Noname manuscript No.
(will be inserted by the editor)

User privacy and modern mobile services: Are they on the same path?

D. Damopoulos · G. Kambourakis · M. Anagnostopoulos · S. Gritzalis · J. H. Park

Received: date / Accepted: date

Abstract Perhaps the most important parameter for any mobile application or service is the way it is delivered and experienced by the end-users, who usually, in due course, decide to keep it on their software portfolio or not. Most would agree that security and privacy have both a crucial role to play towards this goal. In this context, the current paper revolves around a key question: Do modern mobile applications respect the privacy of the end-user? The focus is on the iPhone platform security and especially on user's data privacy. By the implementation of a DNS poisoning malware and two real attack scenarios on the popular Siri and Tethering services, we demonstrate that the privacy of the end-user is at stake.

Keywords Malware · iPhone · iOS · Siri · mDNS · Tethering

D. Damopoulos (✉) · G. Kambourakis · M. Anagnostopoulos · S. Gritzalis
Department of Information and Communication Systems Engineering, University of the Aegean, GREECE
E-mail: ddamop@aegean.gr

G. Kambourakis
E-mail: gkamb@aegean.gr

M. Anagnostopoulos
E-mail: manag@aegean.gr

S. Gritzalis
E-mail: sgritz@aegean.gr

J. H. Park
Department of Computer Science and Engineering, Seoul National University of Science and Technology, REPUBLIC OF KOREA E-mail: jhpark1@seoultech.ac.kr

1 Introduction

Over the last few years, mobile devices have experienced a rapid shift from pure telecommunication devices to small and ubiquitous computing platforms. Nowadays, such devices (also known as handheld computers or smartphones) are equipped with enough facilities to even replace the usage of laptops [1]. For instance, smartphones are able to store a rich set of personal information and at the same time provide powerful services, e.g. location services, Internet sharing via tethering, and intelligent voice assistants to name just a few [2]. As expected, this situation draws the attention of aggressors to steal or misuse private information, or to disrupt the information flow [3]. Typical methods to achieve such goal are gaining root permissions (known as Jailbreak [4] on iOS, or Root on Android platforms [5]), exposing new vulnerabilities [6], and developing smart and perilous malware [7]. In fact, every new facility or service offered for modern smartphones may be susceptible to attacks and/or privacy leaks.

While more than sixty five billion mobile devices are expected to be in use by 2012, Lookout Mobile Security Center in its 2011 Mobile thread report [8], estimates that almost one million people have been affected by Android malware only in the first half of 2011. In the same report it is stated that the 33.9% of free iOS applications had some sort of hidden capability to access user's location and 11.2% of them to access personal contacts. Malware like iSAM [7] (for iOS) and DroidDream [9] (for Android) are only some examples of the dangers the users of such devices have to face and clearly show the path of what should be expected in the near future. Hence, knowing the increasing risk of mobile malware, designing a secure mobile device that protects user's privacy is still a very challenging task.

Under this prism, the work at hand deals with user privacy in modern mobile devices and especially the newly introduced iPhone 4S. Our primary aim is to elaborate on privacy risks that may come with the introduction of new mobile services. This means that new services for smartphones may expose private information without the user consent. In this direction, we concentrate on two very popular services (a) the Personal Hotspot (PH), which is a very common way of tethering an iPhone Internet connection with other WiFi devices, and (b) Siri, the new intelligent personal voice assistant available since iOS version 5. In the following, we detail how an attacker can take advantage of such technologies to trample on user privacy.

To unveil user's privacy risks that may stem from the aforementioned services, we implement as a first step a DNS poisoning malware. On the one hand, this malware is capable of poisoning the iPhone tethering service and thus redirecting all users connected via it to a fake Facebook website aiming to phish their credentials while they trying to access their profile. On the other hand, by targeting on the Siri facility, the malware manipulates the DNS service of the device in an effort to expose sensitive user information including its geographical location, account credentials, telephone numbers etc. As far as we are aware of, this is the first work in the literature to discuss and analyze ways to expose user privacy by leveraging on such popular mobile services.

The next section brings into the foreground basic requirements toward realizing a DNS poisoning malware for iOS devices. It also contains necessary information about the Siri and Tethering services. Details specific to the implementation of our malware are given in section 3. Section 4 focuses on the architecture and the inner workings of the attack. Related work on the topic is addressed in section 5. The last section holds the conclusion of our paper.

2 Preliminaries

Although this paper assumes a minimum level of familiarization with iOS programming and mobile services by the reader, this section is necessary for reasons of completeness. Thus, we briefly provide background information on iOS and jailbreaking. Also, we discuss the basic components and functionality of both the tethering and Siri services as an essential prelude to the following sections.

2.1 Malware implementation requirements

iPhone was the first multi–touch smartphone equipped with iOS (formerly iPhone Operating System). iOS has been derived from Mac OS X and relies on the Darwin foundation kernel. Therefore, it is a Unix–like Operating System by nature. On Feb. 2008, Apple released the first iOS SDK allowing developers to create third–party native applications. In this context, the implementer of any malware needs to take into account two basic requirements. First of all, it requires gaining root permissions to be able to hook and override OS internal functions of interest. Second, it needs to run continuously in the background of the OS being stealthy to the end–user. But, whether rightly or wrongly, only applications inspected and signed by Apple's Certificate Authority (CA) can be released and are allowed to run on an iOS device. So, considering the first requirement, the only way to run unsigned software is by gaining root permissions on the device using an exploitable vulnerability. This process is generally referred to as Jailbreak [6]. Upon jailbreaking, the entire iPhone file system becomes open for use. Also, Jailbreaking allows creating and executing third–party software using both the official public and the unofficial private frameworks. Public frameworks are provided by the native SDK allowing developers to build AppStore applications. The private frameworks on the other hand are used only by Apple to provide high–level programming features on the original applications. Unfortunately, private frameworks are neither available by the iOS SDK nor documented. The only way to overcome this issue (as in our case) is by retrieving the private framework directly from the files of a jailbroken iPhone and then use the class–dump utility to generate the (still undocumented) header file(s).

