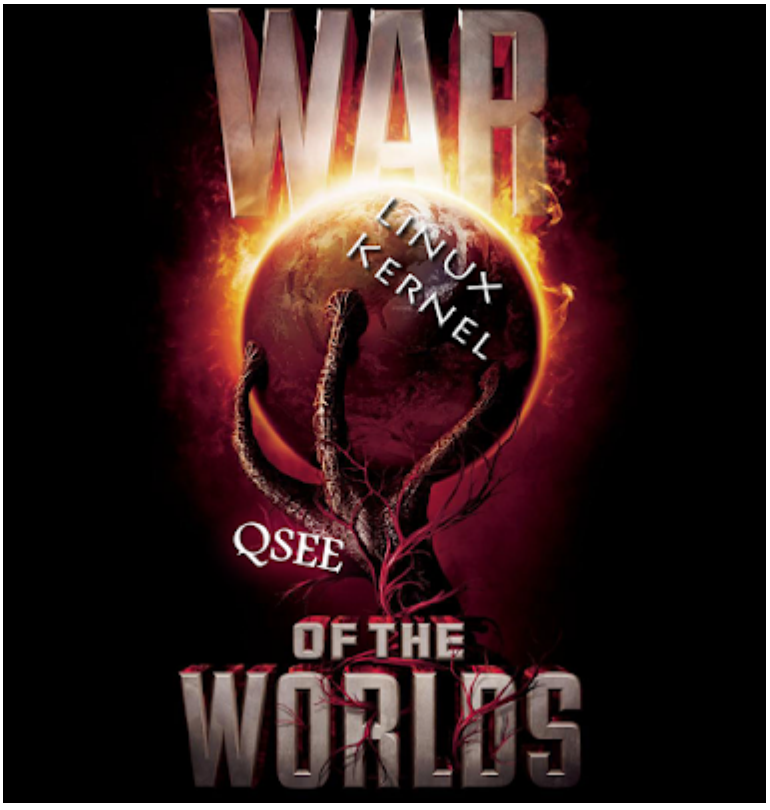
War of the Worlds - Hijacking the Linux Kernel from QSEE

Archived: 2026-04-05 17:50:19 UTC

After seeing a full QSEE vulnerability and exploit in the previous blog post, I thought it might be nice to see some QSEE shellcode in action.

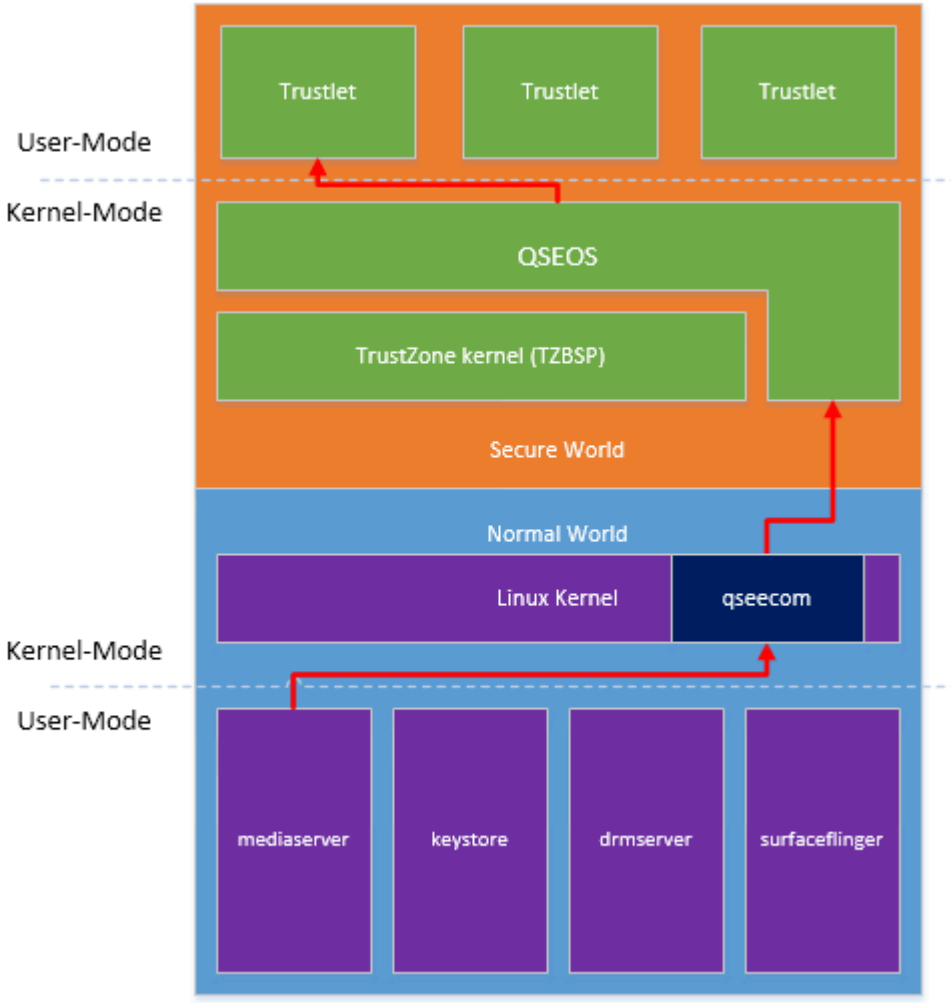
As we've previously discussed, QSEE is extremely privileged - not only can it interact directly with the TrustZone kernel and access the hardware-secured TrustZone file-system (SFS), but it also has some direct form of access to the system's memory.

