Google

Overcoming Stagefright

Integer Overflow Protections in Android

Dan Austin (oblivion@google.com)

May 2016

Agenda

$ whoami

Stagefright

Sanitizers

Sanitizers in Practice

The Future

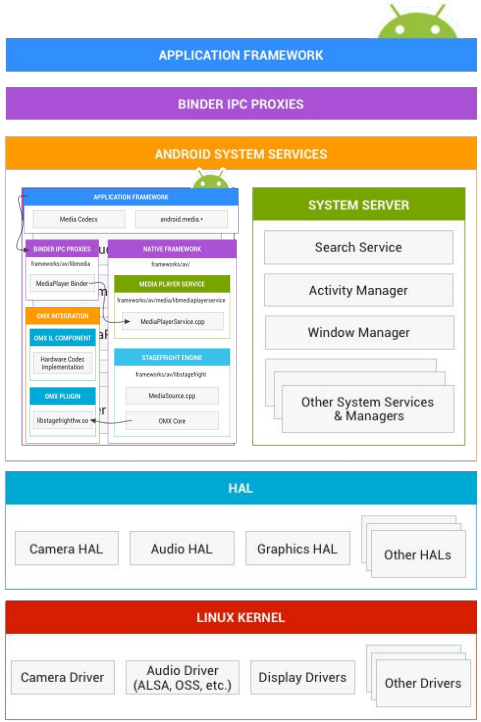
$ whoami

$ whoami

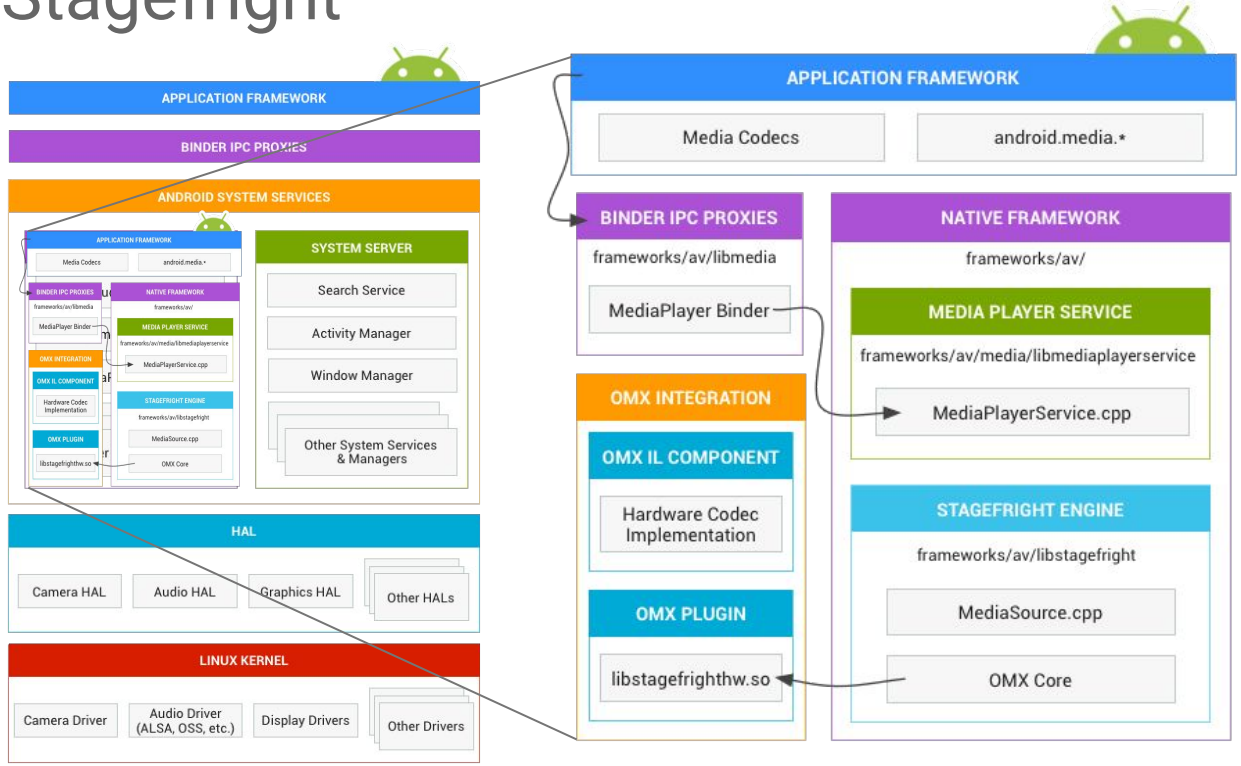
- Dan Austin
- Google since August 2015
- Android Platform Security
- I work on fuzzing and fuzzing accessories!
  - Scalable Fuzzing
  - Smart Fuzzing
  - Compiler-based Defenses
  - Vulnerability Mitigations

