MobiDeke: Fuzzing the GSM Protocol Stack

Sébastien Dudek
Guillaume Delugré
Sogeti / ESEC
Hack.lu 2012
Who we are

Sébastien Dudek
- Has joined the ESEC R&D lab this year (2012) after his internship
- Subject: Attacking the GSM Protocol Stack
- Developer of a GSM fuzzing framework (‘MobiDeke’)

Guillaume Delugré
- Researcher working at Sogeti ESEC R&D lab
- Working on embedded devices / reverse engineering
- Developer of ‘qcombbdbg’ (Qualcomm 3G key Icon255 debugger) and ‘Origami’
Summary

1 Introduction

2 Fuzzing over-the-air

3 The MobiDeke Framework

4 Conclusion
The GSM (2G) network
User Equipments (mobile phones)

Radio control functions are highly timing dependent, so most of the phones use two separated CPUs nowadays.

**The application processor**
- Runs the user OS: Android, iOS, Windows Mobile, and so on
- Documented (often)

**The baseband processor**
- Responsible for handling telecommunications
- Includes stacks for telephony protocols
- Closed binary blob running RTOS

See the talk of Guillaume at 28c3 (Chaos Computer Congress) for more information.
Existing attacks

Many papers have been published:

- Harald Welte: Fuzzing your GSM phone using OpenBSC (2009);
- Collin Mulliner: Fuzzing the Phone in your Phone (2009);
- Collin Mulliner, Nico Golde and Jean-Pierre Seifert: SMS of Death: from analyzing to attacking mobile phones on a large scale (2011);
- Nico Golde: SMS Vulnerability Analysis on Feature Phones (2011);
- Ralf-Philipp Weinmann: Baseband Attacks, WOOT 2012
- ...
What we are looking for

- The baseband: new angle of attack
- Setup a network is easy today thanks to SDR contributions (OpenBTS, OpenBSC, and so on)
  ⇒ Attacking a cellphone ‘over-the-air’ could be fun! (+ not completely explored)

**Typical scenario**

- The attacker controls a rogue base station
- The victim joins the cell and gets remotely exploited

*SDR: Software-Defined Radio*
To reach our goal

- Baseband code is proprietary

**How to find bugs?**

- Reverse-engineering: Hard because the code is very complex
- Fuzzing: The easier way, but need some knowledge and a radio device like the USRP (we’ve won one at hack.lu;) or a nanoBTS, UmTRX, Phi card...

For more information: Harald Welte’s presentation at SSTIC 2010 gives also a good overview of the GSM industry and security
Fuzzing

4 very important points

- Choose your target
  - Targeting the baseband GSM stack (2G), not 3G, HSDPA, LTE...
  - SMS are usually not parsed by the baseband (passed as raw PDUs to the APP)
  - Smartphones: HTC Desire S, Desire Z, iPhone 4S...
- Inject malformed data
- Monitor target activity
- Classify bugs, behaviours
Fuzzing

4 very important points

- Choose your target
- Inject malformed data
  - Smart generation of GSM packets (specs, existing libraries...)
  - Mutate each field of the generated message (describe a structure with Sulley)
- Monitor target activity
- Classify bugs, behaviours
Fuzzing

4 very important points

- Choose your target
- Inject malformed data
- Monitor target activity
  - Quite difficult because we don’t have a debugger
  - Check if the phone is responding
  - Look for strange behaviors / side-effects
  - ...
- Classify bugs, behaviours
Fuzzing

4 very important points

- Choose your target
- Inject malformed data
- Monitor target activity
- Classify bugs, behaviours
  - Crash reporting: report with indicators
  - Replay the recorded payload and see what happens
Summary

1 Introduction

2 Fuzzing over-the-air

3 The MobiDeke Framework

4 Conclusion
GSM layers

Layers

- **Layer 3**
  - Radio Ressource: Channel set up and tear-down
  - Mobility Management: User location...
  - Connection Management: Call (CC), SMS and other services

- **Layer 2**
  - Fragmentation
  - Integrity check

- **Layer 1**
  - Transfers data over the air interface
  - Uses GMSK modulation
  - F/TDMA for multiple accesses
L3 Messages structure

