Boomerang: Exploiting the Semantic Gap in Trusted Execution Environments

Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna
Trusted Execution Environment (TEE)

- Hardware-isolated execution environments (e.g., ARM TrustZone)
  - Non-secure world
    - Untrusted OS and untrusted applications (UAs) (e.g., Android and apps)
  - Secure world
    - Higher privilege, can access *everything*
    - Trusted OS and trusted applications (TAs).
ARM TrustZone

NS Bit
- 0 - Secure or Trusted
- 1 - Non-secure or Non-trusted or Untrusted

Picture reused from arm.com
Untrusted OS ↔ Trusted OS

- Untrusted applications (UAs) request trusted applications (TAs) to perform privileged tasks.

- TAs should verify the request and perform it only if the request is valid.
  - **Example:** Sign the contents of a memory region
    - TA should check if the **requested memory region belongs to untrusted OS** before computing the signature of it.
Untrusted OS ↔ Trusted OS

Non-Secure World | Secure World
---|---
Untrusted Application (UA) | Trusted Application (TA)
Untrusted OS | Trusted OS
Untrusted OS ↔ Trusted OS

Non-Secure World | Secure World

Untrusted Application (UA) <-> Trusted Application (TA)

Untrusted OS

Trusted OS

Userspace

Supervisor
Untrusted OS ↔ Trusted OS

Non-Secure World

Untrusted Application (UA)
Library
Driver Interface (ioctl)
Untrusted OS

Secure World

Trusted Application (TA)
Trusted OS

Userspace

Supervisor
Untrusted OS ↔ Trusted OS

Non-Secure World

Untrusted Application (UA)
Library

Driver Interface (ioctl)

Untrusted OS

TEE Interface

SMC

Trusted OS

Secure World

Trusted Application (TA)

Userspace

Supervisor
Untrusted OS ↔ Trusted OS

Non-Secure World | Secure World

Untrusted Application (UA)

Library

Driver Interface (ioctl)

Untrusted OS

TEE Interface

Trusted Application (TA)

Trusted OS

Userspace

Supervisor
Untrusted OS ↔ Trusted OS

Non-Secure World | Secure World

Untrusted Application (UA)

Library

Driver Interface (ioctl)

Untrusted OS

Trusted Application (TA)

Trusted OS

Userspace

Supervisor
Communication with TA

- Requests to TA can contain pointers.

```
struct keymaster_sign_data_cmd {
    uint32_t data_ptr; // Pointer to the data to sign
    size_t dlen; // length of the data to sign
};
```

Structure of a sign request to KeyMaster TA.
Pointer translation and sanitization in untrusted OS

- Memory model could be different in untrusted and trusted OSes.

- One should use physical address for all pointer values between trusted and untrusted OSes.
Pointer translation and sanitization in untrusted OS

- **Sanitization**: Untrusted OS should check that the UA has access to the pointer provided in the request.

- **Translation**: Convert the virtual address to physical address.

- We call this **functionality in untrusted OS** as PTRSAN.
Example PTRSAN

```c
int ptr_san(void *data, size_t len, phy_t *target_phy_addr)
{
    if(!access_ok(VERIFY_WRITE, data, len)) {
        return -EINVAL;
    }
    *target_phy_addr = get_physical_address(data);
    return 0;
}
```
Handling untrusted pointers in trusted OS

- Check if the physical address indicated by the pointer belongs to the non-secure memory.
  - Protect trusted OS against untrusted OS

- Trusted OS (or TA) has no information about the UA which raised the request.
Handling untrusted pointers in trusted OS

- Check if the physical address indicated by the pointer belongs to the non-secure memory.
  - Protect trusted OS against untrusted OS

- Trusted OS (or TA) has no information about the UA which raised the request.
Bypassing Sanitization

Non-Secure World | Secure World

Untrusted Application (UA) | Trusted Application (TA)

Untrusted OS | Trusted OS

Supervisor

Userspace

Untrusted OS

Trusted OS

Unknown

VA

PA
Bypassing Sanitization

Untrusted Application (UA)

Non-Secure World

Untrusted OS

Secure World

Trusted Application (TA)

Userspace

Supervisor

Trusted OS

Untrusted OS

 PTRSAN

 PTRSAN

 PTRSAN
Boomerang flaw

Untrusted Application (UA)

Non-Secure World | Secure World

Trusted Application (TA)

Userspace

Supervisor

Untrusted OS

Trusted OS

Untrusted OS

Untrusted OS

Supervisor

Trusted OS

Trusted OS

Userspace

Supervisor

Malicious

Unknown

VA

PA
Boomerang flaw

- Real world PTRSAN implementations are complex.