Stagefright

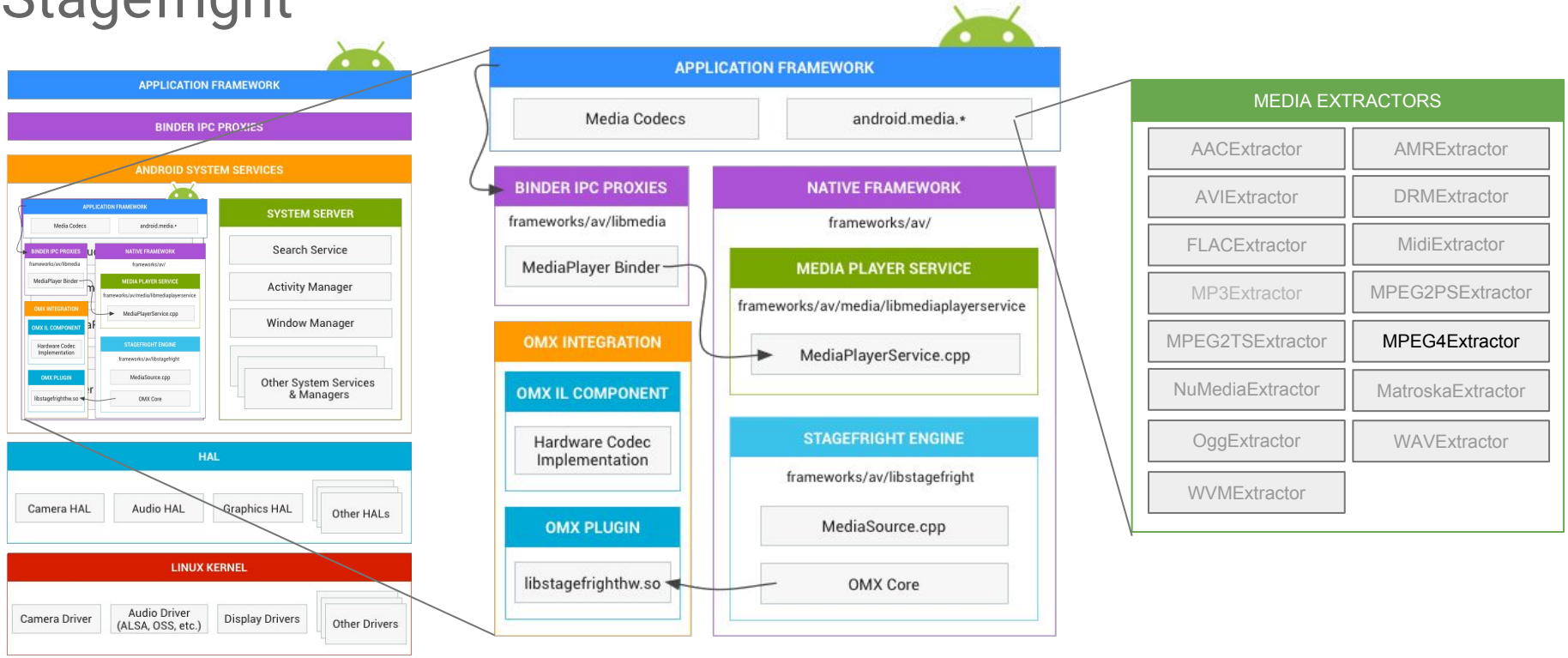
Stagefright



Stagefright

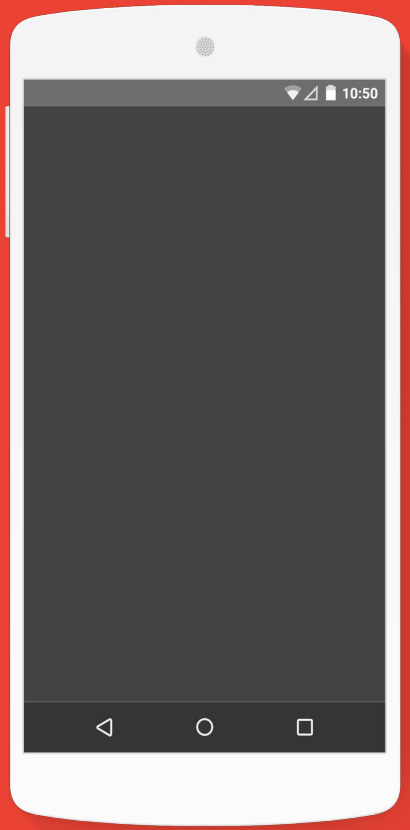


Stagefright



Vulnerability in Stagefright!!!

