# One Quick Way to Drastically Reduce your iOS App's Download Size

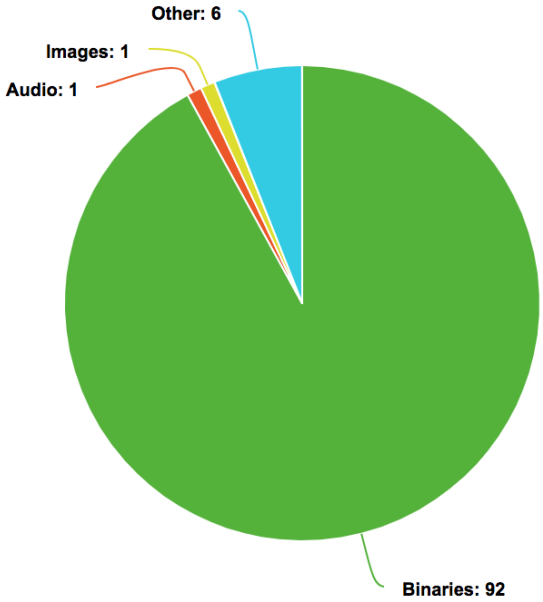
Michael Eisel
Oct 18, 2018 · 6 min read

**Before you read this article: Apple says that it will reduce the download size of all apps by 50% by "packaging" them differently. They are probably fixing the problems that this article provides workarounds for, making this article irrelevant. However, I will leave the article up for general interest.**

App download size is important. The larger your app's download size, the fewer users will download it. Plus, if it goes past 150 MB, then the App Store won't allow users to download it over WWAN.

## What determines an app's download size?

Let's look at the Telegram app (thinned for the iPhone X) and see how much each file type contributes:




Telegram app payload

As you can see, it's dominated by binaries (the main binary, frameworks, and extension binaries). Binaries usually make up 60%+ of an app's download size.

## Why are my binaries so big? Is there really that much code? Guess it's because I'm a 10x programmer 😏

Code can certainly add up, but part of the issue is that Apple encrypts a large portion of the binary before it's added to the compressed payload that the user downloads. Encrypted data looks like random data to the compression program, so it can't compress it at all.



## But encryption is good because it will make my app more tamper-proof, right?

Actually, the encryption is almost useless! You can easily decrypt any app on a jailbroken phone with the help of Clutch. Plus, how important is tamper-proofing for your app in particular?

## How much does the encryption add?

For a typical app, the encrypted portion takes around 70% of the binary (and the binary is in turn, let's say, is 80% of the app). If unencrypted and then compressed, its size would drop by about 60%. So we have 70% of 80% of 60%, i.e. **a ballpark reduction of 34% of your app's overall size**. You can use this script to approximate the savings. You can find more details about what exactly is encrypted in the "Overview of an iOS binary" section below.

### Conclusion: your users are downloading a massively inflated app payload, for no gain!

## Solution: move everything out of the encrypted part of the binary

### Overview of an iOS binary

Each binary is organized into a number of chunks called segments. Each segment consists of sections. There's a section for the code, a section for constant strings, etc. Apple encrypts the `__TEXT` segment, and nothing else. If we move all the sections out of that segment, then the segment will take up almost no space, and encrypting it won't have much impact on our app. The sections can be moved to a new segment, where they won't be encrypted and can be properly compressed. The decompressed size of our app will not change, but the compressed size will.

