Fuzzing the Phone in your Phone (Black Hat USA 2009)

Collin Mulliner
TU-Berlin / T-Labs
`collin@sec.t-labs.tu-berlin.de`

Charlie Miller
Independent Security Evaluators
`cmiller@securityevaluators.com`

June 25, 2009

Abstract

In this talk we show how to find vulnerabilities in smart phones. Not in the browser or mail client or any software you could find on a desktop, but rather in the phone specific software. We present techniques which allow a researcher to inject SMS messages into iPhone, Android, and Windows Mobile devices. This method does not use the carrier and so is free (and invisible to the carrier). We show how to use the Sulley fuzzing framework to generate fuzzed SMS messages for the smart phones as well as ways to monitor the software under stress. Finally, we present the results of this fuzzing and discuss their impact on smart phones and cellular security.

1 Introduction

The Short Message Service (SMS) is the oldest service existing in addition to the actual mobile telephony service. SMS is loved by the users because it is simple and easy. You punch in a short text message such as `what's up dude?` and hit send. The text almost instantaneously appears on the receivers phone, and thats it from the user's perspective. It is simple and easy. For the mobile phone operator SMS is a good additional business since the messages are relatively expensive and SMS is counted as unreliable therefore messages can be delayed or lost without any repercussions. Also in practice SMS seems pretty reliable.

Under the hood the Short Message Service is much more complex then the user expects. SMS is used for all kinds of other mobile phone related services such as voice mail notification, the Wireless Application Protocol (WAP), multimedia messaging (MMS - the Multimedia Messaging Service), Over-the-Air (OTA) remote phone configuration, and for vendor specific features like the iPhone's visual voice mail. To incorporate all these features the Short Message Service supports sending binary messages that contain additional control information, therefore making the SMS system very complex. Complexity often leads to implementation faults. Implementation faults often lead to security issues.

In the past years we have seen many different SMS-related security problems on mobile phones. The problems range from crashing and rebooting devices to bugs that prevent further reception of SMS messages. From a security stand point SMS is the worst possible case since it is an always on technology. As long as a mobile phone is connected to the mobile phone network SMS messages can be delivered to the device. Further it is uncommon for mobile phone service operators to filter SMS messages on the network. Also since SMS is an essential component of the mobile phone service it cannot be deactivated on a mobile phone. Even if the possibility existed users would most likely never deactivate it, therefore, making SMS the perfect attack vector against mobile and smart phones.

Until now most of the SMS related security issues have been found by accident. People trying to use a specific feature on a specific phone or trying to implementing some SMS-based application and in the process discover a bug or vulnerability. One reason why no systematic analysis of SMS implementations was conducted until now is the fact that sending SMS messages costs money. Because of the lack of access to source code of SMS implementations all testing has to take the black-box approach therefore requiring a huge amount of SMS messages to be sent.

In this work we present a novel way for conducting vulnerability analysis of SMS implementations that does not require sending SMS messages using a mobile operator network. For our approach we inject SMS messages locally into our test phones. The injection is done in software only and requires only application level access to the phone. We inject SMS messages below the software telephony stack and therefore are able to analyze and test all SMS-based services that are implemented on the mobile telephony stack on the respective phones.

The vulnerability analysis itself was conducted using fuzzing. In this paper we present the possibilities for fuzzing-based testing of SMS implementations. We further present our testing tools and test methodology.

2 The Short Message Service

The Short Message Service is a store and forward system, messages sent to and from a mobile phone are first sent to an intermediate component in the mobile phone operators network. This component is called the Short Message Service Center (SMSC). After receiving a message, the SMSC forwards the message to another SMSC or if the receiving phone is handled by the same SMSC, it delivers the message to the recipient without invoking another party.

The SMSC can receive messages from all kinds of sources besides a mobile phone for example from the voice mail system that wishes to inform the mobile phone about a waiting voice mail message.

2.1 The SMS Message Format

SMS messages exist in two formats: SMS_SUBMIT and SMS_DELIVER. The SMS_SUBMIT format is used for messages sent from a mobile phone to the SMSC. The SMS_DELIVER format is used for messages sent from the SMSC to the mobile phone. Since our testing method is based on local message injection that replicates an incoming message, we are only interested in the SMS_DELIVER format.

2.1.1 The SMS_DELIVER Format

An SMS_DELIVER message consists of the fields shown in Table 1. The format is simplified since our main fuzzing targets are the Protocol ID, the Data Coding Scheme, and the User Data fields. Other fields such as the User

Name	Bytes	Purpose
SMSC	variable	SMSC address
DELIVER	1	Message flags
Sender	variable	Sender address
PID	1	Protocol ID
DCS	1	Data Coding Scheme
SCTS	7	Time Stamp
UDL	1	User Data Length
UD	variable	User Data

Table 1: SMS_DELIVER Message Format

Field	Bytes
Information Element (IEI)	1
Information Element Data Length (IEDL)	1
Information Element Data (IED)	variable (IEDL defines length)

Table 2: The User Data Header (UDH).