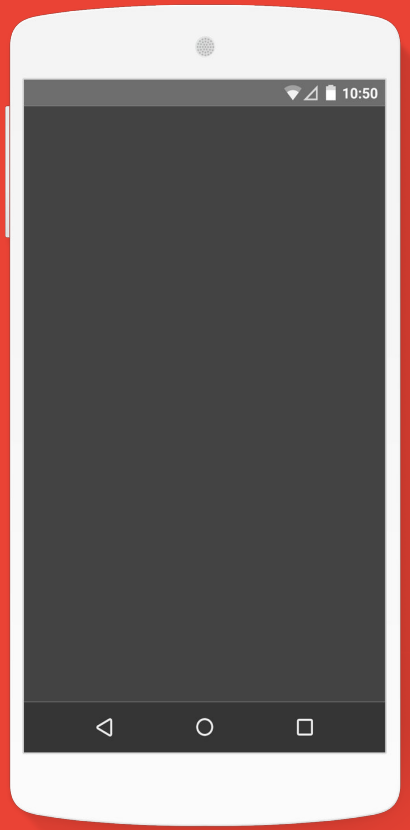
Vulnerability in MPEG4Extractor!



Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

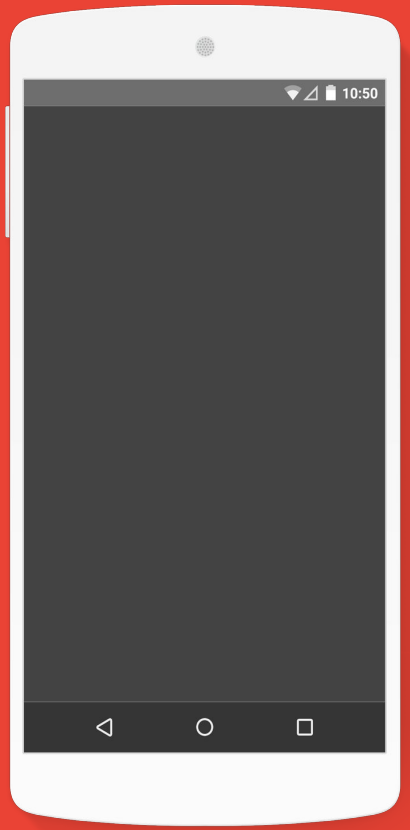


Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

Of type tx3g



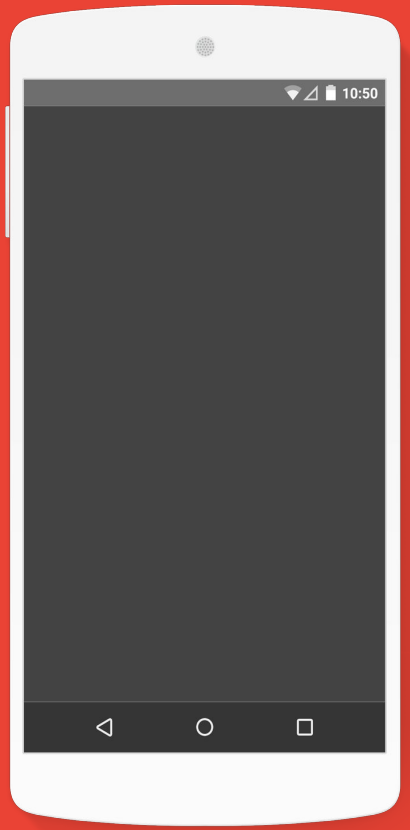
Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

Of type tx3g

That contains a size field



Vulnerability in Stagefright!!!

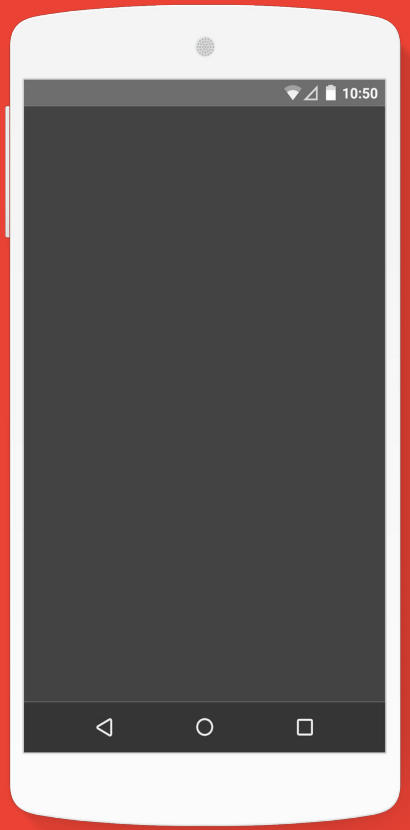
Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

Of type tx3g

That contains a size field

Which is not validated



Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

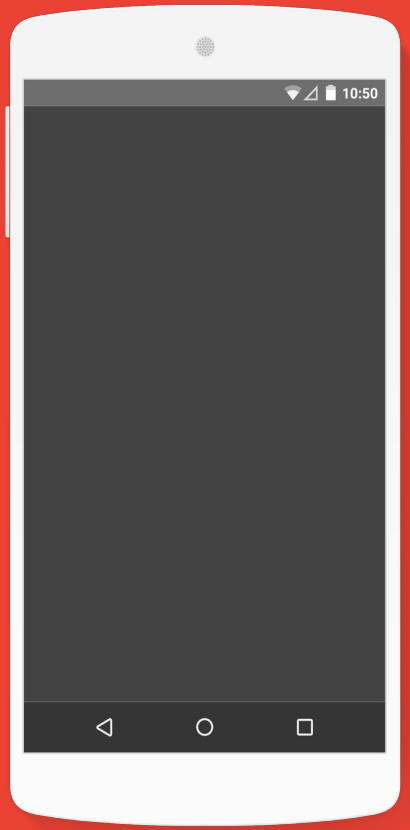
Specifically in parseChunk which, well parses chunks

