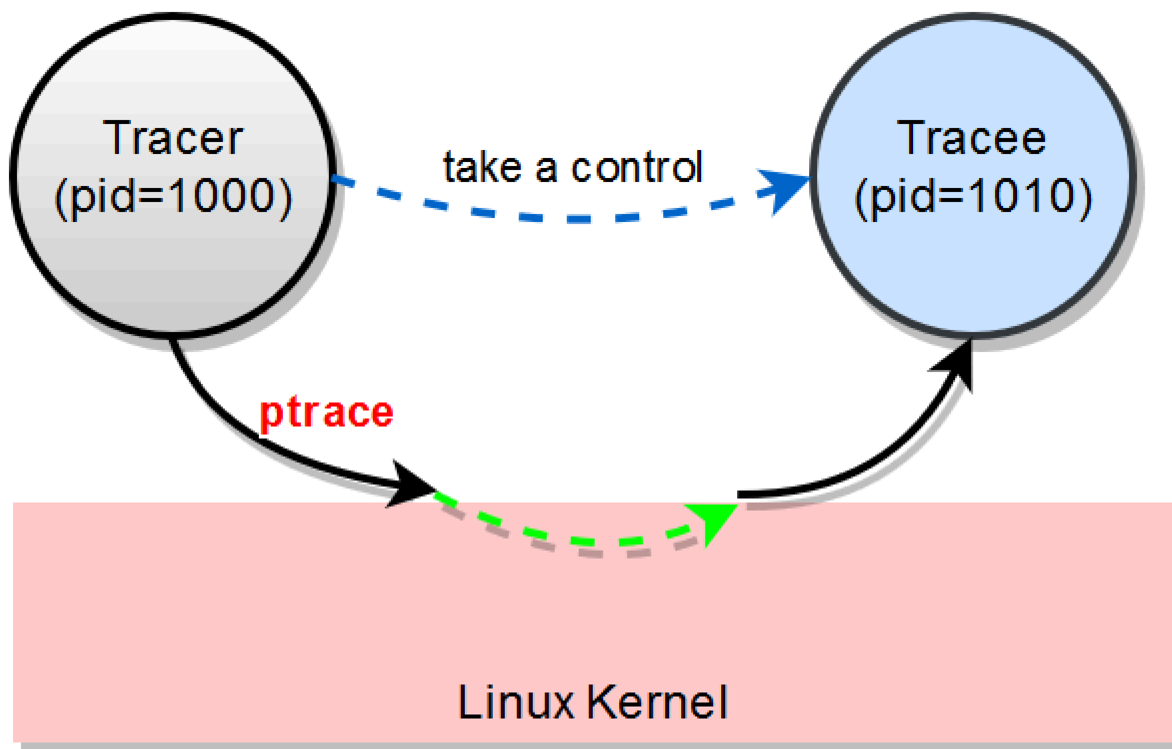
Shared Library Injection on Android 8.0

By Alexandr Fadeev

Published: 2018-08-25 · Archived: 2026-04-05 19:50:04 UTC

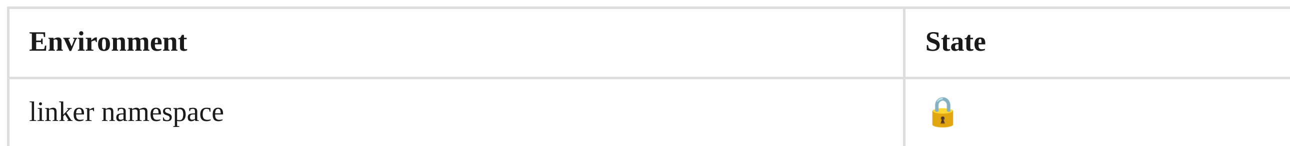
Full solution: https://github.com/fadeevab/TinyInjector

One of the ways to carry out the shared library injection is to use `ptrace` system call (syscall). One process (a **tracer**) attaches to a **tracee** and calls `dlopen` inside tracee's virtual memory space.



Superuser privileges (root) are required to attach to any process in the system and to overcome other security controls. It is possible to use `ptrace` without the root, but both processes must have the same user ID. In the current shared library injection it is assumed that SELinux is enforced as well as dm-verity and a few other security controls.

Environment	State
root privileges	🔓
SELinux	🔒
dm-verity	🔒
ASLR	🔒



Environment	State
linker namespace	🔒

SELinux - *LSM module aimed to provide Mandatory Access Control (MAC).*
dm-verity - *Linux kernel module to protect read-only partitions (/system, /vendor).*
ASLR - *address space layout randomization.*
Linker namespace - *restrictions applied to dynamic linking.*

When a tracer attaches to a target process (tracee), it's possible to read/write data from/to the process' memory, read/write registers, to call `mmap`, `mprotect`, to manipulate with memory, to call any system API.