Data Length and the DELIVER flags will be set to corresponding values in order to create valid SMS_DELIVER messages.

2.1.2 The User Data Header

The User Data Header (UDH) provides the means to add control information to an SMS message in addition to the actual message payload or text. The existence of a User Data Header is indicated through the User Data Header Indication (UDHI) flag in the DELIVER field of an SMS_DELIVER message. If the flag is set, the header is present in the User Data of the message. The User Data Header consists of the User Data Header Length (UDHL), followed by one or multiple headers. The UDHL is the first byte of the User Data in the SMS_DELIVER message. The format for a single User Data Header is shown in Table 2.

3 Mobile Phone Side SMS Delivery

Most current smart phones are build out of two processors. The main CPU, called the application processor, is the processor that executes the smart phone operating system and the user applications such as the mobile telephony and the PIM applications. The second CPU runs a specialized real time operating system that controls the mobile phone interface and is called the modem. The modem handles all communication with the mobile phone network and provides a control interface to the application processor.

Logically the application processor and the modem communicate through one or multiple serial lines. The mobile telephony software stack running on the application processor and communicates with the modem through a text-command-based interface using a serial line interface provided by the operating system running on the application processor. The physical connection between the application processor and the modem solely depends on the busses and interfaces offered by both sides but is irrelevant for our method.

The modems of our test devices (the iPhone, the HTC G1 Android, and the HTC-Touch 3G Windows Mobile) are controlled through the GSM AT command set. The GSM AT commands are used to control every aspect of the mobile phone network interface, from network registration, call control and SMS delivery to packet-based data connectivity.

3.1 The Telephony Stack

The telephony stack is the software component that handles all aspects of the communication between the application processor and the modem. The lowest layer in a telephony stack usually is a multiplexing layer to allow multiple applications to access the modem at the same time. The multiplexing layer also is the instance that translates API-calls to AT commands and AT result codes to status messages. The applications to allow the user to place and answer phone calls and to read and write short messages exist on top of the multiplexing layer.

3.2 SMS Delivery

Short messages are delivered through unsolicited AT command result codes issued by the modem to the application processor. The result code consists

```
+CMT: ,22
07916163838450F84404D0110020009030329021810007040102000880000
```

Figure 1: Unsolicited AT result code that indicates the reception of an SMS message.

of two lines of ASCII text. The first line contains the result code and the number of bytes that follow on the second line. The number of bytes is given as the number of octets after the hexadecimal to binary conversion. The second line contains the entire SMS message in hexadecimal representation. Figure 3 shows an example of an incoming SMS message using the CMT result code which is used for SMS delivery on all of our test devices. There are multiple result codes for delivering incoming SMS messages to the application processor but our investigation results show that all our devices are configured to use the CMT result code. Upon reception of the message the application processor usually has to acknowledge the reception by issuing a specific AT command to the modem. All interaction to the point of acknowledging the reception of the CMT result is handled by the multiplexing layer of the telephony stack.

3.3 The Stacks of our Test Devices

We will shortly describe the parts of the telephony stack that are relevant for SMS handling on each of our test platforms.

3.3.1 iPhone OS

On the iPhone, the telephony stack mainly consists of one application binary called CommCenter. CommCenter communicates directly with the modem using a number of serial lines of which two are used for AT commands related to SMS transfers. It handles incoming SMS messages by itself without invoking any other process, besides when the device notifies the user about a newly arrived message after storing it in the SMS database. The user SMS application is only used for reading SMS messages stored in the database and for composing new messages and does not itself directly communicate with the modem.

3.3.2 Android

On the Android platform the telephony stack consists of the radio interface layer (RIL) that takes the role of the multiplexing layer described above. The RIL is a single daemon running on the device and communicates with the modem through a single serial line. On top of the RIL daemon, the Android phone application (com.android.phone) handles the communication with the mobile phone network. The phone application receives incoming SMS messages and forwards them to the SMS and MMS application (com.android.mms).

3.3.3 Windows Mobile

In Windows Mobile, the telephony stack is quite a bit larger and more distributed compared with the iPhone and the Android telephony stacks. The parts relevant to SMS are: the SmsRouter library (Sms_Providers.dll) and the tmail.exe binary. The tmail.exe binary is the SMS and MMS application that provides a user interface for reading and composing SMS messages. Other components such as the WAP PushRouter sit on top of the SmsRouter.

