Injecting SMS Messages into Smart Phones for Security Analysis

Collin Mulliner
Deutsche Telekom Laboratories / TU-Berlin
`collin@sec.t-labs.tu-berlin.de`

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

Abstract

The Short Message Service (SMS) is one of the building blocks of the mobile phone service. It is used for text messaging by users as well as for services that work under the hood of every mobile phone. The security of SMS-implementations is critical because attacks can be carried out remotely without any user interaction and because SMS can not be disabled or filtered on current mobile phones. This paper presents a novel method for vulnerability analysis of SMS-implementations. The presented approach is independent from any service operator, does not produce costs, and guarantees reproducible results. Our approach was able to identify previously unknown security flaws that can be used for Denial-of-Service attacks against current smart phones.

Keywords: SMS, Smart Phones, Vulnerability Analysis, Fuzzing.

1. Introduction

The Short Message Service (SMS) is the most popular secondary service on mobile phones beloved by both the users and the service providers for it's ease of use and for the generated revenue, respectively. Besides the use for simple text messaging, the Short Message Service has many applications on a mobile phone. It is used as a control channel for services like voice mail where it is used to notify the user about new messages. Another SMS-based service is remote over-the-air (OTA) phone configuration. SMS further is used as a transport for the Wireless Application Protocol (WAP).

The Short Message Service in all its functionality is complex and therefore security issues based on implementations faults are common. In the past years SMS-based security issues for almost every mobile phone platform were known. Furthermore, no possibility exists to firewall or filter SMS messages, therefore, SMS-based attacks are hard to prevent especially since user interaction is not required. Therefore, it is necessary to develop techniques and tools to analyze and improve the security of SMS-implementations and SMS-based applications.

In this paper we present a novel approach to the vulnerability analysis of SMS-implementations on smart phones. To the best of our knowledge, no attempt has been made before to analyze and test Short Message Service implementations and SMS-based applications in a methodical way. We believe that the main reason for this situation is that SMS testing would be very cost intensive since SMS messages would have to be sent to the tested phones in large quantities.

The analysis of SMS-implementations on smart phones is difficult for several reasons besides the cost factor. The reasons all tail from the fact that SMS messages are delivered through infrastructure controlled by an operator and thus is outside the control of the researchers who are conducting the vulnerability analysis. One problem is the uncertainty of whether a message is delivered to the target in its original form. This is because mobile phone operators have the ability to filter and modify short messages during delivery. Also, it is possible that the operator might not filter messages on purpose but might use equipment that can not handle certain messages. Second, SMS is an unreliable service, meaning messages can be delayed or discarded for no deterministic reason. This makes the testing very time-consuming and hard to reproduce.

We addressed these problems by removing the need for a mobile phone network all together through injecting short messages locally into the smart phone. Injection is done in software only and requires only application level access to the smart phone. The injection is taking place below the mobile telephony software stack and therefore we are able to analyze and test all SMS-based services that are implemented in the mobile telephony software stack.

The vulnerability analysis itself was conducted using fuzzing. In this work, we present the possibilities for fuzzing-based testing of SMS-implementations. Further, we present our testing methodology and the tools we have developed in the process. To show that our approach is generic we implemented and tested our analysis framework for three different smart phone platforms.

So far we have found several flaws in the tested SMS-implementations, some of which can be exploited for Denial-of-Service attacks. One particular vulnerability allows us to disconnect a device from the mobile phone network through crashing the telephony application, leaving the phone in state where it cannot receive calls.

Contributions of this paper are the following:

- We introduce a novel method to test SMS-implementations that circumvents filters and any other restrictions that might be put in place by a mobile phone service operator. It further prevents the provider from easily detecting that testing is taking place. Furthermore, it allows analysis of services and applications built on top of SMS such as WAP.
- We developed a testing framework that allows one to perform SMS vulnerability analysis at high speeds and without costs.
- We developed a tool that performs security testing of SMS-implementations through fuzzing. The tool found a number of previously unknown vulnerabilities.

The rest of this paper is structured as follows: Section 2 presents related work. Section 3 describes the Short Message Service and how messages are transferred between devices. Section 4 describes in great detail how SMS messages are received and handled on a smart phone. Section 5 describes our SMS injection framework. In Section 6 we present our fuzzing tools, the methodology, the results of our fuzzing approach, and the possible attacks based on the results. In Section 7 we briefly conclude.

2. Related Work

Previous research in the area of SMS security can be divided into two areas. The first area consists of research that investigated protocols that facilitate SMS as a transport such as WAP and MMS. Here the creators of the PROTOS [8] testing suite implemented test cases for WAP implementations. In [6] the authors build a framework for analyzing the security of MMS client implementations.

The second area of research conducted in [10] focused around the possibility of mobile phone service disruption based on the ability to send excessive amounts of short messages from the Internet to individuals and groups of people in a certain area.

In the past, SMS bugs [3, 7, 5] were found by accident rather than through thorough testing. One notable example of this kind of bug discovered is the Curse of Silence [9] bug which existed in most of Nokia's Symbian S60-based smart phones. The bug consisted of a single malformed SMS message that, upon reception, prevented further SMS messages from being displayed to the user. The work presented in this paper provides a method for conducting thorough security analysis of SMS-implementations without the burden of services fees.

3. 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.

3.1. The SMS Message Format

SMS messages exist in two formats [2]. 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.

An SMS_DELIVER message consists of the fields shown in Figure 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 Data Length and the DELIVER flags will be set to corresponding values in order to create valid SMS_DELIVER messages.

