nccgroup
freedom from doubt

An NCC Group Publication

A few notes on usefully exploiting libstagefright on Android 5.x

Prepared by:
Aaron Adams

Contents

1 Introduction

At NCC Group, a colleague and I recently spent some time trying to develop a more robust exploit for the Android libstagefright bug CVE-2015-3684. This is a bug that persisted through the patches Joshua Drake (jduck) originally provided to Google, so a few more firmware versions are vulnerable. In this white paper, I will discuss a few tricks we came up with to make the exploit a bit more robust with regards to address space spraying, dealing with SELinux sandbox restrictions, automating device identification, and staging a kernel exploit.

Unfortunately I didn't have any breakthroughs on the ASLR bypass front, and similarly I couldn't come up with a reliable exploit when using dlmalloc feng shui on Android 4.x devices, because the combined brute force complexity resulting from layout instability (thanks to so many noisy `mediaserver` threads) and no ASLR bypass makes the timing required for exploitation unrealistic. It's possible I missed some useful approach though, so feedback is more than welcome.

Despite the noted failures, I think some of the improvements I made could be interesting to some and so are worth documenting. As is often the case, I highly recommend reading a few other blog posts before reading this post, as they provide good background information and provide details I don't bother replicating here. Jduck's original presentation and exploit is the best starting point; then Exodus Intelligence's bug writeup and reports about CVE-2015-3684; the Google Project Zero blog on exploiting the bug on Android 5.x with the jemalloc heap; and the Keen Team write up on CVE-2015-3636 exploitation. I glaze over many technical details under the assumption you have first read and understood these write ups.

Also please note that I will refer to MP4 headers as either an 'atom', typically when referring to the actual type indicator of the header, or a 'box header', typically when referring to the header as a whole (both the type and length fields). This is consistent with the terminology used by the actual MP4 standard. Some sources seem to use the term chunk, which is especially confusing when you're also talking about heaps which have their own meaning for the word chunk.

1.1 Credits

I talk about a few different exploits and bugs in this paper, and want to give credit to those that did the work on these before me. Kudos to Joshua Drake from Zimperium for originally finding the libstagefright bugs and releasing an exploit targeting 4.0.4; to Jordan Gruskovnjak from Exodus Intel for finding and explaining the hole in the libstagefright patches that led to CVE-2015-3864; to Mark Brand from Google Project Zero for releasing details on exploiting CVE-2015-3864 on Android 5.x; to Wen Xu and Yubin Fu of Keen Team for finding and exploiting CVE-2015-3636; and to fi01 for posting exploit source for CVE-2015-3636 on GitHub. If I missed anyone, please let me know and I'll update the paper.

1.2 tl;dr

I built a slightly more reliable exploit for Android 5.x, which works across multiple devices and can stage a kernel exploit that breaks out of the SELinux sandbox and turn SELinux off. ASLR bruteforce is still required however.

2 A few notes on usefully exploiting libstagefright on Android 5.x

2.1 MMS as a realistic exploit vector?

You will notice that we focus on the HTTP vector rather than MMS and maybe you would wonder why? There are at least few problems related to the MMS vector:

1) Sending a large file is not a very effective spray mechanism across devices and the optimal spray approach I will show later in the paper only works over an HTTP transport.
2) MMS gateways seem to actually vary on how large of MMS files they accept. This wasn't thoroughly investigated on our part, but in practice we were unable to even send 2MB media files between devices.
3) Assuming you had a solution for problems 1 and 2, you would still need to bruteforce ASLR which doesn't seem feasible over MMS (pending some other trick someone hasn't published). This means you'd be limited to non-ASLR devices (as shown by jduck), which are Android 4.0.4 devices and earlier. There are still plenty of those devices floating around, but we found even exploit reliability on 4.0.4 and earlier isn't ideal due to heap layout issues.
4) Finally, for generic MMS-based exploitation on ASLR-enabled devices to really be effective you'd also have to have a way to either ignore device versions or automatically determine them as part of the ASLR bypass. For instance, a blind ASLR bypass that lets you re-use existing code that you could predict at static offsets from some partially corruptible pointer still means you need to know the exact model and build you're targeting, which you won't easily know through the MMS vector, without having done some pre-exploitation reconnaissance.

Maybe someone out there has figured out a way to practically overcome all of these problems and if so I do hope to see a post about the solution someday!

2.2 Leveraging recursion for more efficient heap spray

The bugs in libstagefright can be triggered locally or remotely, with the latter being possible over HTTP. I started working with the exploit released by Google Project Zero, which only works over an HTTP transport, although this isn't explicitly mentioned in their post. The reason is that it relies on the allocation and corruption of a new `MPEG4DataSource` object, and in order for this to be allocated the data source being used to read the MP4 must be cache-enabled, which is only the case for the HTTP transport. It's probably worth noting that you can target a different object (like a `SampleTable` object) for corruption, which allows for exploitation over non-HTTP transport, but then you lose the ability to heap spray effectively in the way I am about to describe.

In the Project Zero blog the following observation was made:

"*After a bit of experimentation, it seemed that the best way to achieve this in practice is by wrapping a number of our 'pssh' chunks inside a valid sample table ('stbl'). This triggers the creation of a caching MPEG4DataSource, which will then allocate and save all the data for the contained chunks; and will then be used to parse out the chunks. This essentially doubles the size of our spray, reducing the size of file needed.*"

This was a pretty interesting finding, and to me seemed worth thinking about more. First, let's revisit the main idea. Assuming we send an approximately 2MB MP4 file, where most of the data is contained with a sample table (designated by the `stbl` atom), that means we get approximately 4MB of data copied into memory. This isn't too bad, but also really not great in practice. I found in testing across a large number of different builds, on a few different devices, that spraying this amount of memory wasn't very reliable as far as being able to use a constant address for every target. The address that worked on some devices would often not work on others. So the question I inevitably asked was whether or not this observed functionality could be leveraged to spray an

almost arbitrary amount of memory with a minimal amount of in-file overhead, and the answer turns out to be yes.

First, to understand why this is an HTTP-only trick, it is important to understand that the creation of a caching `MPEG4DataSource` is actually only possible if the primary data source itself supports caching, as specified by the `kIsCachingDataSource` flag. Let's take a look at the related libstagefright source, since it wasn't actually shown in the Project Zero blog post. The part we're interested in right now is in the `parseChunk()` method (which holds the main logic for handling most atom types).

```
    switch(chunk_type) {
        […]
        case FOURCC('s', 't', 'b', 'l'):
        […]
        {
            if (chunk_type == FOURCC('s', 't', 'b', 'l')) {
                […]
                if (mDataSource->flags()
                        & (DataSource::kWantsPrefetching
                            | DataSource::kIsCachingDataSource)) {
                    sp<MPEG4DataSource> cachedSource =
                      new MPEG4DataSource(mDataSource);
                    if (cachedSource->setCachedRange(*offset, chunk_size) == OK) {
                        mDataSource = cachedSource;
                    }
                }
                mLastTrack->sampleTable = new SampleTable(mDataSource);
            }
         }
```

We see that a `cachedSource` is only created if the `kWantsPrefetching` or `kIsCachingDataSource` flags are set. You can see in `NuCachedSource2.cpp` that the flag is set when the `flags()` method is called:

```
uint32_t NuCachedSource2::flags() {
    // Remove HTTP related flags since NuCachedSource2 is not HTTP-based.
    uint32_t flags = mSource->flags() & ~(kWantsPrefetching | kIsHTTPBasedSource);
    return (flags | kIsCachingDataSource);
}
```