4 SMS Injection

Based on the results of our analysis on how SMS messages are delivered to the application layer, we designed our SMS injection framework.

Our method for SMS injection is based on adding a layer between the serial lines and the multiplexer (the lowest layer of the telephony stack). We call this new layer `the injector`. The purpose of the injector is to perform a man-in-the-middle attack on the communication between the modem and the telephony stack. The basic functionality of the injector is to read commands from the multiplexer and forward them to the modem and in return read back the results from the modem and forward them to the multiplexer.

To inject an SMS message into the application layer, the injector generates a new CMT result and sends it to the multiplexer just as it would forward a real SMS message from the modem. It further handles the acknowledgement commands sent by the multiplexer. Figure 2 shows the logical model of our injection framework.

We implemented our injection framework for our three test platforms: iPhone OS, Android, and Windows Mobile. We believe that our approach

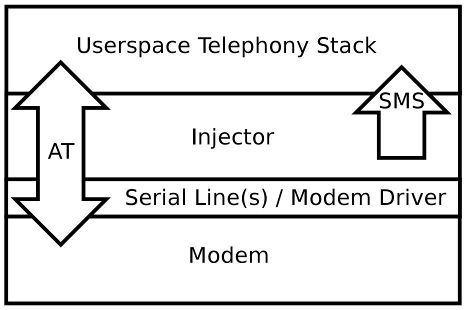


Figure 2: Logical model of our injector framework.

for message injection can be easily ported to other smart phone platforms if these allow application level access to the serial lines of the modem or the ability to replace or add an additional driver that provides the serial line interface.

We noticed several positive side effects of our framework, some of which can be used to further improve the analysis process. First of all, we can monitor and log all SMS messages being sent and received. This ability can be used to analyze proprietary protocols based on SMS, such as the iPhone's visual voice mail. The ability to monitor all AT commands and responses between the telephony stack and the modem provides an additional source of feedback while conducting various tests. On the iPhone, for example, messages are not acknowledged in a proper way if these contain unsupported features.

4.1 The Injection Framework

Below we will briefly describe the implementation issues of the injection framework for each of our target platforms. Every implementation of the framework opens TCP port 4223 on all network interfaces in order to receive the SMS messages that should be injected. This network based approach gives us a high degree of flexibility for implementing our testing tools independent from the tested platform.

So far we are able to install our injection framework on all the test targets and continue to use them as if the injection framework was not installed, therefore giving us high degree of confidence in our approach.

4.2 iPhone

On the iPhone, SMS messages are handled by the CommCenter process. CommCenter is the central control for all communication functionalities of the iPhone such as WiFi, Bluetooth, and the 2G or 3G modem for voice, short messages, and packeted-based communication. The interface for CommCenter consists of sixteen virtual serial lines, `/dev/dlci.h5-baseband.[0-15]` and `/dev/dlci.spi-baseband.[0-15]` on the 2G and the 3G iPhone, respectively.

The implementation of our injection framework for the iPhone OS is separated into two parts, a library and a daemon. The library is injected into the CommCenter process through library pre-loading. The library intercepts the `open(2)` function from the standard C library. Our version of open checks for access to the two serial lines used for AT commands. If the respective files are opened the library replaces the file descriptor with one connected to our daemon. The corresponding device files are the serial lines `3` and `4` on the 2G and 3G iPhones. The library's only function is to redirect the serial lines to the daemon. The daemon implements the actual message injection and log functionality.

Figure 3 shows the shell script that loads our injector framework on the iPhone. In the first step the injector daemon is started. In the second step, the library is copied in to the Libraries directory. Third, the plist responsible for CommCenter is overwritten with our version that contains the necessary settings in order to load our library that hijacks the open(2) call. Our plist is shown in Figure 4.

4.3 Android

The implementation for the Android platform consists of just a single daemon. The daemon talks directly to the serial line device connected to the modem and emulates a new serial device through creation of a virtual terminal.

The injection framework is installed in three steps. First, the actual serial line device is renamed from `/dev/smd0` to `/dev/smd0real`. Second, the daemon is started, opens /dev/smd0real and creates the emulated serial device by creating a TTY named /dev/smd0. In the third step, the RIL process (`/system/bin/rild`), is restarted by sending it the TERM signal. Upon restart, rild opens the emulated serial line and from there on will talk

```
#!/bin/sh
./injector &
mkdir -p /System/Library/Test/
rm -f /System/Library/Test/libopen*
cp libopen* /System/Library/Test/
cp com.apple.CommCenter.plist /System/Library/LaunchDaemons/
 com.apple.CommCenter.plist
>ldpre.log
launchctl unload -w /System/Library/LaunchDaemons/
 com.apple.CommCenter.plist
launchctl load -w /System/Library/LaunchDaemons/
 com.apple.CommCenter.plist
```

