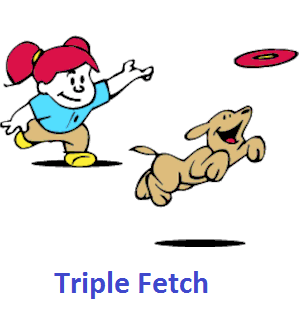
Triple\_Fetch Analysis and another iOS NSXPC Sandbox Escape Vulnerability

Author: Min(Spark) Zheng (SparkZheng@twitter)

0x00 Introduction

Ian Beer@google released an exploit of CVE-2017-7047 Triple\_Fetch [1] in the last week. Then, chenliang@keenlab wrote a Chinese article to analyze it [2]. However, there are lots of highlights we could learn in the vulnerability and exp. In addition, Triple\_Fetch vulnerability and exploit technique are very similar to another NSXPC sandbox escape vulnerability we used in our private OverSky jailbreak. Therefore, we plan to write this article to analyze Triple\_Fetch and share another NSXPC vulnerability – MLSqlite SBE.

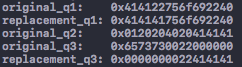
0x01 CVE-2017-7047 Triple\_Fetch Vulnerability



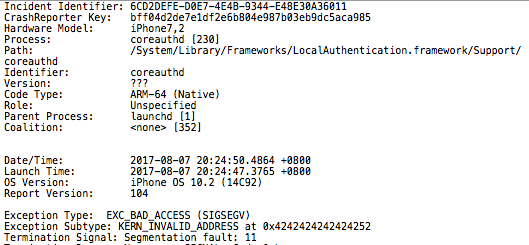
Chenliang and beer has a very detailed write up, so I only briefly summarize the vulnerability here. Because of the efficiency issue, for OS\_xpc\_data which is larger than 0x4000 in XPC translation, iOS system will use mach\_vm\_map to map the memory using physical pages. This means the sender and receiver shared same physical pages of memory. If the sender changes the shared memory, the receiver’s memory will be changed as well.

Therefore, using race condition, we could make the receiver get different data and the receiver may treat them as the same one. It’s will cause some serious problems. For example, the sender first sends @”ABCD” (includes @ and “) to the receiver, the receiver will allocate 7 bytes buffer for this string. However, the receiver will copy the string to the buffer until it meets ‘“’. If we change the string to @"ABCDOVERFLOW\_OVERFLOW\_OVERFLOW" after the allocation, it will cause a heap overflow in the process of the receiver.

The function Triple\_Fetch chooses to attack is \_\_\_NSMS1() in the CoreFoundation frameowork. This function reads the shared string multi-time. If the sender changed the shared string during the process, it may cause a logic error of the function and then trigger the heap overflow. By using three groups of strings, it’s possible for the attacker to control the pc register of the receiver. That’s why this exploit is called Triple\_Fetch. The picture shows three groups of strings which are used for the exploit:



The exploit chooses to attack the “com.apple.CoreAuthentication.daemon” service (/System/Library/Frameworks/LocalAuthentication.framework/Support/coreauthd), because this process has root privilege and can get other processes’ send right by using processor\_set\_tasks()[3]. The picture shows the crash report of coreauthd:



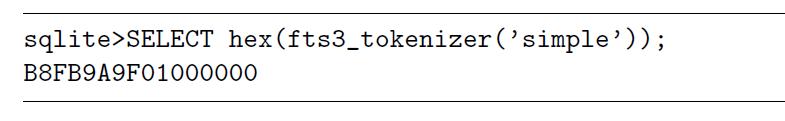
0x02 MLSqlite Sandbox Escape Vulnerability

This vulnerability was found by Team OverSky (Cererdlong, Eakerqiu, Min (Spark) Zheng) and it is used in our OverSky private jailbreak. Because Apple enhanced the sandbox rules in iOS 10, the vulnerability cannot be triggered inside the sandbox now. But this vulnerability can still be used in iOS 9.3.5 which is the latest iOS version of iPhone 4s.