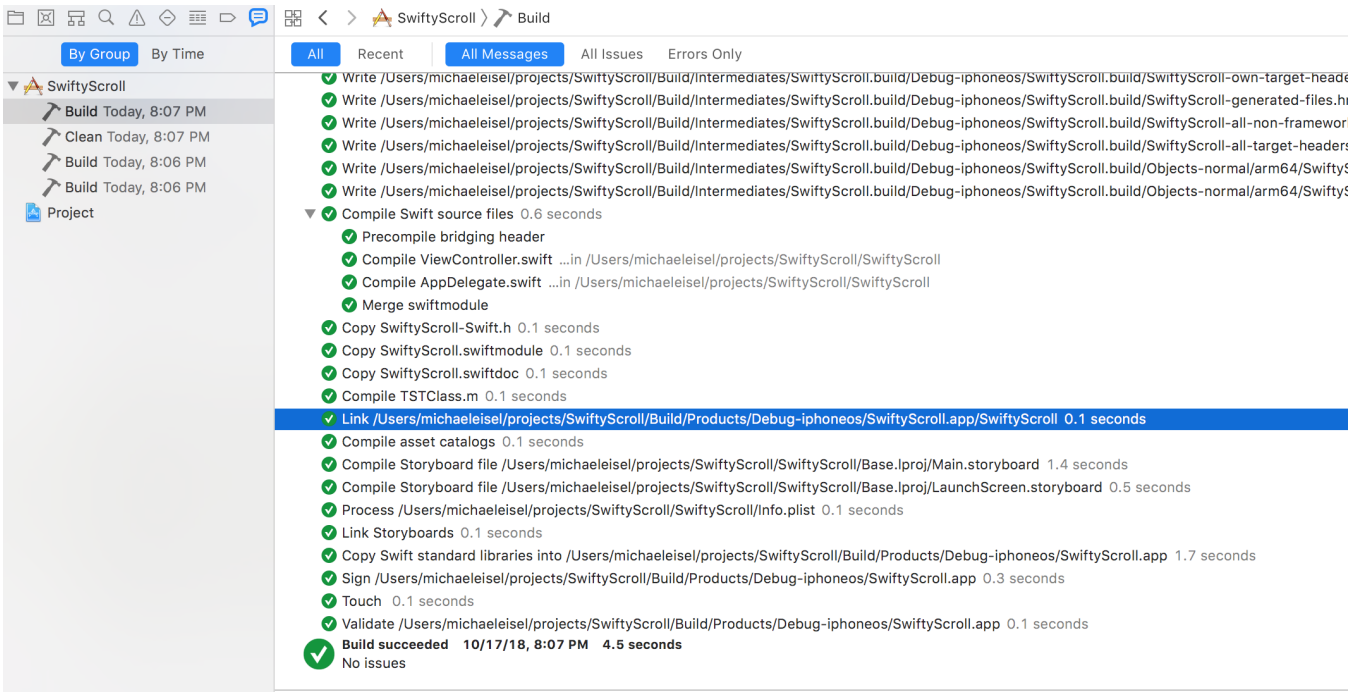
Why not just get rid of `__TEXT` entirely? There may be a way to do it, but when I tried, I encountered a number of problems, so we'll just move the sections.

## 1. Disable bitcode, if it's on

Our linker flags won't work with bitcode, so we'll have to turn bitcode off. Are there any drawbacks to turning off bitcode? In my opinion, not really, but for more information, read through this article. Plus, if you decide you want bitcode for some later release, you can always just undo this trick.

## 2. Examine the current state of your app

Get the path to your executable from the build log:




Then, run `xcrun size -x -l -m <path to your app>` and you should get:

```
Segment __PAGEZERO: 0x100000000 (vmaddr 0x0 fileoff 0)
```

```
Segment __TEXT: 0x8000 (vmaddr 0x100000000 fileoff 0)

Section __text: 0x1fb0 (addr 0x10000407c offset 16508)

Section __stubs: 0x240 (addr 0x10000602c offset 24620)

... more sections

Segment __DATA: 0x4000 (vmaddr 0x100008000 fileoff 32768)

...
```