The main exploit currently used to Jailbreak iOS 5 is Corona [10]. This is a userland exploit (i.e., an exploit related to software) that uses both the Racoon String Format and the HSF Heap overflows to jailbreak the device. Besides, Apple does not offer any frameworks that override iOS functions. To fill the gap, J. Freeman has created the MobileSubstrate extension, a framework that allows developers to deliver run–time patches to system functions using Objective–C dynamic libraries (dylib) [11]. Also, D. L. Howett has contributed Theos, a cross–platform suite of development tools for managing, developing, and deploying jailbreak–oriented iOS development [12]. By creating a dylib and connecting it with the MobileSubstrate extension, developers are able to build applications capable of hooking internal system functions.

Taking into account the second requirement, the malware needs to run continuously in the background of

the underlying OS. Although iOS ver. 5 supports multitasking through seven official Apple's APIs for backgrounding, it is not the best way to launch a malware since it needs to remain hidden from the end–user. Fortunately, iOS being a Unix–based OS, is able to provide multitasking using launchd, a launch system that supports daemons and per–user agents as background services. Once an iOS device has been jailbroken, any installed application or shell script is able to behave as daemon by creating a launch property list (plist) file and placing it into the "/Library/LaunchDaemons" iOS directory. Another way to support multitasking is with dynamic libraries (dylib). Upon launching an application, the iOS kernel loads the application code and data into the address space of a new process. At the same time, the kernel loads the dynamic loader, namely /System/MobileSubstrate/DynamicLibraries into the process space and passes the control to it. It is also possible to load a dylib at any time through an Objective–C function.

It is also relevant to note that although the primary aims of a smart malware is to infect the target, self–propagate to other targets and finally connect back to a bot master server [7], in this paper we do not analyze new jailbreak methods, but use already referenced ones [6] to gain root permissions on the device and infect it with our malware. Moreover, it is straightforward that the malware implemented in the context of this paper can be integrated with other similar applications like iSAM [7], or propagate individually by incorporating some of the existing infection methods already presented in the literature [6].

2.2 mDNS

As already stated, the aim of our malware is to compromise the DNS service running on the device. This is a sine qua non for the attacks described further down to be successful. Toward this direction, one of the main technologies used in iOS for networking is Bonjour. Bonjour enables a device to allocate an IP address and advertise a service to other computers or devices plugged into the same TCP/IP network. Also, Bonjour includes service discovery, address assignment, and name resolution. On top of that, Bonjour, being a Zero Configuration Networking (ZCN) facility, needs to be able to translate name–to–address even without the presence of a DNS server. To meet this requirement the Multicast DNS (mDNS) protocol is used. This protocol uses the same packet format, name structure, and DNS record types as unicast DNS. However, two main differences apply. The first one is that mDNS queries are sent to all local hosts using multicast in contrast to the DNS protocol, which queries are sent to a specific, preconfigured name server. The second is that mDNS listens on UDP port 5353, in contrast to DNS which listens on standard UDP port 53. Also note that mDNS requests use the multicast address 224.0.0.251. In case a device triggers the Bonjour service, it listens to the multicast requests and if it knows the answer, it replies to a multicast address. mDNSResponder is the application which is responsible for handling Bonjour on Mac OS X and iOS devices and for listening for services out of the box.

As expected, iOS supports a hosts file configuration in order to be able to map already visited hostnames to IP–addresses before DNS can be referenced. This temporary mapping per hostname is kept in the /etc/hosts, which is also manipulated by our malware as described further down in section 3.1. Last but not least, iOS holds in the Network.identification.plist the settings of all the wireless networks with which the device has been associated sometime in the past. This happens as part of a new feature that allows the iOS device to remember the network settings and automatically connect to it, using the same settings, without user intervention. Therefore, our malware needs to replace the DNS IP address of all networks logged in the Network.identification.plist with a bogus one (where our server resides) and to restart the mDNS service in order the new settings to take effect. This situation is discussed in detail in section 3.1.

2.3 The Tethering and Siri services

As already mentioned, the purpose of the attacks described in this paper is to compromise the privacy of the end–user by capitalizing on two popular services; Tethering and Siri. Tethering is a network service which gives the end–user the ability to share their mobile phone cellular data connection with other devices (users). This sharing can be offered over a wireless LAN (WiFi), Bluetooth, or by a physical connection via a cable. Currently, Tethering is available only for the two latest iPhone devices (4 and 4S), which incorporate a software functionality known as Personal Hotspot (PH). The PH service is in charge to transform the device into a wireless Access Point (AP), so that iPhone users are able to share their 3G connection. Once the PH starts up, the device selects the first empty 802.11b/g wireless channel to emit the signal using the device name as the Extended Service Set ID (ESSID) name for the AP. From this point on, PH can support and share the Internet connection with up to five simultaneous devices. The PH service functions by default in the WPA2 Pre–shared key (PSK) mode.