There are 5 standards defining information elements (described in 11.1.14 in the TS 04.07)
State Machines: Originating a Call example (simplified)

- Setup
- Assign
- Connect
- Setup
- End

Observations
- There is often a way to exit from a state machine (e.g., The RELEASE message)
- Sometimes a state requires user interaction
- There are ‘obscure’ elements: present in specs, but never seen in real life...
Message exchanges

Some message exchanges can be considered as stateless, but there are also finite state machines

**Stateless exchanges**
- Simple to fuzz

**Finite state machines**
- Complex and harder to fuzz
- Also harder to program correctly: potential surprises
Let’s fuzz it!

We have set up our network with OpenBTS as follows

But how to send a payload to a targeted cellphone? ⇒ Use the ‘testcall’ feature
The ‘testcall’ feature

- Included in OpenBTS (since 2.5 version) and OpenBSC
- Opens a channel for each targeted IMSI*
- The channel ties to an UDP socket on local computer
- Takes packets as Layer 3 messages and forwards them to the mobile

*IMSI: International Mobile Subscriber Identity
Fuzzing problems...

Testcall exists for fuzzing handsets, but it’s not enough because

- It works for a limited time
- OpenBTS crashes a lot
- The reserved channel is not very stable
  - You’re stuck to your chair while trying to send all your testcases...
- What about the monitoring?
Summary

1 Introduction

2 Fuzzing over-the-air

3 The MobiDeke Framework
    Testcases generation and mutation
    Monitoring
    Report
    Future enhancement

4 Conclusion
To perform our fuzzing tests, we created a framework that:

- Generates and mutates L3 messages
- Sends payload ‘over-the-air’
- Checks if a handset is ready to receive our payload
- Monitors states (Phone and BTS)
- Records a final report
MobiDeke Architecture Diagram
MobiDeke: Data generation and mutation
Methods for data generation and mutation

Creating crafted L3 messages

- Dumb: using captures (MBUS Nokia, OsmocomBB...) and bit-flipping
- Smarter: knowing the structure of the messages
  - gsm_um for scapy: interesting but not complete
  - libmich developed by Benoît Michau: we have chosen this solution for the most part

Mutations

- ‘libmich’ Mutor
- Sulley mutation engine

It is better to combine multiple generation methods to cover as much testcases as possible.
MobiDeke: Monitoring
Methods used to monitor crashes

**Problems**

- Blackbox monitoring
  - Did the baseband crash?

**Solutions**

- Check if the baseband still responds correctly to ‘AT’ commands
- Look for bugs on the application processor by checking crashlogs
- Check the radio channel state reserved by OpenBTS
Check the reserved channel: 'over-the-air'

**Motivations**

- OpenBTS crashes a lot! During that time your fuzzer continues to send payloads...
- Is the reserved channel stable enough?
- Is the baseband ready to receive the next payload?
- Did the baseband crash?

**Solutions**

- Check the radio channel state regularly ⇒ Transaction entries, paging states in OpenBTS.
- Send ‘ping’ requests to the baseband ‘over-the-air’
  - Send a IDENTIFY REQUEST, the mobile will respond with an IDENTIFY RESPONSE
Check ‘AT’ responses with the ’injecATor’ locally

- We checked for phone responsiveness on the radio side
- What about on the local interface?

We modified Collin Mulliner’s ‘injector’ to forward ‘AT’ responses over the opened socket.

