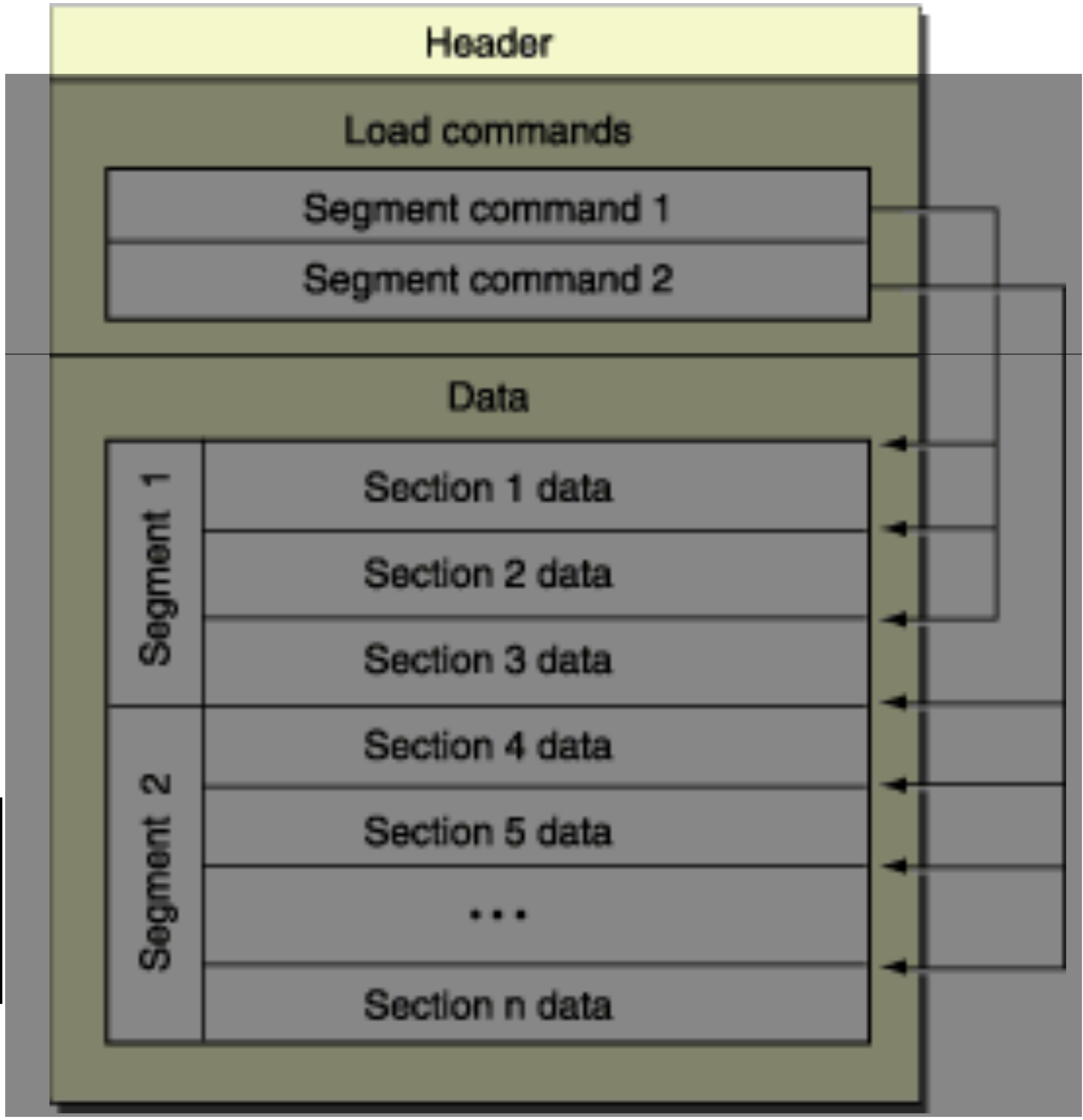
- The .app is a bundle
- Inside is the Application binary
- The binary format is **Mach-O**
- This is the format for macOS and iOS



