The Misuse of Android Unix Domain Sockets and Security Implications

Yuru Shao$^1$, Jason Ott$^2$, Yunhan Jack Jia$^1$, Zhiyun Qian$^2$, Z. Morley Mao$^1$

$^1$University of Michigan, $^2$University of California, Riverside
Motivation

- Android inherits IPC mechanisms from Linux
  - Sockets, signals, pipes, shared memory, etc.
- Android also has unique IPCs mainly for framework and apps’ use
  - Intents, Message handler, etc.

<table>
<thead>
<tr>
<th>IPC</th>
<th>Usage in AOSP</th>
<th>Documentation</th>
<th>Studied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android unique</td>
<td>Framework, apps</td>
<td>Detailed</td>
<td>Yes [1][2]</td>
</tr>
<tr>
<td>From Linux</td>
<td>Daemons, native</td>
<td>Sketchy</td>
<td>No</td>
</tr>
</tbody>
</table>

Motivation (cont’d)

- IPCs inherited from Linux have been overlooked
  - Vulnerabilities reported (e.g., CVE-2011-1823)
- Unix domain sockets the **only** Linux IPC being used by many **apps and system daemons**

<table>
<thead>
<tr>
<th>Linux IPC</th>
<th>Primary usage</th>
<th>Bidirectional</th>
<th>Java APIs</th>
<th>Native-Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signals</strong></td>
<td>Async notification</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Netlink sockets</strong></td>
<td>Kernel-userspace</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Unix sockets</strong></td>
<td>Userspace</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Pipes</strong></td>
<td>Parent-child</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>
Contributions

• We develop *SInspector* for discovering misuse of Unix domain sockets

• We perform **the first study** of Unix domain sockets on Android
  1. Categorization of usage
  2. Existing security measures being enforced
  3. Common flaws and security implications

• We conduct an in-depth analysis and suggest countermeasures
Unix domain sockets on Android
Unix domain sockets

• For IPC between processes **on the same host**
• 3 address namespaces on Android
  • FILESYSTEM: address is *a file path*, and file exists on FS

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Socket file</th>
<th>File permissions</th>
<th>SEAndroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILESYSTEM</td>
<td>✔</td>
<td>✔</td>
<td>N/A for apps</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>✗</td>
<td>✗</td>
<td>N/A for apps</td>
</tr>
</tbody>
</table>

**ABSTRACT sockets are less secure, as less access control is imposed on the socket channel**

*RESERVED is essentially a sub-namespace of FILESYSTEM. Socket files locate under /dev/socket/.*
Threat model

• A malicious app with low privilege is installed
  • Only INTERNET permission is the must-have
  • Repackaging or building a suspicious exploit app

/proc/net/unix is a publicly readable file!
ABSTRACT is the default

• Android APIs **by default** use ABSTRACT addresses

![LocalServerSocket constructor](image)

• Possible reasons to make ABSTRACT as default
  • More reliable
  • More convenient to use
Authentication is needed

• Especially for ABSTRACT sockets
• No file permissions access control on the socket channel
• But developers are doing bad on this
• Lack of authentication

It covers everything but security
Detecting vulnerable apps & system daemons
Overview

• SInspector: Vetting apps and daemons for finding out potentially vulnerable ones
Highlights

Q1 Which apps are using Unix domain sockets?

Q2 How to evaluate the security of a socket address?

Q3 Which types of authentications are strong?
Apps using Unix sockets (Q1)

• INTERNET permission is required
• Unix domain socket APIs or syscalls present in an app’s code
  • socket(AF_UNIX, ...)
  • LocalSocket
  • LocalServerSocket
• Code is reachable at runtime
  • Unix domain socket related logic is executed at runtime
Socket address analysis (Q2)

- FILESYSTEM addresses are secure only if socket file permissions are correctly set
  - Is there any operations that change file permissions?
- Some socket addresses are not just constant strings

```java
public static String addr() {
  return String.format("com.qihoo.socket\x",
    Long.valueOf(System.currentTimeMillis() & 65535));
}

protected void b(...) {
  ... String addr = addr();
  this.serverSock = new LocalServerSocket(addr);
  ...
}
```

Tracking the construction of the socket address string

A sever socket listening on ABSTRACT address addr is created
Authentication analysis (Q3)

• Authentication is **the last chance** to prevent unauthorized access

• Client and server can get their peer’s credentials
  • PID, UID, and GID

<table>
<thead>
<tr>
<th>Credentials</th>
<th>getPeerCredentials()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Retrieves the credentials of this socket's peer.</td>
</tr>
</tbody>
</table>

• Look at control dependency between r/w operations and credentials

• PID-based authentication is considered to be weak
  • Process ID allocation is nondeterministic
Implementation

• App DEX code
  • Analysis built on top of Amandroid [1]
  • Inter-component control and data flows
  • Control and data dependency