Of type tx3g

That contains a size field

Which is not validated

And attacker provided



Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

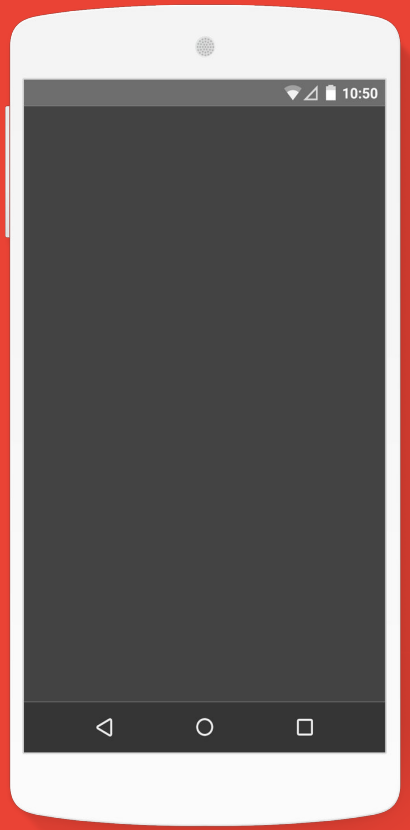
Of type tx3g

That contains a size field

Which is not validated

And attacker provided

That results in an integer overflow



Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

Of type tx3g

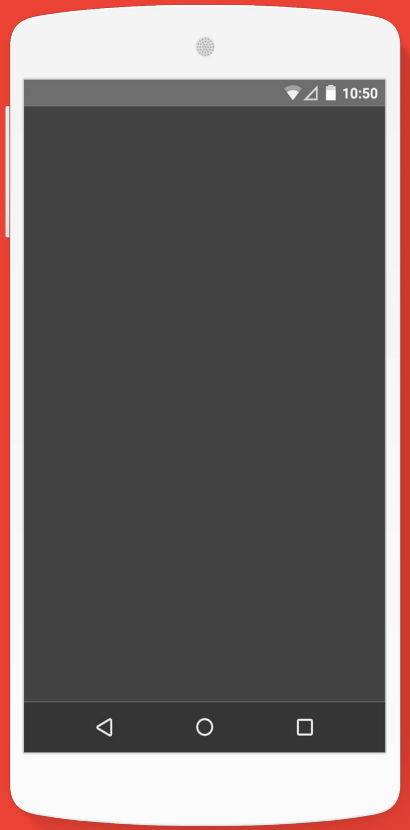
That contains a size field

Which is not validated

And attacker provided

That results in an integer overflow

And memory corruption



Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

Of type tx3g

That contains a size field

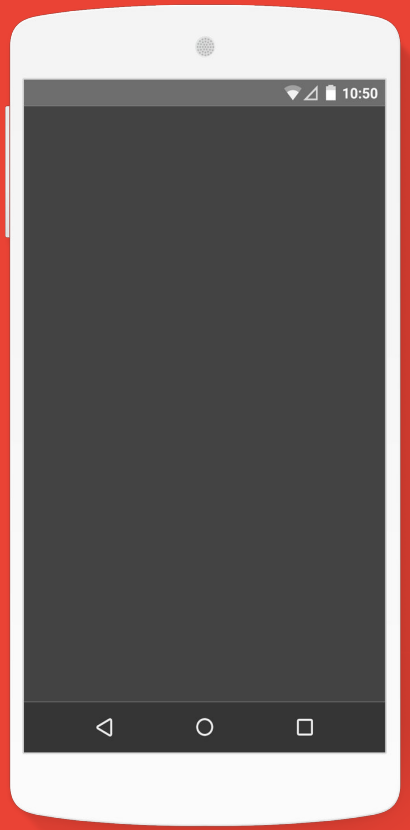
Which is not validated

And attacker provided

That results in an integer overflow

And memory corruption

And ultimately execution...



Vulnerability in Stagefright!!!

Vulnerability in MPEG4Extractor!

Specifically in parseChunk which, well parses chunks

Of type tx3g

That contains a size field

Which is not validated

And attacker provided

That results in an integer overflow

And memory corruption

And ultimately execution...

EVERYBODY FREAK OUT!!!

It’s not all bad...

Vulnerability Researcher provided a patch!

Android was patched in August 2015

Raised visibility of Android's Monthly Security Update Program

It’s not all bad...

