

Software Security

Security Testing

especially

Fuzzing

Erik Poll

Radboud Universiteit Nijmegen



Security in the SDLC

Last week: static analysis aka **code review tools** aka **SAST**

This week: dynamic analysis aka **security testing** aka **DAST**



Focus of this lecture – and group assignment – is on **testing C(++) code for memory corruption**

The security testing paradox

- Security testing is harder than normal, functional testing
 - We have no idea what we are looking for!
A bizarre input may trigger an obscure bug that is exploitable in some bizarre way, and finding that input with testing is hard
 - Normal users are good testers, as they will complain about functional problems, but they will not complain about many/any security flaws
- Security testing is easier than normal, functional testing
 - We *can* test for some classes of bugs in partly automated way using **fuzzing**
 - Fuzzing is the great success story in (software) security in the past decade .

Fuzzing group project

- Form a team with 4 students
- Choose an **open-source C(++)** application that can take **input from the command line** in some **complex file/input format**
 - For instance, any graphics library for image manipulation
 - Check on <http://lcamtuf.coredump.cx> if it has already been fuzzed with afl; if so, you will have to test old release
- Try out **fuzzing tools (afl, Radamsa, zuff, ...)** to look for security vulnerabilities (esp. memory corruption bugs)
 - with/without **instrumentation (ASan, MSan, valgrind,...)** for additional checks on memory safety
 - Optional variations: investigate bugs, check against known CVEs, introduce bugs, test older releases, try different settings or seed inputs, try other fuzzing tools, ...

Fuzzing group project

1. **Coming days/week: pick an application and hand in Brightspace assignment with application & its input format**
 - Maybe we'll have some discussion about suitability & feasibility
2. **For the rest of Oct & Nov: spend > 4hrs per week to see how far you get & collect results in some report**
 - Good to pick one day to work and/or sync on this with your groups
3. **We will discuss & compare experiences at the end**
 - And maybe along the way

For the fuzzing you can use your university Azure cloud account - and you own computer, of course.

Overview

1. Testing basics
2. Abuse cases & negative tests
3. Fuzzing
 - a) Dumb fuzzing
 - b) Mutational Fuzzing
 - example: OCPP
 - c) Generational fuzzing aka grammar-based fuzzing
 - example: GSM
 - d) Whitebox fuzzing with SAGE
 - looking at symbolic execution of the code
 - e) Code coverage guided evolutionary fuzzing with afl
 - grey box instead of black box

Beware: terminology for various forms of fuzzing is messy

Testing basics

SUT, test suite & test oracle

To test a SUT (System Under Test) we need two things

1. test suite, ie. collection of input data
2. test oracle to decide if response is ok or reveals an error
 - ie. some way to decide if the SUT behaves as we want

Both defining test suites and test oracles can be *a lot of work!*

- In the worst case, a test oracle is a long list which *for every individual test case, specifies exactly what should happen*
- A simple test oracle: *just looking if application doesn't crash*

Moral of the story: crashes are good ! (for testing)

Code coverage criteria

Code coverage criteria can measure how good a test suite:

- **statement coverage**
- **branch coverage**

Statement coverage does not imply branch coverage; eg for

```
void f (int x, y) { if (x>0) {y++};  
                y--; }
```

Statement coverage needs 1 test case, branch coverage needs 2

- More complex coverage criteria exists, eg **MCDC (Modified condition/decision coverage)**, commonly used in avionics

Code coverage metrics can also be used **to guide test case generation** (as afl does)

Possible perverse effect of coverage criteria

High coverage criteria may *discourage* defensive programming, eg.

```
void m(File f) {  
    if <security_check_fails> {log (...);  
                                throw (SecurityException);}  
  
    try { <the main part of the method> }  
    catch (SomeException) { log(...);  
                            <some corrective action>;  
                            throw (SecurityException); }  
}
```

If **defensive code**, ie. the if- & catch-branches, is hard to trigger in tests, programmers may be tempted (or forced?) to remove this code to improve test coverage...

