*Journal of Information Security*, **2010, 1, 74-87**
doi:10.4236/jis.2010.12009 Published Online October 2010 (http://www.SciRP.org/journal/jis)

# iPhone Security Analysis

**Vaibhav Ranchhoddas Pandya, Mark Stamp**
*Department of Computer Science, San Jose State University, San Jose, USA*
*E-mail*: *stamp@cs.sjsu.edu*
*Received August* 10, 2010; *revised September* 21, 2010; *accepted October* 15, 2010

## Abstract

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 angered many for not allowing installation of third party applications and for working exclusively with AT & T wireless services (in the US). Besides the US, iPhone was only sold only in a few other selected countries. Software attacks were developed to overcome both limitations. The development of those attacks and further evaluation revealed several vulnerabilities in iPhone security. In this paper, we examine some 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.

**Keywords:** iPhone, Wireless, Mobile, Smartphone, Jailbreaking, Reverse Engineering

## 1. Introduction

The release of Apple's iPhone on June 29, 2007 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 [1]. By any measure, the iPhone has been a commercial success—in spite of being a first-timer in the smart phone industry, Apple immediately outpaced traditional cell phone giants like Nokia, Motorola, and LG. 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 more [2]. It has helpful widgets for maps, weather, in addition to email and other Internet capabilities [2].

### 1.1. Features

The iPhone confirms that Apple understands consumers' desires, not only in terms of functionality, but also in terms of appearance and style. While other smart phone companies have offered products that include features offered by the iPhone, none have approached the iPhone in terms of popularity and sales. Phone features include a soft keypad with the ability to easily merge calls and visually obtain voicemail information. Apple took advantage of iPod's popularity by including complete iPod functionality in the iPhone. A full-functional web browser with zoom in/out functionality made internet surfing experience on a mobile phone better than ever. The Multi-Touch touch screen display allows for gliding and scrolling besides zooming. The accelerometer detects the orientation of the phone. These features put iPhone above and beyond other smartphones such as Blackberry and Motorola Q.

### 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 [3]. **Figure 1** shows how different functions within the iPhone interface with one another [4]. **Figure 2** shows an image of the board inside an iPhone.

## 2. Motivation

iPhones are supposed to only be used with AT & T wireless service (in the US). 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 viewed this as an "unfair" move by the two companies. People felt that they should be

 

