Tufts University

Computer Science

Introduction to Computer Security

Fuzzing an iOS Application

Author:
Aaron Wishnick

Mentor:
Ming Chow

December 13, 2013

Abstract

The question of iOS and mobile security in general has been on the minds of many people since the creation of the smartphone. The smartphone is essentially a computer that we carry around with us everywhere we go and that stores our most personal information, so naturally it has become a large target for malicious intent. And with apps such as Google Wallet coming out on the market now, it is even more important that these devices be secure. In this paper I will discuss iOS security and, more specifically, exploiting apps using a technique known as fuzzing.

1 Introduction

Since the creation of the iPhone in 2007, the amount of smartphones in existence has increased to almost 25% of the mobile market. This has forced smartphone makers and more specifically for this paper, Apple, to focus a large amount of time on making their devices more secure.

1.1 Securing iOS

The first version of iOS began as simply a stripped down version of OS X and came with essentually no added security measures. And since at the time very few people carried iPhones, this was not a huge problem. But when popularity of the device sky-rocketed, the first big step they took towards securing it was introducing the dual-core processor which caused many pre-existing exploits to become much less reliable. Another large step they took was disabling java and flash from being able to run, both of which have a history of security vulnerabilities, thereby greatly decreasing the attack surface of iOS when compared to a much heavier operating system like OS X. They also removed the shell from the original operating system preventing any possible execution of malicious shell scripts, which is often the goal of an OS X exploit.

The last two security measures I am going to cover are code signing and sandboxing, these two have made it very difficult, but not impossible, to discover vulnerabilities in iOS. Code signing means that in order for an application to run on an iPhone it must (in most cases) have come from the Apple App Store. And in order for an application to be available on the app store, it must first have passed an inspection to make sure the app does not have malicious intent. Once accepted to the app store all components of the app receive a specific signature which must be available in order for it to execute. What this means for security is that even if the app is able to download malicious, executable content on to the phone after being accepted by the App Store, that content will not get executed because it is not signed. This eliminates a large number of potential exploits. And in the unlikely case that an attacker is able to create an app that successfully downloads and executes malicious content, sandboxing prevents the damage from spreading. The term sandboxing refers to the organizational structure of applications within the iOS operating system. Each app exists in its own "sandbox" where it can do whatever it wants within that box, but cannot venture very far outside of it. There are certain things, such as photos and contacts, which are accessible to all applications (with user permission), but for the most part sandboxing prevents apps from talking to each other. For example, an application would be able to download all of a person's contacts but would not be able to send text messages from that phone, because that function is reserved for the messages app.

There have been many more actions taken by Apple to protect the iPhone from attacks, the one's listed above are just a few of the more notable ones.

1.2 Exposing iOS Vulnerabilities

It would appear, then, that Apple has made their operating system impenetrable. However, while the attacks are different from that of OS X, they do exist. One way to find a vulnerability within iOS is simply brute force. Jose "Barraquito" Rodriguez has employed this method many times throughout his career as a Spanish soldier while waiting around in cars. He is the person credited with discovering the lock-screen vulnerability within iOS 7. Clearly, this is not the most efficient way to discover vulnerabilities. Fuzzing, the topic of this paper, is the next step up. It is still a brute force attack, the difference being that the brute forcing is done by the computer and not by the attacker himself. This paper will discuss fuzzing in greater detail later on.

Another, much more difficult attack utilizes return-oriented programming(ROP). This is a technique by which the attacker inserts and executes machine instructions, known as "gadgets", into the call stack. There are a few reasons why this exploit is so difficult on iOS: 1. due to code signing, the entire exploit must be written in ROP and 2. it is absolutely necessary to understand both the ARM architecture basics and the calling convention used on iOS[1]. However, in his article on exploiting the iPhone, Dan Goodin points out that Apple has done attackers the favor of leaving in a few functions from the original OS X operating system that allow easier access to the kernel, where all of the ROP will take place. One of these leftovers is the kernel debugger, which serves no purpose within the iOS operating system. But if an attacker is able to force a crash within the kernel, the debugger will give him access to the CPU allowing him to read/write memory and read/change register values. Essentially all the bases are now belong to the

[1]Charlie Miller, Chapter 8 - Return-Oriented Programming

attacker.