Annotations as test oracle

- Annotations, eg SAL annotations of C/C++ code, can be used as test oracle by doing **runtime assertion checking**
 - So annotations provide a **test oracle for free!** You can test by sending random data & checking if annotations are violated
- Information flow policies can also be used as test oracles
 - Eg SAL's `Tainted=SA_YES` annotations or nicer policy languages discussed later in this course

But: runtime checking for these require heavy instrumentation of the code, to trace the origin of data *inside* the running application, aka **dynamic taint tracking**

**Security testing:
Abuse cases & Negative testing**

testing vs security testing

Difference in focus

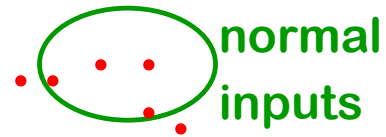
- Normal testing focuses on **correct, desired behaviour** for sensible inputs (aka **the happy flow**), but will include some inputs for borderline conditions
- Security testing also – especially – looks for **wrong, undesired behaviour** for really strange inputs
- Similarly, normal use of a system is more likely to reveal **functional problems** than **security problems**

Security testing is HARD

space of all possible inputs

• some input

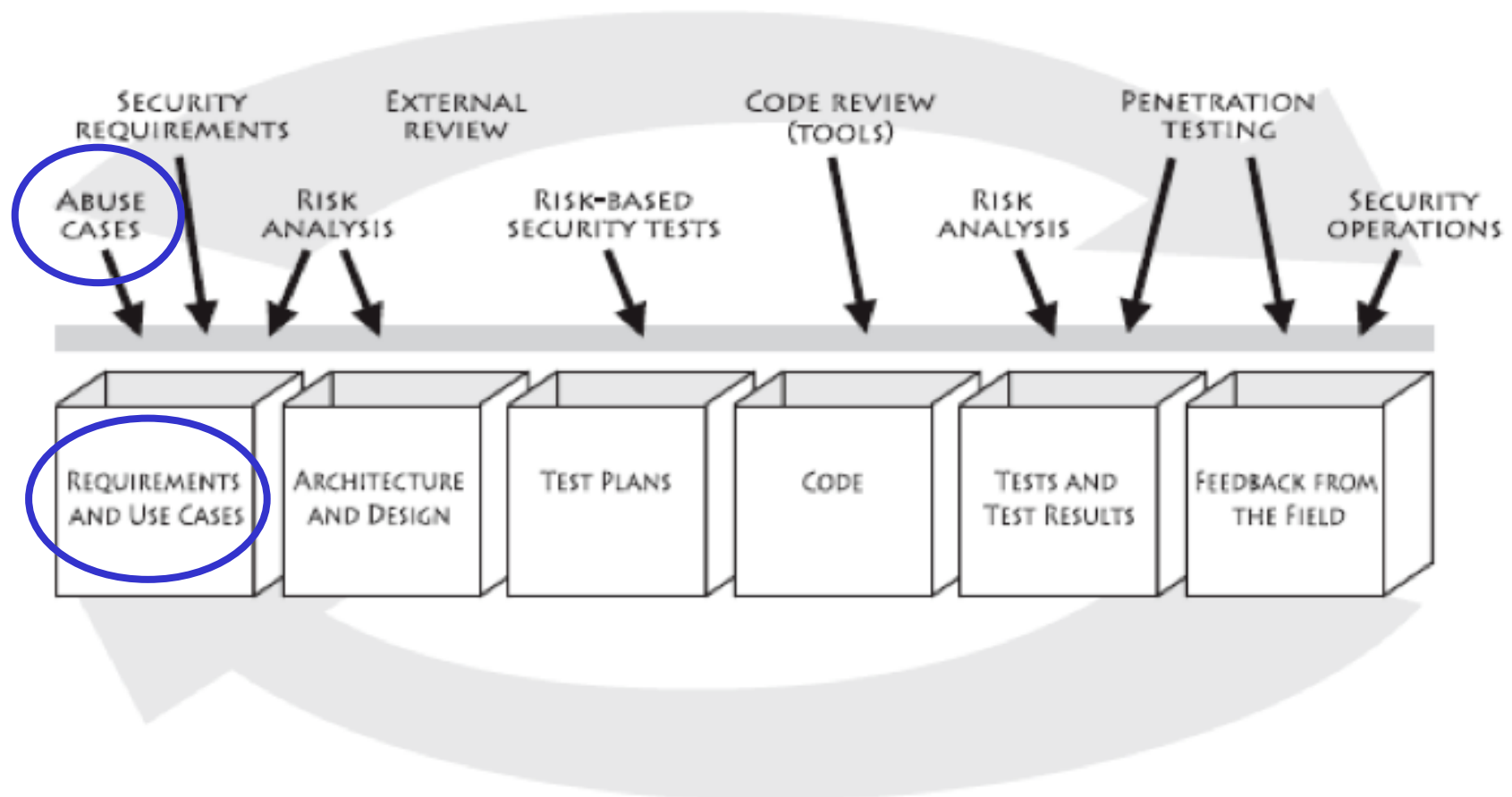
• input that triggers security bug



Abuse cases aka negative test cases

- Thinking about **abuse cases** is a useful way to come up with security tests
 - *what would an attacker try to do?*
 - *where could an implementation slip up?*
- This gives rise to **negative test cases**,
 - i.e. test cases which are *supposed to fail***as opposed to **positive test cases**, which are meant to ***succeed***