Siri, on the other hand, is one of the highlights of iOS 5 only provided for the iPhone 4S. It is a personal intelligent software assistant that uses a natural language interface to interact with the user and execute their requests. Although, Siri is still in beta version, it is able to carry out a variety of tasks (e.g. send SMS, E–mail, set up meetings, make questions about the weather, points of interest etc). To accomplish such tasks, Siri communicates securely via https with a remote server residing at `https://guzzoni.apple.com:443`. This server is responsible to translate user voice requests into text commands, and text commands into actions. To fulfill a task, Siri can exchange a wide range of data with the Guzzoni server, such as raw audio data, plist files, confidence score of each word in a sentence, time–stamps, location information, and more importantly, information derived directly from the device local databases (calendar, contacts etc).

Applidium [13] has very recently reverse engineered the Siri protocol. They also provided the first evidence about its structure as well as the open source tools they used. For using Siri, the device must firstly authenticate the Siri server. This is done during the SSL handshake as the server certificate, namely `guzzoni.apple.com`, is preinstalled on every iPhone 4S device. Note that the authentication is unilateral, i.e. the client (device) does not authenticate itself to the service by means of a certificate. Upon successful authentication and under the protection of the SSL tunnel, Siri sends four keys to the Guzzoni server x–ace–host, assistantID, speechID, validationData. Where: *x–ace–host* is a unique identifier generated by Siri on the device and updated every two weeks; *assistantID* is a string containing information about the user. It is generated by Siri on the device at every use; *speechID* is a speech identifier, generated by Siri on the device on–the–fly at every use; *validationData* is a string that gets generated every 24 hours on the device via FairPlayed. By using this quadruple of keys, the Guzzoni server authenticates the device.

From the above discussion it becomes clear that attacking Siri is not trivial. Specifically, as already mentioned, Siri is a proprietary software designed to communicate securely (https) with the original Siri server(s) controlled by Apple. Therefore, to fool the protocol, one has to somehow hijack the device–to–Siri_legitimate_server communication in an undetectable manner. In this direction, as described in [19], a solution is to create a fake SSL Certification Authority (CA) and inject it into the device replacing in this way the original one. This is necessary to create and sign a fake certificate for `guzzoni.apple.com`. After that, the same team managed to redirect all iPhone packets, using a VPN connection, through a custom DNS server for further analysis. A few weeks later, P. Lamonica [14] created an open source server, namely SiriProxy, having the ability to handle Siri packets. Also, through the creation of customized plugins he has been able to execute certain actions (e.g., control a thermostat over Siri).

3 Implementation

In this section we delve into the internal workings of the malware responsible for poisoning the DNS service running on the device. This is a first step towards executing the two attack scenarios described further down in section 4.

3.1 The DNS poisoning malware

To manipulate the mDNs service running on iOS we implement a malware which, as we show in what follows, acts as a rootkit. The malware was written in Objective-C and compiled for iPhone ARM CPU using Theos. It was tested to run on iOS version 5 and above. Also, it has been built using the unofficial ways for backgrounding (daemons and dylibs), the public and private frameworks for developing iOS applications, and the MobileSubstrate framework with the substrate.h header that overrides iOS internal functions. That is why certain modules of our malware can be classified as rootkit and more specifically as a DNS poisoning one. The malware assaults over the mDNS protocol, thus making possible the execution of a man–in–the–middle assault at a later time depending on the attack scenario. That is, to take over the control of the Siri service upon its activation by the user, or if tethering is in use, redirect any connected device to a fake website.

As depicted in Fig. 1, the heart of the malware consists of a main daemon combined with a proper launch plist (activated at device boot time) and six subroutines written as Objective–C functions and dylibs. The daemon is responsible for managing all subroutines, namely SirInvervine, HUpdate, NIUpdate, mDNSReloader, NetDetector and PHDetector, which in turn carry out the malware tasks. In the following, we elaborate on the functionality of each subroutine.

Recall that for using Siri, the device must first authenticate the Siri server. This is done in a unilateral fashion i.e., the client (device) does not authenticate itself to the service. So, to act as man–in–the–middle and hijack the https session one needs to replace the original Siri certificate stored in the device with a fake one. This is accomplished by *SirIntervine*. Upon execution, this routine installs a custom SSL CA into iOS and at the same time adds into the com.apple.assistant.plist

file, which is a Siri setting file, the Domain Name of our man–in–the–middle server (in this case spe.samos.icsd.gr). This is needed to create and sign a fake certificate for `guzzoni.apple.com`.

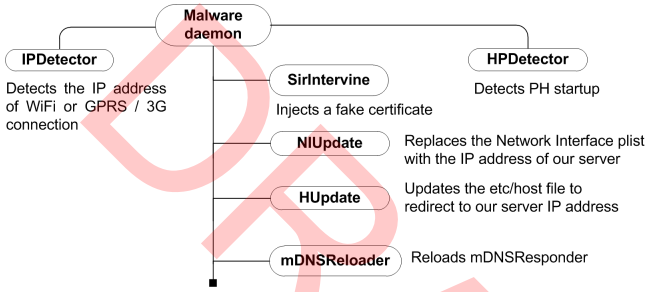

Fig. 1 Malware module.