And finally you can see in `DataSource.cpp` that the `CreateFromURI()` method when building an HTTP-based data source specifically creates a backing `NuCachedSource2` object:

```
      source = new NuCachedSource2(httpSource,
                    cacheConfig.isEmpty() ? NULL : cacheConfig.string());
```

Next it is worth reviewing the `setCachedRange()` function, which is the code responsible for actually allocating the cached memory and copying the current atom's data into it:

```
status_t MPEG4DataSource::setCachedRange(off64_t offset, size_t size) {
    Mutex::Autolock autoLock(mLock);
    clearCache();
    mCache = (uint8_t *)malloc(size);
    if (mCache == NULL) {
```



```
        return -ENOMEM;
    }
    mCachedOffset = offset;
    mCachedSize = size;
    ssize_t err = mSource->readAt(mCachedOffset, mCache, mCachedSize);

    if (err < (ssize_t)size) {
        clearCache();
        return ERROR_IO;
    }
    return OK;
}
```

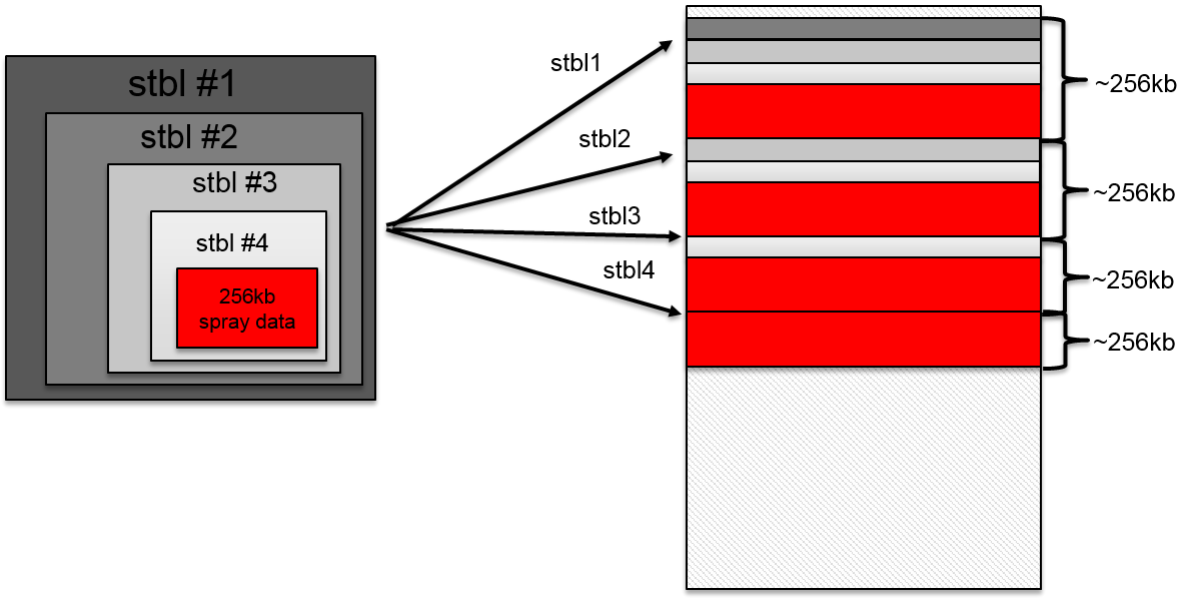
We see that first the size of the `stbl` box header is allocated, and then the data is read from the current `mSource` (which could also be cached) and copied into the new buffer.
Once the cached data source is allocated and the data is copied into it, we hit the following code that will recursively parse over the cached range that was just set.

```
off64_t stop_offset = *offset + chunk_size;
*offset = data_offset;
while (*offset < stop_offset) {
    status_t err = parseChunk(offset, depth + 1);
    if (err != OK) {
        return err;
    }
}
```

The take-away from all this is that an `stbl` box embedded inside the cached range setup by a higher level `stbl` box can re-reference the same set data the outer table referenced, aside from the outer `stbl` header, and cause a new cached source allocation of that same data. We can use this to our advantage to greatly improve heap spray efficiency without needing to increase the file size.
The main reason we can abuse this is because the `parseChunk()` method is recursive and there is no logic to prevent sample tables from containing their own embedded sample tables, nor is there any maximum recursion depth enforced (this alone also makes it easy to crash `mediaserver` through memory exhaustion). This ability to embed recursively means that each encountered sample table results in the allocation of a new cached data source, which causes another recursive call to `parseChunk()`.

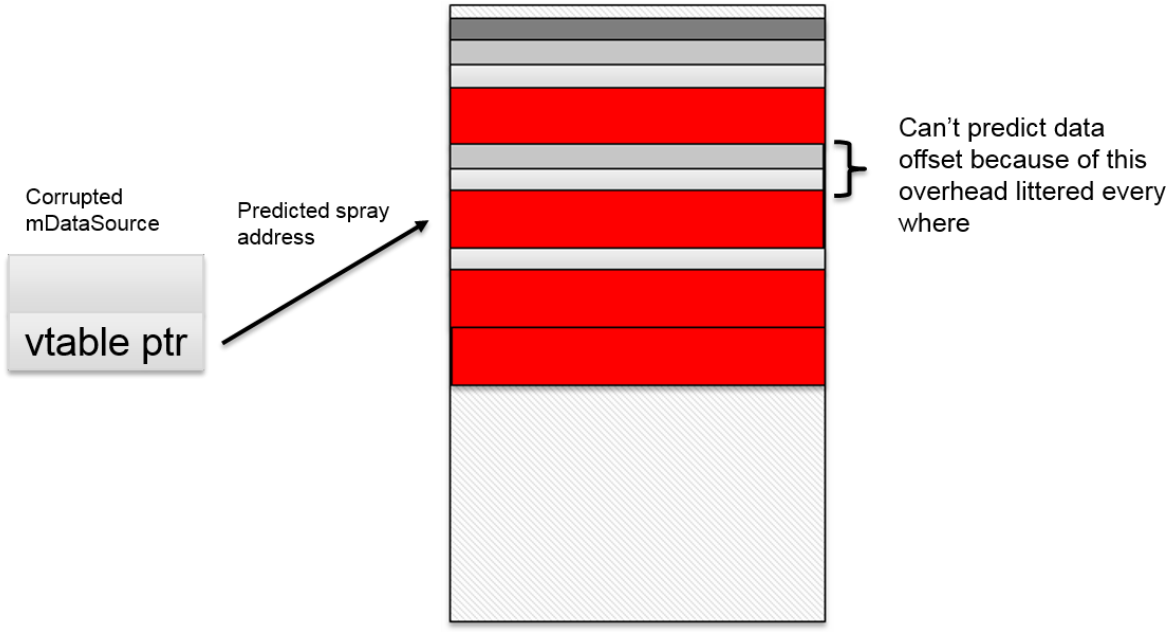
For the sake of example let's assume that we have a 256kb block of data that we want to spray into memory, such that we consume a total of approximately 256mb of memory. If we wrap the 256kb block in a sample table, we get a representative total file size of approximately `256kb + sizeof(box header)`, which (as described by Project Zero) doubles our spray size. In these examples `sizeof(box header) == 8`, as per the spec. So, if we wrap the first sample table in another sample table, then we get three times 256kb sprayed into memory, with an overhead of `256kb + (sizeof(box header)*2)`. To reiterate why there are three copies: the first is from the initial mapping of the MPEG4 into memory, the second is from the cached data source created by the first encountered `stbl`, and the third is from the cached data source created by the second encountered `stbl`. Conveniently, this behaviour can be scaled to whatever total size we want to spray into memory. So, if we represent the number of total sample table box headers wrapping the 256kb blob of data by n, then we can see that we can effectively spray `256kb * (n+1)` bytes of memory (the +1 is to represent the original 256kb when the file is originally read into memory) where the actual file overhead is only `256kb + (sizeof(box header) * n)`. So if we've included a 256kb block of data we want to use to spray 256mb of memory with, we would just need to wrap the data in sixteen sample tables.