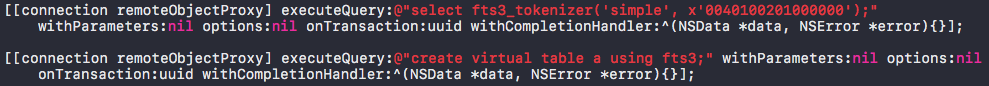
The vulnerability was found in com.apple.medialibraryd.xpc NSXPC service (/System/Library/PrivateFrameworks/MusicLibrary.framework/Support/medialibraryd). This NSXPC service has an interface related to sqlite. Because there is no privilege check for the caller, we could use [[connection remoteObjectProxy] beginTransactionForDatabaseAtPath] NSXPC interface to create and open arbitrary sqlite files on the iOS system. In addition, we can use [[connection remoteObjectProxy] executeQuery] NSXPC interface to execute the query command on the sqlite file.

However, the query command is not enough for us to control the pc register. Lucky, there is a classic SQLite fts3\_tokenizer vulnerability on iOS. A talk from Chaitin introduce the usage of this vulnerability in browsers on BH2017[4]. Today, we will introduce another usage of this vulnerability to do sandbox escape on iOS. An FTS tokenizer is a set of rules for extracting terms from a document or basic FTS full-text query. It provides built-in "simple" and other tokenizers and also provides an interface for applications to implement and register custom

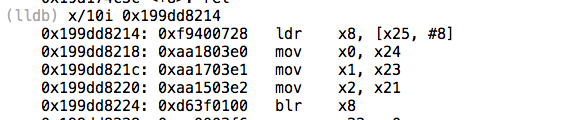
tokenizers. Therefore, we could use fts3\_tokenizer('simple') to leak the memory address:



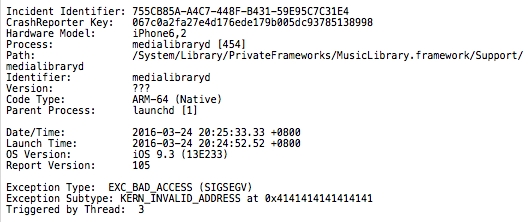
In addition, we could use fts3\_tokenizer('simple’, addr) to control the virtual table and then use “create virtual table a using fts3;” to control the pc register. The picture shows related exploit code:



In the debugging, we can find x25 register value is controlled by us:



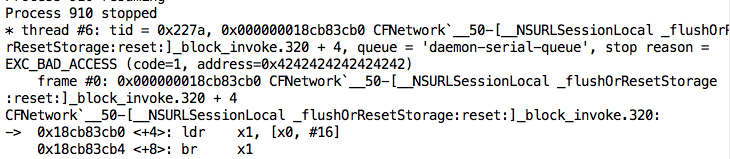
And, the picture shows the crash report of medialibraryd:



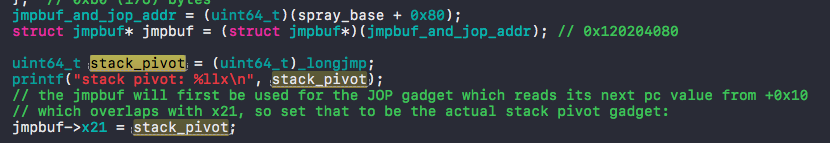
We will introduce how to do JOP and ROP later.

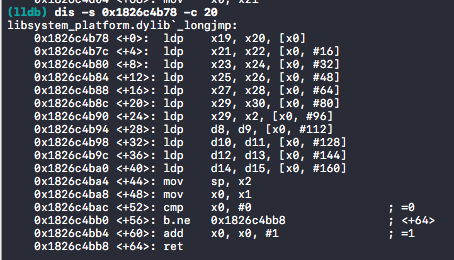
0x03 Triple\_Fetch JOP & ROP & Arbitrary Code Execution

Although you can control the pc register through Triple\_Fetch vulnerability, you cannot control the stack. Therefore, we need to do stack pivot first. A good news is the X0 register which points to the xpc\_uuid object is controlled by us:

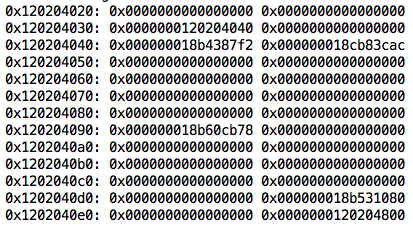


Therefore, we can use the \_longjmp function to do the stack pivot using JOP. Then we can control the stack:

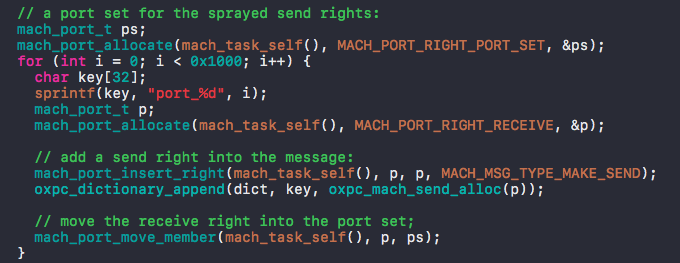




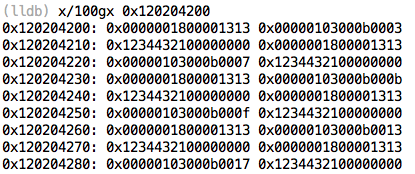
The layout of xpc\_uuid object and the JOP we used to do stack pivot are shown below:

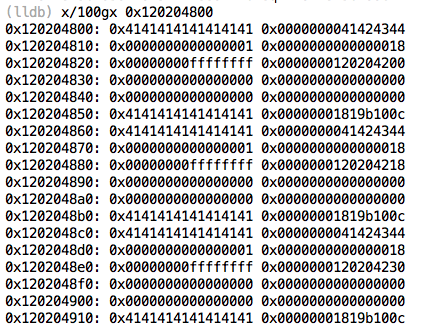


It’s easier for us to write ROP when we control the stack. But the goal of beer is not only ROP, but arbitrary code execution as well. Therefore, beer’s exp used an exploit technique to get the task port of the receiver. Besides the fake xpc\_uuid object, the exploit also send 0x1000 mach msgs with send right port to the target process:.

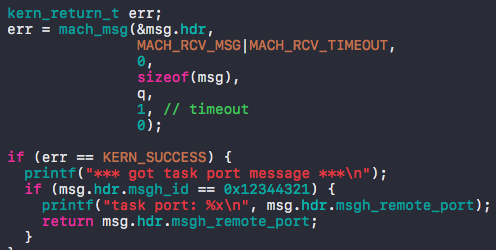


The content and location of these mach msgs (with 0x12344321 msgh\_id) are shown below:



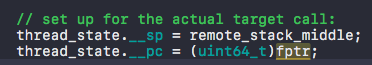
After that, the exp used ROP to traverse the Mach msgs and send them back to the attacker: 

Then, the attacker will receive the Mach msg. If it receives the Mach msg with 0x12344321 msgh\_id, it means the attacker get the task port of the target process successfully:



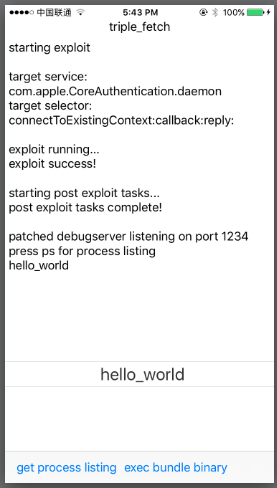
After getting the task\_port (in sploit()), the exploit moves to the next step - do\_post\_exploit(). do\_post\_exploit() also did a lot of things. Firstly, it uses the task port of coreauthd and processor\_set\_tasks() to get the task port list of all processes.

So how does it work? Because we have the task port of coreauthd, we could use mach\_vm\_\* API to modify the memory and register values of coreauthd. So, we can allocate a block of memory for the stack of function and then set the pc register to the function pointer we want to invoke. Therefore, we can let the target process to execute any function we wish to. The implementation can be found in call\_remote() function:



Then, the exploit lets coreauthd execute a serial of functions (e.g., task\_get\_special\_port(), processor\_set\_default(), host\_processor\_set\_priv() and processor\_set\_tasks()) to get the task port list of all processes and send it back to the attacker. Then, the exploit will search for the task ports of amfid, launchd, installd and springboard in the task port list. After that, it will patch amfid using the same trick in mach\_portial. Last but not least, it starts debugserver.