The Mach-O binary structure - Header

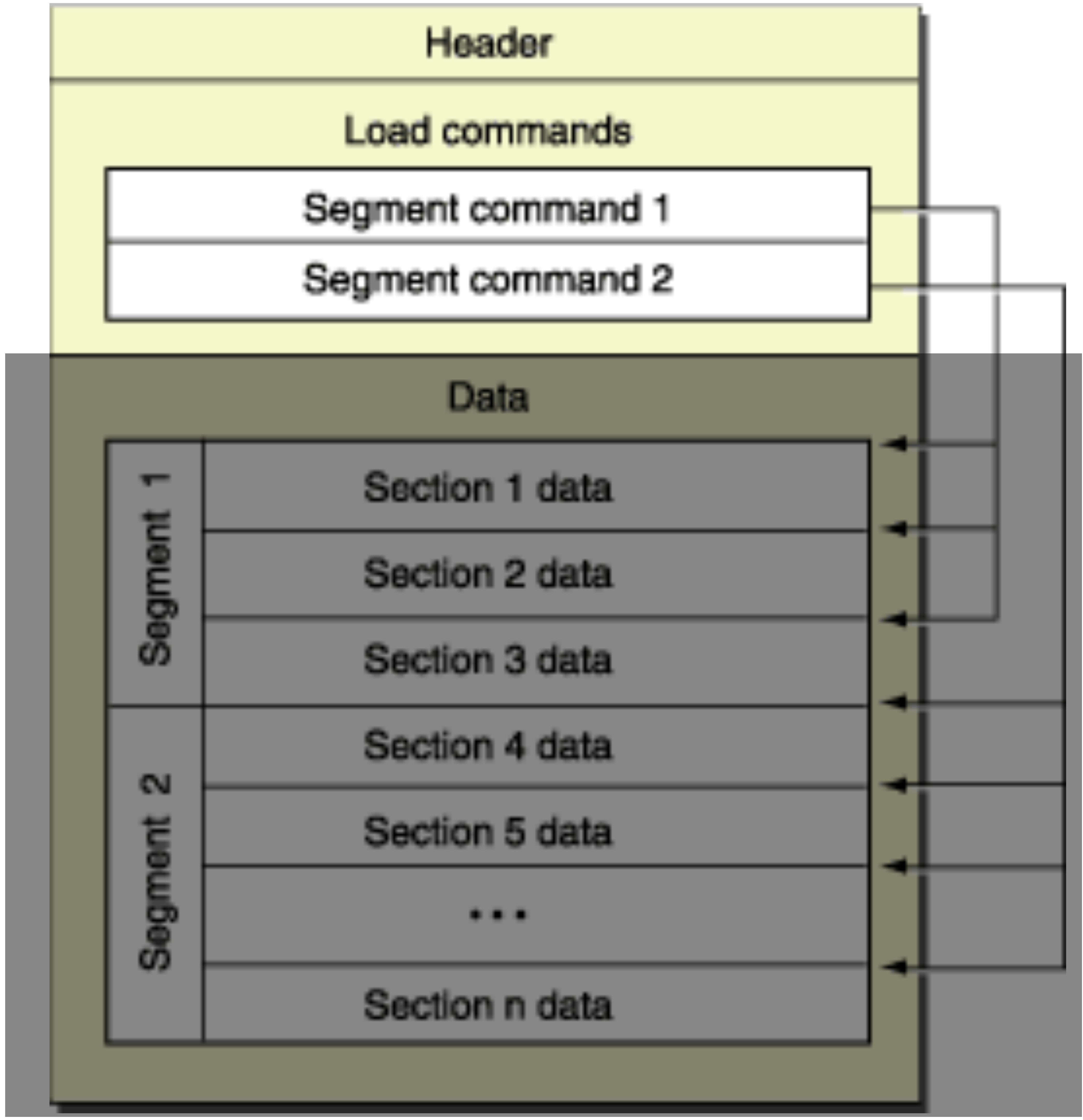
- Contains general information about file
- Target architecture / cpu
- And the number of load commands that follow
- You can use the otool -h to inspect the header:

```
christoskoninis ~ integrity otool -h /Users/christoskoninis/Desktop/Application.app/Application
/Users/christoskoninis/Desktop/Application.app/Application:
Mach header
      magic  cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777223          3  0x00           2    32       4144 0x00200085
```



