Lane Gramling
EECS 700
Reading 22

Summary - Jekyll on iOS: When Benign Apps Become Evil

This paper focuses on the security of iOS and its procedures for maintaining the security of apps available on the app store. As explained, apps with the goal of being accepted onto the app store must complete a review process carried out by Apple. In this process, the app review inspects the binaries of every submitted app, rejecting apps that violate the app store's regulations. Presented in this paper is an explanation of how a malicious actor can slip through this process undetected, as well as a demonstration of this process in action, where the authors actually implement their attack and get approved to the store. With this attack, the paper describes a large number of possible attacks that can be used to steal private information, send messages and tweets, hijack link clicks or device drivers, and other malicious actions.

Contributions

The high-level idea of the attack model is that a malicious actor can implant remotely exploitable vulnerabilities into the app which are disguised within legitimate functionalities in small code gadgets. After doing this, the adversary can then remotely launch the attack after the app has passed the review process and landed on end user devices. At this stage, the implanted vulnerabilities can execute their malicious logic at runtime and chain together to carry out their exploits. This is described from an abstract perspective in the paper as a dynamic reconfiguration of the control flow graph from a benign form into a malicious form. This concept is one of the key contributions of this paper. In one given example of this, a method supposedly downloads 'greeting cards' from a malicious server which deliberately forces a buffer overflow, contaminating the stack layout and changing the control flow of the app to send a buffer containing the user's contacts to the server despite the proper flow of the code showing a benign failure report status being sent to the server. The reason this can be done consistently is because despite iOS' use of ASLR and DEP as security mechanisms against calculated overflow attacks, the app can deliberately leak its memory layout information to the server. Generally speaking, the attack scheme is enabled by using planted vulnerabilities to circumvent the securities of ASLR and DEP to enabled calculated overflow attacks which change the control flow of the app.

After exploring the implementation and providing a deeper elaboration on a number of the aforementioned attacks, the paper also gives a novel contribution in the form of a new proposed technique for using dynamic analysis to discover private APIs of other services which can be used to perform malicious examples. This is how the attacks for sending tweets, SMS, and emails are enabled. iOS uses Objective-C, which dispatches all object method invocations through a generic message handling function. By developing a dynamic analysis tool based on iOS' LLDB debugger, the authors wrote a script to set a conditional breakpoint at the *objc_msgSend* method that handles the message dispatching. This, in combination with a knowledge of the ARM standard calling convention for passing arguments through registers (and the stack), provides enough information for Jekyll to dynamically load the framework in question (Twitter in the paper's example) and create instances of these classes to gain access and control over the private API.

Limitations

Due to the rapid evolution of iOS, the versions investigated in the paper are already far out of date. Thus, it is feasible that the demonstrated attack has already been guarded against in subsequent versions of iOS. And if not, the attack's lifespan is inevitably limited. Additionally, the attack fundamentally relies on carefully calculated buffer overflows to get started, which are a single point that the review process already in place could expand its scrutiny and caution upon. Adding to that, it's quite a bit more work that is necessary to expand the functionality by gaining access to private APIs, which makes this attack quite difficult to implement in its full power; likewise, additional obstacles compound the difficulty due to the expensive costs implicit with registering to the app store and being approved. In reality, the attack is powerful in the world of iOS where attacks are typically few and far between, but it

requires lots of effort, money, and patience for the malicious actor to implement - only to inevitably be shot down in one of the frequent updates to iOS.

Future Work

One place where more work could be done on this lies in the responsibility of Apple. They could implement extra checks to the review process that increase the level of scrutiny given to where buffer overflows may occur in code. Additionally, more stringent policies could be emplaced on communications with third-party APIs, such as the one used in the paper's demonstration to set off the initial chain reaction. On the offensive side, it would be interesting to see a redux of this attack method to see whether it is still feasible after the 6+ major updates* to iOS that have happened since the initial investigation was done.

* This amount of change is why it seems so uncertain that the attack is still effective at least without evolving to keep up