The *HUpdate* routine is responsible for poisoning the device's /etc/hosts file. Although information stored in this file is mostly used when a DNS server is not available on the network, as pointed out in section 2.2, this file is always queried upon Bonjour activation. Once we gain root permissions, the /etc/hosts file is vulnerable as it is stored in plaintext. For our attack scenarios, HUpdate inserts two hostname records into the /etc/hosts file which correspond to the IP–address of our man–in–the-middle server. The two hostnames that are poisoned are "`guzzoni.apple.com`" and "`facebook.com`". In this way all packets sent to the aforementioned domain names will eventually be sent to man–in–the–middle entity controlled by us. HUpdate adds the poisoned hostnames in the /etc/hosts file using the public NSFileManager class (only if not poisoned already). Fig. 2 depicts a snapshot of the /etc/hosts file after poisoning has taken place. Note that the two last entries correspond to our man–in–the–middle server.

```
# Host Database
# localhost is used to configure the loopback interface
# when the system is booting. Do not change this entry.

127.0.0.1          localhost
255.255.255.255    broadcast host
::1                localhost
Fe80::1%lo0        localhost

195.251.166.50     facebook.com
195.251.166.50     guzooni.apple.com
```

The last two entries have been altered to redirect the traffic to our man-in-the-middle server

Fig. 2 The /etc/hosts file after poisoning.

NIUpdate is the subroutine responsible for poisoning the IP address of any DNS server found in the Network Identification file, with a malicious one. Every time an iPhone device connects to a WiFi or a 3G / GPRS network, an entry is created in the Network.identification.plist file containing all settings specific to this network, i.e. router's IP address, subnet mask, DNS server IP address, MAC address etc. Hence, every time the device tries to connect to a known network, it will load the settings used during the previous session. Once NIUpdate is activated, it changes all the predefined DNS servers' IP addresses with the one of our man–in–the–middle server. Once again, the Network.identification.plist file is stored in plaintext (plist), thus it can be easily modified using the NSMutalbeDictionary class.

mDNSReloader is a dylib responsible for shutting down or restarting the mDNSResponder service (deamon) running on the device aiming to activate new network settings. Specifically, by disabling the mDNSResponder service one also terminates the Unicast DNS resolution. By doing so, we block the mDNS service, meaning that instantly the device cannot resolve hostnames. Once the service gets restarted, the mDNSResponder will parse the /etc/hosts and Network.identification.plist files in an effort to use the default settings before obtaining new ones. Note that mDNSReloader enables or disables the service by simply modifying the "ProgramArguments" settings (in the com.apple.mDNSResonder.plist file) which is responsible for the activation of the service into "Yes" or "No". Fig. 3 depicts a snapshot of the source–code responsible for this modification.

```
NSMutableDictionary *mDNSR = [NSMutableDictionary dictionaryWithContentsOfFile:
        @"/System/Library/LaunchDaemons/com.apple.mDNSResponder.plist"];
...
if (mDNSR) {
  ...
  [mDNSR setObject:args forKey:@"ProgramArguments"];
  [mDNSR writeToFile:@
        "/System/Library/LaunchDaemons/com.apple.mDNSResponder.plist" atomically:YES];
}
```

Fig. 3 Source code snippet for disabling / enabling mDNSResponder.

Both *NetDetector* and *PHDetector* are dylibs triggered directly from the iOS Notification Center (more specifically the CFNotificationCenterGetDarwinNotifyCenter) every time the device connects to any wireless network interface, e.g. WiFi, GPRS, 3G, or after PH activation. As soon as one of these dylibs is executed, it will re–run all the aforementioned subroutines to update the network settings for the device.

Lastly, our man–in–the–middle server incorporates three basic modules:
(a) A typical DNS recursive server that provides fabricated answers for every domain name that is queried for. Specifically, for the first scenario this is the Siri le-

gitimate server, while for the second, a bogus version of the Facebook website.
(b) The open source SiriProxy Ruby script [14] which allows us to manipulate Siri packets and create our own custom plugins to violate user's privacy though the Siri technology.
(c) An http server used during the first attack scenario.

The server runs on a typical laptop machine which incorporates a 2.53 GHz Intel Core 2 Duo T7200 CPU and 4 GB of RAM. The OS of this machine is OS X Leopard Snow. The lightweight open source DNS Server named Dnsmasq has been used to provide DNS service. We also tinkered with the pre–alpha version of the SiriProxy that runs on our server to handle (i.e., decipher, encipher, modify) Siri packets.

Both Dnsmasq and SiriProxy server, which is the main software employed for realizing man–in–the-middle and handling Siri Packets, are able to accommodate multiple users by design.

4 Attack Scenarios

In this section it is demonstrated how the aggressor is in position to collect private user information while they using Tethering or interact with Siri. We analyse these two attacks cenarios in detail and show that any private information the user provides for the benefit of both of these services (e.g., passwords, account numbers, telephone numbers, emails, user's location etc) is at stake. The overall attack architecture is given in Fig.4 . It is stressed that all experiments had 100% accuracy in logging private and sensitive information without exposing any malicious behavior to the user of the device.

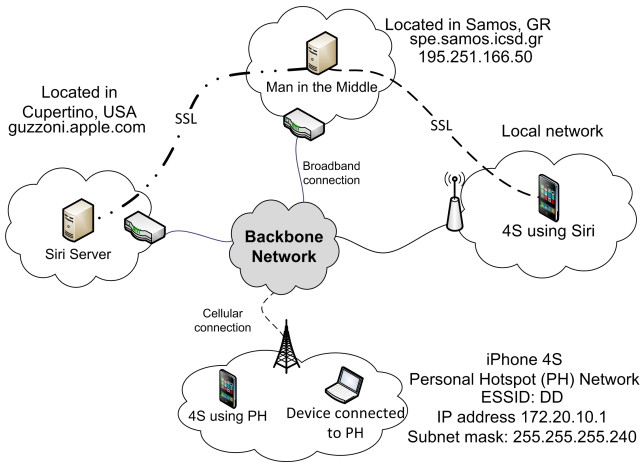

Fig. 4 Network architecture used during the attack scenarios.