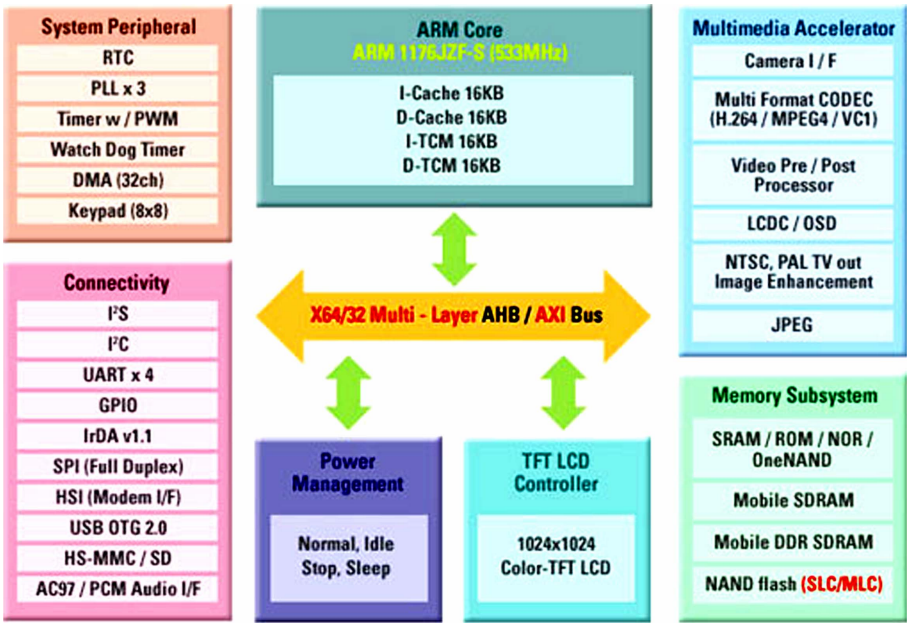


**Figure 1. iPhone architecture from a high level [4].**

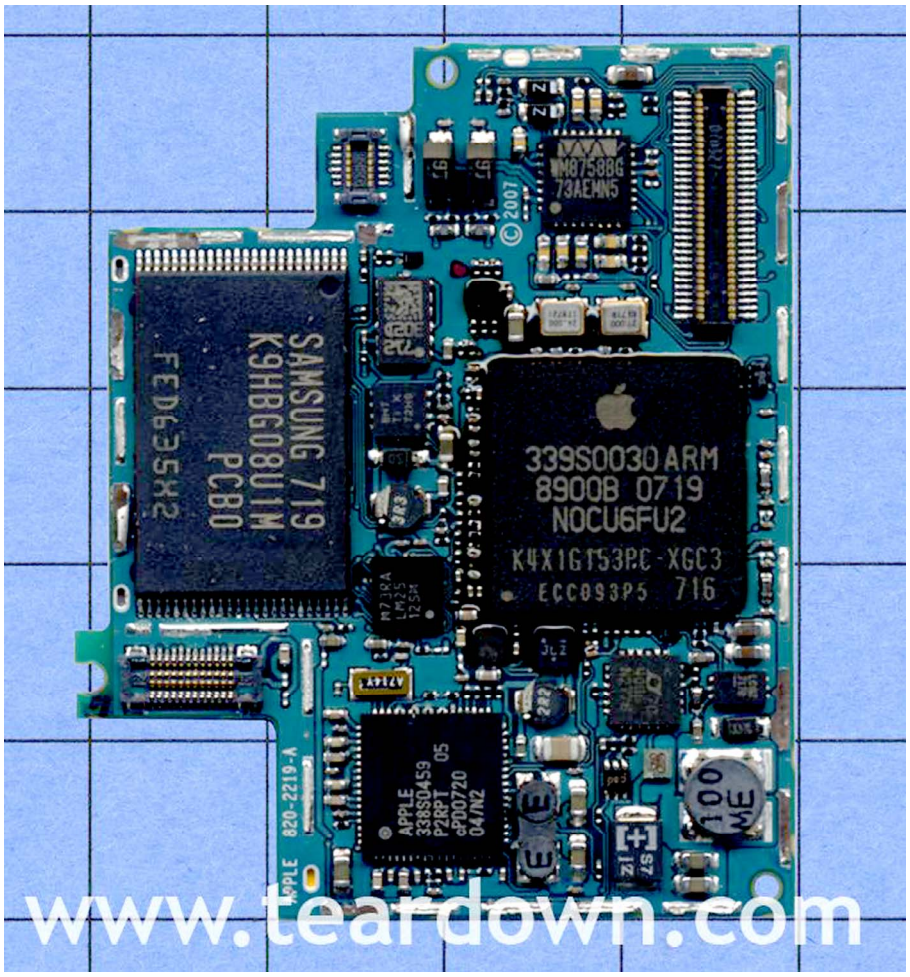


**Figure 2. Board showing different parts in iPhone.**

 

able to choose whatever wireless service they prefer and should not be forced to use a particular one.

There was another reason that some 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 who wanted to install third-party applications such as widgets and games were unable to do so.

These two limitations placed on iPhone users prompted a series of hack and attack efforts by iPhone enthuseasts 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. 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. Perhaps surprisingly, 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 discuss 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 obviously has had significant financial implications. In addition, with more millions of users worldwide, any security holes in iPhone can jeopardize the privacy of millions of people. We believe that these issues make the security analysis of iPhone a worthwhile and important topic.

## 3. Jailbreaking

The process of gaining root access to the iPhone so that third party tools can be installed is called Jailbreaking [5]. Without gaining read-write access to the root system, one cannot install third party applications. Note that this limitation prevents users from doing what they want to do with their iPhones—products that they own. This is somewhat analogous to buying a computer and not being allowed to install new programs on it. There are several websites (see, for example, [6]) that provide interesting gadgets and games for iPhone. Some of the most popular games are iSolitaire, iZoo, Tetris, iPhysics, and NOIZ2SA. Beyond providing access to such applications, jailbreaking is essential for another reason: it is the first step in unlocking.