Figure 3: Installing and starting the injector.

```
<key>EnvironmentVariables</key>
<dict>
 <key>DYLD_FORCE_FLAT_NAMESPACE</key>
 <string>1</string>
 <key>DYLD_INSERT_LIBRARIES</key>
 <string>
 /System/Library/Test/libopen.0.dylib
 </string>
</dict>
```

Figure 4: The plist entry that injects our library into CommCenter.

```
#!/bin/sh
mv /dev/smd0 /dev/smd0real
/data/myrild &
kill 33 # PID of rild (/system/bin/rild)
```

Figure 5: Installing and starting the injector.

to our daemon instead of the modem.

Figure 5 shows the shell script the loads our injector on the Android platform.

4.4 Windows Mobile

The Windows Mobile version of our injection framework is based on the simple log-driver written by Willem Hengeveld. The original log-driver [4] was designed for logging all AT communication between the user space process and the modem. We added the injection and state tracking functionality. To do this, we had to modify the driver quite a bit in order to have it listen on the TCP port to connect our test tools. The driver replaces the original serial driver and provides the same interface the original driver had and loads the original driver in order to communicate with the modem. The driver is installed through modifying several keys of the Windows Mobile registry at: `/HKEY_LOCAL_MACHINE/Drivers/BuiltIn/SMD0`. The most important change is the name of the Dynamic Link Library (DLL) that provides the driver for the interface, whose key is named `Dll`. Its original value is `smd_com.dll`.

In order to install our injector DLL some steps need to be performed. First the device has to be `Application Unlocked`. This is necessary in order to install unsigned libraries. The unlocking can be performed through changing certain registry values. The details of the Application Unlocking [1] can be easily found on the Internet. Second, the installation steps from the original logdev [4] driver need to be followed.

Once our injector framework is installed it can be activated and deactivated through changing the value of the `Dll` key in `/HKEY_LOCAL_MACHINE/Drivers/BuiltIn/SMD0`. Activate it by changing it from `smd_com.dll` to `injector.dll`. To deactivate change it back to `smd_com.dll`. After the change the device needs to be rebooted.

5 Fuzzing

Fuzzing is one of the easiest and most efficient ways to find implementation vulnerabilities. With this framework, we are able to quickly inject fuzzed SMS messages into the telephony stack by sending them over the listening TCP port. In general, there are three basic steps in fuzzing. The first is test generation. The second is delivering the test cases to the application, and the final step is application monitoring. All of these steps are important to find vulnerabilities with fuzzing.

5.0.1 Fuzzing Test Cases

We took a couple of approaches to generating the fuzzed SMS messages. One was to write our own Python library which generated the test cases while the other was to use the Sulley [2] fuzzing framework. In either case, the most important part was to express a large number of different types of SMS messages. Below are some examples of the types of messages that we fuzzed.

5.0.2 Basic SMS Messages

As from Table 1, we fuzzed various fields in a standard SMS message including elements such as the sender address, the user data (or message), and the various flags.

5.0.3 Basic UDH Messages

As seen in Table 2, we fuzzed various fields in the UDH header. This included the UDH information element and UDH data.

5.0.4 Concatenated SMS Messages

Concatenation provides the means to compose SMS messages that exceed the 140 byte (160 7-bit character) limitation. Concatenation is achieved through the User Data Header type `0` as specified in [3]. The concatenation header consist of five bytes, the type (IEI), the length (IEDL), and three bytes of header data (IED) as seen in Table 3. By fuzzing these fields we force messages to arrive out of order or not at all, as well as sending large payloads.

IED Byte Index	Purpose
0	ID (same for all chunks)
1	Number of Chunks
2	Chunk Index

Table 3: The UDH for SMS Concatenation.

IED Byte Index	Purpose
0 - 1	Destination Port (16bit)
2 - 3	Source Port (16bit)

Table 4: The UDH for SMS Port Addressing.

5.0.5 UDH Port Scanning

SMS applications can register to receive data on UDH ports, analogous to the way TCP and UDP applications can do so. Without reverse engineering, it is impossible to know exactly what ports a particular mobile OS will have applications listening on. We send large amounts of (unformatted) data to each port. The structure of the UDH destined for particular applications of designated ports is indicated in Table 4.

5.0.6 Visual voice mail (iPhone only)

When a visual voice mail arrives, an SMS message arrives on port 5499 that contains a URL in which the device can receive the actual voice mail audio file. This URL is only accessible on the interface that connects to the AT&T network, and will not connect to a generic URL on the Internet. The URL is clearly to a web application that has variables encoded in the URL. We fuzz the format of this URL.