The basic idea of what we're doing is illustrated in the following diagram:



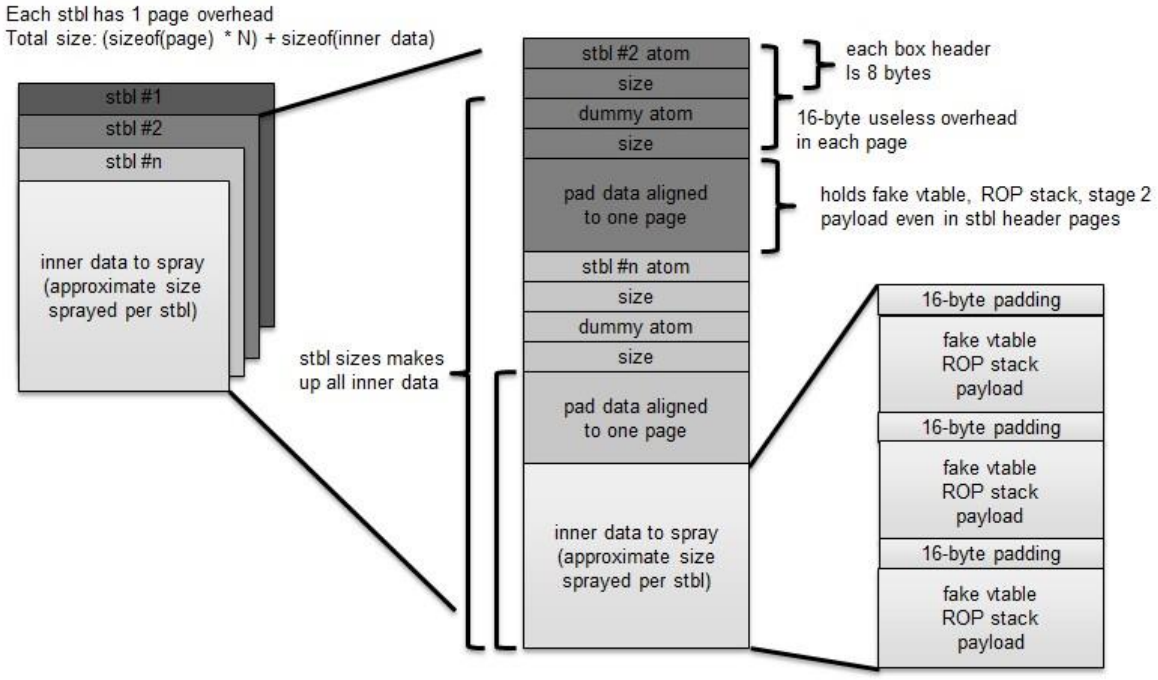
In the illustration above, each `stbl` being handled (from the outside in) results in the remaining inner set of embedded `stbl` headers and the spray data being mapped into memory; effectively stripping off one additional box header each time.

Now with that primitive understood, there are a few additional quirks we have to work around in order to actually use it for exploitation. The primary hurdle is with the alignment of sprayed data within each page. The problem is that, as already illustrated, each time a new sample table atom is encountered, it will be stripped off the data already copied into memory, leaving only the remaining data in the newly allocated cached data source. If you don't know exactly what part of the sprayed memory you'll be landing within, when using your guessed spray address, you risk not encountering the correct data, because the relative offsets per page keep changing. From reading the background material on exploiting this bug, you'll remember that the guessed spray address needs to point reliably to a fake vtable. If you can't predict which page of your sprayed data will have what data offset, then you can't reliably point to the vtable. This is illustrated in the following diagram:



The easiest way to ensure that these relative offsets don't matter quite so much is to leverage one page worth of padding for each `stbl` wrapping the inner data. This approach also requires that each page of inner data that you spray needs to be padded with some fake box headers that take up as much space as the `stbl` box header itself. This way you can have a predictable data offset in either the `stbl` page or in an inner data back.

So the modified layout I used for this is shown in the following illustration:



To summarise the illustration: On the top left is a representation of the full amount of in-file data you send in order to spray the heap with the data you want. This is a large block of data wrapped in numerous sample tables. The middle block is a zoomed-in representation of what the actual innermost encapsulating `stbl` looks like, showing the sixteen bytes of atom header overhead. Finally on the bottom right is a representation of the actual inner-data being sprayed, which consists of multiple pages that are designed to basically mimic the layout of the `stbl` wrappers themselves: sixteen bytes of unusable padding followed by a usable payload.

As long as you guess the sprayed range correctly this all means that no matter what page you hit, whether the inner spray data or the outer `stbl` pages, you can be confident that the data you need is present. Going this route just means that you increase the proportion of in-file data needed to spray memory, so you have `256kb * (n+1)` bytes of memory sprayed represented by `256kb + (sizeof(one page) * n)` bytes of file data.

In practice I found spraying about 128mb of memory, which took an MP4 file that was smaller than 2mb, was reliable enough to have a range that worked across all tested vulnerable builds and devices (running Android 5.x).

One other thing to note about the recursive data construct illustrated above, is that you don't need to explicitly use legitimate atom types to fill out the pages or for the inner data portion you want to spray. The `parseChunk()` function by design skips unrecognised atoms, so you can also just use some random invalid atom type and be sure that no side-effect allocations or anything else actually occur, but that the data still gets allocated and then skipped as needed.

With all that understood there is one last caveat worth noting, which dictates what exact `spray address + page offset` value you need to use. This has to do with some limitations imposed by the jemalloc heap, which I won't go into great detail about. For an exhaustive background on jemalloc please see the fantastic papers and presentations by argp and huku. To summarise, allocation of our spray data will still occur on the jemalloc heap. Typically, allocations are placed inside a structure called a run, and multiple runs exist in a larger mapping called a chunk. By default on Android a chunk is a 4mb mapping. At a high level what ends up happening is that our allocations will fill out each new 4mb chunk as best as they can fit, and once no room for a new allocation is left on the chunk, a new chunk will be allocated and will be filled and this process is repeated as necessary until all of our data is stored.

There are two important things we have to cope with related to this. The first couple of pages of the chunk aren't usable in practice, as they have some heap metadata present and are otherwise zeroed out. Secondly, because you won't fill each 4mb chunk precisely, there will be some number of pages at the end of each chunk that are empty. So things won't be quite as nicely aligned adjacently as earlier illustrations would have you believe. This just means that there are periodic, but deterministic, gaps in sprayed memory that you don't want to hit when guessing your spray address. When calculating your guessed spray address you just need to ensure it is at least a few pages into the start of a 4mb aligned address, and otherwise falls within a range that you can predictably spray across devices.