- Can we **bypass the validation** and make PTRSAN translate arbitrary physical address?
YES!!

- We can bypass PTRSAN *in all of the* popular TEE implementations.

<table>
<thead>
<tr>
<th>TEE Name</th>
<th>Vendor</th>
<th>Impact</th>
<th>Bug Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrustedCore</td>
<td>Huawei</td>
<td>Arbitrary write</td>
<td>CVE-2016-8762</td>
</tr>
<tr>
<td>QSEE</td>
<td>Qualcomm</td>
<td>Arbitrary write</td>
<td>CVE-2016-5349</td>
</tr>
<tr>
<td>Trustonic</td>
<td>As used by Samsung</td>
<td>Arbitrary write</td>
<td>PZ-962*</td>
</tr>
<tr>
<td>Sierra TEE</td>
<td>Sierraware</td>
<td>Arbitrary write</td>
<td>No response from vendor</td>
</tr>
<tr>
<td>OP-TEE</td>
<td>Linaro</td>
<td>Write to other application’s memory</td>
<td>Github issues [13, 14]</td>
</tr>
</tbody>
</table>

*concurrently found by Google Project Zero ([laginimaineb](#))
How to exploit Boomerang flaws?
Automatic detection of vulnerable TAs

- Goal: Find TAs which accepts pointers

- Static analysis of the TA binary:
  - Recover CFG of the TA
  - Paths from the entry point to potential sinks
  - Output the trace of Basic Block addresses
Results

<table>
<thead>
<tr>
<th>TEE Name</th>
<th>Number of TAs</th>
<th>Vulnerable TAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSEE</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>TrustedCore</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

- ✓ Arbitrary kernel memory read on Qualcomm phones.
- ✓ [Demonstrated at GeekPwn](http://example.com).
- ✓ Geekpwn Grand Prize ($$$)
Impact

- Compromising untrusted OS == Rooting your device.

- Hundreds of millions of devices on the market today.

- Fixes yet to be released.

- Your device may be vulnerable!!!
Expectation
Reality

\[
\text{Android} + \text{TrustZone}^\text{®} \quad \text{System Security by ARM} \quad = \quad \text{Android}
\]
How to prevent Boomerang attacks?
Just fix PTRSAN? NO!!

This requires to understand the semantics of current and future TAs.

- Structure of the TA request?

- Which fields within the structure are pointers?
Root Cause

- **Semantic Gap**: Inability of the TA (or TEE) to verify whether the requested UA has access to the requested memory

- Should have a mechanism for the TA (or TEE) to verify or bridge the semantic gap.
Existing Defenses

- Page Table Introspection
- Dedicated Shared Memory Region (DSMR)
Page Table Introspection

- Implemented in NVIDIA Trusted Little Kernel.

- Untrusted OS sends an id (e.g., pid) of the requested app (UA) along with every request.

- TA or TEE verify the access of all untrusted pointers by referring to the requested app page table.
Page Table Introspection

Pros:

- Easy to implement.

Cons:

- Trusted OS depends on Untrusted OS
- Increases attack surface
- Page table walking could be dangerous
Dedicated Shared Memory Region (DSMR)

- Implemented in Open Platform -Trusted Execution Environment (OP-TEE).
- Dedicated memory region for communication between trusted and untrusted OS.
- UA should request access to the shared memory.
- TA or TEE verify that all untrusted pointers are within the dedicated memory region.
Dedicated Shared Memory Region (DSMR)

Pros:
- Simple
- Independence from Untrusted OS

Cons:
- UA can interfere with other UAs via TAs (Partial Boomerang)
- Additional copying to/from shared memory
- Allocation of shared memory could become bottleneck in case of multithreaded applications.
- Some applications (integrity monitoring) are hard to implemented using DSMR.
Cooperative Semantic Reconstruction (CSR)