Without jailbreaking, one cannot install the necessary application to use a wireless service other than AT & T. Close to a million new iPhones were not activated with AT & T in the first six months after its release [1]. 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 necessary to enable the addition of third party applications to the iPhone.

### 3.1. Looking for Ideas

Immediately after its release, iPhone enthusiasts and hackers all around the world were looking for a way to gain root access. 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 [7,8].

A previous hacker success was using buffer overflow techniques on the Sony PSP. By exploiting 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 otherwise prohibited [9].

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 of libtiff [10,11]. 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 [10]. This vulnerability is also documented and tracked in the U.S. National Vulnerability Database [12]. A malicious TIFF file can be created to include the desired rogue code. When attempting to view the malicious tiff file in a vulnerable version of MobileSafari, 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 a stack buffer overflow to inject code and the "return-to-libc" technique to execute it. To illustrate how a stack buffer overflow can be created and how a return-to-libc attack works, we first consider a generic example.

Consider the piece of code below [13]:

```
void func (char *passedStr) {
        char localStr[4]; // Note that only 4 bytes allocated
```

 

```
        strcpy(localStr, passedStr); // length of passedStr is not checked
}
int main (int argc, char **argv) {
        func(argv[1]);
}
```

Suppose that we have a program is called myprog. Now, let us look at a simplified representation of the stack when myprog is executed with "hi" as the input parameter—see **Table 1** below.

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

As it is clear from the tables above(**Table 1, 2**), 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 [14]. Of course, the 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 this is potentially vulnerable to a stack buffer overflow [14,15].

Instead of entering "good security," a carefully crafted string could be used. In the example above, suppose we replace "good security" with, say, "good secu\x12\x34\x56\x78." In little-endian, the last 4 bytes are 0x78563412, which might be the address of a function, say, system(). Then 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 so that desired parameter values are passed to a pre-existing function [16]. Such an attack is generally known as the return-to-libc attack. By discovering the address of such a desirable function, an attacker can potentially exploit a buffer overflow to execute the function and thereby achieve the desired behavior. Furthermore, by passing a carefully crafted malicious input that exploits a stack overflow, an attacker can even inject malicious code that results in a chain of calls to such pre-existing functions.

**Table 1. Simplified stack representation with proper input.**

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

**Table 2. Simplified stack representation with corrupting input.**

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

## 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 [10]. 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 [17]. Below we give excerpts of this function as it appears in libtiff versions 3.8.1 and 3.8.2. First, we 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. The fix is 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 discuss the TIFF format 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 [18]. 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 [19]. 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 be used to exploit a buffer overflow in poorly written programs that read and manipulate tiff images [19]. 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 ("value" column) with corresponding descriptions [18].

| **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). The IFD then consists of a sequence of tags, 12 bytes each, where the first two bytes identify the field, and the next two identify the field type: short int, long int, byte, or ASCII. The next four bytes specify the number of values, and the final four specify the value itself or an offset to the value [18]. Since TIFF files are not intended to be human-readable, their contents are best viewed in a hex editor.

### 3.5. Arm Processor

Since the ARM1176JZF-S processor 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. **Figure 3** gives a high-level diagram of ARM1176JZF-S.

