From Quantitative Change to Qualitative Change -- A New Fuzzing Method on Android

Zhang Qing@xiaomi and Bai Guangdong@SIT

Self Introduction

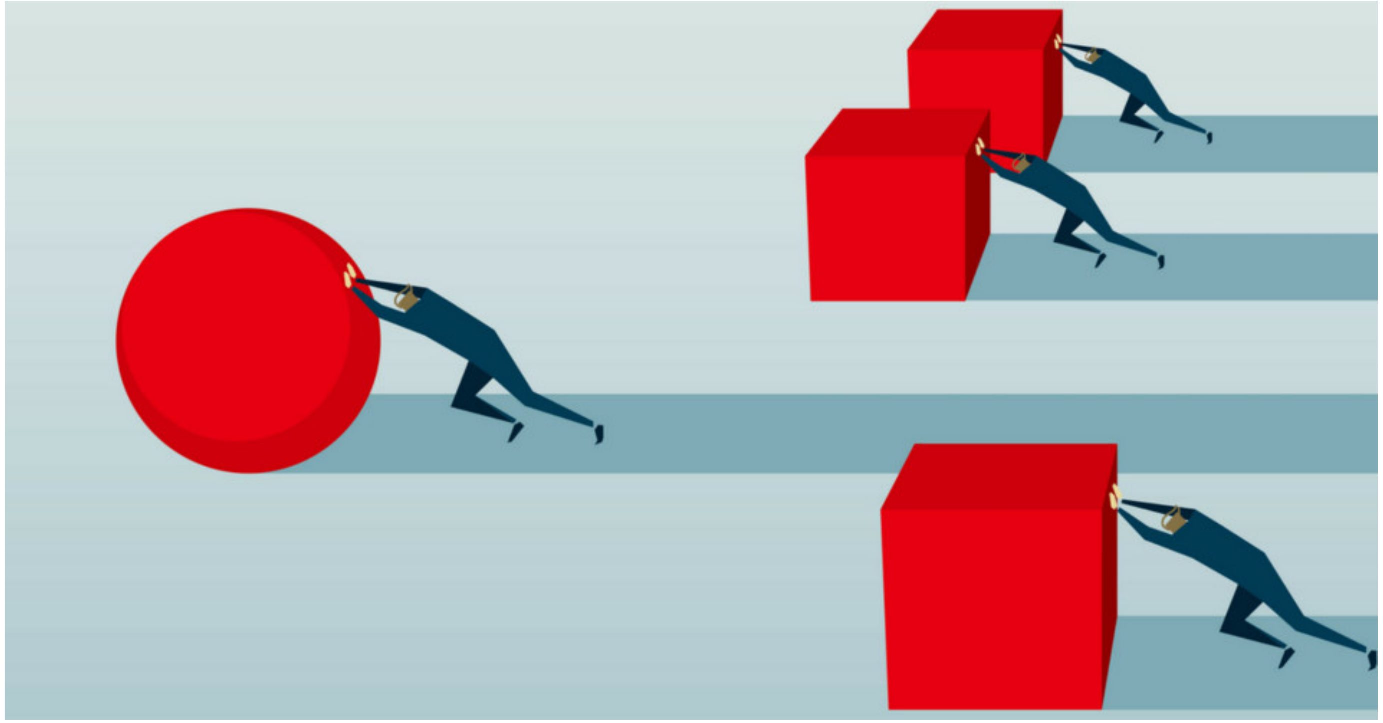
- Zhang Qing
  - Senior Android security researcher from Xiaomi Inc., China
  - Research on Android security and payment security
- Bai Guangdong
  - Lecturer from Singapore Institute of Technology (SIT), Singapore
  - Research on mobile security and protocol analysis

Agenda

- Harder and harder to find new vulnerabilities?
- Traditional fuzzing methods
- Our new approaches
- Case study: several vulnerabilities
- Q&A

Harder and harder to find new vulnerabilities?

- APP
- Third-party libraries
- Binder
- Framework
- Kernel
- … …



Harder and harder to find a Vulnerability?