In this blog post we'll see how we can make use of this direct memory access in the "Secure World" in order to hijack the Linux Kernel running in the "Normal World", without even requiring a kernel vulnerability.



Interacting with QSEE

As we've seen in the previous blog post, when a user-space Android application would like to interact with a trustlet running in QSEE, it must do so by using a special Linux Kernel device, "qseecom". This device issues SMC calls which are handled by QSEOS, and are passed on to the requested trustlet in order to be handled.



Each command issued to a trustlet has a pair of associated input and output buffers, which are usually used to convey all the information to and from the "Normal World" and the trustlet.

However, there are some special use-cases in which a faster mode of communication is required - for example, when decrypting (or encrypting) large DRM-protected media files, the communication cost must be as small as possible in order to enable "smooth" playback.

Moreover, some devices include trustlets which are meant to assure the device's integrity (mostly in corporate settings). For example, Samsung provides "**T**rustZone-based **I**ntegrity **M**easurement **A**rchitecture" (**TIMA**) - a framework used to assure device integrity. According to Samsung, TIMA performs (among other things) periodic measurements of the "Normal World" kernel, and verifies that they match the original factory kernel.

So... Trustlets need fast communication with the "Normal World" and also need some ability to inspect the system's memory - sounds dangerous! Let's take a closer look.

Sharing (Memory) is Caring

Continuing our research on the "widevine" trustlet, let's take a look at the command used to DRM-encrypt a chunk of memory:

```
signed int __fastcall OEMCrypto_Generic_Encrypt(unsigned int session_idx,
                                                int input_buf, int buf_len,
                                                int cipher_type, int unused,
                                                int output_buf)
{
  void* session = wv_get_session(session_idx);
  if ( session && buf_len && !unused )
  {
    if ( cacheflush_register(input_buf, buf_len, output_buf, buf_len) )
    {
      qsee_log(
        3,
        "[%s:%d] Error:  OEMCrypto_Generic_Encrypt cacheflush_register failed!\n",
        "OEMCrypto_Generic_Encrypt",
        6337);
      res = 10003;
    }
    else
    {
        //Do actual encryption of the given buffer...
        <...SNIP...>
    }
    if (cacheflush_deregister_0(input_buf, buf_len, output_buf, buf_len) )
    {
        qsee_log(
          3,
          "[%s:%d] Error: OEMCrypto_Generic_Encrypt cacheflush_deregister failed!\n",
          "OEMCrypto_Generic_Encrypt",
          6428);
        res = 10003;
  }
  if ( res )
    qsee_log(
      3,
      "[%s:%d] Error: OEMCrypto_Generic_Encrypt return %d!\n",
      "OEMCrypto_Generic_Encrypt",
      6434,
      res);
  return res;
}
```

As we can see above, the function receives two pointers denoting the "input" and "output" buffers, respectively. These can be any arbitrary buffers provided by the user, so it stands to reason that some preparation would be needed in order to access them. Indeed, we can see that the preparation is done by calling "cacheflush_register", and, once the encryption process is done, the buffers are released by calling "cacheflush_deregister".

Upon closer inspection, "cacheflush_register" and "cacheflush_deregister" are simple wrappers around a couple of QSEE syscalls (each):

cacheflush_register	**cacheflush_deregister**	
qsee_register_shared_buffer	qsee_prepare_shared_buf_for_nosecure_read	
qsee_prepare_shared_buf_for_secure_read	qsee_deregister_shared_buffer	

So what do these syscalls do?

Looking at the relevant handling code in QSEOS reveals that these names are a little misleading - in fact, "qsee_prepare_shared_buf_for_secure_read" merely invalidates the given range in data cache (so that QSEE will observe the updated data), and similarly "qsee_prepare_shared_buf_for_nosecure_read" clears the given range from the data cache (so that the "Normal World" will see the changes made by QSEE).