5.1 Delivery

Once the test cases are generated, they need to be delivered to the appropriate application. In this case, due to the way we have designed the testing framework, it is possible to simply send them to a listening TCP port. All of this work is designed to make it easy to deliver the test cases.

5.2 Monitoring

It does no good to generate and send fuzzed test cases if you do not know when a problem occurs. Device monitoring is just as important as the other steps. Unfortunately, monitoring is device dependent. There are two important things to monitor. We need to know if a test case causes a crash. We also need to know if a test case causes a degradation of service, i.e. if the process does not crash but otherwise stops functioning properly.

5.2.1 iPhone

On the iPhone OS, the crash of a process causes a crash dump file to be written to the file system compliments of Crash Reporter. This crash dump can be retrieved and analyzed to determine the kind and position of the crash. In between each fuzzed test case, a known valid test case is sent. The SMS database can be queried to ensure that this test case was received and recorded. If not, an error can be reported. In this fashion, it is possible to detect errors that do not necessarily result in a crash.

5.2.2 Android

The Android development kit takes a different approach by suppling a tool called the Android Debug Bridge (ADB), this tool allows us to monitor the system log of the Android platform. If an application crashes on Android the system log will contain the required information about the crash. If a Java/Dalvik process crashes, it will contain information including the back trace of the application. Like on the iPhone we query the SMS database to be more certain to catch messages that cause problems but not necessarily cause a crash.

5.2.3 Windows Mobile

The Windows Mobile development kit on the other hand provides the tools for on-device debugging. This means Windows Mobile allows traditional fuzzing by attaching a debugger to the process being fuzzed.

6 Results

We found multiple problematic SMS messages on the iPhone and on Android. At the point of writing this white paper we are still busy fuzzing Windows Mobile.

In order to determine if a problematic SMS message that was found using our local injection framework can be abused for an attack it needs to be determined if the specific message can be sent over the real mobile phone network of an operator. The test is quite simple since the message body just has to be copied from the generated SMS_DILVER message to a SMS_SUBMIT message. The SMS_SUBMIT message then can be easily sent using any mobile phone or GSM/3G modem that supports sending of binary messages via AT commands.

In the following we will briefly describe the bugs and vulnerabilities we have found. Of course we will not reveil the actual SMS messages that cause the crash until our presentation.

6.1 iPhone

Our iPhone OS targets were running OS version 2.2 and 2.2.1. We discovered a couple of bugs of which we will describe two particularly interesting ones.

The first bug, or first kind of bug since we discovered multiple instances of it, crashes SpringBoard (the iPhone OS window manager). When SpringBoard crashes it interrupts any currently running application, when it restarts it locks the device and forces the user to slide an unlock. If a pass code is set the user has to enter the pass code. Crashing SpringBoard takes a couple of seconds since it has to write a crash dump, therefore this bug can be utilized for a Denial-of-Service attack where the victim is barred from using his iPhone.

The second bug is much more interesting since it is able to crash the CommCenter process. If you remember our description from earlier section you know that CommCenter manages the iPhones connectivity. If CommCenter crashes the phone completely looses network connectivity (GSM/3G and WiFi). If a phone call is in progress the call is interrupted. This bug can be utilized for a serious Denial-of-Service attack since the victim can be effectively barred from making and receiving phone calls. Figure 6 below shows the result of such an SMS message, the iPhone searches for its mobile phone network.

6.2 Android

Our Android targets were running OS version 1.0, 1.1, and 1.5. We found several bugs on all Android versions of which we will discuss one particular bug that is present in a different form on every Android version we tested.

The bug is similar to the second iPhone bug in the way that it kills the telephony process (com.android.phone) and thus kicks the Android device from the mobile phone network. On Android the bug is a little more interesting since it will permanently kick the target device off the network if the SIM card residing in the phone has a PIN set. The problem is that when com.android.phone is restarted it resets the modem and therefore clears the PIN. After the attack the Android phone is disconnected from the network until the user enters the PIN of the SIM. Until the PIN is entered the user cannot be called, does not receive SMS messages, and of course email and other IP-based services are disabled. The attack is silent, the user is not notified about the bad SMS in any way, and, therefore, very effective. The only way to detect it is to regularly check the phone to see if it shows that the SIM card is locked. Figure 7 shows the crashed com.android.phone process. Figure 8 shows the Java/Dalvik trace of the crash and Figure 9 shows the locked SIM card.