It is of low-efficiency to find vulnerability by reading the source code

Modern projects are becoming bigger and bigger

What’s more, source code is not always available → have to reverse engineering

- What do we do?
  - Go after the low-hanging fruit
    - Third-party libraries
    - Small-scale apps
    - ……
  - AI?
    - Are you kidding me!?!?
    - There is still a long way to go

Fuzzing

We would also like to thank all security researchers that worked with us during the development cycle to prevent security bugs from ever reaching the stable channel. An additional $14,500 in rewards were issued for security bugs present on non-stable channels.

As usual, our ongoing internal security work was responsible for a wide range of fixes:

[591402] CVE-2016-1642: Various fixes from internal audits, fuzzing and other initiatives.

Many of our security bugs are detected using AddressSanitizer, MemorySanitizer or Control Flow Integrity.

http://googlechromereleases.blogspot.com

http://code.google.com/p/address-sanitizer/wiki/AddressSanitizer
https://code.google.com/p/memory-sanitizer/wiki/MemorySanitizer
https://sites.google.com/a/chromium.org/dev/developers/testing/control-flow-integrity
https://sites.google.com/a/chromium.org/dev/developers/testing/libfuzzer

Fuzzing

Fuzzing has been well researched, and extensively used by android security researchers to identify vulnerabilities.

Basically, current fuzzing methods take into account two coverages:

- Path coverage: static analysis, symbolic execution, etc…
- Input range coverage: test cases of AFL

Fuzzing tools

- Path coverage: cover as many paths as possible
  - Symbolic Execution
  - Soot
  - Bunny
  - Model checking
- Input range coverage: offer enough varieties of inputs
  - Peach
  - AFL
- Other
  - Binder fuzzing
  - Drozer-based fuzzing

A New Fuzzing Perspective

- Philosophy: quantitative change leads to qualitative change (by G. W. F. Hegel)
  - We know something happen because quantitative change leads to qualitative change.
  - Is it the same for vulnerability detecting?
- Use this philosophy to improve existing fuzzing tools
- The derived fuzzing are different from existing approaches
  - Can find much more security vulnerabilities that traditional fuzzing approaches cannot find
  - Why? How?

Core Ideas

- Single function point
  - Quantitative change: multiple times

- Multiple function points
  - Quantitative change: combination

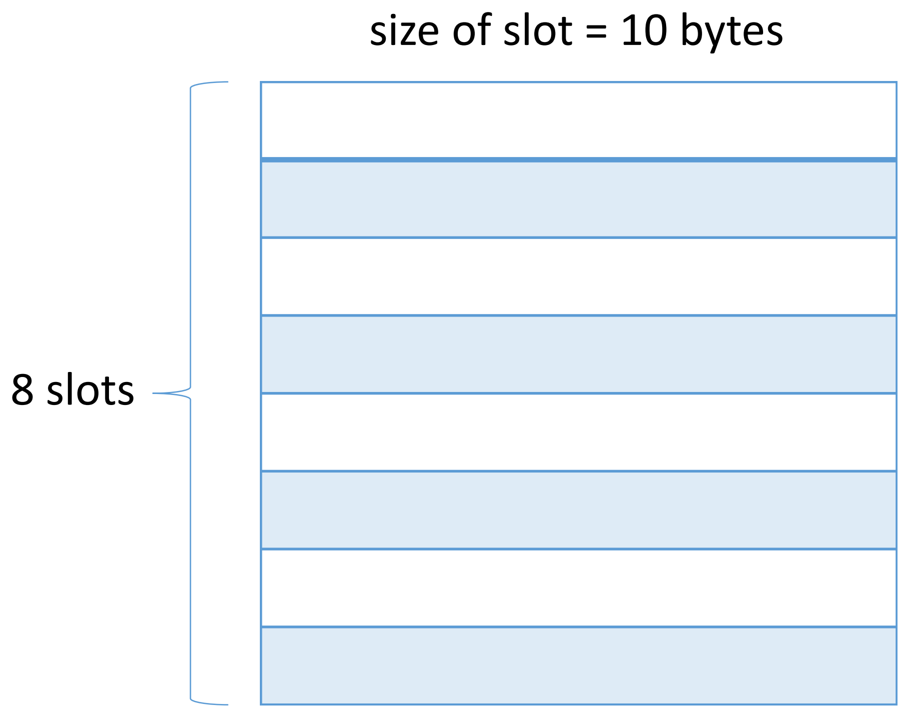
Quantitative Change for Single Function Point