3.1.1. 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

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

Figure 1. SMS_DELIVER Message Format

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

Figure 2. The User Data Header (UDH).

Data of the message. The User Data Header consists of the User Data Header Length (UDHL), followed by one or multiple headers. The format for a single User Data Header is shown in Figure 2.

4. Mobile Phone Side SMS Delivery

Most current smart phones are composed 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 [4]. 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.

```
+CMT: ,22
07916163838450F84404D0110020009030329
0218100070401020008800O
```

Figure 3. Unsolicited AT result code that indicates the reception of an SMS message.

4.1. 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.

4.2. SMS Delivery

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

4.3. 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.

4.3.1. 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.

4.3.2. 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).

4.3.3. 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.

5. 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 4 shows the logical model of our injection framework.

We implemented our injection framework for our three test platforms. We believe that our approach 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.

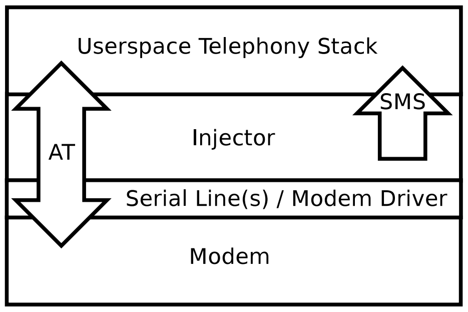

Figure 4. Logical model of our injector framework.

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.

5.1. 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.

5.1.1. iPhone On the iPhone, SMS messages are handled by the CommCenter process. 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.

5.1.2. 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 to our daemon instead of the modem.

5.1.3. Windows Mobile The Windows Mobile version of our injection framework is based on the simple log-driver [11] written by Willem Hengeveld. The original log-driver 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`.

6. Fuzzing SMS Implementations

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.

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

Figure 5. The UDH for SMS Concatenation.

6.1. 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 [1] 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.

Basic SMS Messages As from Figure 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.

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

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 [2]. The concatenation header consist of five bytes, the type (IEI), the length (IEDL), and three bytes of header data (IED) as seen in Figure 5. By fuzzing these fields we force messages to arrive out of order or not at all, as well as sending large payloads.

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 Figure 6.

Visual voice mail (iPhone only) When a visual voice mail arrives, an SMS message arrives on port 5499 that

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

Figure 6. The UDH for SMS Port Addressing.

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.

6.2. 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.

6.3. 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.

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.

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.

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.

6.4. Fuzzing Results

Our iPhone OS target was running software version 2.2. One of the flaws we discovered here is a `null pointer dereference` in the handling code of Flash-SMS messages. The flaw causes SpringBoard (the iPhone OS window manager) to crash forcing the user to `slide to unlock` his iPhone. For Android, the targets were the development firmware versions 1.0, 1.1 and 1.5. Here we found several flaws that cause an `array index out of bounds` exception. Multiple of the flaws cause com.android.phone to crash and thereby disconnect the phone from the mobile phone network. The fuzzing of our Windows Mobile device is still work in progress.

In order to determine if a specific flaw can be exploited the particular SMS message needs to be sent over the mobile phone network. If the message is delivered to the target, and was not modified in the process, it can be utilized for an attack.

7. Conclusions

We presented a novel method for performing vulnerability analysis of SMS-implementation on smart phones. Our method removes the cost factor and thus enables large scale fuzz-based testing. In addition, it removes the intermediate infrastructure that otherwise would make obtaining conclusive results difficult. Removing the infrastructure further creates the possibility to discover flaws that could not haven been discovered through testing using a service providers infrastructure. Through the use of our testing tools, we identified a number of vulnerabilities that can be abused for critical Denial-of-Services attacks.

Future work will focus on porting our framework to other mobile phone platforms for testing and analyzing more SMS-implementations. We further believe that our injection framework can be used beyond the focus of fuzz-based testing, since it provides an unfiltered and cost free path for delivering SMS messages to a smart phone.

Acknowledgments

The authors would like to thank Willem Hengeveld for his kind help with his Windows Mobile log-driver.

References

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

[2] 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.

[3] B. Jurry XFocus Team. Siemens Mobile SMS Exceptional Character Vulnerability. http://www.xfocus.org/advisories/200201/2.html, January 2002.

[4] European Telecommunications Standards Institute (ETSI). GSM 06.06 (ETS 300 642): Digital cellular telecommunication system (Phase 2); AT Command set for GSM Mobile Equipment (ME). http://www.etsi.org, 1999.

[5] J. de Haas. Mobile Security: SMS and a little WAP. http://www.itsx.com/hal2001/hal2001-itsx.ppt, August 2001.

[6] C. Mulliner and G. Vigna. Vulnerability Analysis of MMS User Agents. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, December 2006.

[7] O. Whitehouse @stack Inc. Nokia Phones Vulnerable to DoS Attacks. http://www.infoworld.com/article/03/02/26/HNnokiados_1.html, February 2003.

[8] Oulu University Secure Programming Group. PROTOS Security Testing of Protocol Implementations. http://www.ee.oulu.fi/research/ouspg/protos/, 2002.

[9] T. Engel. Remote SMS/MMS Denial of Service - Curse Of Silence. http://berlin.ccc.de/~tobias/cursesms.txt, December 2008.

[10] W. Enck, P. Traynor, P. McDaniel and T. La Porta. Exploiting Open Functionality in SMS-Capable Cellular Networks. In *Conference on Computer and Communications Security*, 2005.

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