The final attack this paper will cover is the baseband attack. This refers to an attack on the internal chip - the digital baseband processor. This is the chip that interfaces between the actual phone component within the iPhone and cellular towers operated by the carrier. To attack a device over the air, an adversary would operate a rogue base station in close enough proximity to the target device such that the two can communicate [2]. Setting up a base station is as simple as downloading open-source software such as OpenBTS [8] and running it on a computer. Once the attacker gets another person's phone on their base station instead of their carrier's, he/she will be able to corrupt the memory and execute custom malicious code on the baseband processor.

2 Fuzzing

Fuzzing is a form of vulnerability testing that can be used to test any kind of application (web,mobile,etc...). It is a very important tool that anyone planning on creating an application, iOS or not, should know about. It is of course not the end all be all of penetration testing, but it is a really good place to start, excellent for exposing very serious vulnerabilities. The concept behind it is simple: repeatedly send malformed data to the application in the hopes(or not) that it causes it to crash. Since this is essentially all that is involved in fuzzing, it is important to note then, that source code is not required, making it an easier place for an attacker to start. There are two forms of fuzzing: mutation-based and generation-based, each with it's benefits and detriments. A key concept to note is that without a program running that monitors the crash, it doesn't matter which technique of fuzzing is used because the attacker won't know what causes a crash or any information about the crash.

2.1 Mutation-Based Fuzzing

Mutation-based fuzzing could be considered the naive way of approaching fuzzing. It requires little to no working knowledge of the input being provided and can be done in under 100 lines of code (see 6.1). As the figure indicates, the fuzz_buffer function takes in a buffer and than selects a random number of

[2]Charlie Miller, Chapter 11 - Baseband Attacks

bits to switch and then assigns that number of bits to random values between 0 and 256. So the ease in which this was created is a clear benefit to this form of fuzzing. However, the program is not knowledgable about the specific protocal being fuzzed and therefore is unable to really delve into potentital application crashing bugs, which is why it often referred to as a "dumb" fuzzer. As technology and security is improved, mutation-based fuzzing is becoming less effective. However on earlier versions of Mac OS X(<10.5.7) and iOS(<2.2.1), it was able to expose the Adobe JBIG2 vulnerability [3]. This attack exposed a vulnerability where an image that was compressed with JBIG2 and then corrupted with a mutation-based fuzzer and then passed to the Adobe Reader which used the JBIG2Decoder caused the reader to crash. By embedding a trojan within the file the attacker was then able to gain remote access [4]. And so while mutation-based fuzzing did in fact work for that specific vulnerability, more often than not, a more in-depth knowledge of the protocol being fuzzed is required, and for that we use generation-based fuzzing.

2.2 Generation-Based Fuzzing

This form of fuzzing requires a very full understanding of how the specific file format/protocal works, which is why it is often referred to as "smart" fuzzing. This is because rather than simply taking a file and mutating it, it is in fact generating the files from scratch, based on the specifics of each component of whatever is being fuzzed. To gain a better understanding let us return to the PDF example. With the mutation-based fuzzer, the attacker simply passed a valid pdf file through the python script below which flipped some bits and then spit it back out again slightly corrupted. However, with generation-based fuzzing, the attacker would need to go through the entire PDF specification provided by adobe[5] and then construct a fuzzer that would test each and every component of the pdf, like the JBIG2Decoder. The benefit to this type of fuzzing is that it is a much more in depth fuzz, and since it is not random, it is more likely to find vulnerabilities. However it is very difficult and time consuming to get it set up. For this a number of security

[3]Miller, "Adventures in PDF Fuzzing", Chapter 6 - Fuzzing iOS Applications

[4]See `http://secunia.com/secunia_research/2009-24/` for more details on the exploit

[5]See http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf for full spec

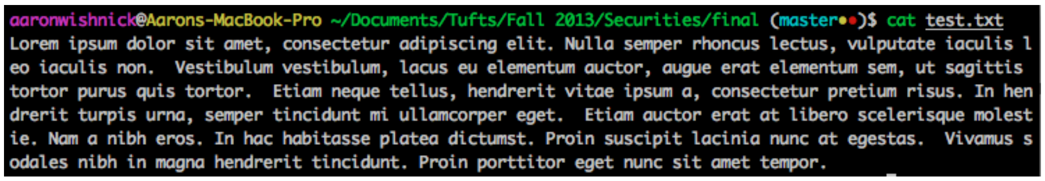
software developers have created generation-based fuzzing frameworks which provide an easier way to accomplish this.