2.3 Bruteforce complexity woes

With the more reliable spraying technique available, we're left with just bruteforcing the libc (or some either librarie) base address. However, even with this ASLR bruteforce working quite reliably, you occasionally run into a non-ideal heap layout even on jemalloc. Occasionally the region (jemalloc version of what most of us think of as a chunk) you are overwriting happens to be the last region on a run and the region you're trying to target will be at the start of some new run in a totally different location. This means that even if you happened to have guessed the right base address on this

attempt, you won't know and have to continue bruteforcing. Even though this layout is relatively rare, it's still one of those things you have to accept might work against you.

2.4 Semi-effective multi-device support on Android 5.x

With a more predictable spray address that can work across devices, we still have the problem of not being able to bypass ASLR effectively without brute force, which means we're relying on ROP gadgets which will be specific to library versions on each firmware build we target. The best bet for fingerprinting the target seems to be relying on the web browser user-agent. If the victim is using Chrome or the AOSP browser then the user-agent is verbose enough (arguably too verbose) to give you almost all the information you need. An example of this is the following user-agent:

```
Mozilla/5.0 (Linux; Android 4.1.1; Nexus 7 Build/JRO03D) AppleWebKit/535.19 (KHTML,
like Gecko) Chrome/18.0.1025.166 Safari/535.19
```

We see the exact details for the Android version, the device model, and even the firmware build number. This is enough information to pull out the data we need from a pre-constructed database of firmware versions from which we've already mined all the necessary information. That said, this approach is easier said than done; the sheer volume of Android device models and build versions combined is staggering in practice. Building out a database of all of the gadgets for all of the models requires terabytes of disk space, scripts catering to each firmware type, and even then you're limited to what firmware you can find online or otherwise acquire and have had time to actually download and analyse. Even if you manage to obtain a decent amount of firmware it won't always reflect the number of actual firmware and device combinations you will see in the wild. What's more, some devices will actually identify as the same device, even though the underlying firmware builds will differ. For example, compare the razor and razorg devices, which are the Nexus 7 Wi-Fi-only and Nexus 7 Wi-Fi + Mobile tablets respectively, but both just identify as Nexus 7 in the user-agent..

Despite the shortcomings of this approach, if you hope to target multiple devices in the wild as part of a red team exercise then building out some sort of database like this is necessary. And if you actually archive all of the obtained firmware you can at least benefit from it for future Android exploits you build that might have similar constraints.

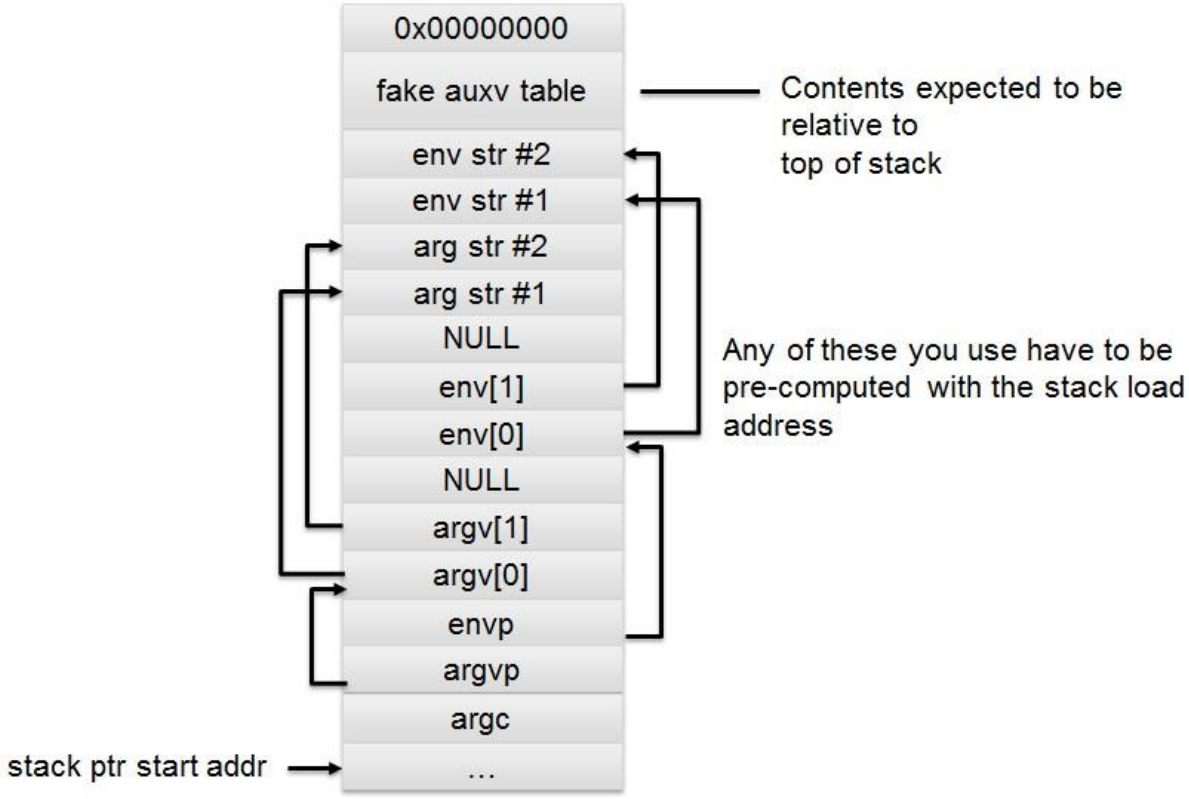
2.5 Working within the Android SELinux sandbox

Android 5.0 has SELinux policies in enforcing mode for all domains by default. This means the `mediaserver` process we're targeting when exploiting libstagefright is protected by a SELinux policy; specifically, it runs inside the `media` domain. This means if you do manage to get code execution you're in a fairly restrictive sandbox, which is pretty useless in practice. On the latest 5.x builds, for instance, the domain is restricted such that you can't spawn a shell, you can't execute any other useful binaries on disk, you can't drop and execute your own file, and you don't really have access to many interesting data locations (depending on what type of data you're actually after).

So ideally in this scenario we want to stage a secondary payload that can cope with these SELinux restrictions, and preferably something generic that we can reuse whenever we run into a similar situation. There was at least one recent public exploit, targeting CVE-2015-1528, that does diskless execution from within the `mediaserver` SELinux sandbox in an interesting way, but it explicitly requires local execution. The technique worked by pre-mapping a secondary exploit binary into a shared memory segment (via ashmem) and then linking it all up from the target address space once initial code execution was running and before it transitioned completely to the second-stage exploit. When exploiting `mediaserver` remotely this is not possible, as we can't control any shared memory mappings from a separate process.