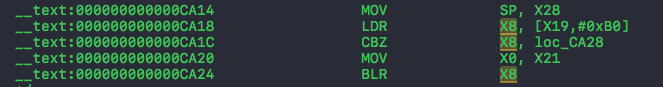
In fact, the exploit not only can execute debugserver, but other binaries outside the sandbox as well. The only thing you should do is to swap the bin file in the pocs folder, recompile the app and then click exec bundle binary:



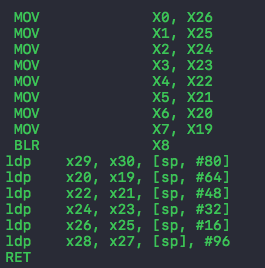
The magic is in the spawn\_bundle\_binary(). The function first executes “chmod 0777” for the third party bin and then invokes a serial of posix\_spawn(like fork()) APIs to execute the third party bin.

0x04 MLSqlite JOP & ROP & Arbitrary Code Execution

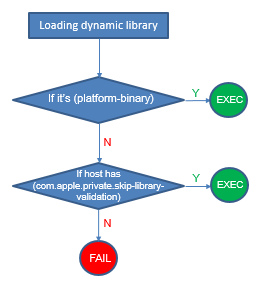
Because there is no exploit code for us to refer, in our MLSqlite SBE exploit, we choose another stack pivot in the frameworks:



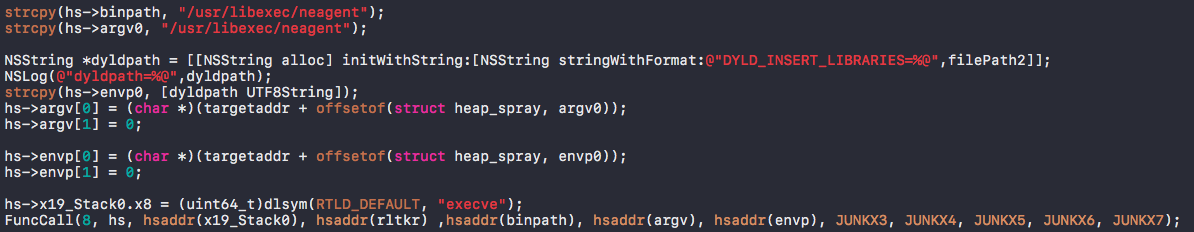
In order to set the value of X28, we found a very long JOP chain (with 5 gadgets) to do the stack pivot and then used a universal gadget in the CoreFoundation.framework to call functions through ROP:



We know that using ROP to write a program is inefficient. Therefore, we need to find a way to do arbitrary code execution. However, iOS denied to execute or load any binary outside the sandbox unless it has “platform-binary” team-id. But, there is one exception: “com.apple.private.skip-library-validation” entitlement. After searching the iOS system, we found that neagent is the only one with this entitlement. Because neagent needs to load third-party VPN libraries, it must has this entitlement.



Therefore, injecting a dynamic library (through DYLD\_INSERT\_LIBRARIES) into neagent process, we can achieve arbitrary code execution (not ROP) outside the sandbox:



Code execution outside the sandbox opens more interfaces to attack the kernel. In addition, we could read and modify lots of files on the iOS system through the SBE vulnerability. For example, the attacker can read personal files (e.g., SMS, chat logs and photos) and then send them to the hacker’s server:



0x05 Conclusion

In this article, we describe two NSXPC vulnerabilities respectively: the Triple\_Fetch vulnerability found by Beer and the MLSqlite vulnerability found by us. Then we introduce two ways to do stack pivot through JOP and gain arbitrary code execution through ROP. However, if we want to control the kernel, we still need one or two vulnerabilities in the XNU or IOKit. In addition, Apple fixed the kpp bypass technique which are used in the yalu 102. Therefore, even we have Triple\_Fetch exploit, there is still a long way for jailbreakers to finish the jailbreak on iOS 10.3.2.

0x06 Reference

1. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1247>
2. <http://keenlab.tencent.com/zh/2017/08/02/CVE-2017-7047-Triple-Fetch-bug-and-vulnerability-analysis/>
3. <http://newosxbook.com/articles/PST2.html>
4. <https://www.blackhat.com/docs/us-17/wednesday/us-17-Feng-Many-Birds-One-Stone-Exploiting-A-Single-SQLite-Vulnerability-Across-Multiple-Software.pdf>