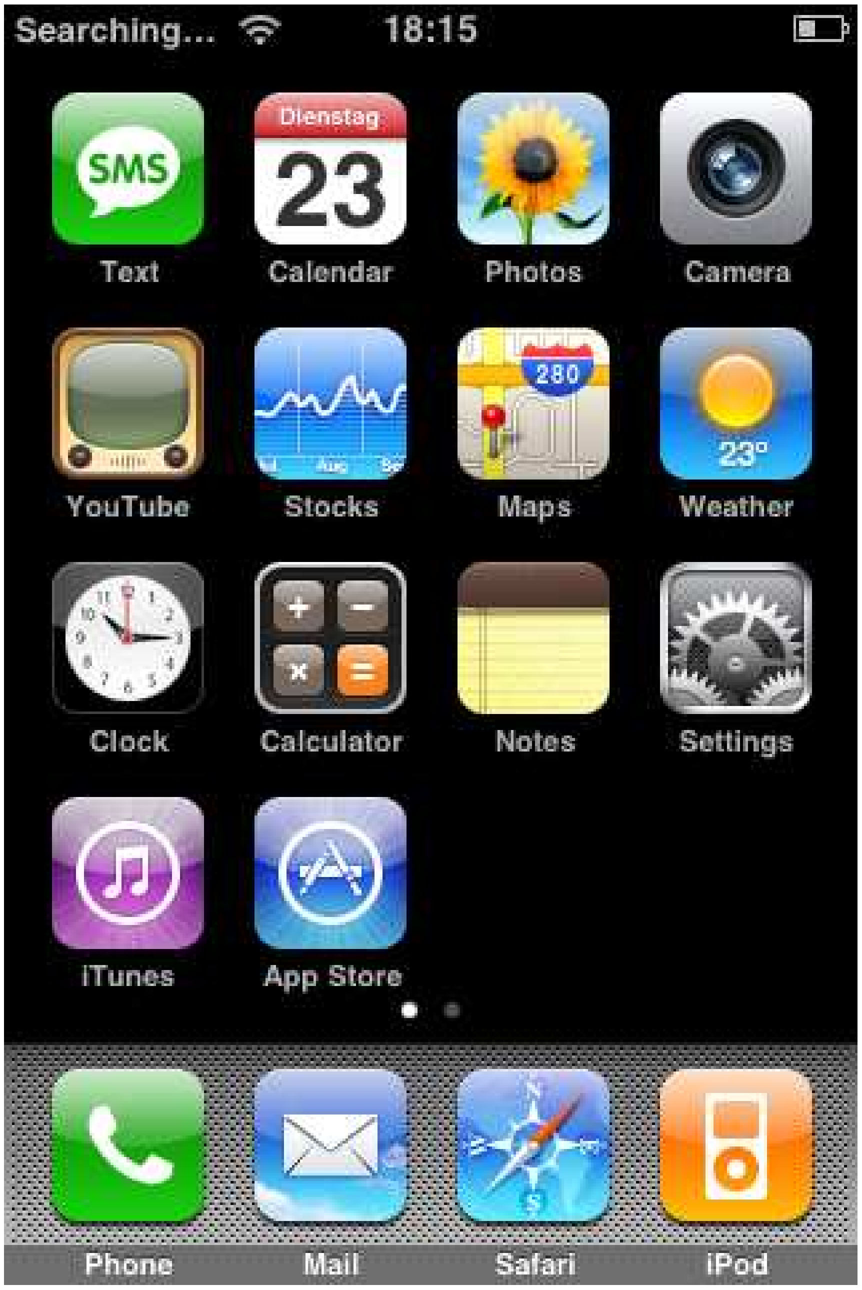
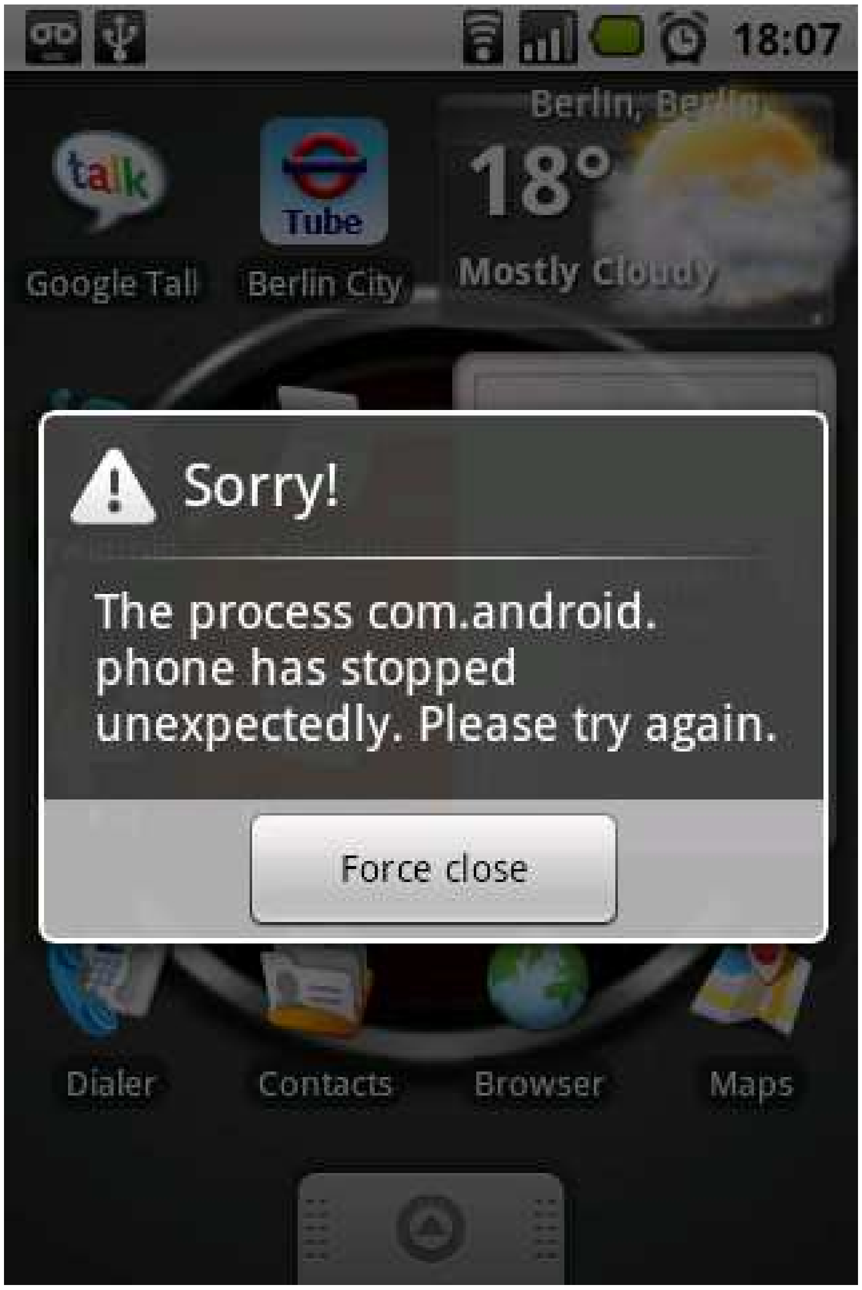