Abuse cases – early in the SDCL



iOS goto fail SSL bug

```
...  
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)  
    goto fail;  
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)  
    goto fail;  
    goto fail;  
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)  
    goto fail;  
err = sslRawVerify(...);  
... .
```

Negative test cases for flawed certificate chains

- David Wheeler's 'The Apple goto fail vulnerability: lessons learned' gives a good discussion of this bug & ways to prevent it, incl. [the need for negative test cases](#)

<http://www.dwheeler.com/essays/apple-goto-fail.html>

- The FrankenCert test suite provides (broken) certificate chains to test for flaws in the program logic for checking certificates.

[Brubaker et al, Using Frankencerts for Automated [Adversarial Testing](#) of Certificate Validation in SSL/TLS Implementations, Oakland 2014]

- Code coverage requirements on the test suite would also have helped.

Fuzzing

The idea

Suppose some C(++) binary asks from some input

```
Please enter your username
```

```
>
```

What would you try?

1. ridiculous long input, say a few MB

If there is a buffer overflow, a long input is likely to trigger a SEG FAULT

2. `%X%X%X%X%X%X%X%X`

To see if there is a format string vulnerability

On the command line, we cannot include a **null terminator \0** in an input, but in other situations we may be able to

3. **Other malicious inputs, depending on back-ends, technologies & APIs used:** eg SQL, XML, JSN, Unicode character encodings,...

Out of scope for the project assignment

Fuzzing

- **Fuzzing** aka **fuzz testing** is a highly effective, largely automated, security testing technique
- **Basic idea: (semi) automatically generate random inputs and see if an application crashes**
 - So we are **NOT** testing functional correctness (compliance)
- **The original form of fuzzing: generate very long inputs and see if the system crashes with a segmentation fault.**

How to fuzz

Depending on input type

- very long inputs, very short inputs, or completely blank input
- min/max values of integers, zero and negative values
- depending on what you are fuzzing, include special values, characters or keywords likely to trigger bugs, eg
 - nulls, newlines, or end-of-file characters
 - format string characters `%s %x %n`
 - semi-colons, slashes and backslashes, quotes
 - application specific keywords `halt, DROP TABLES, ...`
 -

Good **validation** and/or **sanitisation** would catch these problems.

More on this in later lecture on secure input handling.

Pros & cons of fuzzing

Pros

- **Very little effort:** test cases are automatically generated, and test oracle is trivial
 - Fuzzing of a C/C++ binary quickly gives a good indication of robustness of the code

Cons

- Only finds **'shallow'** bugs and not **'deeper'** bugs
 - If a program takes **complex inputs** or the program is **stateful**, 'smarter' fuzzing is needed to trigger bugs.
- Crashes may be hard to analyse; but a crash is a clear *true positive* that something is wrong!
 - unlike a complaint from a static analysis tool like PREfast

Improved crash/error detection

Making systems crash on errors is useful for fuzzing!

So when fuzzing C(++) code, the memory safety checks listed in the SoK paper (discussed last two weeks) can be deployed to make crash in the event of memory corruptions more likely

Tools for this include

- `ASan - AddressSanitizer`
- `MSan - MemorySanitizer`
- `valgrind`
 - `MemCheck`

Ideally checks for both spatial bugs (buffer overruns)

& temporal bugs (malloc/free bugs)

Types of fuzzers

- 1) **Mutation-based**: apply random mutations to set of valid inputs
 - Eg observe network traffic, than replay with some modifications
 - More likely to produce interesting invalid inputs than just random input
- 2) **Generation-based** aka **grammar-based** aka **model-based**: generate semi-well-formed inputs from scratch, based on knowledge of file format or protocol
 - with tailor-made fuzzer for a specific input format, or a generic fuzzer configured with a grammar
 - *Downside?*
 - more work to construct this fuzzer or grammar
- 3) **Evolutionary**: observe execution to try to learn which mutations are interesting
 - For example, **afl**, which uses a **greybox** approach
- 4) **Whitebox approaches**: analyse source code to construct inputs
 - For example, **SAGE**