3 Creating Fuzzed Inputs

So now that we have seen the different types of fuzzing, we will go into further depth on ways to go about creating fuzzed files to test an application.

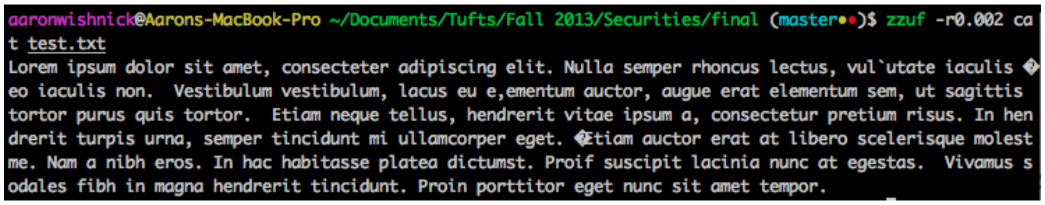
3.1 ZZUF

ZZUF [10] is a mutation-based fuzzing tool that focuses on switching bits in order to corrupt the input data. It is essentially a framework to do a more complex version of what the python function in section 6.1 does. It allows the user to control how much of the data gets changed by having them declare a ratio upon running it. This ratio is the percentage of bits that will get changed.



```
aaronwishnick@Aarons-MacBook-Pro ~/Documents/Tufts/Fall 2013/Securities/final (master••)$ cat test.txt
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla semper rhoncus lectus, vulputate iaculis l
eo iaculis non.  Vestibulum vestibulum, lacus eu elementum auctor, augue erat elementum sem, ut sagittis
tortor purus quis tortor.  Etiam neque tellus, hendrerit vitae ipsum a, consectetur pretium risus. In hen
drerit turpis urna, semper tincidunt mi ullamcorper eget.  Etiam auctor erat at libero scelerisque molest
ie. Nam a nibh eros. In hac habitasse platea dictumst. Proin suscipit lacinia nunc at egestas.  Vivamus s
odales nibh in magna hendrerit tincidunt. Proin porttitor eget nunc sit amet tempor.
```

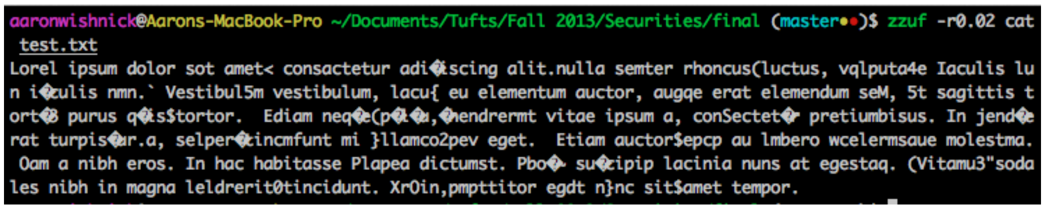
Figure 1: This is the input text to be passed into ZZUF.



```
aaronwishnick@Aarons-MacBook-Pro ~/Documents/Tufts/Fall 2013/Securities/final (master••)$ zzuf -r0.002 ca
t test.txt
Lorem ipsum dolor sit amet, consecteter adipiscing elit. Nulla semper rhoncus lectus, vul`utate iaculis �
eo iaculis non.  Vestibulum vestibulum, lacus eu e,ementum auctor, augue erat elementum sem, ut sagittis
tortor purus quis tortor.  Etiam neque tellus, hendrerit vitae ipsum a, consectetur pretium risus. In hen
drerit turpis urna, semper tincidunt mi ullamcorper eget. �Etiam auctor erat at libero scelerisque molest
me. Nam a nibh eros. In hac habitasse platea dictumst. Proif suscipit lacinia nunc at egestas.  Vivamus s
odales fibh in magna hendrerit tincidunt. Proin porttitor eget nunc sit amet tempor.
```

Figure 2: Running zzuf with a ratio of .002 or .2%.