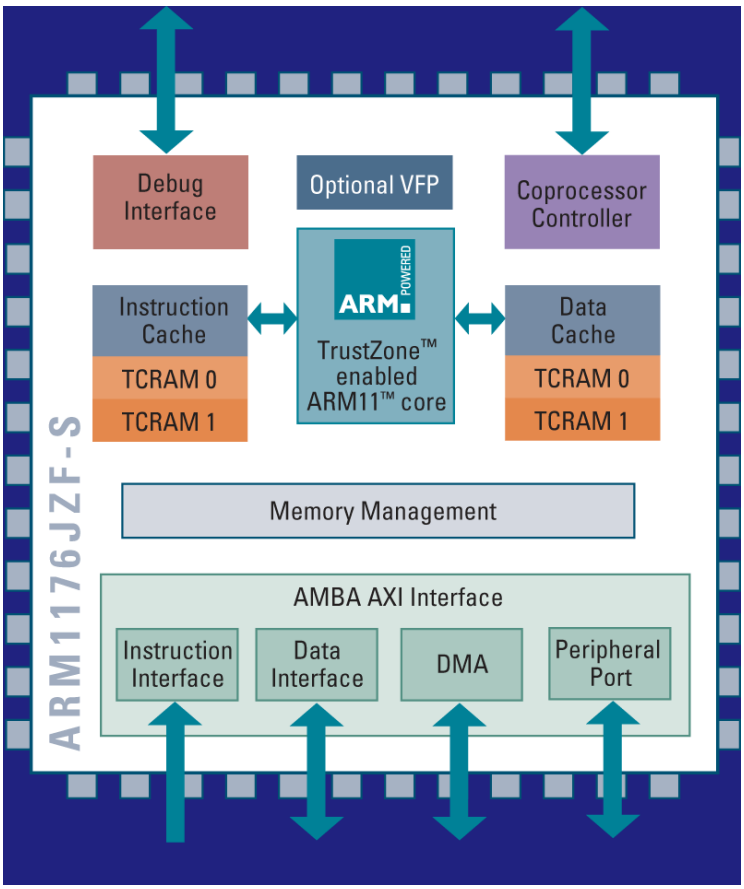


**Figure 3. ARM 1176JFZ-S processor [3].**

 

The ARM processor can be configured in either little- or big-endian modes to access its data [20]. The iPhone runs the ARM processor in little-endian mode. For example, if a value in a register is 0x12345678, in little-endian mode it appears in memory "byte-reversed", that is, as 0x78 0x56 0x34 0x12. This is illustrated in the **Figures 4** and **5** below.

The ARM processor can be configured in either little- or big-endian modes to access its data [20]. The iPhone runs the ARM processor in little-endian mode. For example, if a value in a register is 0x12345678, in little-endian mode it appears in memory "byte-reversed", that is, as 0x78 0x56 0x34 0x12. This is illustrated in the **Figures 4** and **5** below.

### 3.6. Dre And Niacin'S Tiff Exploit Jailbreak

We now have accumulated the background required to understand and reverse-engineer the libtiff exploit for jailbreaking developed by two teenagers known as Dre and Niacin. The source code for the attack is available on Dre and Niacin's website [23]. However, little explanation is provided, so we found it necessary to reverse engineer various aspects of the attack.

First, we verify and demonstrate the overflow problem. Though the exploit was created for the iPhone, we 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 badDotRange.tiff was created.

An interesting outcome occurred when we attempted to create the code badDotRange.tiff. The file creation was blocked by Norton AntiVirus software running on the machine, and it claimed the file was "Bloodhound. Exploit.166" [24]. Further information on the vulnerability shows Norton characterizing badDotRange. tiff as a Trojan and a Virus, as shown in **Figure 6** [24].

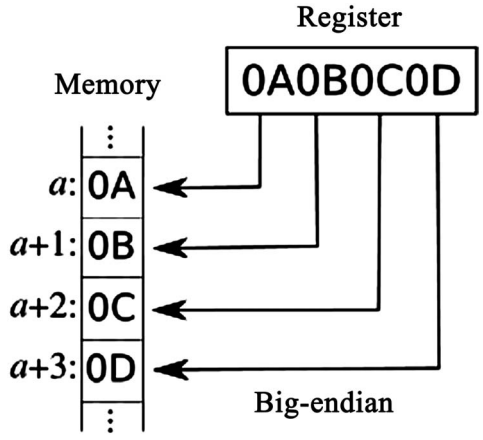


**Figure 4. Big-endian [22].**

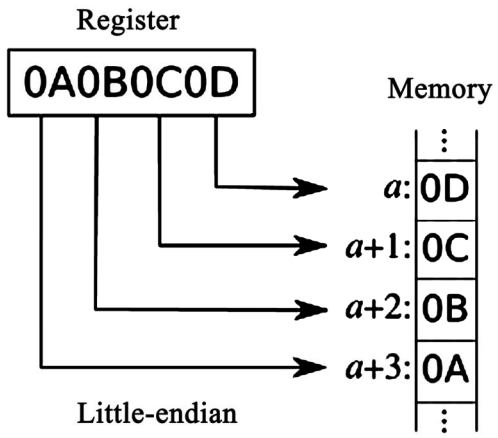


**Figure 5. Little-endian [22].**

Once the work area was put in the list of directories to be excluded by Norton AntiVirus, badDotRange.tiff was created; a hex editor view of the file is available in [25].