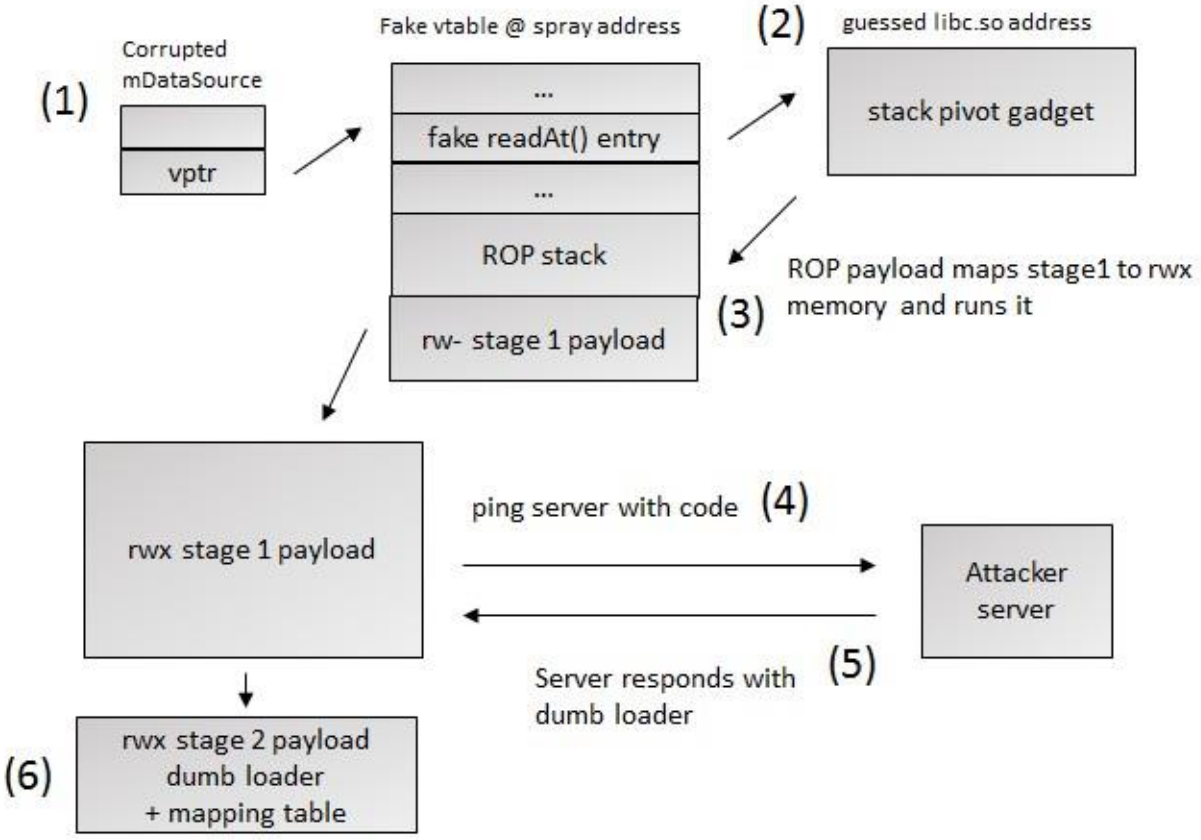
I chose an approach which was documented quite some time ago in phrack, which is an evolution of an even earlier technique called ul_exec; first publicly documented (as far as I am aware) by the grugq. The premise is to map a static ELF executable into memory, set up the stack to mimic how the kernel would initialise everything when starting up a new process, and then just jump to the ELF's entry point. The way I chose to do this differs slightly from how it has been done publicly in the past, in that I chose to implement a rudimentary mapping stub in assembly and do all of the setup on the exploit-side in python. This greatly simplifies what needs to be done in assembly, especially if things like auxiliary vectors set up by the kernel ever change.

Basically, the Python portion of the code chooses a static stack address and builds the argument and environment vectors, as well as the auxiliary vectors using the static address. This can be mimicked fairly easily by just reading the related kernel source. For those unfamiliar with this, the illustration below shows how the stack is laid out in memory, which we can fairly easily pre-construct in Python.



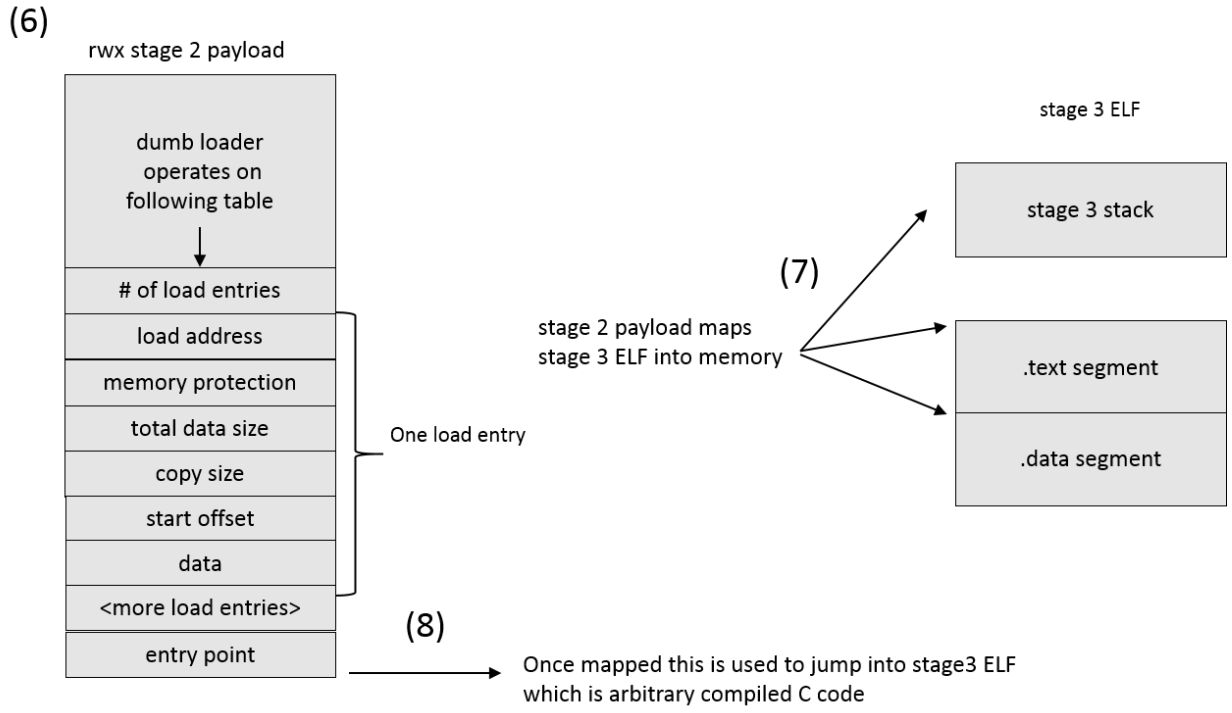
The Python script then reads in and sets up a mapping for every `PT_LOAD` segment from the static ELF file to be loaded. Care must be taken to ensure that all of the padding requirements are fulfilled, as the `rw-` segment typically starts at some offset into the first `rw-` page. Once all of the stack and loadable ELF segments are ready, a special load map structure is built and sent to the payload.

To fully understand what's going on here, I've created two illustrations. The first, below, shows getting up to the point of mapper payload described above:



The illustration above shows the initial six steps in getting from initial corruption to executing our diskless ELF execution payload. First we corrupt the vtable pointer of a target `mDataSource` object and issue a `readAt()` call. This triggers a stack pivot (2) in `libc.so` that jumps to a ROP payload (3). The ROP payload maps a stage1 payload into executable memory and executes it (4). The payload contacts the attacker controlled server requesting a stage2 payload (the diskless ELF mapper in assembly), which the server responds with (5). The stage2 payload is then executed (6).