4.1 Scenario I: Tethering DNS Hijacking

According to this scenario, we use an already infected with our malware iPhone 4S to tether its 3G connection and therefore enable it to act as an IEEE 802.11 hotspot. This situation is given in the lower part (cloud) of Fig. 4 . From this time forth, the device behaves as a Wi–Fi router meaning that any Wi–Fi device (the laptop in Fig. 4) will be able to connect via the iPhone PH service to the Internet. Once a device gets connected, it will allocate an IP address in the range of 172.20.10.2 to 172.20.10.14 using the Dynamic Host Configuration Protocol (DHCP). Since then, all network packets will be routed via the smartphone behaving as PH. One of the main iPhone tasks when acting as a PH is to translate any hostname into a valid IP address. To do so, firstly it lookups into the /etc/hosts file and if it does not find the answer, it will query the DNS server. Nevertheless, the device is infected with our malware and both the /etc/hosts file and the network DNS IP address have been fabricated to contain the IP address of our man–in–the–middle server. This means that all the traffic generated by the users connected via the PH will be redirected to a server under the control of the attacker. To show the hazardous effects of this attack we have built a webpage that appears exactly the same as that of Facebook and stored the page on our server. We chose Facebook as it is a very popular website and most people check their profiles once they connect to the Internet. In fact, the only functionality of our fake webpage is to log into a MySQL database the credentials of the user in plaintext, once they try to login into the site. As soon as the credentials are stored, the fake website returns a message that the page is temporally unavailable due to heavy loads.

4.2 Scenario II: Privacy leak over Siri

The second attack scenario takes advantage of the Siri service. Once more, the malware compromises the mDN protocol with a view to redirect all (or selected) Internet traffic to our man–in–the–middle server. In this way we achieve to place a malicious entity between the device and the legitimate Siri server controlled by Apple. After that, we are able to intercept user's private information transferred over Siri. At present, this is realized through the implementation of three custom plugins for SiriProxy [14]. To exemplify these, in Fig. 5 we present the basic message flow happening between the Siri service running on the mobile device and its legitimate server, but when our server is placed in the middle.

Upon Siri activation (1), an SSL handshake between Siri and our man–in–the–middle server is performed

and at the same time a second handshake is conducted between the man–in–the–middle server and the Apple's original Siri server.

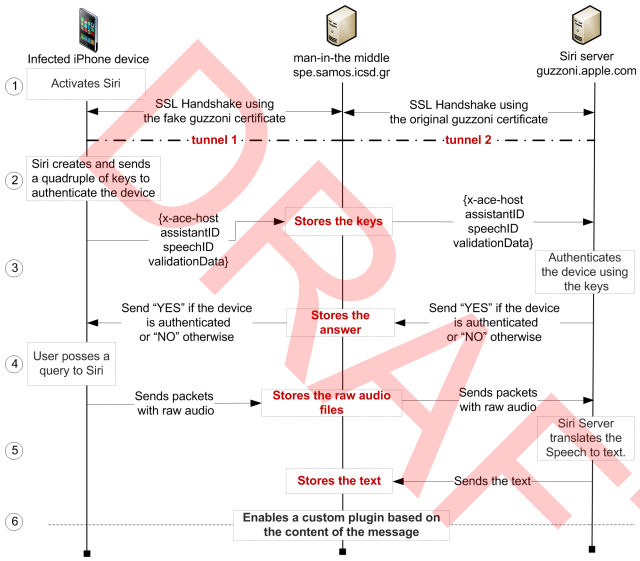

Fig. 5 Siri protocol flow.

Recall, that SiriProxy runs on our server to handle (i.e., decipher, encipher, modify) Siri packets. Specifically, to initiate the handshake, Siri sends a "Hello" message which is redirected to our server and forwarded to the original one. The Siri server replies and sends over its original server certificate containing its public key. The man–in–the–middle entity transmits to Siri its fake server certificate containing the corresponding (fake) public key. This certificate has been created from the same CA authority with the one been injected to the device when infected by our malware. Once Siri verifies the fake certificate and subsequently authenticates our fake server, it sends a premaster secret (premaster secret 1) to our server encrypted with the corresponding fake public key. At this moment, both sides (Siri and man–in–the–middle) calculate a session key (session key 1) and establish an SSL session (tunnel 1). After that, our server acting as Siri client sends a second premaster secret to Siri server encrypted with the original server's public key. As a result, the man–in–the–middle entity and the Siri server calculate another session key (session key 2) and establish a second SSL session (tunnel 2).

Under the protection of tunnel 1, Siri generates and sends the quadruple of keys necessary to authenticate the device with the server (2). Our server captures the keys and forwards them to the original Siri server but this time through tunnel 2. Upon reception, Siri server will check if the received keys correspond to a legitimate iPhone 4S and if true, it will answer with "YES" (else "NO") (3). Assuming a positive answer, Siri is ready to listen to user commands (4,5). Otherwise, it will respond with a "Siri server unavailable" message. From this point on, the user is able to make questions by speaking to the service. Siri records the voice containing the user query (or an answer to a question posed by Siri during a transaction), converts it into raw audio files and sends them to Siri server. The server translates the audio file to text and sends back the translated text which is eventually passed to the user by synthetic speech. It is therefore obvious that every personal information being transmitted from the user side it becomes available to the man–in–the–middle entity as well.