- Lack of AT response can indicate a baseband crash/reboot
- Can also be used to simulate user interactions (e.g. accept a phone call)
Application OS bug report

- Even though we are targeting the baseband, some messages still get parsed by the application OS
- Can be the case for SMSs, Information Messages.

```
----- STACK TRACE -----
java.lang.IllegalArgumentException: val.length > 91
   at android.os.SystemProperties.
      set(SystemProperties.java:122)
   at com.android.internal.telephony.PhoneBase.
      setSystemProperty(PhoneBase.java:222)
   at com.android.internal.telephony.gsm.
      GSMPhone.setSystemProperty(GSMPhone.
         java:695)
   at com.android.internal.telephony.gsm.
      GsmServiceStateTracker.
      updateNetworkNameDisplay(GsmServiceStateTracker.
         java:905)
   at com.android.internal.telephony.gsm.
      GsmServiceStateTracker.
      pollStateDone(GsmServiceStateTracker.java:1362)
   at com.android.internal.telephony.gsm.
      GsmServiceStateTracker.
      handlePollStateResult(GsmServiceStateTracker.
         java:1154)
   at com.android.internal.telephony.gsm.
      GsmServiceStateTracker.
```
Logfiles: Android logcats

Some useful data...

- When something crashes, it is likely to be reported in the logcat (Android syslogs)

Extract the information

- Fetch the logs using adb and filter this information
- Check for any known vocabulary in the log that could be related to a crash: ‘***’, ‘uncaught exception’, ‘Error Process’...
Logfiles: iOS CrashReporter

- iOS CrashReporter records application bugs
- Path: `/var/wireless/Library/Logs/CrashReporter`

Note: On Infineon X-Gold (iPhone 1 to iPhone 3) it’s possible to save baseband core dumps in ‘CrashReporter’ if the CORE option is enabled.
The final report

**Indicators**

- 0: The state changed, but everything is fine
- 1: The baseband takes a little bit longer to respond
- 2: Maybe something happened (takes too long to respond, applicative crash...)
- 3: It is probably a crash (can’t talk with the baseband at all...)
- ...

You can define new indicators depending on your analysis.
Sample of a crash in the report

```xml
<?xml version="1.0" ?>
<report>
  <informations>
    <started>
      Fri, 07 Sep 2012 16:18:47
    </started>
    <finished>
      Fri, 07 Sep 2012 16:21:07
    </finished>
  </informations>
  <events>
    <event time="16:18:49" id="0" lastpayload="1658" level="0">
      Fuzzing (re)started
    </event>
    <event time="16:19:52" id="1" lastpayload="1665" level="2">
      AT answer: Timeout!
    </event>
    <event time="16:19:54" id="2" level="0">
      AT is working once again
    </event>
    <event time="16:20:00" id="3" lastpayload="1666" level="3">
      AT Error
    </event>
    <event time="16:20:04" id="4" level="0">
      AT is working once again
    </event>
    <event time="16:20:57" id="5" lastpayload="1674" level="4">
      AT answer: Strange Oo!
    </event>
  </events>
</report>
```
Monitoring enhancement

- Hard without a debugger: a lot of states to check
- We lately managed to get a `qcombbdbg` running on some phones: HTC Desire S/Z
- It’s also possible to debug using the JTAG interface and additional hardware (e.g.: RIFFBOX)
- With a debugger: we don’t need heuristics to detect crashes
Demo!
The fuzzing platform (injecATor, OpenBTS and MobiDeke)
Summary

1 Introduction
2 Fuzzing over-the-air
3 The MobiDeke Framework
4 Conclusion
Problems

- Mostly the unstability of OpenBTS for fuzzing tests
- Deadlocked phones can require human intervention to reboot
- Did not have time to test all layers yet:
  - A lot of fixes required on the monitoring part
- Checking the phone state slows down fuzzing
- We don’t have debuggers for every phone models
- A debugger is always needed to decide about exploitability
Our results

- MobiDeke is a handy way to automate fuzzing tests on GSMs
- Not a lot of bugs have been found with stateless messages
  - **MM_INFORMATION**: few DoS and applicative crashes
  - **TMSI_RELOCATION_COMMAND**: few DoS
  - 1 state of Call Origin: 1 crash and a lot of DoS
  - **LOCATION UPDATING**: not tested completely, few DoS
- A fuzzing test takes time: days, weeks or months (depends on the number of testcases and complexity)

*DoS: The phone was not responding*
Todo

- There are still plenty of vectors to fuzz
- Integration with a debugger (e.g. JTAG)
- Implement state machines
- Source code not to be released at the moment
Thank you! ;)
Any questions?