The following illustration expands on what (6) involves in more detail:



We see that the stage2 payload (6) consists of a dumb loader, followed by an actual mapping table containing everything that needs to be loaded. It maps each entry in the table into the designated area (7), which will typically consist of a stack, text segment, and data segment. Though other things could be mapped as needed. Finally the dumb loader jumps to the entry point indicated by the mapping table (8).

As shown above, by keeping this model very simple, the first-stage shellcode can simply read in each entry in the load map structure and copy it as specified, with the only exception being that it stores the stack address starting point in order to set it right before it jumps into executing the static binary.

So this allows us to fairly trivially execute arbitrary binaries, as long as they are statically mapped. As the SELinux sandbox does have permissions to map the loader into memory, this could also be done relatively easily for dynamically-linked binaries, as we could also just map the loader and jump to its entry pointer rather than our own.

If you want to just poke around, a custom and statically-compiled copy of BusyBox that reads commands in a loop is relatively easy to build, and can at least let you see your id, poke at some typical shell commands, and so on. However, as I noted earlier, the SELinux policies make the sandbox environment nearly useless in practice, so it's better to try to break out of the sandbox

entirely.

2.6 Breaking out of the SELinux sandbox

I decided I would stage a custom-built CVE-2015-3636 (aka pingpongroot) exploit, which targets a socket-related use-after-free. In practice I found the public version of the exploit, which I started working from, fairly unreliable on a number of fronts, and also found it didn't cope well with certain SELinux policy scenarios, so made a few changes that I document in the following sections.

If you are not familiar with the pingpongroot exploit I suggest you read the presentation by k33n team, who found the bug and wrote the original exploit. I won't go over all the details of this vulnerability or exploit either, but will just touch on a few things I did to improve it for my needs.

It's possible that some of what I do here is overly complicated, but I don't have much experience dealing with SELinux, so just did what came to mind rather than what might be more normal practices! Again, feel free to school me on anything I'm missing.

2.6.1 Improving reliability

The way the pingpongroot exploit works in general is to spray a number of vulnerable sockets into different parts of memory, free them, and then try to re-map the memory indirectly by spraying userland mappings. The goal here is to extend a kernel construct called the *physmap*, which is a kernel space virtual address range that points to physical memory allocated by the system. The exploit repeatedly tries to extend the *physmap* in hopes of overlapping the freed (but still referenced) vulnerable socket structures, and each time tries to access the freed sockets to see if one of them has been replaced with controlled data (using a clever information leak). On my Nexus devices I found in practice that the *physmap* extension overlapping trick was actually fairly unreliable with the default set of sockets that were sprayed, and in some cases the exploit spun trying to remap indefinitely (or at least long enough that I gave up waiting).

I fixed this problem fairly simply by implementing a counter that prompted the exploit to spray some new vulnerable sockets after some number of tries, under the assumption that the current *physmap* extension was not going to overlap any of our existing structures. This doesn't introduce any new unreliability, as the process is already holding a bunch of unstable sockets, so adding a few more to that list doesn't change anything other than to place them in new parts of memory that might have a better chance of overlapping with *physmap*. I found that by simply adding that feature the exploit pretty much always worked, and typically in a much shorter time period.

2.6.2 Improved stability by cleaning up

Another issue with the exploit I used as my starting point was that it expects you to reboot the device after the process has been rooted, because there are stale freed socket references lying around in the kernel. I never actually saw many cases of successful exploitation where these invalid sockets weren't retained in memory, which always prevents the process from being torn down (as otherwise the stale sockets are closed, which will crash the kernel), and so basically always a reboot scenario was encountered. I found that this caused instability when trying to execute new processes and actually getting a practically useful root shell.

I actually ended up solving this in two ways, and I guess this is a good example of the iterative process of building an exploit sometimes. The lazy solution to this problem eventually helped me with something I needed later to find important SELinux-related identifiers, so in the end I was still happy I did it first.

I also attempted to do this by as device-agnostic a method as possible.

2.6.2.1 The lazy way

Originally I decided that doing a proper socket cleanup might be annoying, so instead I wanted to isolate the unstable process so that it would never die unless it was explicitly killed, but that it would also be safe to root some process and carry on with the attack as if there was no unstable process that existed.

The way I originally approached this was to introduce an additional process that is the parent of the actual process doing the exploitation. Once kernel code execution was triggered from the child, the child could escalate the privileges of its parent and then basically enter a zombie-like state. Since the parent doesn't hold any references to any of the bad sockets, it is mostly free to do whatever it wants without risking a crash. The key problem here is that the kernel payload needed modification in order to find the `task_struct` pointer to the parent process, which requires a different search pattern than the usual trick for finding the `cred` structure. This new search is relatively easy, as the parent pointer is at a relative offset to some easily discoverable values. The following is the relevant part of the `task_struct` structure (which you can find in `include/linux/sched.h`) in the kernel source:

```
struct task_struct {
        [...]
         unsigned long atomic_flags; /* Flags needing atomic access. */
        struct restart_block restart_block;

        pid_t pid;
        pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
        /* Canary value for the -fstack-protector gcc feature */
        unsigned long stack_canary;
#endif
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->real_parent->pid)
         */
        struct task_struct   rcu *real_parent; /* real parent process */
        struct task_struct   rcu *parent; /* recipient of SIGCHLD, wait4() reports
*/
        /*
         * children/sibling forms the list of my natural children
         */
        struct list_head children; /* list of my children */
        struct list_head sibling;  /* linkage in my parent's children list */
        struct task_struct *group_leader; /* threadgroup leader */
        [...]
}
```

We see that not far above the `parent` member are the `pid` and `tgid` values (highlighted in bold), which we can easily obtain for our own process. This means we can search from the base of the `task_struct` structure for these values. The only hurdle then is determining dynamically if the

`stack_canary` member is actually present, since we don't want to have to check every device kernel configuration, but as long as our process isn't in some abnormal state, the `real_parent` and `parent` members should both hold the same address. So once we know the offset of `tgid`, we can test if the next value in memory matches the one after it. If not, then `stack_canary` is present and needs to be skipped.

2.6.2.2 The right way

After living with the approach described above for a little while, and not having any real stability issues, it started to bug me that things still weren't properly cleaned up, so I decided to see if I could do it even better.

The ideal approach would be to find the file descriptor map dynamically and just remove every reference to the bad sockets, which we can keep references to by file descriptor. This way none of them are touched when the exploit process is torn down on exit.

I came up with a fairly reliable approach to doing this, which relies on yet another search pattern. To start, we can leverage the fact that we already know the offset of the `comm` array (from looking up the `cred` structure), which is a bit before the file descriptor reference as shown below:

```
struct task_struct {
       [...]
       const struct cred   rcu *cred;   /* effective (overridable) subjective task
                                         * credentials (COW) */
       char comm[TASK_COMM_LEN]; /* executable name excluding path
                                  - access with [gs]et_task_comm (which lock
                                    it with task_lock())
                                  - initialized normally by setup_new_exec */
/* file system info */
       struct nameidata *nameidata;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
       struct sysv_sem sysvsem;
       struct sysv_shm sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
       unsigned long last_switch_count;
#endif
/* filesystem information */
       struct fs_struct *fs;
/* open file information */
       struct files_struct *files;
/* namespaces */
       struct nsproxy *nsproxy;
/* signal handlers */
       struct signal_struct *signal;
       struct sighand_struct *sighand;

       sigset_t blocked, real_blocked;
       sigset_t saved_sigmask;    /* restored if set_restore_sigmask() was used */
       struct sigpending pending;
       [...]
}
```


Note that between `comm` and the `files` member there are some configuration-specific entries, so we couldn't reliably use a static offset from `comm`. It is better to make no assumptions about configuration-specific entries because, as noted earlier, we don't want to have to keep track of and accommodate every device's particular configuration.