- Novel defense proposed by us.
- Provides a channel for Trusted OS to query Untrusted OS for validation.
Cooperative Semantic Reconstruction (CSR)

Non-secure World
- Untrusted Application (UA)
  - User Mode
  - Untrusted OS

Secure World
- Trusted OS
  - Supervisor Mode
- Trusted Application (TA)
  - User Mode

Physical Memory
- VA within application memory map
- Non-secure memory
- Unknown
- Secure memory

Normal flow
Cooperative Semantic Reconstruction (CSR)
Cooperative Semantic Reconstruction (CSR)
Cooperative Semantic Reconstruction (CSR)

Non-secure World

1. Untrusted Application (UA)
   - Append PID
   - Untrusted OS

2. Secure World
   - Trusted OS

3. Trusted Application (TA)

4. Physical Memory
   - VA within application memory map
   - Non-secure memory
   - Unknown
   - Secure memory

Verification flow
Normal flow
Cooperative Semantic Reconstruction (CSR)
Cooperative Semantic Reconstruction (CSR)

Non-secure World

1. Untrusted Application (UA)
2. Untrusted OS

Secure World

3. Trusted OS
4. Trusted Application (TA)

Physical Memory

5. Physical Memory

6. Supervisor Mode

7. Supervisor Mode

User Mode

Application memory
VA within application memory map
Non-secure memory
Secure memory
Unknown

Verification flow
Normal flow
Cooperative Semantic Reconstruction (CSR)
Implementation

- Open Platform-Trusted Execution Environment (OP-TEE)
  - Easy to use
  - Helpful community
  - Has DSMR already implemented

- HiKey Development board (Lemaker Version)
Evaluation: CSR vs DSMR

- Microbenchmark: Time to validate single memory pointer/page.

<table>
<thead>
<tr>
<th>Defense Name</th>
<th>Overhead Component</th>
<th>Overhead (μs)</th>
<th>Total Overhead (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR</td>
<td>Untrusted OS verification</td>
<td>21.909</td>
<td>26.891</td>
</tr>
<tr>
<td></td>
<td>Mapping in trusted OS</td>
<td>4.982</td>
<td></td>
</tr>
<tr>
<td>DSMR</td>
<td>Shared memory allocation</td>
<td>13.795</td>
<td>21.777</td>
</tr>
<tr>
<td></td>
<td>Shared memory release</td>
<td>7.982</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation: CSR vs DSMR

- XTEST

- Default OP-TEE Test suite.

- 63 Tests covering sanity, functionality, benchmarking and compliance.
## Evaluation: CSR vs DSMR

<table>
<thead>
<tr>
<th>Tests Category</th>
<th>Overhead (CSR - DSMR) averaged over 30 runs</th>
<th>Avg Time(%)</th>
<th>Avg Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Functionality</td>
<td></td>
<td>-0.58%</td>
<td>-7.168</td>
</tr>
<tr>
<td>Trusted-Untrusted Communication</td>
<td></td>
<td>4.45%</td>
<td>0.510</td>
</tr>
<tr>
<td>Crypto Operations</td>
<td></td>
<td>-1.72%</td>
<td>-901.548</td>
</tr>
<tr>
<td>Secure File Storage</td>
<td></td>
<td>0.03%</td>
<td>0.694</td>
</tr>
<tr>
<td><strong>Average over All Categories</strong></td>
<td></td>
<td>-0.0344%</td>
<td>-189.919 ms</td>
</tr>
</tbody>
</table>

CSR faster than DSMR

DSMR faster than CSR
Evaluation: CSR vs DSMR

- DSMR is slow in practice:
  - Synchronized access for shared memory allocation.
  - Additional copying.

- CSR can be slow for simple requests.
  - Setup of tracking structures.
Conclusion

✓ Boomerang: New class of bugs

✓ Automated attack vector detection

✓ Novel, practical, and efficient solution against boomerang: Cooperative semantic reconstruction (CSR)

✓ Detection, exploits (?), and defenses available at [github](https://github)