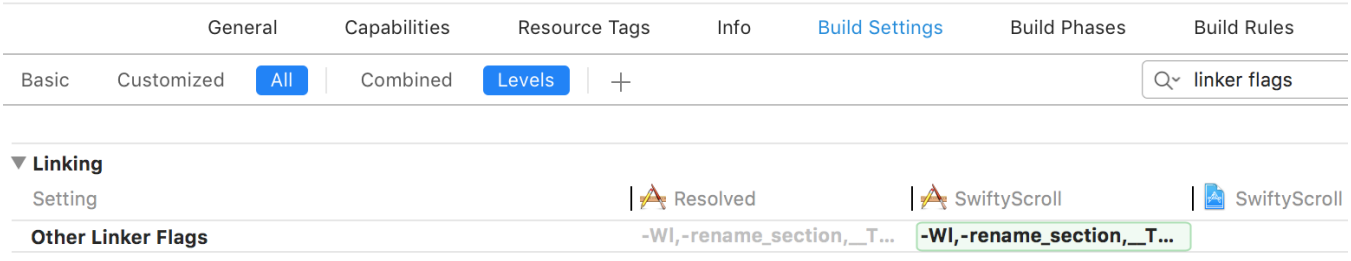
We can see that the `__TEXT` segment has a number of sections (in the lines below `Segment __TEXT: ...` and above `Segment __DATA: ...`).

## 3. Figure out the linker flags

We'll call our new segment `__MY_TEXT`. For each section in the `__TEXT` segment, we'll add a flag, `-rename_section __TEXT <section name> __MY_TEXT <section name>`, to the linker invocation. Also, we need to add `-segprot __MY_TEXT rx rx` to give our new segment `rx` permissions (read and execute, but not write). Lastly, to pass a flag to the linker during an Xcode build, it needs to get changed from `<arg1> <arg2> …` to `-Wl,<arg1>,<arg2>,…`.

You can get a quick set of flags that will probably do the job with this gist. Or, you can be more thorough and run this script for your binary, which will give you flags for all the sections.

To add these flags, go to the Xcode build settings for your target and add them under "Other Linker Flags":




## 4. Verify

Now run another build and run the `xcrun size …` command again to see if it worked. You should see something like:

```
Segment __PAGEZERO: 0x100000000 (vmaddr 0x0 fileoff 0)
Segment __TEXT: 0x4000 (vmaddr 0x100000000 fileoff 0)
Segment __DATA: 0x4000 (vmaddr 0x100004000 fileoff 16384)
Section __got: 0x88 (addr 0x100004000 offset 16384)
Section __la_symbol_ptr: 0x180 (addr 0x100004088 offset 16520)
... more sections
Segment __MY_TEXT: 0x4000 (vmaddr 0x100008000 fileoff 32768)
Section __text: 0x1fb0 (addr 0x100008000 offset 32768)
Section __objc_methname: 0xa2a (addr 0x100009fb0 offset 40880)
... all the sections previously in __TEXT
Segment __LINKEDIT: 0x10000 (vmaddr 0x10000c000 fileoff 49152)
total 0x10001c000
```

Caveat: in my experience, the `__const` section is not always moved. Oh well, it's not a big section anyways.

### 5. Rinse and repeat for other binaries in your app, if necessary

Although doing this for the main binary will usually be the biggest win by far, you can also do this for other binaries in your app, if you have any. The two categories that come to mind are app extensions (like a share extension) and dynamic libraries AKA dylibs (such as the Swift dylibs). For the extensions, you can simply repeat the above process. For dylibs, you will only be able to repeat the above process if you're doing the linking for them yourself (meaning you can't do this for Swift dylibs). I haven't tested the process for these other binaries, but I imagine it would be equally smooth.

## Voila!

Now you have an app that will be compressed properly, at minimal cost to your its security. If you'd like to double-check what the users are downloading, you can download the app to a jailbroken phone or to your desktop (via iTunes).

## Disclaimer

I have verified with an app of mine that it can be distributed on the App Store with a properly compressed binary and still work. I've also played around with symbolication and Objective-C introspection, and they seem to work, too. ***However***, I can't guarantee that this is risk-free. It's possible that some crazy runtime thing in some library is affected by this. Also, Apple might someday start rejecting apps that use this technique. Use at your own risk.

Nonetheless, I'd recommend trying it for any app with a good beta testing process. If you do decide to try it, let me know how it goes in the comments section!

## Further reading

**More optimizations for the binary size**
https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/CodeFootprint.html

**An overview of Mach-O executables**
https://www.objc.io/issues/6-build-tools/mach-o-executables/

`man ld` on OSX, the man page for the linker, which has lots of cool flags

*Special thanks to Jeb Boniakowski for his feedback for this piece.*

. . .