• Native code
  • Leverage IDA Pro’s disassembly engine and control flow analysis
  • Only support ARM32, intra-procedural data flow analysis for now

Results & findings
Overview

• 14,644 Google Play apps (top ones in each cate.)
  • 3,734 (25.5%) have Unix domain socket APIs/syscalls
  • The majority (3,689) use ABSTRACT addresses
  • 45 out of 67 reported are exploitable

• 60 daemons from 3 rooted phones
  • 20 use RESERVED addresses
  • 9 out of 12 reported are exploitable
Real-world usage

1. IPC, (e.g., debugging interface)
2. Implementing watchdog
3. Realizing singleton services & global locks
   • ABSTRACT socket addresses are used exclusively
   • Could be easily DoS’ed

Starts a socket server listening on ADDR

Fails to listen on ADDR

Preemptively takes ADDR

??

share

Push Service
App1

Push Service
App2
Identified libraries

- **12** libraries were identified by socket addresses
- **10/12** are using ABSTRACT namespace
  - The other 2 (Amazon, OpenVPN) use FILESYSTEM
- Only *Facebook Stetho* has authentication

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Usage</th>
<th># Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baidu, Tencent, Umeng,</td>
<td>Singleton/Globlock</td>
<td>145</td>
</tr>
<tr>
<td>Facebook SockLock, Yandex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facebook Stetho, QT5</td>
<td>Debugging interface</td>
<td>107</td>
</tr>
<tr>
<td>Sony, Samsung, Amazon, CM</td>
<td>Data transmission</td>
<td>40</td>
</tr>
<tr>
<td>OpenVPN</td>
<td>Cmd &amp; control</td>
<td>7</td>
</tr>
</tbody>
</table>
Weak authentication

• Qualcomm time daemon checks client process name
  • By reading proc file `/proc/PID/comm`
  • Access allowed if client proc name contains “`comm.timeservice`”

• Apps are able to change their process names freely
  • `Process.setArgV0(String s)`
Strong authentications

• UID/GID checks
• Username checks
• Permission checks
  • Facebook Stetho only accepts connections from adb shell
  • See if the peer has \texttt{android.permission.DUMP}
• Token-based checks
  • Client and server shared a token through a secure channel, e.g., \textit{a permission-protected broadcast}
  • Deny access if the peer doesn’t own the right token
Common mistakes

• Intended to use FILESYSTEM but actually used ABSTRACT
  • **ES File Explorer** listens on an ABSTRACT address: 
    `/data/data/com.estrongs.android.pop/file/comm/su_port`

• Socket file permissions wrongly configured
  • **Hideman VPN** changed socket file permissions to 777
    `/data/data/net.hideman/cache/OpenVpnManagementInterfaceSocket`
  • **LG AT daemon** allows accesses from apps in inet group

• Lack of strong peer authentication
What attackers can do

• Privilege escalation
  • LG AT daemon, etc.
  • Turn on/off SIM card, factory reset phone

• Data injection
  • KingRoot, etc.
  • Grant any app root access

• Data theft
  • Samsung health library, etc.

• DoS
  • Baidu push, etc
Case study: KingRoot

1. Socket file permissions wrongly configured
2. No authentication when accepting root request decision
Mitigations

• More fine-grained SEAndroid domain assignment
  • Currently all 3rd-party apps belong to untrusted_app
  • They can access each other’s socket channels

• System daemons
  • Implement a proxy service for talking to the daemon and performing authentication
Mitigations (cont’d)

• Apps exposing functionalities to other apps
  • The server doesn’t know who are potential client apps
    • UID/GID-based authentication is not applicable
  • A secure way to exchange token is desired
    • Secure Android IPC such as broadcast
Summary

• We performed the first study on Unix domain sockets on Android
  • Presented a tool for detecting potentially vulnerable apps and daemons
  • Reported high-severity vulnerabilities
  • Conducted in-depth analysis and discussed mitigations
Demos

1. Exploit *KingRoot* to gain root access

2. Exploiting *ES File Explorer* to modify system file

3. Exploiting *LG AT daemon* to factory reset a phone
Thank you

• Q/A

Scan to watch demos
Dynamic analysis for daemons (Q2)

- We cannot pull binaries out without root
- Once we have root access we can collect a lot more runtime information about sockets
- We don’t need to worry about coverage

Socket channel is securely protected if connection failed due to insufficient permission
Limitations

• Human efforts required for validating vulnerabilities
• We may have false negatives
  • Encrypted/packed daemons cannot be analyzed
  • We are unable to handle dynamically loaded code
• Lack of ground truth to evaluate false negatives
Case study: LG AT daemon

- LG AT daemon listens on /dev/socket/atd

```
$ ls -l /dev/socket/atd
srw-rw---- system inet
```

- **All** apps having INTERNET permission belong to Linux user group **inet**
- No SEAndroid policy enforced, no authentication
- CVE-2016-3360