Figure 6: iPhone lost its network connectivity and starts searching.



Sorry!

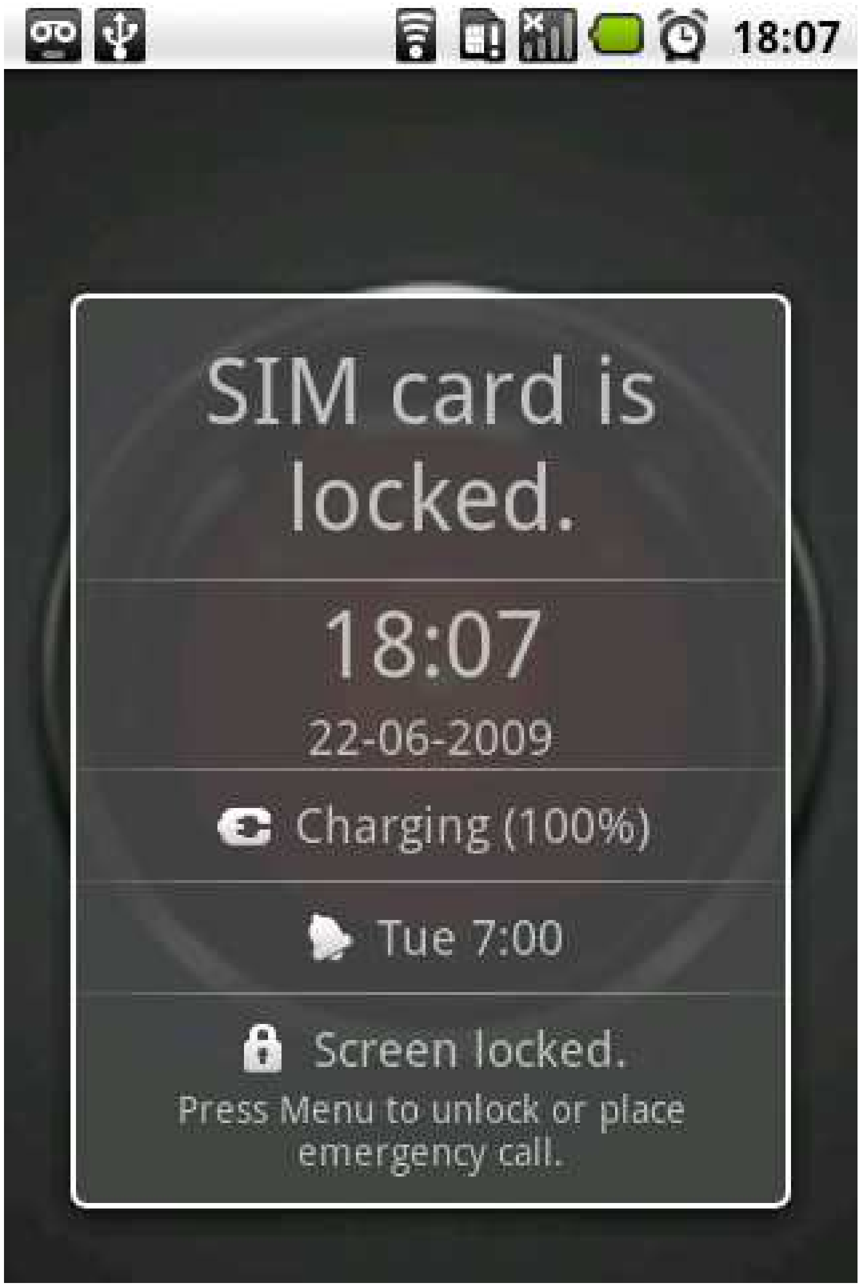
The process com.android.phone has stopped unexpectedly. Please try again.