Example mutational fuzzing

Example: Fuzzing OCPP [research internship Ivar Derksen]

- OCPP is a protocol for **charge points** to talk to a back-end server
- OCPP can use XML or JSON messages

Example message in JSON format

```
{ "location": NijmegenMercator215672,  
  "retries": 5,  
  "retryInterval": 30,  
  "startTime": "2018-10-27T19:10:11",  
  "stopTime": "2018-10-27T22:10:11" }
```



Example: Fuzzing OCPP

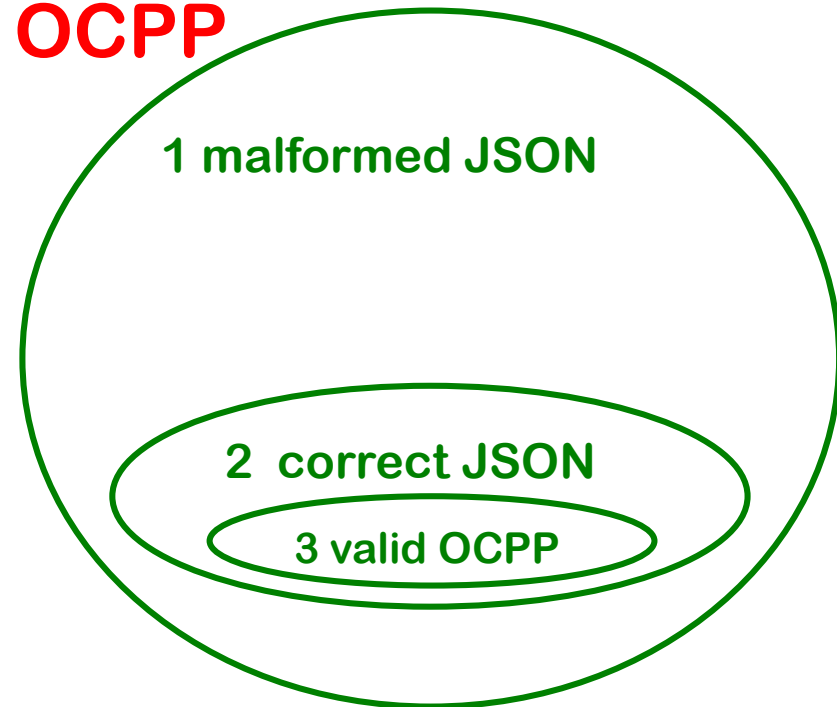
Classification of messages into

1. **malformed JSON/XML**
eg missing quote, bracket or comma
2. **well-formed JSON/XML, but not legal OCPP**
eg with field names not in OCPP specs
3. **well-formed OCPP**

can be used for a simple test oracle:

- **The application should never crash**
- **Malformed messages (type 1 & 2) should generate generic error response**
- **Well-formed messages (type 3) should not**

Note: this does not require *any* understanding of the protocol semantics!
Figuring out correct responses to type 3 would require that.



Test results with fuzzing OCPP server

- Mutation fuzzer generated 26,400 variants from 22 example OCPP messages in JSON format
- Problems spotted by this simple test oracle:
 - 945 malformed JSON requests (type 1) resulted in malformed JSON response
 - Server should never emit malformed JSON!*
 - 75 malformed JSON requests (type 1) and 40 malformed OCPP requests (type 2) result in a valid OCPP response that is not an error message.
 - Server should not process malformed requests!*
- One root cause of problems: the Google's gson library for parsing JSON by default uses **lenient** mode rather than **strict** mode
 - Why does gson even have a lenient mode, let alone by default?
- Fortunately, gson is written in Java, not C(++), so these flaws do not result in exploitable buffer overflows

Postel's Law aka Robustness Principle

“Be conservative in what you send,
be liberal in what you accept”

[Named after Jon Postel, who wrote early version of TCP]

Is this good or bad?