Exploitation of the stagefright vulnerability on its own was in the context of mediaserver

- Privesc possible with an additional exploit

Led to a full re-architecture of mediaserver with security in mind

Original PoC required sending an MMS

- Repeatedly
- Which is a bit noticeable

Integer Overflows

Integer Overflows

Integers are kept in a container of finite space

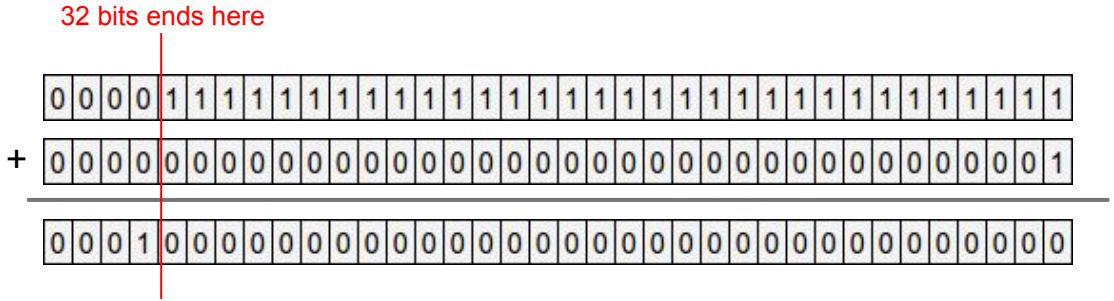
If an arithmetic operation results in a value that can’t be fully kept in that finite space, integer overflow occurs!

Example: 4294967295 + 1 = ?

Integer Overflows

Example: 4294967295 + 1 = ?

Represented as 32 bit values:



So 4294967295 + 1 = 0?

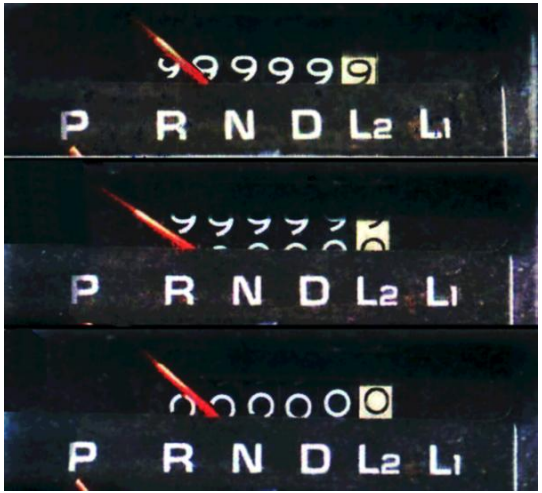
Integer Overflows

In C & C++:

For unsigned values: the result is taken modulo 2^{bits}

For signed values: the result is undefined

Can lead to memory corruption! (CVE-2015-3864!!!)



Exploitable Integer Overflows: How do they work?

```
void overflowable(uint8_t *in_buf, uint32_t in_buf_size) {
  uint8_t *buffer;
  uint32_t buf_size;

  buf_size = in_buf_size + 16;

  buffer = (uint8_t *)malloc(buf_size * sizeof(uint8_t));

  if (buffer != NULL) {
    memcpy(buffer, in_buf, in_buf_size);
  }
}
```

Exploitable Integer Overflows: How do they work?

`in_buf_size` is user controlled, so it can be anything...

```
void overflowable(uint8_t *in_buf, uint32_t in_buf_size) {
  uint8_t *buffer;
  uint32_t buf_size;

  buf_size = in_buf_size + 16;

  buffer = (uint8_t *)malloc(buf_size * sizeof(uint8_t));

  if (buffer != NULL) {
    memcpy(buffer, in_buf, in_buf_size);
  }
}
```

Exploitable Integer Overflows: How do they work?

`in_buf_size` is user controlled, so it can be anything...

If `in_buf_size > 0xfffffffe`, then `buf_size < in_buf_size`

```
void overflowable(uint8_t *in_buf, uint32_t in_buf_size) {
  uint8_t *buffer;
  uint32_t buf_size;

  buf_size = in_buf_size + 16;

  buffer = (uint8_t *)malloc(buf_size * sizeof(uint8_t));

  if (buffer != NULL) {
    memcpy(buffer, in_buf, in_buf_size);
  }
}
```

Exploitable Integer Overflows: How do they work?

`in_buf_size` is user controlled, so it can be anything...

If `in_buf_size > 0xfffffffe`, then `buf_size < in_buf_size`

```
void overflowable(uint8_t *in_buf, uint32_t in_buf_size) {
  uint8_t *buffer;
  uint32_t buf_size;

  buf_size = in_buf_size + 16;

  buffer = (uint8_t *)malloc(buf_size * sizeof(uint8_t));

  if (buffer != NULL) {
    memcpy(buffer, in_buf, in_buf_size);
  }
}
```

If `buf_size < in_buf_size`, then the memcpy will write past the allocated amount, resulting in memory corruption :(

Coding is Hard

Unfortunately, the patch had a flaw…

```
uint32_t type;
const void *data;
size_t size = 0;
if (!mLastTrack->meta->findData(
        kKeyTextFormatData, &type, &data, &size)) {
    size = 0;
}

if (SIZE_MAX - chunk_size <= size) {
    return ERROR_MALFORMED;
}

uint8_t *buffer = new uint8_t[size + chunk_size];

if (size > 0) {
    memcpy(buffer, data, size);
}
```