The `ptrace` syscall is not the only way to trap into a remote process, but the most obvious and powerful one.

I wish to express my thanks to the author *shunix* for the series of articles about Shared Library Injection in Android.

1. Shared Library Injection in Android (ASLR bypass is there as well).
2. Bypass SELinux on Android.
3. ARM and Thumb Instruction Set.

I use his TinyInjector as a basis, and I extend one to the Android 8.0 realms, particularly to the ARM64 architecture (original version supports 32-bit ARM), SELinux, and the linker namespace.

More About Ptrace

The help topic of "SELinux Policy Manager" has a pretty nice overview of ptrace (lockdown_ptrace.txt):

> Most people do not realize that any program they run can examine the memory of any other process run by them. Meaning the computer game you are running on your desktop can watch everything going on in Firefox or a programs ... that attempts to hide passwords.
> ptrace allows developers and administrators to debug how a process is running using tools like **strace**, **ptrace** and **gdb**. You can even use gdb (GNU Debugger) to manipulate another process running memory and environment.
> The problem is this is allowed by default.
> ... It would be more secure if we could disable it by default.
> **Note**: Disabling ptrace can break some bug trappers that attempt to collect crash data.

If you ever thought about any system security hardening, you would have to think about disabling ptrace (together with the root user). However, you cannot disable them altogether, the problem lays in **CIA** principle (Confidentiality, Integrity, Availability): improving the integrity/confidentiality usually leads to degradation of availability. Disabling the ptrace makes the system to be almost undebuggable and unmaintainable.

Preconditions

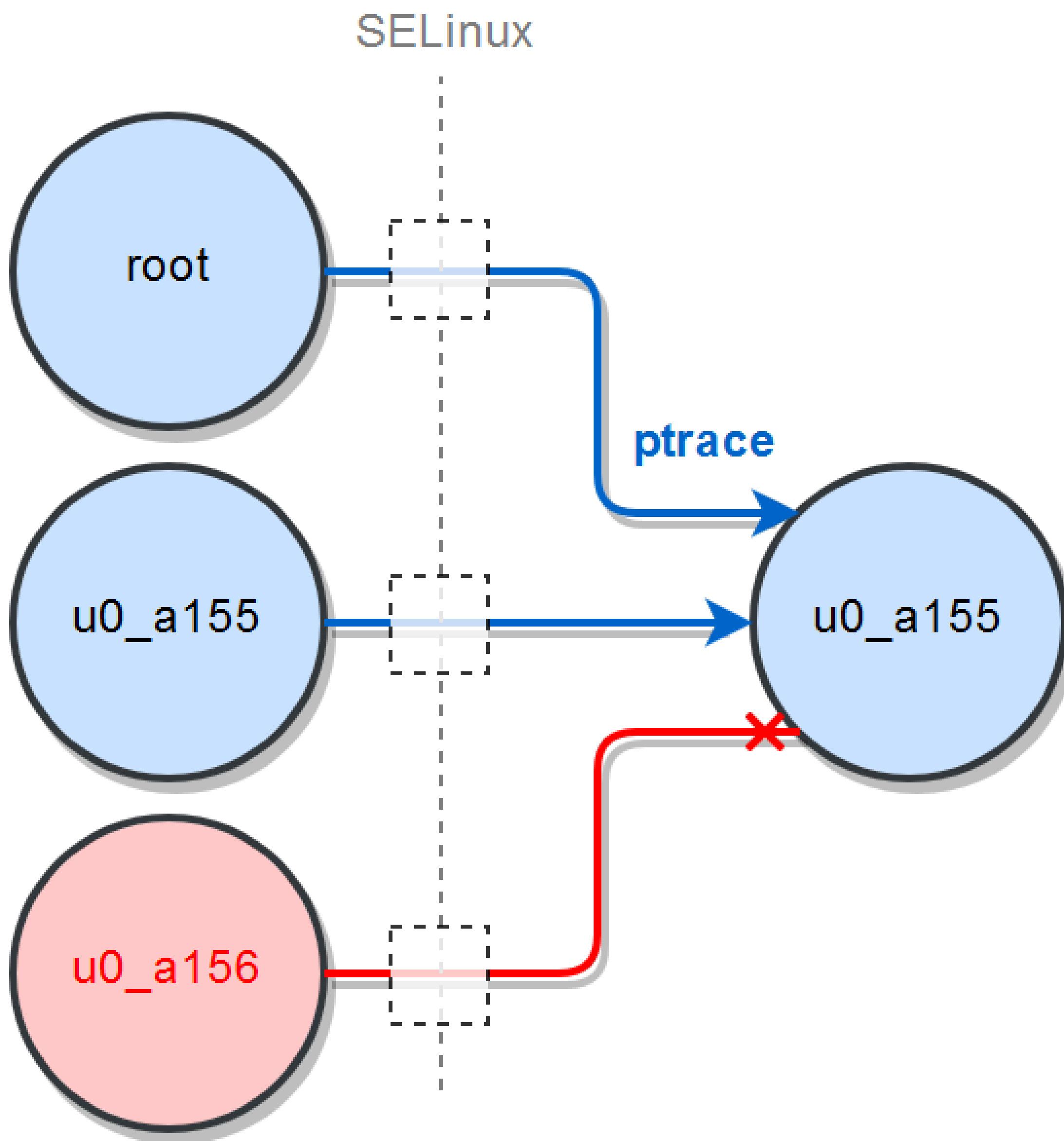
In order to attach to any process via `ptrace` system call, it is necessary:

1. to have root privileges
2. to be in a proper SELinux context allowed to use ptrace

Root is needed to:

1. attach via `ptrace` to *any* process
2. list all processes
3. read /proc/<pid>/maps
4. change SELinux labels

If you don't have superuser privileges, the injector (tracer) needs to have a tracee's UID to be able to attach to a tracee process.



If */proc* filesystem is mounted with `hidepid` option, then an ordinary non-root process is not even able to see other processes in the system.

```
 $ grep proc /proc/mounts
proc /proc proc rw,relatime,gid=3009,hidepid=2 0 0

$ ps
PID   USER    COMMAND
17677 u0_a155/data/data/bash/bash -i
21955 u0_a155ps
```

Disabling SELinux: setenforce 0

I haven't had to disable SELinux on my Android 8.0 device. Only root. I tested TinyInjector, launching one from the /data partition, and it works just fine in Android 8.0 *having only root*:

```
 # On the device under root
/data/local/tmp/injector com.android.example.app /data/local/tmp/libinject.so
```

But, on your device, you might be not able to attach the injector to another process with SELinux enforced. The current version of TinyInjector contains the code to disable SELinux through **selinuxfs**. The trick to disable SELinux can be shortly described as the following shell script:

```
 # get actual selinuxfs directory here, e.g. /sys/fs/selinux
grep selinuxfs /proc/mount
# disable SELinux; under the root
echo 0 > /sys/fs/selinux/enforce
```

Disabling SELinux: playing with labels

I find the way above to be pretty rough to disable SELinux. Depending on a vendor of your device the *selinuxfs* may be not even mounted (*/sys/fs/selinux* may not be present in such case). Disabling SELinux may be not allowed on the device. In such case there are a few other ways to deal with SELinux to allow *ptrace*:

1. Search for a context which is allowed to use ptrace,
2. either set a desired label to injector using `semanager` or `chcon` tool,

   ```
   chcon -v u:object_r:apk_data_file:s0 injector
   ```

3. or place injector near the tracee and trap into the context using `restorecon`,
4. or find any other way to trap into desired SELinux context, e.g. through any standard Android behavior which leads to obtaining the desired context or the tracee's user ID.

Overcome dm-verity

dm-verity protects read-only partitions such as */system* and */vendor*. Pushing the binaries into */data* partition allows to evade altering the protected partitions.

```
 adb push injector /data/local/tmp/
adb push libinject.so /data/local/tmp/
adb shell /data/local/tmp/injector com.android.example.app /data/local/tmp/libinject.so
```

How Injection Is Performed

1. An injector attaches to a target process via `ptrace`.
2. The target process is being suspended.
3. The injector gets a content of registers saving the running context of the target.
4. The injector finds symbols `mmap`, `dlopen` and `dlsym` in the virtual memory of the *target process*.
5. `mmap` is used to allocate memory for the string parameters and for the stack.
6. `dlopen` loads a shared library.
7. `dlsym` retrieves some function from the newly loaded shared library.
8. The function obtained via `dlsym` could be called.
9. The injector restores registers of the target process restoring a running context.
10. The target process continues as usually while the new shared library is injected and some custom function is called from that library.

Porting TinyInjector to AARCH64 (ARM64)

ptrace() ARM64 Implementation

Initially, TinyInjector supports ARM32 only. One of the differences lays around the low-level implementation of the ptrace() system call.

ptrace() prototype is fairly generalized, but functionality is strongly tied to hardware: ptrace allows to manipulate registers, and concrete list of registers depends on assembler, and assembler comes from architecture of CPU. "Back-end" of `ptrace` is provided by Linux kernel implementation.

```
 #include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

For some reasons ARM64 `ptrace` usage pretty differs from ARM32.