Next, we demonstrate the malicious TIFF file causing a buffer overflow in libtiff. We also show a well formed TIFF file being handled properly by libtiff. A program was written to simulate the stack buffer overflow. Below is a snippet from driver.cpp file.

```
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, as shown below.

```
$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 the following: 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);
```

 

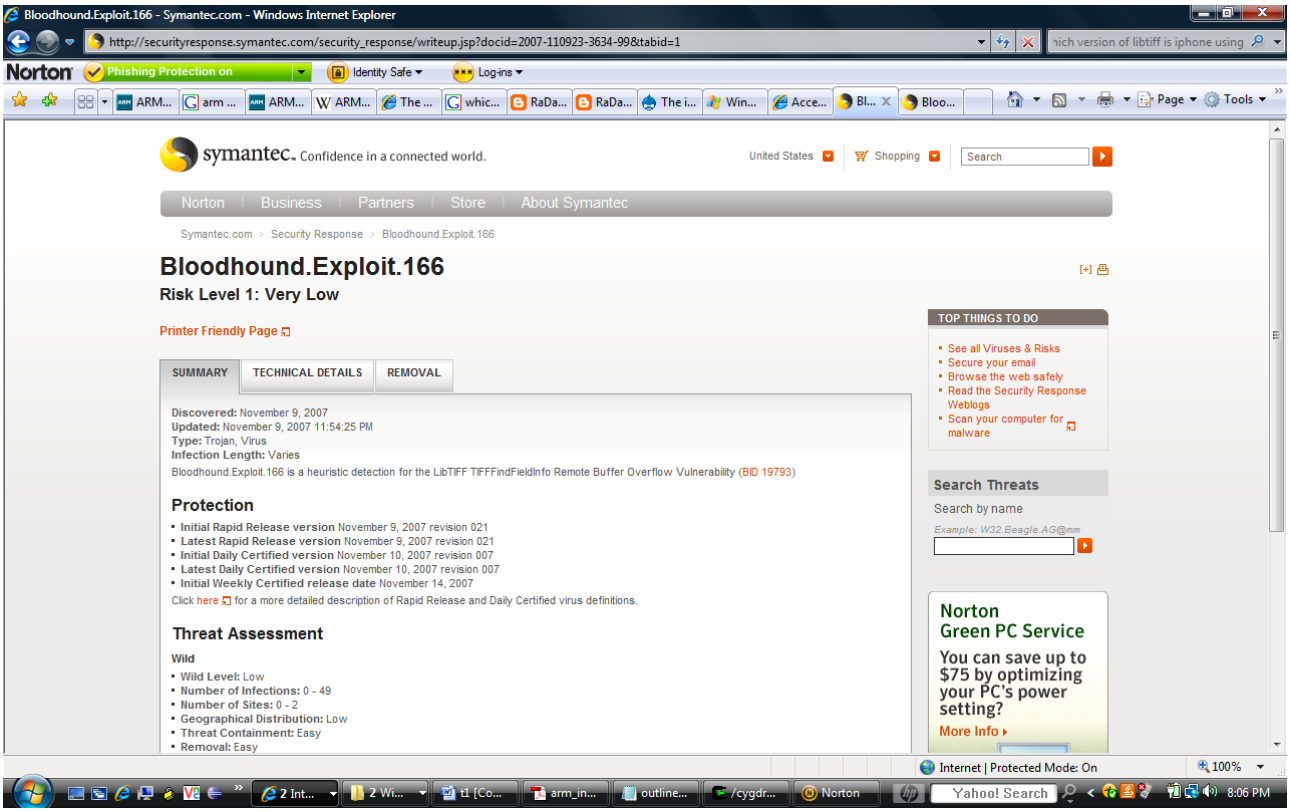


**Figure 6. Bloodhound.Exploit.166 trojan [24].**

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,

Since we are assuming little-endian representation, the first two bytes become 0x0150, which represents the DotRange tag. The next two bytes give us the value 0x0003, which means the data type is SHORT. The next four bytes give us the number of different values for this tag, which is 0x000000ff or 255 in decimal. Finally, the final four bytes give us 0x00000084, which is the offset to the actual values for the tag [18].

By looking at the TIFF specification [18] and also looking at the code for the version of libtiff with sanity check [17], we see that the number of parameters expected by DotRange is two. 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. 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 confirmed 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 now consider 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 [26]. 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. It then creates a symbolic link with /var/root/Media pointing to root, “/” and next it remounts root with the “MNT_UPDATE” flag to make it writable

 

[23]. 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.

### 3.7. Summary of Jailbreaking

Let's recap the tools needed and the process taken for jailbreaking method used above. Vulnerability in tiff library was targeted to create a stack buffer overflow and inject desired code. Then return-to-libc technique was used to execute desired code to make the root directory of iPhone writeable – i.e. to jailbreak it. During the process knowledge of TIFF was necessary in order to construct a vulnerable TIFF file. Also, knowledge of ARM processor architecture and it's deficiencies were required to ensure the attack works consistently on any given iPhone. Furthermore, knowledge of ARM instructions was required to construct the code for the attack. In summary, a great deal of research and learning was required in order to pick up the necessary tools to successfully create the Jailbreak attack.

## 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, likely because it is free. It is developed by a group of people who call themselves the iPhone dev team.

AnySim works by patching the firmware on the baseband [27]. 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 [27]. 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 "un-unlocked", as the patch that skips the check will almost certainly no longer be part of the code.