This is the check that was added in the patch. Unfortunately, SIZE_MAX and size are 32 bits, while chunk_size is 64 bits, which means **overflow can still happen**

```
libc    : Fatal signal 11 (SIGSEGV), code 1, fault addr 0x3232324e in tid 3794 (mediaserver)
pid: 3794, tid: 3794, name: mediaserver  >>> /system/bin/mediaserver <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x3232324e
    r0 b2e90220  r1 32323232  r2 000002a4  r3 00000000
    r4 b2e90240  r5 ffffffe0  r6 b2e90200  r7 00000000
    r8 fffd1da4  r9 bedcf6b8  sl b604b980  fp b604b9d4
    ip bedcece8  sp bedcf1c0  lr b67dff67  pc b67dff76  cpsr 600f0030

backtrace:
    #00 pc 0008ff76  /system/lib/libstagefright.so
(android::MPEG4Extractor::parseChunk(long long*, int)+7613)
    #01 pc 0008fac1  /system/lib/libstagefright.so
(android::MPEG4Extractor::parseChunk(long long*, int)+6408)
    #02 pc 0008fac1  /system/lib/libstagefright.so
(android::MPEG4Extractor::parseChunk(long long*, int)+6408)
    #03 pc 0008de7f  /system/lib/libstagefright.so (android::MPEG4Extractor::readMetaData()+78)
    #04 pc 0008de0b  /system/lib/libstagefright.so
(android::MPEG4Extractor::getMetaData()+8)
    #05 pc 000c0e6f  /system/lib/libstagefright.so
(android::StagefrightMetadataRetriever::parseMetaData()+38)
```

… and exploitation was still possible.

Thanks Project Zero!

UndefinedBehaviorSanitizer

C & C++ have the concept of undefined behavior

Often the cause of subtle bugs...

...such as signed integer overflow…

LLVM has an UndefinedBehaviorSanitizer!

Which adds checks at the code generation level to detect and prevent undefined behavior

Authors also included unsigned integer overflow, which is nice

UBSan: Integer Overflow Sanitization: How does it work?

Implemented in clang as of the CodeGen module (CGExpr & CGExprScalar)

Arithmetic operation (+, -, *) detected and passed to EmitOverflowCheckedBinOp

LLVM Intrinsic corresponding with the operation checks for overflow

Generate code to branch to abort or handler function if an overflow is detected

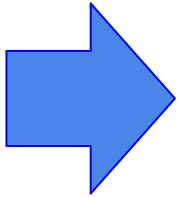
Overflow cannot occur!

What if this were applied to libstagefright?

Sanitizers In Practice

Stagefright before patch

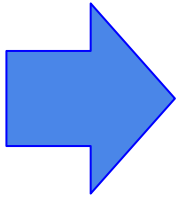
```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```



```
BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
CMP             R0, #1
ITE NE
STRNE           R7, [SP,#0x30]
LDREQ           R7, [SP,#0x30]
LDR             R6, [SP,#0x28]
ADDS            R0, R7, R6
BLX             _Znaj    ; operator new[](uint)
MOV             R8, R0
CBZ             R7, loc_7E6A6
LDR             R1, [SP,#0x40]
MOV             R0, R8
MOV             R2, R7
BLX             __aeabi_memcpy
```

Stagefright after patch v1

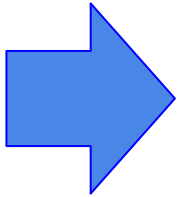
```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    if (SIZE_MAX - chunk_size <= size) {
      return ERROR_MALFORMED;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```



```
BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
CMP             R0, #1
ITE NE
STRNE           R6, [SP,#0x30]
LDREQ           R6, [SP,#0x30]
LDRD.W          R7, R0, [SP,#0x28]
MOVS            R2, #0
MVNS            R1, R7
CMP             R1, R6
MOV.W           R1, #0
IT LS
MOVLS           R1, #1
CMN             R2, R0
ITT EQ
MOVEQ           R2, #1
MOVEQ           R2, R1
CMP             R2, #0
BNE.W           return_ERROR_MALFORMED
ADDS            R0, R6, R7
BLX             _Znaj   ; operator new[](uint)
MOV             R8, R0
CBZ             R6, loc_7E6BC
LDR             R1, [SP,#0x40]
MOV             R0, R8
MOV             R2, R6
BLX             __aeabi_memcpy
```

Stagefright after patch v1, sanitized

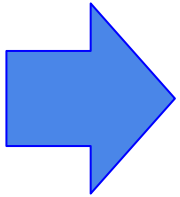
```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    if (SIZE_MAX - chunk_size <= size) {
      return ERROR_MALFORMED;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```