With that in mind we want to find another way. If you take note of the `blocked` and `real_blocked` members, you can see they are at a static offset after the `files` member, which means we might be able to search forward to those if we could populate one of them with a signature.

If you're familiar with signal handling on Linux, you'll remember that when you define a handler for a specific signal, you can also designate a list of signals that become blocked while handling the signal. It turns out the `blocked` member is where this data is stored. So what this means is that we can define a handler with some special magic value in the list of `blocked` signals, and then just trigger the actual kernel code execution from inside a signal handler by raising the signal we registered our handler with. The following code shows how you can easily trigger the bug from inside a handler:

```
struct sigaction sa = {0};
sa.sa_handler = trigger;
sa.sa_mask = 0xdead9eef;
sigaction(SIGUSR1, &sa, NULL);
trigger_sock = socks[i];
raise(SIGUSR1);
break
```

Where the `trigger` function will simply close the socket designated by the global `trigger_sock`, to trigger our payload. Then, once we find the `0xdead9eef` signature from without our payload, we can use a static offset backwards to find the `files` pointer. Since we basically control this entire search signature value (aside from a few masked off bits), we don't need to actually even search relative to the `comm` array for it to be stable, but can also just search from the start of the `task_struct` in memory.

Once you have the `files` pointer things are fairly straightforward, as you can just mimic what the kernel does when it closes a file. The relevant structure, of type `files_struct`, looks like this:

```
struct files_struct {
  /*
   * read mostly part
   */
        int count; // was atomic_t
        struct fdtable *fdt;
        struct fdtable fdtab;
  /*
   * written part on a separate cache line in SMP
   */
        unsigned file_lock; // was spinlock_t
        int next_fd;
        unsigned long close_on_exec_init[1];
        unsigned long open_fds_init[1];
        struct file * fd_array[NR_OPEN_DEFAULT];
}
```

We're specifically interested in the `fdtable` pointed to by `fdt`, and the `next_fd` value. We will want to set the `next_fd` value if any of the stale descriptors to which we're removing references has

a higher value. The `fdtable` structure, which will need multiple modifications, is defined as follows:

```
struct fdtable {
        unsigned int max_fds;
        struct file **fd;      /* current fd array */
        unsigned long *close_on_exec;
        unsigned long *open_fds;
        struct rcu_head rcu;
        struct fdtable *next;
}
```

For each descriptor we want to remove we'll be indexing into the `fdt->fd` table to `NULL` out the file pointer that was previously referenced. We will also want to modify the `fdt->close_on_exec` and `fdt->open_fd` bitmaps which indicate certain descriptors are in use.

That's basically enough to make the system stable and allow you to kill the process that used to hold the stale references. There could be more to do to make it perfect, but in practice it seems unnecessary. So you end up with code similar to the following:

```
static void
remove_uaf_socks(struct files_struct * fsp, int * socks)
{
    if (NULL == socks) {
        return;
    }
    struct fdtable * fdt = fsp->fdt;
    int32_t i;
    for (i = 0; socks[i] != -1; i++) {
        uint32_t s = socks[i];
        // This should never happen for us, but just in case
        if (s >= fdt->max_fds) {
            continue;
        }
        // Remove the struct file * reference fdt-
        >fd[s] = NULL;
        // Unset the close on exec bit
        __clear_bit(s, fdt->close_on_exec);
        // Remove it as an open fd
        __clear_bit(s, fdt->open_fds);
        if (s < fsp->next_fd) {
            fsp->next_fd = s;
        }
    }
}
```

2.6.3 Disabling SELinux

Having not really done much with SELinux in the past, one interesting problem I didn't expect to encounter is that SELinux restrictions were imposed on what we run as root. I was originally using the default kernel payload in the public exploit, which places the process into the `kernel` domain (using a security identifier value of 1), which is still fairly restricted on the devices I tested.

You can see what the domain is allowed to do by running the following command from Linux, after downloading the sepolicy from the device or extracting it from downloaded firmware:

```
$ sesearch --allow -s kernel sepolicy
```

Unfortunately you seem to have to extrapolate what a domain can't do by noting what is omitted from the list of what is allowed, so there is no easy way to show everything it can't do. But in general, the domain can't make outgoing network connections, nor can it execute a shell, so there is no easy remote root shell that way. Most importantly it can't manipulate existing SELinux policies, which means we can't just change to permissive mode for the entire system.

In the exploit I was starting from, the code used to escalate the privileges of the task looks like this:

```
static void
root_task(struct cred *cred)
{
  struct task_security_struct *security;

  cred->uid = 0;
  cred->gid = 0;
  cred->suid = 0;
  cred->sgid = 0;
  cred->euid = 0;
  cred->egid = 0;
  cred->fsuid = 0;
  cred->fsgid = 0;
  cred->cap_inheritable.cap[0] = 0xffffffff; cred-
  >cap_inheritable.cap[1] = 0xffffffff; cred-
  >cap_permitted.cap[0] = 0xffffffff; cred-
  >cap_permitted.cap[1] = 0xffffffff; cred-
  >cap_effective.cap[0] = 0xffffffff; cred-
  >cap_effective.cap[1] = 0xffffffff; cred->cap_bset.cap[0] =
  0xffffffff;
  cred->cap_bset.cap[1] = 0xffffffff;

  security = cred->security;
  if (security) {
    if (security->osid != 0
     && security->sid != 0
     && security->exec_sid == 0
     && security->create_sid == 0
     && security->keycreate_sid == 0
     && security->sockcreate_sid == 0) {
      security->osid = 1
      security->sid = 1;
    }
  }
}
```

The `osid` and `sid` values in the code above end up indicating what domain the task is in. I spent some time trying to see a way to abuse this domain to do something more useful, but I eventually decided it would be easier to transition to a domain that is still more powerful, specifically one that can manipulate existing policy, as dictated by the rule `kernel : security load_policy`. It turns out the only domain that can do this, at least on the devices I tested, is the `init` domain, which

naturally only the `init` process is in:

```
shell@flo:/ $ ps -Z | grep init
u:r:init:s0                    root      1     0     /init
```

You can find what domains have this ability by using the command:

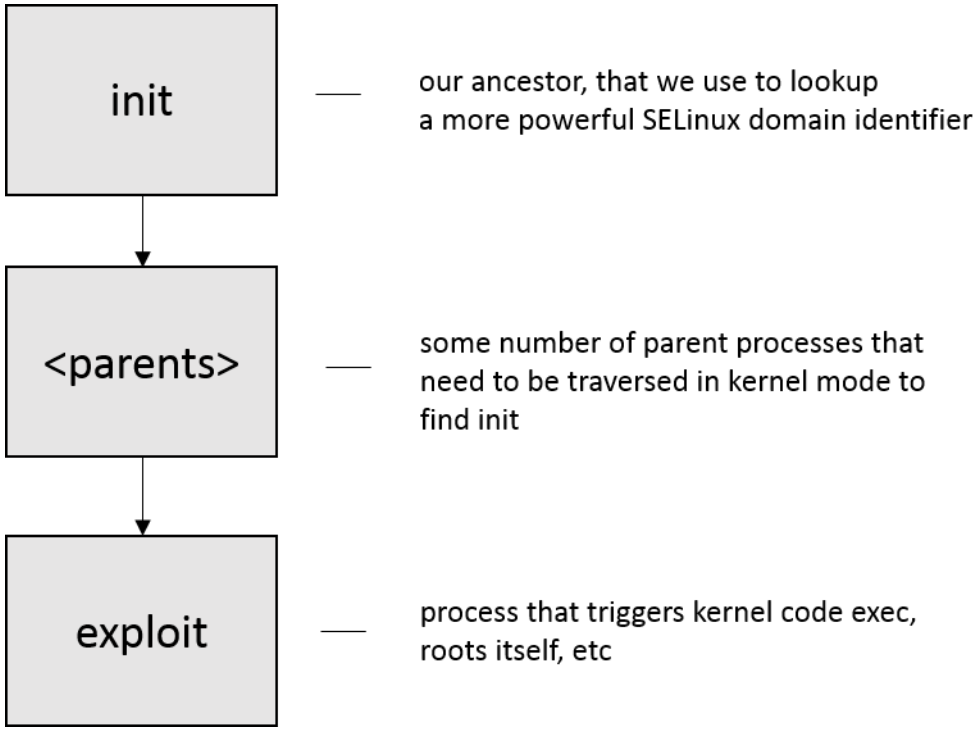
```
$ sesearch --allow sepolicy  | grep load_policy
```

In order to switch to the `init` domain, instead of overwriting the security identifiers in our `cred` structure with the value of 1, we need to somehow determine what the `init` domain's `sid` value actually is. In theory this might change, so ideally we want to find it dynamically. At first I tried using some of the common SELinux tools, like `sesearch` and `seinfo`, to see if I could pull this id out of a compiled `sepolicy` file, but neither seemed to be able to tell me. I also implemented a custom tool that tried to pull out the security id value from the `sepolicy` directly, using the `libsepol` library; however, in practice it always returned a static value that didn't actually correspond to the value used by the kernel.

I then decided the better way to approach the problem was to find and borrow `init`'s security id directly from the kernel. This is especially useful because if the identifier happened to change per device policy then I'd not have to track it, or anything else. In order to find the identifier you need a way to find the task structure for `init` inside the kernel, which happens to be fairly easy if you are familiar with process hierarchies on Linux. Normally, when a process forks a child, the child's `task_struct` structure will have a pointer to the parent task from which it was forked. However, if a child's parent process dies then the child is adopted by the `init` process. Similarly the `init` process will always be the ancestor of a process at some point, so if you walk back far enough through your ancestors you can find it.

This means that as long as we have some way to locate the parent task pointer within the `task_struct` structure, which we already described earlier when approaching stability from a lazy perspective, we can walk the parent processes until we find the `init` process. Alternatively, we could deliberately orphan the process that we plan to root, so that we can find `init` by finding its new parent, and simply patch the `init` security identifier into the `cred` structure of our process. In the end it turns out this value is 0, which I suppose could probably have guessed, but it's worth having a generic way to look it up in the event this ever changes.

The process hierarchy leveraged for exploitation looks like the following illustration. If you wanted to guarantee `init` was your immediate parent, as noted earlier you could alternatively just orphan the exploit process, but in practice this isn't necessary:



Once this is set up, we can easily place our orphan process into the `init` domain, which is significantly less restricted than other domains. However, even the `init` domain is technically sandboxed!

Ideally we want to be able to set SELinux to permissive mode so that we aren't restricted at all, but it turns out that even the `init` domain doesn't have the ability to call `setenforce` and change the value. This is because doing so is restricted by the `kernel : security setenforce` permission, which no domain actually has after boot.

Fortunately for us, as noted earlier, we know our new domain at least has the `load_policy` permission. So in order to work around this last limitation, from within our rooted exploited process, we can load the existing `sepolicy` file from disk, using the `libsepol` library, and then insert a custom permissive rule for the `init` and `init_shell` domains and load the new policy, which will mean that it can carry out any restricted action and SELinux will just log an audit warning without actually enforcing a restriction. Once this is done, the `init` domain can actually disable SELinux entirely for the system using the `setenforce 0` shell command. Assuming a loaded policy database, the code for setting up permissive rules looks like the following (which relies on the `libsepol` library API):

```
type_datum_t * type;
type = hashtab_search(policydb.p_types.table, "init_shell");
ebitmap_set_bit(&policydb.permissive_map, type->s.value, 1);
```

Also note that there are other ways to deal with SELinux from a kernel exploit, which have been demonstrated in the past. Spender (@grsecurity) straight up patched SELinux functionality in the

kernel (to break SELinux, but report as if it still works) in his enlightenment exploit framework, but I didn't go this route because I couldn't easily access kernel symbols to find the associated functions in kernel memory, and I didn't want to rely on searching for certain opcodes as they could change across compilations. But it's still a good approach to consider, especially if we eventually can't rely on the `init` having `load_policy` permissions.

3 Conclusion

So, with all of above tweaks and tricks we can reliably and remotely root an Android 5.x device that is vulnerable to both CVE-2015-3684 and CVE-2015-3636, despite the SELinux sandbox. It is done in such a way that staging other kernel exploits to break out of SELinux is easy if necessary.

There are almost always some fun tricks like this you can come up with developing a new exploit or modifying an existing one. But, nevertheless, I think exploiting bugs these days without a useful info leak and on the Android ecosystem is clearly quite a mess. Although I tried to demonstrate a few tricks to improve the exploitation experience, there are still major hurdles. All of what I've said is framed around the assumption that you don't have a useful memory revelation leak primitive, which perhaps others have found privately and I'm just missing. But assuming you don't have one, brute-forcing ASLR, despite weak entropy, still adds a fairly non-ideal brute-force timing window. Furthermore, the sheer volume of Android devices makes it a challenge to build (let alone effectively test) a complete catalogue of usable gadget offsets for every device. And if the browser is ever changed to omit the exact device model and build information, it will get much harder.

So how long did this all take? Well this paper highlights the more interesting results of about two months of work. The work involved collecting test devices and archiving and analysing associated firmware, familiarising ourselves with Android in general and the libstagefright bugs, investigating reliable exploitation of CVE-2015-3684 on both Android 4.x (dlmalloc) and 5.x (jemalloc), investigating other public stagefright bugs to look for good ASLR bypass candidates, investigating and failing at numerous other ASLR bypass ideas, working on non-HTTP transport exploitation for CVE-2015-3684 (which proved to be relatively uninteresting as there was no ASLR bypass, although it did work), learning more about SELinux, etc. Things clearly aren't as easy as they used to be!

Some thoughts on SELinux: Although in my opinion the SELinux policy for `mediaserver` on Google Android images (maybe not the case for all other vendors) is pretty solid, I think in-memory ELF loading makes dealing with SELinux a lot less of a hurdle in general. It allows you to steal data more easily, and as long as there is some other exploit you can stage then it's still fairly easy to break out and then disable. I think if SELinux is to be more effective it would need to be more aggressively baked into the system, such that you can't just turn it off and that you can't easily shut it off at a central point. I appreciate this isn't a new argument against SELinux, but perhaps this paper will further help to prove that point.

Thanks for reading, and hopefully something in the post was useful or interesting to you. As always, feedback is welcome and you can reach me at aaron<.>adams<@>nccgroup<.>trust or on Twitter @fidgetingbits