- We feed the same test case to test one exposed function point, and get some unexpected results

- Why?
  - A *write* operation may fail after we have written enough wrong or abnormal data to the system
  - A *read* operation may fail after we have read enough times from abnormal or unavailable data source
  - Others…

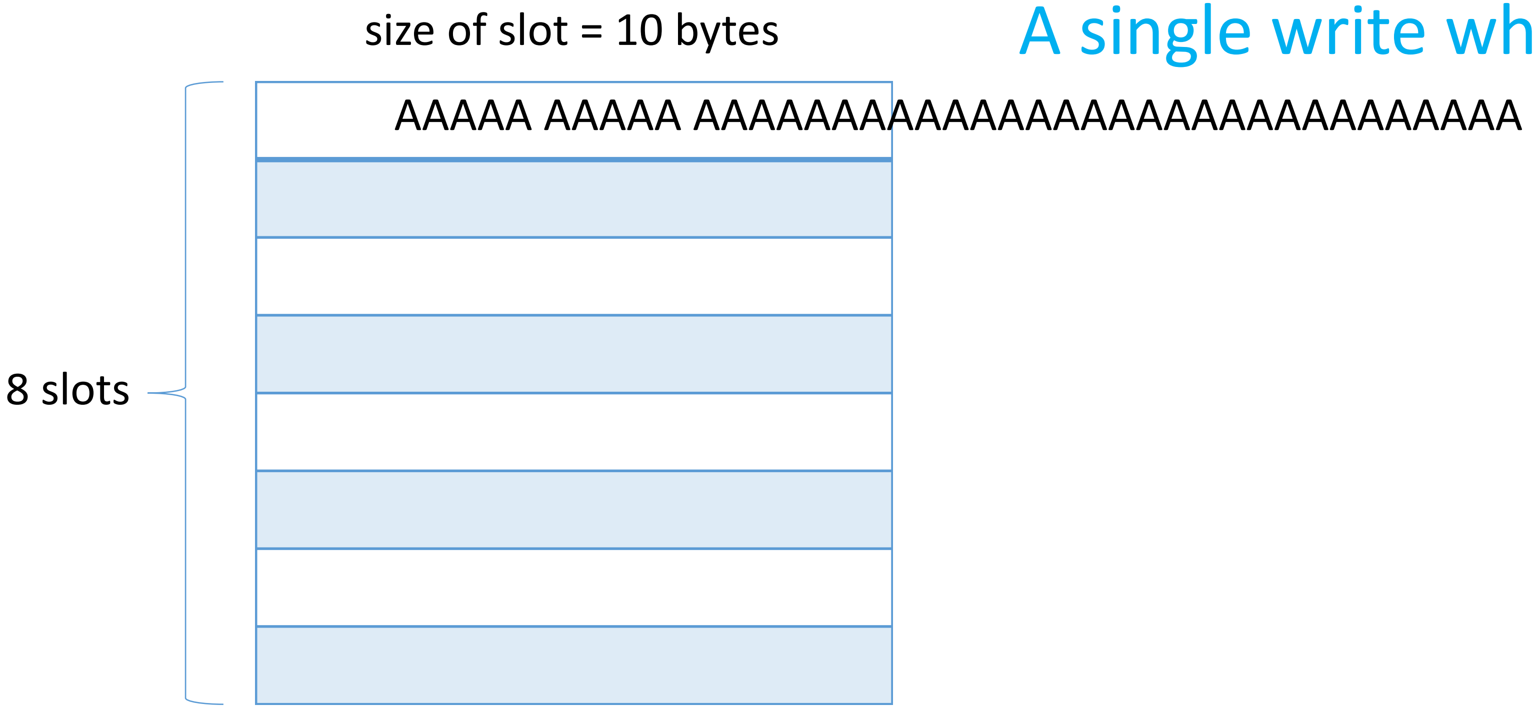
To further elaborate …

- Write



To further elaborate …

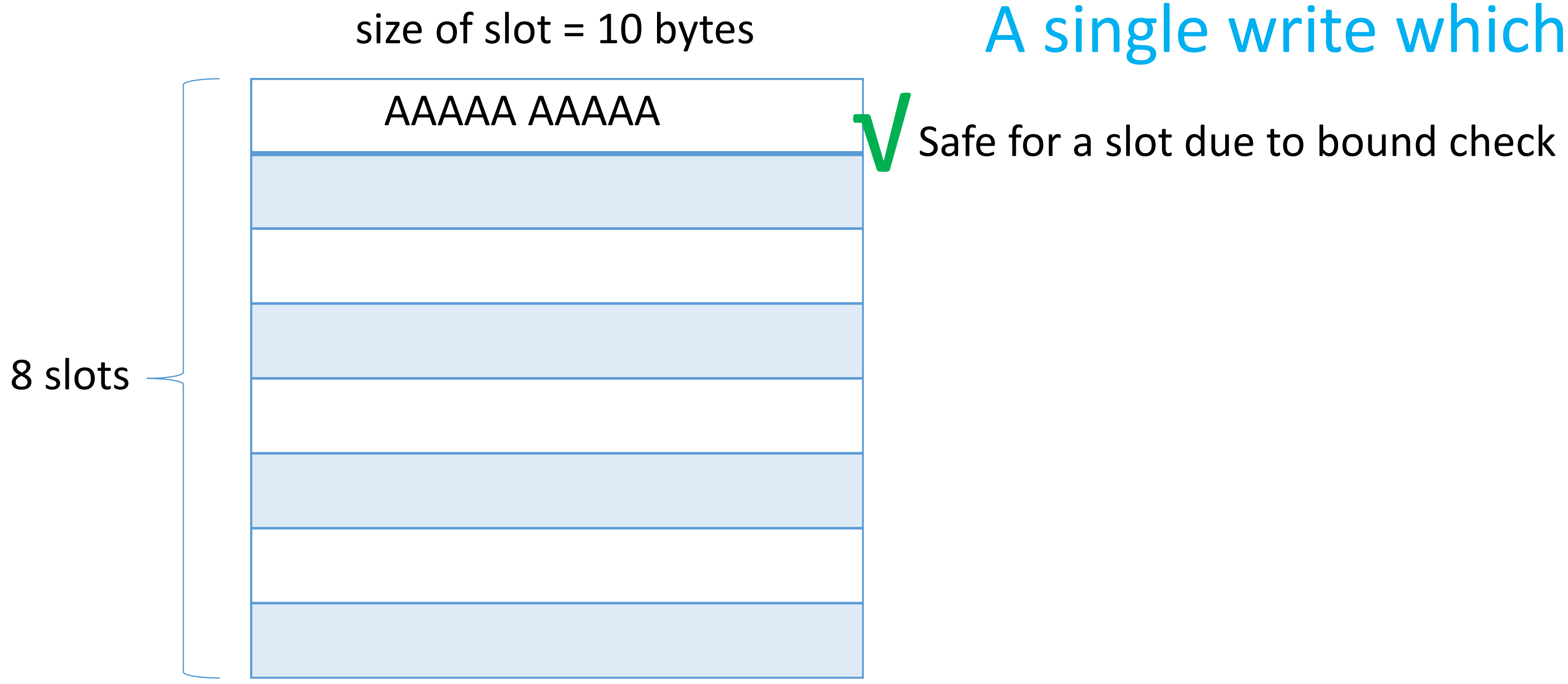
- Write



A single write which exceeds the slot

To further elaborate …

- Write

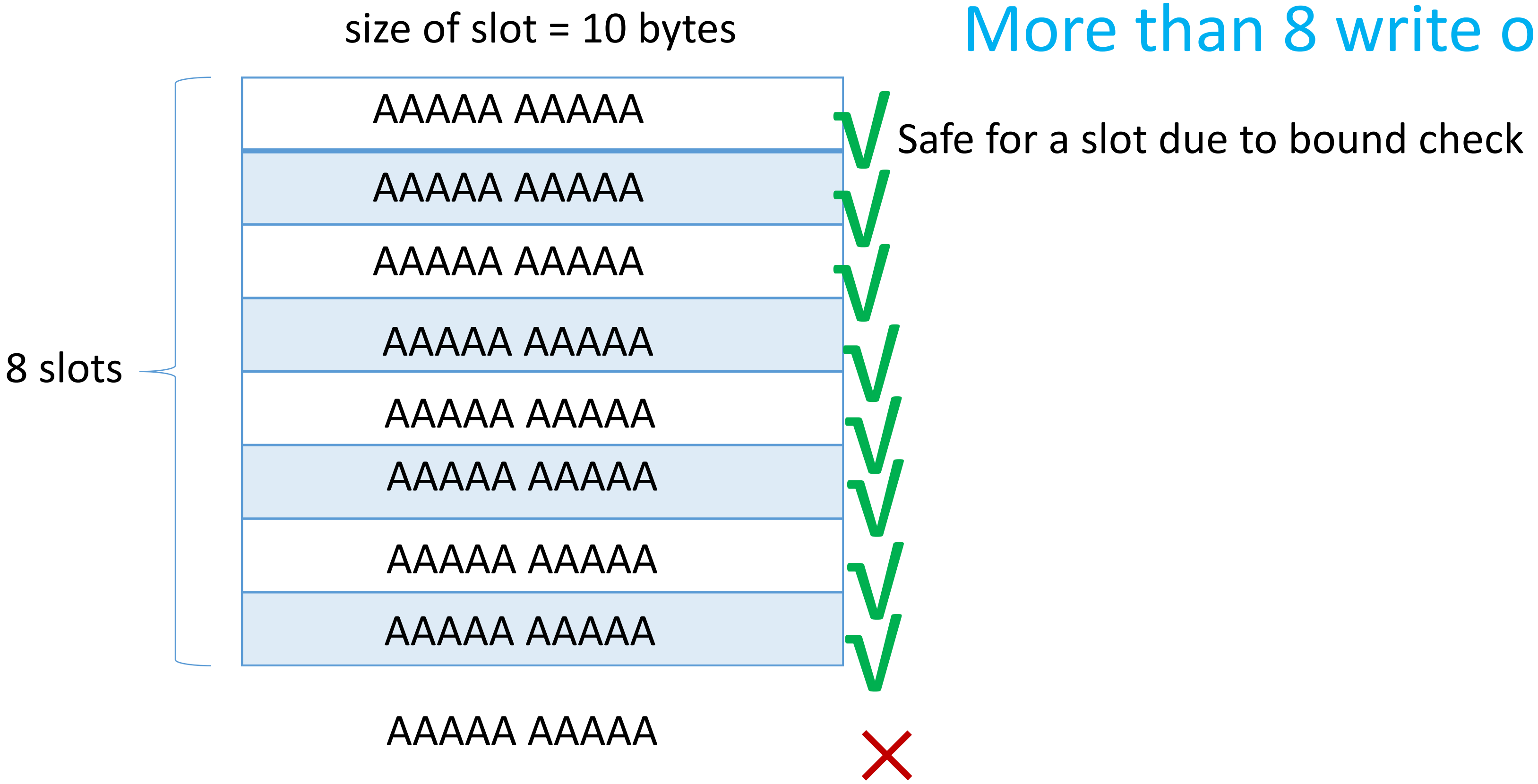


A single write which exceeds the slot

To further elaborate …

- Write

More than 8 write operations



But how do we find this …

1. It happened when we try to locate one vulnerability which cannot be easily identified from the log
2. We had to execute the fuzzing tools for multiple times to reduce the scope
3. During this, we found another vulnerability which is completely different from the one we had been trying to locate
4. The newly found vulnerability is a permanent vulnerability, and initially we didn’t understand the cause. So we had to factory reset the phone for multiple times to analyze it
5. After two-days’ exploration, we found out the cause, which lead to our new idea: quantitative change for vulnerability detection

An example: fuzzing Clipboard

1. Let us test write of clipboard

```
public void test(){
    while (true){
        binderFuzzOnManyWithAAAA("clipboard");
    }
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAA";
        }
    }
}
```

An example: fuzzing Clipboard

1. Let us test write of clipboard
2. Write a string

```
public void test(){
    while (true){
        binderFuzzOnManyWithAAAA("clipboard");
    }
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAA";
        }
    }
}
```