```
                BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
                CBNZ            R0, loc_81F2A
                STR             R5, [SP,#0x38]

loc_81F2A                               ; CODE XREF: .text:00081F26↑j
                LDR             R1, [SP,#0xF4]
                CMN             R5, R1
                BNE.W           call_abort
                LDR             R5, [SP,#0xF0]
                NEGS            R0, R1
                LDR             R7, [SP,#0x38]
                MOVS            R2, #0
                MVNS            R3, R5
                CMP             R3, R7
                MOV.W           R3, #0
                IT LS
                MOVLS           R3, #1
                CMP             R0, #0
                MOV.W           R0, #0
                ITT EQ
                MOVEQ           R0, #1
                MOVEQ           R0, R3
                CMP             R0, #0
                BNE.W           return_ERROR_MALFORMED
                ADDS            R0, R7, R5
                MOV.W           R3, #0
                ADC.W           R1, R1, #0
                CMP             R0, R7
                IT CC
                MOVCC           R3, #1
                CMP             R1, #0
                IT NE
                MOVNE           R3, R2
                CMP             R3, #0
                BNE.W           call_abort
                BLX             _Znaj   ; operator new[](uint)
                MOV             R6, R0
                CBZ             R7, loc_81F86
                LDR             R1, [SP,#0x3C]
                MOV             R0, R6
                MOV             R2, R7
                BLX             __aeabi_memcpy
```

Stagefright before patch v1, sanitized

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size];

    if (size > 0) {
        memcpy(buffer, data, size);
    }

    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);

    delete[] buffer;

    *offset += chunk_size;
    break;
}
```



```
BLX             j__ZNK7android8MetaData8findDataEjPjPPKvS1_
CMP             R0, #1
ITE NE
STRNE           R7, [SP,#0x38]
LDREQ           R7, [SP,#0x38]
MOV             R8, R5
LDRD.W          R5, R1, [SP,#0xF0]
MOVS            R3, #0
MOVS            R2, #0
ADDS            R0, R7, R5
ADC.W           R1, R1, #0
CMP             R0, R7
IT CC
MOVCC           R3, #1
CMP             R1, #0
IT NE
MOVNE           R3, R2
CMP             R3, #0
BNE.W           call_abort
BLX             _Znaj    ; operator new[](uint)
MOV             R6, R0
CBZ             R7, loc_81F62
LDR             R1, [SP,#0x3C]
MOV             R0, R6
MOV             R2, R7
BLX             __aeabi_memcpy
```

UBSan applied to libstagefright

In Summary:

- UBSan with original patch: no integer overflow, stops exploit!
- UBSan with *no patch*: no integer overflow, stops exploit!

SEEMS LEGIT.

UBSan: The Good

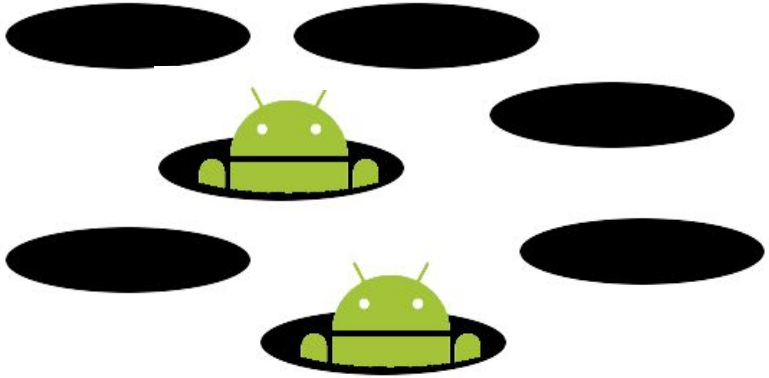
UBSan: The Good

It would have prevented the integer overflow based stagefright vulnerabilities!

It's easy! Just add LOCAL_SANITIZE:=unsigned-integer-overflow to the Android.mk

It's applied everywhere! Catch ALL THE OVERFLOWS!

It's fun! Play whack a mole fixing all that unexploitable undefined behavior in your legacy code base, er, wait...