SimFree, also known as iPhone SimFree or IPSF, is unlocking software that currently sells for approximately $60, and at one point cost $99 [28]. Since it is a paid product, details about how it works are not revealed. It claims not to rely on firmware patching, so a phone unlocked with SimFree should remain unlocked even when a baseband upgrade is performed [27].

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). For TurboSim to work, it must be programmed with a valid AT & T SIM, which it copies for later use [29].

Following table summarizes the above mentioned unlocking methods.

| Unlocking Method | Technique used |
|---|---|
| AnySim | Patch the baseband to skip AT & T SIM card check. |
| SimFree | Proprietary software application that patches the iPhone firmware |
| TurboSim | Tricks iPhone into thinking that it's SIM card is an AT & T SIM card |

## 5. Jailbreaking and Unlocking Newer Versions of Iphone

As mentioned earlier, for the purposes of this project, iPhone firmware version 1.1.1 and baseband bootloader version 3.9 are assumed. As of 2008, Apple had released versions 1.1.2, 1.1.3, and 1.1.4 of the firmware. Also, the baseband bootloader version is 4.6 in some of the phones. 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 use the known attacks to jailbreak and unlock the iPhone. Several hacker websites, including iphone.unlock.no, offer instructions on how to downgrade the firmware and bootloader, and they also have different firmware files available for download [27].

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 [30] and 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 [31]. 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.

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 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 an executable heap [32].

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. The attackers were then secretly and automatically able to obtain personal data including contacts, call history, text message, and voice mail from the attacked iPhone. Attackers concluded that further personal information including passwords and emails could have been obtained had they chosen to do so [32]. 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 [32]. 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 [32]. This vulnerability is particularly dangerous since the phone bill or text message bill could be increased by the attacker, which could cost the iPhone's owner a significant sum. The attacker could also send maliciously provocative messages to the owner's contacts, which could result in personal or professional relationship problems.

These malicious exploits are, collectively, comparable to having one's iPhone stolen. If attacks like these become widespread, there is a potential that customers 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. This information could certainly be used as a guideline for the attacks above, provided one is able to write appropriate payloads. The iPhone uses Webkit, an open source web browser engine used by Mobile Safari [33], 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 [34] 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 for possible attacks on the iPhone. 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 [35]:

```
<SCRIPT LANGUAGE="JavaScript"><!--
var re = new RegExp("[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
```

 

```
[[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]][[**]]
[[**]][[**]]ABCDEFGHIJKLMNOPQRSTUVWXYZAB-
CDEFG[\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, attackers resorted to a technique called "fuzzing" [35], 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 shell code was injected [35]. 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. 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 [35]. The purpose of each function is clear from its name.

To summarize, vulnerabilities in PCRE were targeted by creating a malicious HTML file to create a buffer overflow, which facilitated injection and execution of malicious code.

## 7. Security Analysis

Having briefly examined several vulnerabilities in the iPhone and attacks that exploit those vulnerabilities, we now analyze the iPhone security structure from a high level. What was the approach Apple took while designing the security architecture for the iPhone? Were there flaws in this 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 a vulnerable device with several security holes. The iPhone security philosophy itself has a signifcant flaw. Apple's approach to making the iPhone a secure device was to reduce "the attack surface of device" or "the device's exposure to vulnerabilities" [32]. 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. Mobile Safari was restricted to only execute Javascript code, and only do so in the sandbox area. In short, Apple's approach was to make a controlled, essentially closed-box device. Apple's security approach might be summed up by 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 is debatable, the architecture has significant 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 supported by the fact that all significant processes run as a super user or with administrative privileges—a major mistake from a security perspective. A result of this configuration is that an attacker is likely able to control the entire iPhone if he is able to exploit any vulnerability in any of its applications [32]. 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 a strong lock to stop an intruder from getting in. No effort is made to, say, secure individual room, to put valuables in a safe-deposit box, to use a home security system, etc. While it may be difficult to enter the house, if a thief can do so, he can easily steal all its contents.

A 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. While the use of open source is itself likely a good idea, using (and sharing) of open source projects with old and outdated versions of those projects is clearly a problem. Earlier we looked at examples of an old version of libtiff library facilitating the jailbreak attack, and an old version of the PCRE library allows another malicious attack. By using outdated versions of open source projects, Apple made it relatively easy for hackers to develop ideas and approaches for attacks 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 buffer overflow exploits [32].

The table below summarizes the attacks discussed in this paper.

| Attack | Vulnerability targeted | Tools used | Effects |
|---|---|---|---|
| Jailbreaking | Vulnerable libtiff | TIFF, buffer overflow, return-to-libc, ARM architecture and instructions | Get root access |
| Unlocking | Jailbroken phones allow for installation of unauthorized applications | Installation of unauthorized application | Being able to use the iPhone with non-AT&T wireless services |
| Mobile safari (malicious) | Vulnerable PCRE | Malicious HTML, fuzzing, buffer overflow | Stolen personal data and other malicious effects |

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 a 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. Unfortunately, these have been somewhat countered by the ability to downgrade the firmware. 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 as well.

After evaluating Apple's security for the iPhone, one can safely conclude that overall the company failed to make the iPhone as secure as it could possibly have been. Looking at the security approach and the decisions the company made, it is no surprise that the initial iPhones were considered a fairly 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? Why did they use a vulnerable version of the tiff library? This final question is particularly important because even after three new versions of firmware and a new version of the bootloader, Apple was still paying for this mistake.

It seems implausible that Apple had no knowledge of the vulnerability in libtiff that causes buffer overflow, 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 only speculate as to why Apple used 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 is some cost 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 found that there were other known vulnerabilities in the version used anyway. 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 also used to perform truly malicious acts.

From a short-term perspective, it is hard to argue with the success of the iPhone. However, from the consumer confidence or reputation perspective, the situation is not so clear. Apple is generally regarded as a company that delivers secure and robust products. They may have lost some of that sheen with the iPhone.

## 9. Suggestions to Improve Security Structure

We have pinpointed several flaws in the initial iPhone security structure. A large security hole would have been filled if most of the processes were not run with administrative privileges, or as the super user. This would generally make it more difficult for an attacker to gain full control of an iPhone.

While using open-source based applications is a good idea, Apple needs to be more cognizant about using versions that do not have serious known bugs. Apple should also use a technique such as ASLR for heap and stack address randomization to make it more difficult for hackers to develop stable attacks and distribute them [32]. 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 the 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 would be difficult, as the target address is unreliable due to randomization.

## 10. Conclusions

In this paper, we considered the iPhone security structure and its vulnerabilities. The Jailbreaking attack analyzed here relied on a known vulnerability in the TIFF library. The analysis of the attack required some knowledge of the ARM architecture and the TIFF file format. We showed that using a vulnerable version of the TIFF library proved costly for Apple, in the sense that updates could not easily prevent "rollback" attacks. Interestingly, hackers found ways to jailbreak later iPhone without even losing the new features introduced in newer versions. Perhaps predictably, the attacks on the iPhone and the countermeasures by Apple quickly devolved into a cat and mouse game.