As the figures indicate, increasing the ratio does indeed cause more of the characters to be altered. This tool is useful for a quick fuzzer to generate a lot of test cases, however, does not dig past a very high level perspective of the file. For that we will use a generation-based fuzzing framework.



```
aaronwishnick@Aarons-MacBook-Pro ~/Documents/Tufts/Fall 2013/Securities/final (master••)$ zzuf -r0.02 cat
 test.txt
Lorel ipsum dolor sot amet< consactetur adi�scing alit.nulla semter rhoncus(luctus, vqlputa4e Iaculis lu
n i�culis nmn.` Vestibul5m vestibulum, lacu{ eu elementum auctor, augqe erat elemendum seM, 5t sagittis t
ort�B purus q�is$tortor.  Ediam neq�e(p�t�u,�hendrermt vitae ipsum a, conSectet� pretiumbisus. In jend�e
rat turpis�ar.a, selper�tincmfunt mi }llamco2pev eget.  Etiam auctor$epcp au lmbero wcelermsaue molestma.
 Oam a nibh eros. In hac habitasse Plapea dictumst. Pbo� su�cipip lacinia nuns at egestaq. (Vitamu3"soda
les nibh in magna leldrerit0tincidunt. XrOin,pmpttitor egdt n}nc sit$amet tempor.
```

Figure 3: Running zzuf with a ratio of .02 or 2%.

3.2 Sulley

Sulley [12] is a python framework created by Pedram Amini and Aaron Portnoy, co-authors of "Fuzzing: Brute Force Vulnerability Discovery". Despite the fact that many fuzzing frameworks already existed, Pedram and Aaron felt that they all came with their caviats. A very popular fuzzer called SPIKE is said to be easy to use which likely is the reason for its popularity. However this ease of use did not come without sacrifice, and in fact resulted in a lot of missing functionality. On the other side there is Peach Fuzzer, which, while very powerful, came with a very steep learning curve. And there are many in between. This is where Sulley comes in. Sulley answers the ease of use issue first by being written in python, which is not only powerful but also a very easily understandable language. It also utilizes a block based data representation, similar to SPIKE, which allows for easy computations on specific sections of the protocol being fuzzed. Lastly, Sulley primitives are built to be simple at first but allow for much more complex work by including optional arguments. There are a number of other features which this paper will not address such as crash monitoring, metrics and others, but are good to note to get a full understanding of how powerful Sulley is.

In order to demonstrate the capabilities of Sulley, included is an example of an exploit done by Charlie Miller where he used Sulley to fuzz sms on an iPhone (see section 6.2). The first function "eight_bit_encoder" is used throughout the sulley code as a way of encoding specific bits of data. From then on it is the actual Sulley code. It follows a very clean organizational structure, defining each block of data at a time and at the end combining it all together in one complete fuzzer. It begins with initializing the current "request" as it is referred to by Sulley, in some cases there may be multiple requests which get combined into a session. Then each sequence of s_size, s_block_start, and s_block_end are creating the required components of the text message (sender, recipient, date, message). And that is all, this program will create a series of invalid text messages which can be tested on the device. One feature of Sulley's approach that is very useful is that the s_size function contains a "fuzzable" parameter, which tells Sulley whether or not the program should fuzz the input. This features allows one to test for the efficacy of one's logic before actually creating fuzzed data.

4 Testing the Fuzzed Inputs

Now that we have a large set of invalid data to be tested, we need to figure out if it will cause the application to crash. Unfortunately this part of the problem seems to be a lesser of all evils decision, in that there is no one great way to test the fuzzes, and often it depends on the specific project.

4.1 The iOS Simulator

There are a few different ways in which the fuzzes that have been created can be tested. The first is to test using the iOS simulator. However there are a number of issues with doing it this way. First of all, since the emulator is running on the x86 operating system of Mac OS X, it is not entirely accurate with how the ARM core would handle the inputs. Another issue is that the simulator is very limited in the apps that are running on it, so unless the source code of the application is available or it is one of the preinstalled applications on teh simulator, this will not be a possible medium for testing.

4.2 The Desktop

An alternative is that if the application is a shared application between OS X and iOS, such as MobileSafari, it is possible to test in Safari on the Mac. But this one also comes with problems, the main one being that since MobileSafari is a lighter weight version of Safari, it is not capable of handling all of the same file formats, or may simply handle them differently. There are a few other examples that overlap, such as the Office quick view, where on a Mac a Microsoft Office document can be previewed before opening it, this is also used in iOS to render Office documents within MobileSafari. So it is possible to fuzzes on the desktop version, and it is quite likely (in this case) that any vulnerability present there would also be present on the iOS device, since it is essentially the same.