UBSan: The Bad

UBSan: The Bad

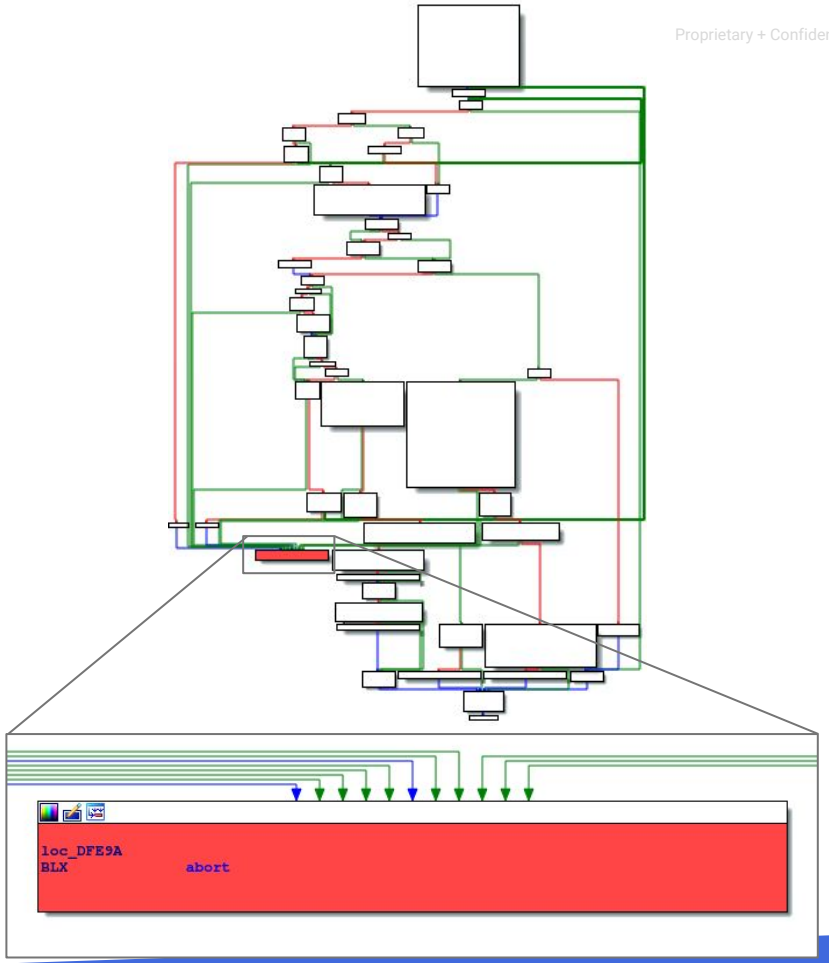
Again, it's applied EVERYWHERE

Even code designed to work with unsigned overflow!

It's not free: some size/execution overhead

Optimized code generation for abort function placement makes debugging hard :(

See ElementaryStreamQueue::dequeueAccessUnitMPEGVideo



UBSan: The Bad

“False Positives”

UBSan is a code health tool being used as a hardening tool

From a security perspective, if an overflow does not influence a memory operation in some way, it’s likely not exploitable

There are lots of overflows in the Android code base that do not influence memory operations at all:

Crypto operations often work modulo $2^{wordsize}$

Codec operations as well

```
while (n--)
```

UBSan: The Ugly

AMR-WB encoder

Legacy code

Lots of arithmetic integer overflows

And stability issues…

Example: “OK, Google” voice recognition

Specifically, this for loop in the coder function

```
for (i = 0; i < L_SUBFR; i++)
{
    Word32 tmp;
    /* code in Q9, gain_pit in Q14 */
    L_tmp = (gain_code * code[i])<<1;
    L_tmp = (L_tmp << 5);
    tmp = L_mult(exc[i + i_subfr], gain_pit);
    L_tmp = L_add(L_tmp, tmp);
    L_tmp = L_shl2(L_tmp, 1);
    exc[i + i_subfr] = extract_h(L_add(L_tmp, 0x8000));
}
```

UBSan: The Ugly

When no integer sanitization, clang generates NEON instructions

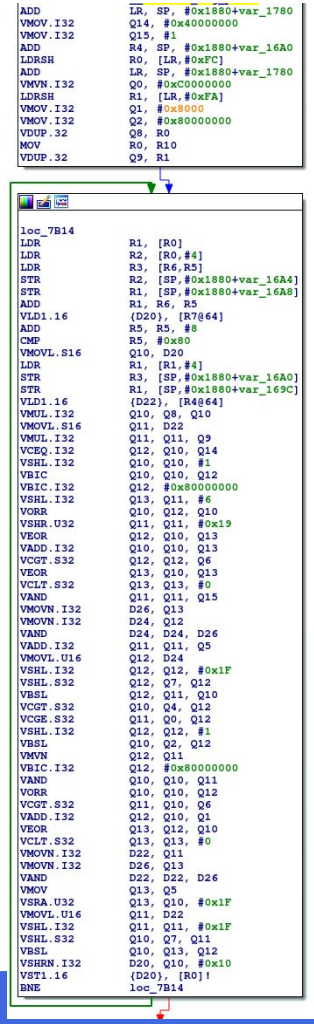
That do not partition the data correctly

With integer sanitization, clang generates normal ARM instructions

Parallelization is broken by the sanitization checks

Data is processed correctly

¯_(ツ)_/¯



The Future

UBSan Runtime

In Android, UBSan overflow detection results in program abort

Great for security, not so good for testing

LLVM upstream UBSan has a runtime library that outputs diagnostic messages instead of aborts

Currently testing the UBSan runtime in Android for platform-wide detection of integer overflows! (AOSP)

Global Integer ~~Domination~~ Sanitization !!!

Integer Overflow Specific Fuzzing

libFuzzer makes fuzzing easy!

1. Write a libFuzzer fuzzer
2. Write a mutator specific to Integer Overflow bugs
3. Include additional logic to better choose paths for further analysis
4. ???
5. Profit!

Questions?

Dan Austin

oblivion@google.com