The security problems discussed here have resulted in financial losses for both Apple and AT&T and, arguably, a reputation loss for Apple. For each iPhone that was unlocked to access an alternate wireless carrier, AT & T stood to lose about $1500 in revenue for the two-year contract period. As we noted earlier, the number of unlocked iPhones was estimated at nearly a million in just its first six months [1]. Apple too missed out on some gains, as it receives 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 relatively secure products. While Apple's exclusive deal with AT & T and its decision to use a closed system undoubtedly increased the motivation to attack the iPhone.

We have also explained that malicious attacks can be created for the iPhone. However, the significant attacks have not been malicious, but were instead focused on enabling people more freedom to do what they want with their telephone product.

We conclude that Apple's initial effort in making the iPhone a secure device was somewhat disappointing. While Apple worked to improve iPhone security, the initial release unnecessarily gave hackers the upper hand, which, to some extent, has continued to this day.

## 10. References

[1] C. Maxcer, "Apple Minus AT&T Equals Lots of iPhones Somewhere Else," Mac News World. http://www.macnewsworld.com/story/61389.html?welcome=1209968031

[2] iPhone, Apple–iPhone. http://www.apple.com/iphone/

[3] ARM, ARM1176 Processor. http://www.arm.com/products/CPUs/ARM1176.html

[4] A. L. Shimpi, "Apple's iPhone Dissected: We did it, so you don't have to," Anandtech, 29 June 2007. http://www.anandtech.com/mac/showdoc.aspx?i=3026&p=3

[5] In brief, Network Security, Vol. 2009, No. 7, July 2009, pp. 3.

[6] Best iPhone Apps. http://www.Installerapps.com

[7] K Dunham, "Mobile Malware Attacks and Defense," Elsevier 2009, pp. 197-265.

[8] B. Haines, "Seven Deadliest Wireless Technologies Attacks," Syngress, 2010.

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

[10] Common Vulnerabilities and Exposures, 2006. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3459

[11] TIFF Library and Utilities, 15 January 2008. http://www.libtiff.org/

[12] National Vulnerability Database, 2006. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-3459

[13] "Stack buffer overflow," Wikipedia. http://en.wikipedia.org/wiki/Stack_buffer_overflow

[14] M. Stamp, "Information Security: Principles and Practice," Wiley 2005.

[15] C. Cowan, *et al.*, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 26-29, 1998.

[16] "Return-to-libc," Wikipedia. http://en.wikipedia.org/wiki/Return-to-libc

[17] Maptools, 15 January 2008. http://dl.maptools.org/dl/libtiff/

[18] Adobe Developers Association, TIFF Revision 6.0 Final, 3 June 1992. http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf

[19] "Tagged Image File Format," Wikipedia. http://en.wikipedia.org/wiki/TIFF

[20] Simple Machines, The ARM instruction set. http://www.simplemachines.it/doc/arm_inst.pdf

[21] "1176JZF-S Technical Reference Manual Revision r0p7," ARM. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301g/DDI0301G_arm1176jzfs_r0p7_trm.pdf

[22] "Little-endian," Wikipedia. http://en.wikipedia.org/wiki/Little_endian

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

[24] "Bloodhound.Exploit.166 Technical Details," Symantec, 9 November 2007.

[25] V. Pandya., IPhone security analysis, Masters Thesis, Department of Computer Science, San Jose State University, 2008. http://www.cs.sjsu.edu/faculty/stamp/students/pandya_vaibhav.pdf

 

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

[27] iPhone UnlockUSA.com. http://iphone.unlock.no

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

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

[30] Hackintosh, iPhone. http://www.hackint0sh.org

[31] G. Hotz, "On the iPhone," 15 February 2008. http://iphonejtag.blogspot.com/

[32] C. Miller, J. Honoroff and J. Mason, "Security Evaluation of Apple's iPhone," Independent Security Evaluators, 19 July 2007. http://securityevaluators.com/files/papers/exploitingiphone.pdf

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

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

[35] C. Miller, "Hacking Leopard: Tools and Techniques for Attacking the Newest Mac OS X," Black Hat Media Archives, 2 August 2007. https://www.blackhat.com/presentations/bh-usa-07/Miller/Presentation/bh-usa-07-miller.pdf

 