Rigged Race Against Firejail for Local Root

This article describes a method that proofed useful in highly reliable exploitation of a file system race condition found in Firejail. Instead of optimizing the exploit to win the real race, the timing of Firejail stderr and stdout output was analyzed. With the correct parameters known the Firejail process can be frozen exactly in the right moment when attempting to write a message to a filled pipe (blocking write). Thus the exploit has any time in the world to modify the file system before restarting Firejail by emptying the pipe again.

TOCTOU and The Hard Life Winning Fair Races

When (privileged) programs process data under control of less privileged users special precautions have to be taken. Even being dangerous, the demand for such operations may be higher than one might expect at first glance. For example Firejail (a SUID program) uses configuration and persistency information provided by the user invoking it. But also the privileged SSH daemon process has to access user-owned private key material or an HTTP-daemon may need to serve files from user directories. Programs ignoring their due diligence accessing such files may quite easily end up processing arbitrary files, pipes, special devices,... as a lack of checks allows attackers to redirect requests from safe locations to arbitrary targets, see CWE-61: UNIX Symbolic Link (Symlink) Following.

Luckily such blatant vulnerabilities in privileged code are quite rare nowadays. Checking the file ownership, file type and other properties BEFORE use of the data is common. Also Firejail performed those checks related to activating per-user persistent overlay storage for use inside the sandbox. But instead of using the secure grab-and-check approach (get a handle/file descriptor for the object of interest, check the handle and use the handle when checks succeeded), Firejail performed a check on the path using *stat()* calls before using the path in file system operations later on (see trace snipplet below), thus creating CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition vulnerabilities.

To exploit such a vulnerability the attacker has to modify the file system structure at some timepoint between the check and the use of the resource. To do that reliably on a wide range of systems may be harder than it sounds due to perfect timing depending on various factors, e.g.

- The system load (concurrent processes)
- The number of CPUs
- RAM (caching of file system related data)
- The underlying file system type (huge differences e.g. between btrfs and ext4)
- Task scheduling of competing tasks (seems to have changed a lot due to Spectre/Meltdown patches - unproven)
- Security modules, auditing in place

To be still able to win the race quite reliably the attacker may apply various techniques. But finding the right mix is often time consuming, based on trial and error and the selected approach may not work the same way on other systems than the one tested. Nonetheless useful methods are:

- Use of inotify to monitor file system access of other (privileged) processes.
- Overloading the task scheduler by forking more processes than CPU cores available (not nice and stealthy).
- Using huge mmaped() files to purge cached file system structures from RAM and thus make the victim program slower.
- Fill directories with huge number of unrelated files to make the victim program loop through numerous directory entries before continuing.

Usually a mix of methods from above is sufficient to win the race occasionally. Thus if the race can be triggered frequently the privilege escalation will happen sooner or later. When the vulnerable code cannot be triggered by the attacker, when losing the race is not an option (e.g. events so rare, a miss would be detected by the victim) or when the attacker just wants to win each time for fun, then such complex or non-deterministic races have to be avoided.

The Joy of Being Unfair

To avoid the non-deterministic race the Firejail local root exploit (UnjailMyHeart.c) used Firejail logging output to block Firejail execution in the right moment completely. Therefore the exploit creates a pipe to capture Firejail stderr and a master/slave pty-pair for stdout. Even being more complex the pty pair was needed to cause stdout being line buffered. Without a terminal attached stdout would enter byte buffered mode, which is way more efficient but would cause a delay between e.g. a printf() and the flushing of the buffer.

Before executing the victim program (Firejail) the pipe (stderr) and the pty (stdout) are filled with test data till writing blocks. The amount of data sucessfully written is recorded to know the size of the pipe buffers. Next the buffers are emptied and filled anew to such extent, that Firejail may only write some of its messages before being blocked. Blocking will happen exactly after the check of a file system resource and before using it. Thus we just need to *win* the race against a sleeping horse (a frozen process). The trace of such a rigged race looks like that:

```
446   [UnjailMyHeart - the attacker process]
465   [Firejail - the victim process]

# The victim checks if the resource is sane to use.
465   stat("/home/test/.firejail/test/owork", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
# The victim is happy with the result and tells the caller,
# but is blocked doing so.
465   write(1, "Mounting OverlayFS\n", 19 <unfinished ...>
# The attacker wakes up quite some time later and renames the
# directory.
446   <... clock_nanosleep resumed>0x7fff70a09048) = 0
# Did I mention I like the RENAME_EXCHANGE atomic operation?
446   renameat2(7, ".firejail", 7, ".firejail-b", RENAME_EXCHANGE) = 0
...
# The attacker reads from the other end of the pipe to unblock
# the victim process.
446   read(5,  <unfinished ...>
# The victim write operation from above is resumed, the problematic
# file system operation takes place using untrusted data.
465   <... write resumed>)              = 19
465   mount("overlay", "/run/firejail/mnt/oroot", "overlay", MS_MGC_VAL,
      "lowerdir=/,upperdir=/home/test/.firejail/test/odiff,
      workdir=/home/test/.firejail/test/owork") = 0
```

Attack Method Recipy

To apply the method mentioned above to other targets, following steps should be performed:

Increase verbosity: Make the victim program as verbose as possible. This includes both activating stdout/stderr messages but may also include use of named pipes instead of log files.

Check system call synteny: Use system call tracing, e.g. strace to monitor the sequence of file system requests and stdout, stderr or logfile writes. For SUID-binaries like Firejail tracing has to be performed as privileged user, e.g. *strace -o result.dump -f -- /bin/su -c '[victim program]' [test user]*.

Calibrate buffering: Measure how much data is written by the victim program till the point during program execution, where you want to freeze the victim program.

Prefill the buffers: Fill the pipe buffers to such extent so that the remaining space is just a little too small to allow the victim program to write the last message before the critical file system operation. Filling the pipes is deterministic but there is some magic (not very intuitive behaviour) involved that cause pipes or especially terminals to react differently when they are filled for the first time or if they are only partially emptied and refilled or completely emptied. Therefore they should always be emptied and filled completely multiple times before use.

Some victim programs may not perform write activity between the two file system operations to race with. Still the method can be useful to synchronize your exploit with the victim program. Instead of timing the exploit actions in relation to e.g. the starting time of the victim program, a much smaller time delay (often only 1%) between synchronization signal and attack can be used, thus increasing precision and chances tremendously.

Notes:

- 20210218: Exploits added, both improved Buster/Bullseye compatible version (sha1 68013d8f8f916c13d2dbae9d107090b02fe77fd1) and the previously announced sha1 f756b79f3c0c682e69a99f472e7b67892a7c7b5d version only valid for Bullseye (/proc mounting is different on Buster, so more generic approach was required).

Comments are welcome, but there is no forum system im place yet. If there is something important to be added to this page, please send it as e-mail. Legal: Appropriate comments will be published, there is no right for you to get them published, use a "Nick:" entry in your comment otherwise for attribution "Anonymous" is used, comment mails are deleted after processing (GDPR), IPR rights for your comment stay with you except that the content may be used to correct or improve the page while referencing to your comment as source of the change, comment data is not submitted to third parties. Phuuu, inhale!