4.3 The Device

So since the simulator and desktop are really not great options, the next thing to do is to run the fuzzes straight on the device. Depending on the application being fuzzed this may or may not be difficult. If the application belongs to the tester, then he/she can simply attach some test inputs to the input debugger. However, if the application being tested is one in which the source is not available, this is where problems present themselves. For something like MobileSafari where it is connected to the outside world, a "dumb" fuzzer could be created where a website is hosted with all of the fuzzed inputs and then javascript requests each one until it crashes or runs through all of them. This is problematic because it is time consuming both on the testers end, having to create a website and upload all of the files to be tested, and on the testing end, potentially having to make thousands of requests. So clearly this is not ideal. There are also proprietary forensics tools, such as Elcomsoft iOS Forensic Toolkit [15], but that requires having $1495 lying around. And lastly, Charlie Miller suggests using a jailbroken iOS device, installing a shell and then putting a custom executable that can open a program from the command line on the device. This will esentually emulate the way fuzzing is done on the desktop, and if possible (which on the current version - 7.0.4, it is not) is probably the best approach.

5 Conclusion

Fuzzing is a very powerful tool for vulnerability testing in general and more specifically for testing iOS applications. It can be used for testing apps within their specific sandbox, and also as a way of determining vulnerabilities within the kernel itself. Especially when testing a personal application, fuzzing is an excellent way to start. And depending on the complexity of the inputs which are being testing, it will determine whether mutation-based or generation-based fuzzing will be necessary. There are a number frameworks out there to help one get started fuzzing and many are simply a matter of preference. However one does it, it is an invaluable technique to have in a vulnerabilitiy testing toolbelt.

6 Supporting Material

6.1 Mutation-Based Fuzzer

The following is code taken from Charlie Miller's, iOS Hacker's Handbook:

```
import random
import math
import subprocess
import os
import sys

def fuzz_buffer(buffer, FuzzFactor):
    buf = list(buffer)
    numwrites=random.randrange(math.ceil((float(len(buf)) /
       FuzzFactor)))+1
    for j in range(numwrites):
        rbyte = random.randrange(256)
        rn = random.randrange(len(buf))
        buf[rn] = "%c"%(rbyte);
        return "".join(buf)

def fuzz(buf, test_case_number, extension, timeout, app_name):
    fuzzed = fuzz_buffer(buf, 10)
    fname = str(test_case_number)+"-test"+extension
    out = open(fname, "wb")
    out.write(fuzzed)
    out.close()
    command = ["./crash", app_name, fname, str(timeout)]
    output = subprocess.Popen(command,
                       stdout=subprocess.PIPE).communicate()[0]
    if len(output) > 0:
        print "Crash in "+fname
        print output
    else:
        os.unlink(fname)

if(len(sys.argv)<5):
    print "fuzz <app_name> <time-seconds> <exemplar>
       <num_iterations>"
    sys.exit(0)
else:
    f = open(sys.argv[3], "r")
    inbuf = f.read()
    f.close()
    ext = sys.argv[3][sys.argv[3].rfind(.  ):]
    for j in range(int(sys.argv[4])):
        fuzz(inbuf, j, ext, sys.argv[2], sys.argv[1])
```