An example: fuzzing Clipboard

1. Let us test write of clipboard
2. Write a string

No matter how complicated and long the string is, no crash is caused

```
public void test(){
    while (true){
        binderFuzzOnManyWithAAAA("clipboard");
    }
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAA";
        }
    }
}
```

An example: fuzzing Clipboard

1. Let us test write of clipboard
2. Write a string

No matter how complicated and long the string is, no crash is caused

3. After we write into clipboard for 200+ times, the system crashed

```
public void test(){
    while (true){
        binderFuzzOnManyWithAAAA("clipboard");
    }
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAA";
        }
    }
}
```

An example: fuzzing Clipboard

1. Let us test write of clipboard
2. Write a string

No matter how complicated and long the string is, no crash is caused

3. After we write into clipboard for 200+ times, the system crashes
4. This is because the garbage data is written into the system partition. What is more, it makes the device a brick: cannot boot any more

```
public void test(){
    while (true){
        binderFuzzOnManyWithAAAA("clipboard");
    }
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAA";
        }
    }
}
```

Another example

When fuzzing one driver, our fuzzer manages to write garbage data to the nv partition, which overwrites the IMEI.

Quantitative Change for Multiple Function Points

- We combine multiple function points in one round of testing, and get some unexpected results

- Why?
  - Some vulnerabilities only happen under a certain system state, which is reached by a sequence of function calls
  - Considering the function calls which may change the system state
    - E.g., Set - Get, Write - Read

But how do we find this …

1. Similar to the previous case, it also happened when we try to locate one vulnerability which cannot be easily identified from the log
2. Again, we had to execute the tools for multiple times to reduce the scope
3. During this, we found another vulnerability which is completely different from the one we had been trying to locate
4. This time, the newly found vulnerability is in a binder service
5. The vulnerability does not appear when executing a function once, but appears in the second time.
6. We found that in the function, the read method A is before the write method B. So the vulnerability does not happen in the first time. When fuzzing in the second time, the vulnerability happens after the read method is executed.

Another Example: Clipboard Again

1. Let us test write of clipboard

Normal fuzzing: sequentially

Method #1

Method #2

Method #3

Method #4

Method #5

Method #6

Method #7

Method #8 read

Method #9

Method #10

Method #11 write

…

Write after read: safe √

```java
public void test(){
    binderFuzzOnManyWithAAAA("clipboard", 11);
    binderFuzzOnManyWithAAAA("clipboard", 8);
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
        }
```

Another Example: Clipboard Again

1. Let us test write of clipboard

Our new method

Read after write

```java
public void test(){
    binderFuzzOnManyWithAAAA("clipboard", 11);
    binderFuzzOnManyWithAAAA("clipboard", 8);
}
public void crashOnServiceWithAAAA(String serviceName, int methodNum) {
    String thevalue = "A";
    for (int i = 0; i < 21; i++) {
        for (int j = 1; j <= i; j++) {
            IBinder iBinder = getTheIbinder(serviceName);
            Parcel parcelData = Parcel.obtain();
            String interfaceName = getTheInterfaceDescriptor(serviceName);
            if (interfaceName == null || interfaceName.equalsIgnoreCase("")) {
                continue;
            }
            parcelData.writeInterfaceToken(interfaceName);
            parcelData.writeString(thevalue);
            setData(parcelData);
            Parcel parcelReply = Parcel.obtain();
            try {
                iBinder.transact(methodNum, parcelData, parcelReply, 1);
            } catch (Exception e) {
                e.printStackTrace();
            }
            parcelData.recycle();
            parcelReply.recycle();
            thevalue += "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";
        }
```

Understand this method

- To some extent, this method is similar to the essence of model checking: exhaustive enumeration
- However, it is not necessary to combine all the function points like model checking does. We only need to combine those core functions
  - E.g., as discussed before, Set - Get, Write - Read

Demonstration

Results

- Using this idea, we have identified about 50 vulnerabilities from various mobile phones
  - Code execution
  - Elevation of privilege
  - Information disclosure
  - Permanent denial of service
- We were ranked #1 in some bounty programs twice in 2017 and 2018

It is not purely philosophic!!

- This new fuzzing approach is under the process of patenting.

Q&A

THANKS

Zhang Qing && Bai Guangdong