- **Good for getting interoperable implementations up & running** 😊
- **Bad for security**, as it leads to implementations with non-standard behavior, deviating from the official specs, in corner cases, which may lead to **WEIRD BEHAVIOUR** and **BUGS** 😞 😞

Generational fuzzing
aka
Grammar-based fuzzing

CVEs as inspiration for fuzzing file formats

- **Microsoft Security Bulletin MS04-028**
Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution
Impact of Vulnerability: Remote Code Execution
Maximum Severity Rating: Critical
Recommendation: Customers should apply the update immediately

Root cause: a zero sized comment field, without content

- **CVE-2007-0243**
Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability
Critical: Highly critical Impact: System access Where: From remote

Description: A vulnerability has been reported in Sun Java Runtime Environment (JRE). ... The vulnerability is caused due to an error when processing GIF images and can be exploited to cause a **heap-based buffer overflow** via **a specially crafted GIF image with an image width of 0**. Successful exploitation allows execution of arbitrary code.

Note: a buffer overflow in (native library of) a memory-safe language

Generation/grammar/model-based fuzzing

Generational fuzzers generate files or data packets that are slightly malformed or hit corner cases in the spec using knowledge of the input format/protocol

Possible starting point:

grammar defining legal inputs,
or a **data format specification**

Typical things to fuzz:

- **many/all possible value for specific fields**
esp undefined values, or values Reserved for Future Use (RFU)
- **incorrect lengths, lengths that are zero, or payloads that are too short/long**

Tools for building such fuzzers:

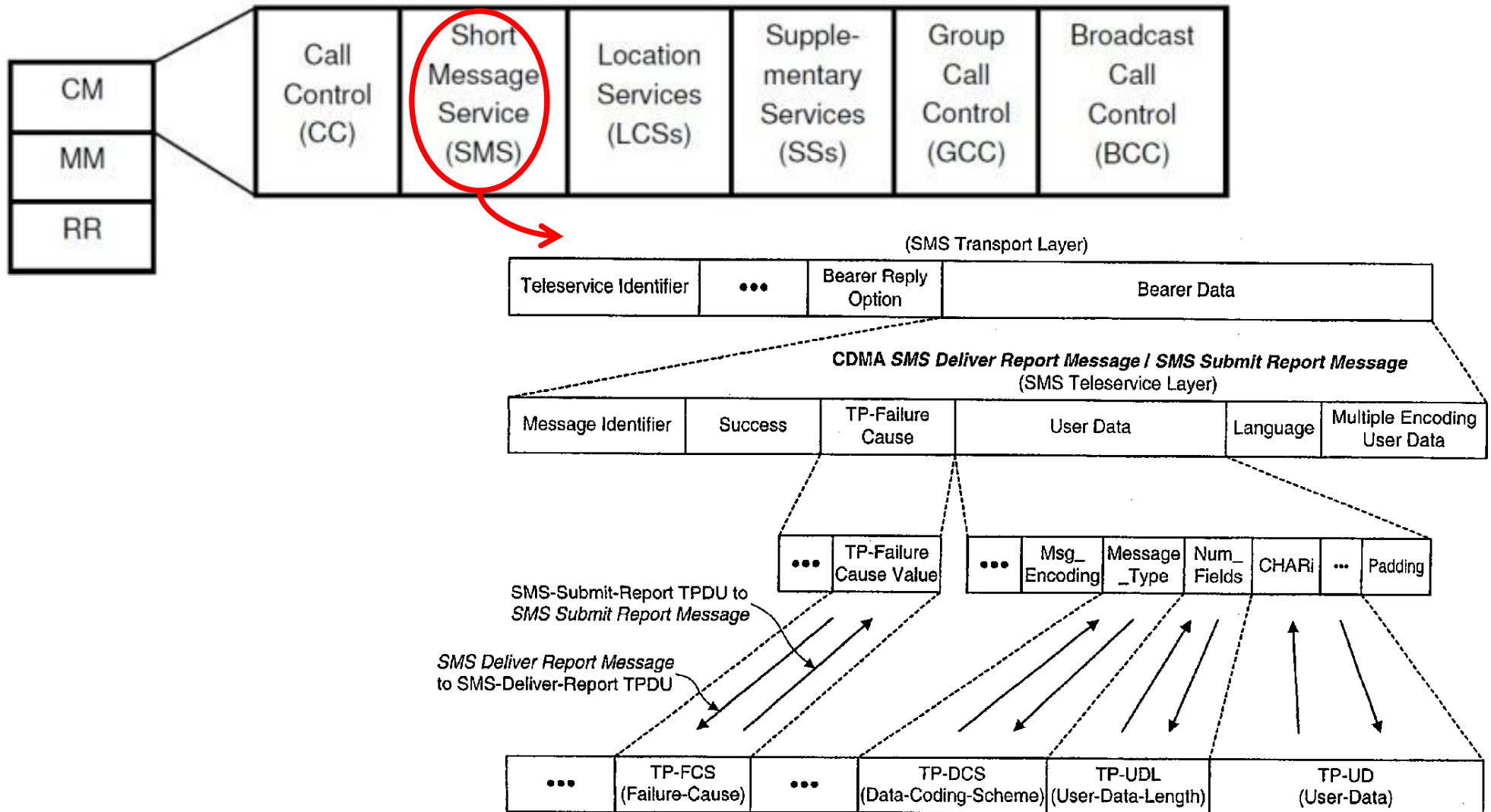
SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