To further analyse this situation, we implemented three custom SiriProxy plugins specially crafted to expose usual private information. This means that once our server receives a Siri message from the device it will try to match its context with one of this plugins. Fig. 6 depicts a basic example of such a plugin that is activated once the translated string coming from the user side is "iPhone privacy". Upon activation, our server will respond with the string "Siri is having some privacy leaks!" to the Siri service. Siri will complete the request by displaying the message on device's screen and at the same time by pronouncing it. The next three sections describe in detail how we were able to intercept valuable user private information through the employment of such plugins.

```
listen_for /iPhone privacy/i do                  # checks the string
    say "Siri is having some privacy leaks!"     # sends a custom answer
    request_completed                            # completes the request
end
```

Fig. 6 Basic source code example of a custom plugin.

4.3 Exposing the user's geographical location

Using the first plugin we were able to successfully retrieve user's location in the form of GPS coordinates. This happened after the user asked Siri about the weather, e.g. "How is the weather today?". Note that with minor modifications, the same plugin is able to retrieve user's location for any posed question such as "How can I get to Ocean Park?", "Where is the nearest metro station and bus stop?" etc. It is stressed that Siri obtains the geographic coordinates without directly asking the user about their location. This happens because Siri has access to the device's location services by default (assuming that the user has not changed the

default settings; in this case the user will be alerted to enable GPS). Fig. 7 depicts part of the plugin source code responsible to retrieve the geographical location of the user. After the Siri server asks Siri about the location of the device, the plugin activates and waits for the standard value (header) "SetRequestOrigin" to filter the exact user's location.

```
filter "SetRequestOrigin", direction: :from_iphone do |object|  # filters the packet for location info
   puts "[Info - User Location]                                 # prints the location on terminal screen
         lat: #{object["properties"]["latitude"]},              # on the side of the man-in-the-middle
         long: #{object["properties"]["longitude"]}"
   return 1                                                     #forwards the object
end
```

Fig. 7 Part of the plugin responsible to retrieve user's location.

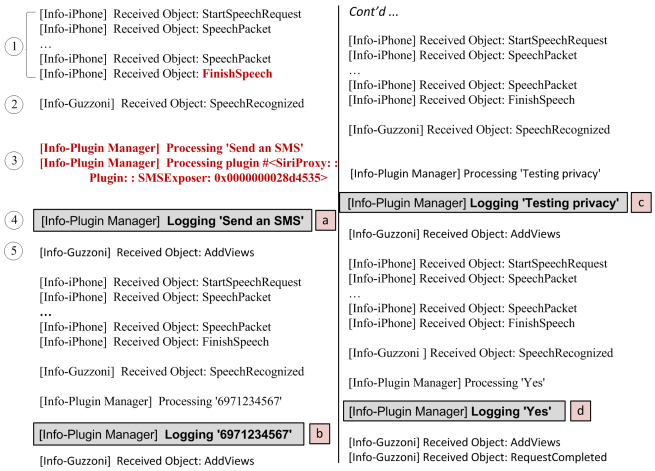

Fig. 8 Log file created by the plugin when sending an SMS.

4.4 Obtaining sensitive information via SMS

The second plugin capitalizes on Short Message Service (SMS). According to this scenario, the user sends an SMS by just speaking to Siri. The plugin intercepts the telephone number of the receiver of the message, the SMS payload and the final outcome, i.e., whether the end–user finally gave their consent to send the SMS or not. By this use case scenario it is made clear that a variety of private information sent to Apple's servers can be exposed to an intruder without the user be aware of it. Fig. 8 illustrates the log file created by the corresponding plugin on our man–in–the–middle entity under this scenario. In the same figure we can easily identify the user's private information leaked out (a, b, c, d). Note that the lines starting with [*Info–iPhone*] correspond to messages sent from Siri, while those starting with [*Info–Guzzoni*] to messages deriving from the Siri original server. Also, messages being transmitted from SiriProxy are marked with [*InfoPlugin Manager*]. For emphasis, each privacy leak is placed within a gray frame.

To exemplify this, once the user activates Siri and starts speaking to it, Siri sends user voice towards the server in many fragmented packets. After the user stops speaking, Siri sends a flag message (1). Then Siri translates the voice into text and sends it back to our server (2). Upon reception, SiriProxy tries to match the translated text with a custom plugin (3). The plugin is in charge to log the translated text when a user tries to send an SMS (4). Once the text is logged, the message is sent to Siri. As a final step, the Siri original server sends a message to inform Siri to create a graphical view for presenting the translated text (5).

4.5 Acquiring user's password

One of Siri highlights is that the user can engage in a form of conversational dialog with the assistant using any of a number of available input and output mechanisms, e.g. speech, graphical user interfaces, text entry, and so on. So, for the last use case, we developed a smarter plugin able not only to eavesdrop on private information but also to interact with the user and ask them custom questions. By doing so, it becomes very likely for our man-in-the-middle entity to intercept confidential information such as the user's e–mail address or even the password of their e-mail account(s). Due to the fact that Siri uses artificial intelligent to interact with the user in order to accomplish a task, e.g. send out an email, the question about the password would not bear any evidence of malicious behavior.