6.2 Generation-Based Fuzzer

This is a generation-based fuzzer built with Sulley to fuzz sms messages:

```
def eight_bit_encoder(string):
        ret =
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret

s_initialize("query")

s_size("SMSC_number", format="oct", length=1, math=lambda x: x/2)
if s_block_start("SMSC_number"):
        s_byte(0x91, format="oct", name="typeofaddress")
        if s_block_start("SMSC_data", encoder=eight_bit_encoder):
                s_string("\x94\x71\x06\x00\x40\x34", max_len =
256)
        s_block_end()
s_block_end()

s_byte(0x04, format="oct", name="octetofsmsdeliver")

s_size("from_number", format="oct", length=1, math=lambda x: x-3)

if s_block_start("from_number"):
       s_byte(0x91, format="oct", name="typeofaddress_from")
       if s_block_start("abyte2", encoder=eight_bit_encoder):
              s_string("\x94\x71\x96\x46\x66\x56\xf8", max_len =
256)
      s_block_end()
s_block_end()

s_byte(0x0, format="oct", name="tp_pid")
s_byte(0x04, format="oct", name="tp_dcs")

if s_block_start("date"):
     s_byte(0x90, format="oct")
     s_byte(0x10, format="oct")
     s_byte(0x82, format="oct")
     s_byte(0x11, format="oct")
     s_byte(0x42, format="oct")
     s_byte(0x15, format="oct")
     s_byte(0x40, format="oct")
s_block_end()

if s_block_start("eight_bit"):
       s_size("message_eight", format="oct", length=1, math=lambda
          x: x / 2, fuzzable=True)
       if s_block_start("message_eight"):
              if s_block_start("text_eight",
encoder=eight_bit_encoder):
                     s_string("hellohello", max_len = 256)
              s_block_end()
       s_block_end()
s_block_end()

fuzz_file = session_file()
fuzz_file.connect(s_get("query"))
fuzz_file.fuzz()
```

References

[1] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philipp Weinmann. 2012. iOS Hacker's Handbook (1st ed.). Wiley Publishing.

[2] Miller, Charlie, and Collin Mulliner. "Fuzzing the Phone in Your Phone." Purdue, n.d. Web. <https://engineering.purdue.edu/dcsl/reading/2011/jevin-fuzzing.pdf>.

[3] McNally, Richard, Ken Yiu, Duncan Grove, and Damien Gerhardy. "Fuzzing: The State of the Art." Australian Government. Department of Defense, n.d. Web.

[4] Lang, Chieh-Jan M., Nicholas D. Lane, Niels Brouwers, Li Zhang, Borje Karlsson, Ranveer Chandra, and Feng Zhao. "Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions." Microsoft. Microsoft, n.d. Web.

[5] "The BUZZ on FUZZING." Buzz on Fuzzing: Introduction to Fuzzing. N.p., n.d. Web. 11 Dec. 2013. <http://www.codenomicon.com/products/buzz-on-fuzzing.shtml>.

[6] Garg, Parul. "Fuzzing Mutation vs. Generation." InfoSec Institute. N.p., 4 Jan. 2012. Web. 11 Dec. 2013. <http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/>.

[7] Goodin, Dan. "Exploit Writer Spills Beans on Secret IPhone Function The Register." The Register. N.p., 4 Aug. 2011. Web. 11 Dec. 2013. <http://www.theregister.co.uk/2011/08/04/secret_iphone_hacking_tool/>.

[8] BURGESS, D. A., AND SAMRA, H. S. The Open BTS project. <http://openbts.sourceforge.net/, Aug. 2008>.

[9] Weinmann, Ralf-Philipp. "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks." University of Luxembourg (n.d.): n. pag. Web. <https://www.usenix.org/system/files/conference/woot12/woot12-final24.pdf>.

[10] Caca Labs. Zuff - Multi-Purpose Fuzzer. <http://caca.zoy.org/wiki/zzuf>.

[11] Mulliner, Collin, and Charlie Miller. "Fuzzing the Phone in Your Phone." (n.d.): n. pag. Black Hat. Web. <http://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf>.

[12] Sulley - Pure Python fully automated and unattended fuzzing framework. http://code.google.com/p/sulley/.

[13] Document management Portable document format Part 1: PDF 1.7. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf

[14] Armini, Pedram, and Aaron Portnoy. Fuzzing Sucks! Introducing Sulley Fuzzing Framework. Tech. N.p., 2007. Web.

[15] Elcomsoft - iOS Forensic Toolkit. http://www.elcomsoft.com/products.html

[16] Deja vu Security - Peach Fuzzer. http://peachfuzzer.com

[17] Juuso, Anna-Maija, and Mikko Varpiola. Fuzzing Best Practices: Combining Generation and Mutation-Based Fuzzing. Tech. Codenomicon Defensics, n.d. Web. <http://www.codenomicon.com/resources/whitepapers/codenomicon-wp-generation-and-mutation.pdf>.

[18] Lukan, Dejan. "Sulley Fuzzing Framework Intro." (n.d.): n. pag. InfoSec Institute. 17 July 2012. Web. <http://resources.infosecinstitute.com/sulley-fuzzing/>.