0	4	8	16	19	31
Version	IHL	Type of Service	Total Length		
Identification			Flags	Fragment Offset	
Time To Live		Protocol	Header Checksum		
Source IP Address					
Destination IP Address					
Options				Padding	

Example: generation based fuzzing of GSM

[Master theses of Brinio Hond and Arturo Cedillo Torres]

GSM is a extremely rich & complicated protocol



SMS message fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

Example: GSM protocol fuzzing

Lots of stuff to fuzz!

We can use a **USRP**



with open source cell tower software (**OpenBTS**)

to fuzz any phone



Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones



Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones

- eg possibility to receive faxes (!?)

you have a fax!



Only way to get rid of this icon; reboot the phone

Example: GSM protocol fuzzing

Malformed SMS text messages showing raw memory contents, rather than content of the text message

(a) Showing garbage



(b) Showing the name of a wallpaper and two games



Our results with GSM fuzzing

- Lots of success to DoS phones:
phone crashes, disconnects from network, stops accepting calls,...
 - eg requiring reboot or battery removal to restart, to accept calls again, or to remove weird icons
 - after reboot, the network might redeliver the SMS message, if no acknowledgement was sent before crashing, re-crashing phone

But: **not all these SMS messages could be sent over real network**
- There is surprisingly little correlation between problems and phone brands & firmware versions
 - how many implementations of the GSM stack did Nokia have?
- *The scary part: what would happen if we fuzz base stations?*

[Fabian van den Broek, Brinio Hond and Arturo Cedillo Torres, Security Testing of GSM Implementations, ESSOS 2014]

[Mulliner et al., SMS of Death, USENIX 2011]

Security problem with more complex input formats



effective.
Power
لُصَّبُّلُصَّبُرَّرَّ ٩ ٩h ٩ ٩
π

Example dangerous SMS text message

- This message *can* be sent over the network
- Different characters sets & characters encoding are a constant source of problems. Many input formats rely on underlying notion of characters.

Example: Fuzzing fonts

Google's Project Zero found many Windows kernel vulnerabilities by fuzzing fonts in the Windows kernel

Tracker ID	Memory access type at crash	Crashing function	CVE
1022	Invalid write of <i>n</i> bytes (memcpy)	usp10!otlList::insertAt	CVE-2017-0108
1023	Invalid read / write of 2 bytes	usp10!AssignGlyphTypes	CVE-2017-0084
1025	Invalid write of <i>n</i> bytes (memset)	usp10!otlCacheManager::GlyphsSubstituted	CVE-2017-0086
1026	Invalid write of <i>n</i> bytes (memcpy)	usp10!MergeLigRecords	CVE-2017-0087
1027	Invalid write of 2 bytes	usp10!ttoGetTableData	CVE-2017-0088
1028	Invalid write of 2 bytes	usp10!UpdateGlyphFlags	CVE-2017-0089
1029	Invalid write of <i>n</i> bytes	usp10!BuildFSM and nearby functions	CVE-2017-0090
1030	Invalid write of <i>n</i> bytes	usp10!FillAlternatesList	CVE-2017-0072

<https://googleprojectzero.blogspot.com/2017/04/notes-on-windows-uniscribe-fuzzing.html>

Even handling simple input languages can go wrong!

Sending an extended length APDU can crash a contactless payment terminal.

APDU Response		
Body	Trailer	
Data Field	SW1	SW2



**Found accidentally, without even trying to fuzz,
when sending legal (albeit non-standard) messages**

[Jordi van den Breekel, A security evaluation and proof-of-concept relay attack on Dutch EMV contactless transactions, MSc thesis, 2014]