Fig. 9 presents the message flow when the user attempts to send an e-mail using Siri. This results to the activation of the corresponding plugin residing on the man–in–the–middle entity (1). Once SiriProxy receives the translated text from the original Siri server - in this case "Send an email" – it will match it against the plugin settings (2). As a consequence, the plugin will temporally block the original text message from being transmitted towards the original Siri server, and instead, it will send back a custom question to Siri asking the user which sender's email address it should use. Since the e-mail address is generally considered public information the user is highly probable to reply providing its email address to Siri (3). As a next step, the plugin shall force Siri to pose a second question to the user. This time Siri will ask for the password of the e–mail address the user gave in the previous step (5). Typically, a naive user will trust Siri and think that the

password is necessary for the e-mail to be sent. Hence, they will respond with the password, thus enabling the plugin to log it in cleartext (6).

Fig. 9 Message flow for acquiring user's password.

5 Related Work

Mobile devices are used in everyday life to store a variety of users' sensitive information. So, it should come as no surprise that they attract the attention of resourceful attackers. In this context, over the last few years, traditional malware also seem to evolve in an effort to catch up with the so called mobile era. Mainly, such malwares affect the most popular OS, namely Android, Symbian and iOS. It is also relevant to note that according to [15] the most common behavior of the propagating malicious application on smartphones is the collection of private information. Due to its novelty, literature about the privacy level of advanced mobile services like Siri and Tethering, other than those mentioned in section 2 is, at least for the time being, scarce. However, malware and fraudulent applications for modern smartphones have been outlined in several lines of work. This section attempts to review the existing literature in chronological order and with respect to the impact of the threat.

The first appearance of malware that spreads through mobile devices infecting them has been reported back in June 2004 with the appearance of Cabir worm for Symbian OS. Soon after, a more serious threat was released; the first trojan spy for Symbian called Pbstealer. This malware has been based on Cabir, and thus spreads via Bluetooth. Once it infects a device searches for the address book and sends the contained data to the first device discovered in range [16]. In the following months, another trojan that violates user privacy has been detected. StealWar presents similar functionality to Pbstealer, but it has been given the ability to propagate via MMS as well [16]. Flexispy is another trojan that takes over control of smartphones running Symbian and sends call information and SMS data to the herder of the Trojan [16]. A different malware is described in [17] with the functionality of taking pictures from the phone's camera without the user's knowledge and send it to a predefined phone number via MMS. The creators of the particular malware aimed to demonstrate that it is quite easy to implement a privacy violation attack for mobile devices.

In 2008 two modern OS platforms for smartphones, Apple's iOS and Google's Android have been introduced. Since then, the research community focused its attention on the security level of these OS. For the case of Android the beginning of privacy concerns arises with the first Android commercial smartphone, namely G1 which was shipped with the initial version of this OS. This version was accompanied with vulnerability to the web browser application. Hence, a determined attacker could gain access to any data that the browser stores, like cookies, text typed in form fields or even saved user passwords [16]. In the same year, as described in [18], some serious privacy concerns appeared with several applications within the AppStore market administered by Apple.

In 2009, the authors in [19] presented a vulnerability in SMS messages, which enables an attacker to inject malformed SMS messages to cause DoS or gain unauthorized permission to the underlying OS. During the same period, the first iPhone worm namely Ikee was released and a barrage of worm attacks started. Ikee was simply changing the iPhone's wallpaper. Also Ikee was a self–propagating worm to attack only jailbroken iPhones using the installed SSH server vulnerability and the default root password. The same vulnerability has been also used by Dutch 5e ransom, a worm that locked the iPhone screen asking 5 to be paid on a PayPal account. Privacy.A, was another worm running in stealth mode aiming to steal personal data from an iPhone device. In Nov. 2009, a new highly hazardous version of Ikee, namely iKee.B appeared [20].

Although iKee.B acts similar to Ikee, it includes Command & Control (C&C) logic to control all infected iPhone devices via a botnet server, making iKee.B the first iPhone botnet. Moreover, iKee.B was the first malware that through a script was able to poison iPhone's DNS cache (/etc/hosts) to redirect specific requests to a given IP address. On the Android camp, LeNa, descendant of Legacy virus family, was hiding in seemingly useful apps to gain the required root privilege and expose private information [16]. Moreover, two known applications, MobileSpy and MobiSteath, exhibit the previously approach by sending the recorded data (e.g. SMS, call history, GPS coordinates, contact details, Video and Pictures Logging) to a remote server. On top of that, the latest trend of malware propagation on Android devices is via Quick Response (QR) codes. According to this, whenever a user reads a malicious QR code, it is directed to a site that instantly downloads a malware to its device [16].

In 2010 the authors in [18] presented some interesting attack scenarios on how a malicious application can use official and public frameworks, provided by Apple. Also, five Android malware families, namely BaseBrigde, JiFake, DroidKungFu, Hongtoutou, Geinimi, have been identified after trying to expose the victim's geographic location, International Mobile Subscriber Identity (IMSI), bookmarks and/or place calls, send SMS to premium phone numbers [16].

In 2011 the first rootkit–similar, multifarious malware that is capable of infecting iPhone devices has been presented [7]. Its aim is to stealthily execute six malware routines, self–propagate wirelessly to other devices and finally connect back to a bot master server to update its programming logic or to obey commands. During the same year several researchers and hacking teams have presented novel techniques that may lead to exposing iOS security [6,10,21]. Another important disclosure occurred when a researcher found that many smartphones OS including Android, BlackBerry, Symbian, and iOS had pre–installed a rootkit / keylogger developed by Carrier IQ [22].