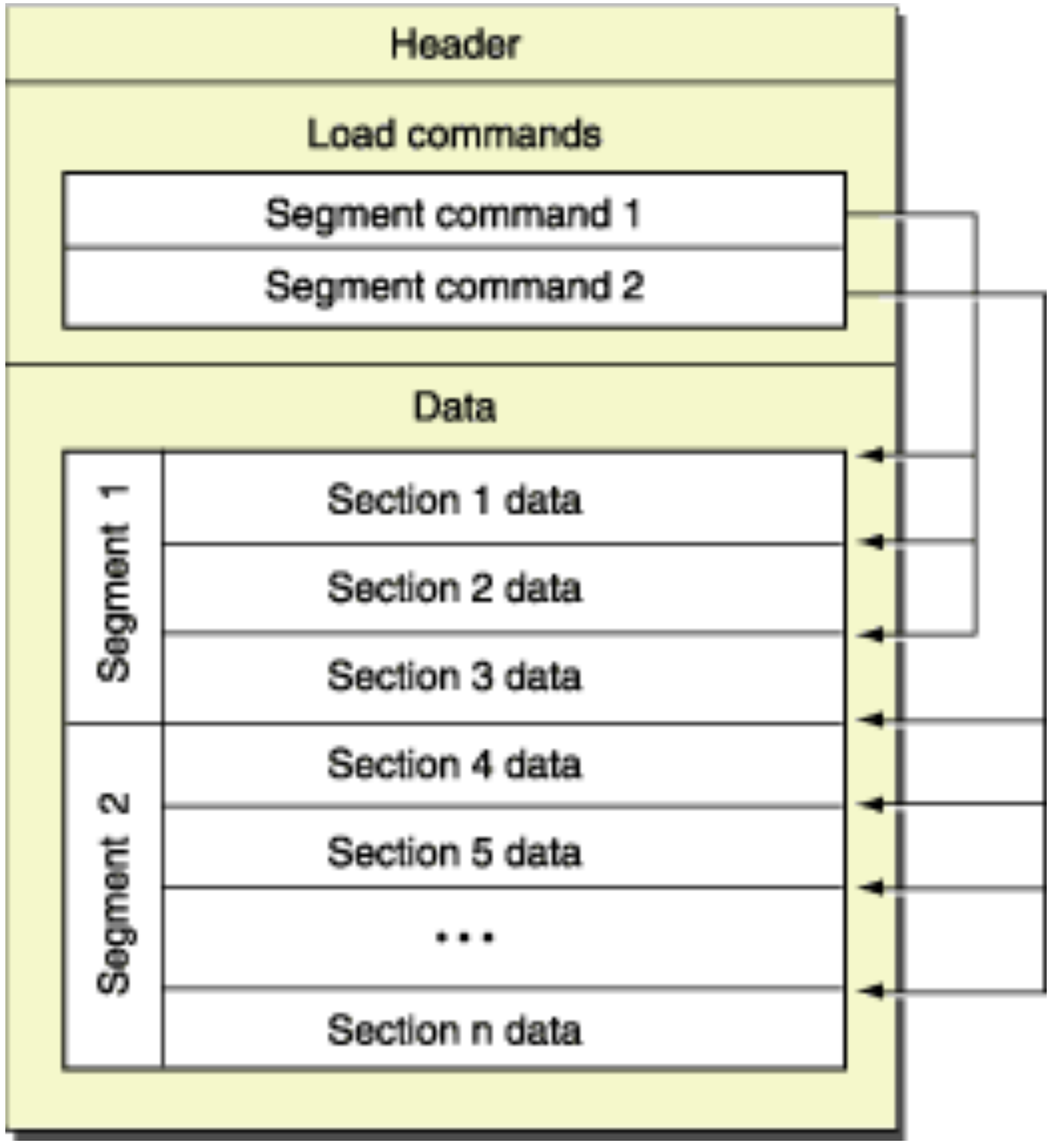
The Mach-O binary structure - Load Commands

- Specify important data that needs to be loaded, where to find them and how to map them to memory
  - **LC_MAIN**: Specifies the entry point of the program. In our case, this is the location of the main() function.
  - **LC_LOAD_DYLIB**: Load a dynamically linked shared library. e.g. /System/Library/Frameworks/UIKit.framework/UIKit or /usr/lib/swift/libswiftCore.dylib
- You can use the otool -l to inspect the LC



The Mach-O binary structure - Data

- The rest of the binary is organised in **Segments**
- Each Segment can have one or more **Sections**
- **__TEXT**: A segment for executable code and other read-only data
    - **__text**: A section for executable machine code
    - **__cstring**: A section for constant C-style strings (like "Hello, world!\n\0").
- The _TEXT seg is loaded by the **LC_SEGMENT_64** load command



Solution steps revised

- Create a build phase script that after the compilation but before the signing
- Calculates the checksum(sha1) for the binary’s machine code (**The __text section only of the __TEXT segment**)
- Add this checksum to your App’s binary (in the **__cstring section of the __TEXT segment**)
- Add a method that will be called during the app’s launch (and preferably at other random places in the app) and will calculate the current checksum and compare the original

- Lets try to tamper with the binary again but now with our **code integrity** checks enabled

Are we safe now?

- What prevents the attacker from removing the code integrity checks?
- Nothing, we just made it a little more **costly** for the attacker that needs to spend more **time**
- What you “buy” with **ALL** security solutions is always **TIME**

Make it more difficult for the attacker

- Try implementing your solutions and add the similar checks in multiple parts of the app
- Obfuscate your code(e.g. swiftshield)
- There are commercial products that work at the compiler level to add overlapping checks at multiple locations, that differ at every build

Limitations of the solution

- Must disable bitcode
- The POC version support only one slice(architecture) in the fat binary
- No a defence for a determined and skilled attacker

References & Resources

- **POC code sample in github**: https://github.com/csknns/AppIntegrity
- **Hopper disassembler:** https://www.hopperapp.com
- **Mach-O executables**: https://www.objc.io/issues/6-build-tools/mach-o-executables/
- **Mach-O structure reference**: https://github.com/aidansteele/osx-abi-macho-file-format-reference, https://www.reinterpretcast.com/hello-world-mach-o
- **OWASP**: https://wiki.owasp.org/index.php/OWASP_Reverse_Engineering_and_Code_Modification_Prevention_Project
- **iOS Security suite**: https://github.com/securing/IOSSecuritySuite
- **Injecting dynamic libraries**: https://blog.timac.org/2012/1218-simple-code-injection-using-dyld_insert_libraries/