```
 #if defined(__aarch64__)
  #define pt_regs user_pt_regs
#endif

static void PtraceGetRegs(pid_t pid, struct pt_regs *regs) {
```

```
  #if defined(__aarch64__)
    struct {
      void* ufb;
      size_t len;
    } regsvec = { regs, sizeof(struct pt_regs) };
    ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS, &regsvec);
  #else
    ptrace(PTRACE_GETREGS, pid, NULL, regs);
  #endif
}
```

ARM32 (AARCH32)	ARM64 (AARCH64)
PTRACE_GETREGS	PTRACE_GETREGSET
pid	pid
NULL	NT_PRSTATUS
struct pt_regs *	{ struct user_pt_regs *ubf, size_t len }

(Obviously, additional control is provided for the 4th parameter `void *data` passing the structure `struct pt_regs` together with length instead of the raw pointer).

Another inconvenience has come from the `struct pt_regs` definition: the similar `struct user_pt_regs` is introduced instead, and the register shortcuts are absent there. I added some macro to make code a bit more portable (I haven't found a better way):

```
 #if defined(__aarch64__)
  #define pt_regs  user_pt_regs
  #define uregs    regs
  #define ARM_r0   regs[0]
  #define ARM_lr   regs[30]
  #define ARM_sp   sp
  #define ARM_pc   pc
  #define ARM_cpsr pstate
#endif
```

X0 is a 64-bit analogue of 32-bit **R0**.
X30 is a return address.
PSTATE is a processor state register.

ARM64/ARM32 ABI Differences

ABI formula:
`ABI` = `PCS` + `ELF` + `DWARF` + ...

In our case, PCS rules contained differences.

PCS stands for a Procedure Call Standard, and it is about CPU rules of how to make procedure calls: how to pass arguments, how to return the value, what registers are expected to get modified during a call, how the stack should be used, and so on.

In the **ARM32 PCS** the first 4 parameters are passed to a procedure call through **R0-R3** registers, and the rest of the arguments are passed through the stack.

```
 for(int i = 0; i < argc && i < 4; ++i) {
  regs.uregs[i] = args[i];
}
```

In the **ARM64 PCS** the first 8 parameters are passed through **X0-X7** registers to a procedure call. Taking `mmap` as the syscall which receives the largest number of arguments (6 arguments) among the syscalls in the TinyInjector, all arguments will be passed via the registers. (*In the loop below I pass only 6 arguments instead of 8 mistakenly, but it's enough to make `mmap` work*).

```
 #if defined(__aarch64__)
  #define uregs regs
#endif

for(int i = 0; i < argc && i < 6; ++i) {
  regs.uregs[i] = args[i];
}
```

The Linker Namespace Bypass

As I mentioned in my review of the Android linker namespace, the caller of `dlopen` can be forged.

By pointing the **X30** register (**LR**) to a code section of any shared library which namespace has wide privileges, we make the dynamic linker of target process to apply desired namespace rules to `dlopen` call.

Currently, I would like to load shared library from */data/local/tmp* (*/sdcard* can be acceptable option as well). `dlopen` of the most of the Android applications is limited about to load shared libraries from such untrusted place. But the linker namespace of */system/lib/libRS.so* allows to load shared libraries from any place of */data*.

Steps:

1. Load the start of the *libRS.so*'s code segment in the target process.

```
 #if defined(__aarch64__)
#define VNDK_LIB_PATH  "/system/lib64/libRS.so"
#else
#define VNDK_LIB_PATH  "/system/lib/libRS.so"
```

```
#endif

long vndk_return_addr = GetModuleBaseAddr(pid, VNDK_LIB_PATH);
```

2. Point the return address to *libRS.so*:

   ```
   long ret = CallRemoteFunctionFromNamespace(pid, function_addr, vndk_return_addr, params, 2);
   ```

   Inside the function:

   ```
   regs.ARM_lr = return_addr;
   ```

3. Perform the remote call:

   ```
   PtraceSetRegs(pid, &regs);
   ```

When `dlopen` call finishes, I watch the same effect as a pointing **LR** to NULL: get the control back after the fault and restoring the running context by setting stored registers.

SELinux Label for Injection Library

The final step is to overcome SELinux that denies to mmap a shared library from */data*. Use the same trick with a label as before:

```
chcon -v u:object_r:apk_data_file:s0 /data/local/tmp/libinject.so
```

Source: https://fadeevab.com/shared-library-injection-on-android-8/