6 Conclusions

Mobile devices have evolved and experienced an immense popularity over the last few years providing users with intelligent services, e.g. personal hotspots, personal assistants over voice, augmented reality etc. Nevertheless, such services are generally regarded to be an attractive target for attackers hoping to compromise the service and expose user's privacy. This paper concentrates on the popular iPhone device and as a case study examines the privacy level of two attractive services, namely Tethering and Siri. To do so, we implement a DNS poisoning malware with the mission of redirecting all or a subset of DNS requests to a DNS resolver which is under the control of the attacker. It is then obvious that such a setting can severely influence the way the user experiences the Internet and expose them to serious threats. On the one hand, the malware poisons the device's tethering service to force all users connected via it and trying to access their Facebook page to be redirected to a bogus Facebook website. After that, those users are left defenseless to phishing attacks. On the other hand, we demonstrate that by leveraging the Siri facility, the attacker is able to intercept sensitive user information including their geographical location, account credentials, address book etc. Generally, such attacks stem from the fact that security and user–privacy is commonly not within the first priorities for new operating systems and features/services for mobile devices. Naturally, this results in poor privacy protection even for the security–savvy user, which most of the time is unaware of such privacy degradation.

As future work we consider the implementation of an intrusion detection tool able to identify smartphone malware and more specifically those having the intent to modify the hosts file. This way DNS poisoning attacks can be thwarted. Apple should also consider updating the Siri protocol to support mutual authentication between the iOS device and the Siri server every time Siri is used. Another effective countermeasure for preventing such attacks relies on iOS per se. It should incorporate a mechanism that inspects the authenticity of the certificates contained in the device's certificate store, i.e. examine if they are issued by a trusted authority.

Acknowledgements This research is partially supported by Seoul National University of Science and Technology.

References

1. Dafir Ech-Cherif El Kettani M, En-Nasry B (2011), MIdM: An Open Architecture for Mobile Identity Management, Journal of Convergence, Vol.2(2), pp. 25-32.
2. Luo H, Shyu M. L. (2011), Quality of service provision in mobile multimedia - a survey, Human-centric Computing and Information Sciences, Vol. 1(5).
3. Chuan D, Lin Y, Linru M, Yua C (2011), Towards a Practical and Scalable Trusted Software Dissemination System, Journal of Convergence, Vol.2(1), pp.53-60.
4. Halbronn C, Sigwald J (2010), iPhone security model & vulnerabilities. In : Proceedings of Hack In The Box SecConf, Kuala Lumpur, Malaysia, 2010.
5. Burns J (2009), Exploratory Android Surgery. In: Proceedings of Black Hat, USA.

6. Miller C, Inside iOS Code Signing, proc. of Symposium on Security for Asia Network (SyScan), 2011.
7. Damopoulos D, Kambourakis G, Gritzalis S (2011), iSAM: An iPhone Stealth Airborne Malware. In: Proceedings of IFIPSec 2011, vol. 354, pp. 1728, Springer.
8. Lookout Mobile Security, Mobile Threat Report, https://www.mylookout.com/mobile-threat-report. Accessed 10 March 2012
9. Lookout Mobile Security, DroidDream, http://blog.mylookout.com/droiddream/. Accessed 10 March 2012
10. Pod2g, Corona, http://pod2g-ios.blogspot.com/2012/01/details-on-corona.html. Accessed 10 March 2012
11. The iPhone Wiki, MobileSubstrate, http://iphonedevwiki.net/index.php/MobileSubstrate. Accessed 10 March 2012
12. The iPhone Wiki, Theos, http://iphonedevwiki.net/index.php/Theos. Accessed 10 March 2012
13. DumasLab, Inside Siri, http://dumaslab.com/2011/11/inside-siri/. Accessed 10 March 2012
14. Lamonica P, SiriProxy, https://github.com/plamoni/SiriProxy. Accessed 10 March 2012
15. Felt A. P., et al. (2011), A Survey of Mobile Malware in the Wild. In: Proceedings of ACM CCS (SPSM), Chicago, USA.
16. La Polla M, Martinelli F, Sqandurra D, A Survey on Security for Mobile Devices, Technical Report, http://puma.isti.cnr.it/dfdownload.php?ident=/cnr.iit/2011-TR-026&langver=en&scelta=Metadata. Accessed 10 March2012
17. Schmidt A, Albayrak S, Malicious Software for Smartphones, Technical Report TUB-DAI 02/08-01, Technische Universitat Berlin, DAI-Labor, Feb. 2008. http://www.dai-labor.de. Accessed 10 March 2012
18. Seriot N (2010), iPhone Privacy. In: Proceedings of Black Hat, USA.
19. Mulliner C, Miller C (2009), Fuzzing the Phone in your Phone. In: Proceedings of the Black Hat, USA.
20. Porras P, Saidi H, Yegneswara V (2009), An analysis of the Ikee.B (Duh) iPhone botnet, SRI International Computer Science Laboratory, Technical Report.
21. Esser S (2011) , iOS Kernel Exploitation - IOKit Edition. In: Proceedings of Symposium on Security for Asia Network, Taiwan.
22. Android Security Test, CarrierIQ, http://androidsecuritytest.com/features/logs-and-services/loggers/carrieriq. Accessed 10 March 2012