Force close

Figure 7: com.android.phone crashed.

```
D/WAP PUSH( 7085): Rx: xxxxxxxx(blocked)xxxxxxxxxxx
W/dalvikvm( 7085): threadid=3: thread exiting with uncaught
 exception (group=0x4000fe70)
E/AndroidRuntime( 7085): Uncaught handler: thread main exiting
 due to uncaught exception
E/AndroidRuntime( 7085): java.lang.ArrayIndexOutOfBoundsException
E/AndroidRuntime( 7085): at com.android.internal.telephony.
 WspTypeDecoder.decodeExtensionMedia(WspTypeDecoder.java:200)
E/AndroidRuntime( 7085): at com.android.internal.telephony.
 WspTypeDecoder.decodeConstrainedEncoding(WspTypeDecoder.java:222)
E/AndroidRuntime( 7085): at com.android.internal.telephony.
 WspTypeDecoder.decodeContentType(WspTypeDecoder.java:239)
E/AndroidRuntime( 7085): at com.android.internal.telephony.
 WapPushOverSms.dispatchWapPdu(WapPushOverSms.java:101)
E/AndroidRuntime( 7085): at com.android.internal.telephony.gsm.
 SMSDispatcher.dispatchMessage(SMSDispatcher.java:554)
E/AndroidRuntime( 7085): at com.android.internal.telephony.gsm.
 SMSDispatcher.handleMessage(SMSDispatcher.java:257)
E/AndroidRuntime( 7085): at android.os.Handler.dispatchMessage(
 Handler.java:99)
E/AndroidRuntime( 7085): at android.os.Looper.loop(Looper.java:123)
E/AndroidRuntime( 7085): at android.app.ActivityThread.main(
 ActivityThread.java:3948)
E/AndroidRuntime( 7085): at java.lang.reflect.Method.invokeNative(
 Native Method)
E/AndroidRuntime( 7085): at java.lang.reflect.Method.invoke(
 Method.java:521)
E/AndroidRuntime( 7085): at com.android.internal.os.ZygoteInit
 $MethodAndArgsCaller.run(ZygoteInit.java:782)
E/AndroidRuntime( 7085): at com.android.internal.os.ZygoteInit.main(
 ZygoteInit.java:540)
E/AndroidRuntime( 7085): at dalvik.system.NativeStart.main(Native
 Method)
I/Process (   56): Sending signal. PID: 7085 SIG: 3
I/dalvikvm( 7085): threadid=7: reacting to signal 3
W/ActivityManager(   56): Process com.android.phone has crashed too
 many times: killing!
```

Figure 8: Crash trace for com.android.phone.



18:07

SIM card is locked.

18:07

22-06-2009

Charging (100%)

Tue 7:00

Screen locked.

Press Menu to unlock or place emergency call.

Figure 9: The SIM card locked, phone is not connected to the network.

References

[1] http://robertpeloschek.blogspot.com/2006/03/howto-application-unlock-your-windows.html.

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

[3] 3rd Generation Partnership Project. 3GPP TS 23.040 - Technical realization of the Short Message Service (SMS). http://www.3gpp.org/ftp/Specs/html-info/23040.htm, September 2004.

[4] W. J. Hengeveld. Windows Mobile AT-command log-driver. http://nah6.com/~itsme/cvs-xdadevtools/itsutils/leds/logdev.cpp.