As for "qsee_register_shared_buffer" - this syscall is used to actually map the given ranges into QSEE. Let's see what it does:

```
signed int __fastcall qsee_register_shared_buffer(unsigned int buf_addr, int buf_len)
{

  //Validity checks to make sure there are no overflows, etc.
  <...SNIP...>

  //Checking for the specially allowed ranges in the "Secure World"
  if ( (is_ns_disallowed_range(buf_addr, buf_len) ||
        !is_ns_allowed_range(dword_FE824920, buf_addr, end_addr - 1))
    && !qsee_is_tag_area(1, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(2, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(3, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(4, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(6, buf_addr, buf_addr + buf_len)
    && !qsee_is_tag_area(5, buf_addr, buf_addr + buf_len) )
  {
    log(5, "{%x:%x %x}", -54, buf_addr, buf_len);
    return -1;
  }

  //Inserting the entry into the mapped buffers list
  <...SNIP...>

  //Mapping the buffer into QSEE!
  qsee_map_region(buf_addr, buf_addr, buf_len, 6, 32773, 1);
  return 0;
}
```

After some sanity checks, the function checks whether the given memory region is within the "Secure World". If that's the case, it could be that the trustlet is trying to attack the TrustZone kernel by mapping-in and modifying memory regions used by TZBSP or QSEOS. Since that would be extremely dangerous, only a select few (six) specific regions within the "Secure World" can be mapped into QSEE. If the given address range is not within any of these special "tagged" regions, the operation is denied.

However - for any address in the "Normal World", there are no extra checks made! This means that QSEOS will happily allow us to use "qsee_register_shared_buffer" in order to map in any physical address in the "Normal World".

...Are you pondering what I'm pondering?

Since QSEE has read-write access to all of the "Normal World"'s memory (all it needs to do is ask), we should theoretically be able to locate the running Linux Kernel in the "Normal World" directly in physical memory and inject code into it.

As a fun exercise, let's create a QSEE shellcode that doesn't require any kernel symbols - this way it can be used in any QSEE context in order to locate and hijack the running kernel.

Recall that after booting the device, the bootloader uses the data specified in the Android boot image in order to extract the Linux Kernel into a given physical address and execute it:

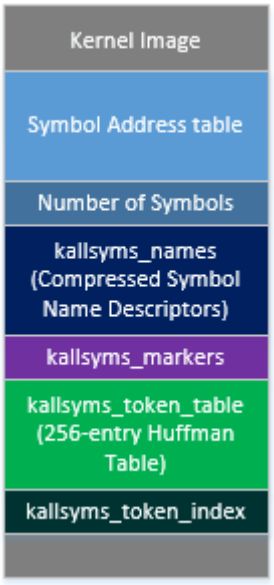
```
struct boot_img_hdr {
        unsigned char magic[BOOT_MAGIC_SIZE];

        unsigned kernel_size;  /* size in bytes */
        unsigned kernel_addr;  /* physical load addr */

        unsigned ramdisk_size; /* size in bytes */
        unsigned ramdisk_addr; /* physical load addr */

        unsigned second_size;  /* size in bytes */
        unsigned second_addr;  /* physical load addr */

        unsigned tags_addr;    /* physical addr for kernel tags */
        unsigned page_size;    /* flash page size we assume */
        unsigned unused[2];    /* future expansion: should be 0 */

        unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */

        unsigned char cmdline[BOOT_ARGS_SIZE];

        unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};
```

The physical load address of the Linux Kernel is then available to any process via the world-readable file /proc/iomem:

```
shell@shamu:/ $ cat /proc/iomem
00000000-0d3fffff : System RAM
  00008000-00f02cbb : Kernel code
  0110e000-014e63ef : Kernel data
0fe00000-bfffffff : System RAM
```

However, simply knowing where the kernel is loaded does not absolve us from the need to find kernel symbols - there is a large amount of kernel images and an equally large amount of symbols per kernel. As such, we need some way to find the all of the kernel's symbols dynamically using the running kernel's memory. However, all is not lost - remember that the Linux Kernel keeps a list of all kernel symbols internally (!), and allows kernel modules to lookup these symbols using a special lookup function - "kallsyms_lookup_name". So how does this work?

As we've previously seen - the names in the kernel's symbol table are compressed using a 256-entry huffman coding generated at build time. The huffman table is stored within the kernel's image, alongside the descriptors for each symbol denoting the indices in the huffman table used to decompress it's name. And, of course, the actual addresses for all of the symbols are similarly stored in the kernel's image.



In order to access all the information in the symbol table, we must first find it within the kernel's image.

As luck would have it, the first region of the symbol table - the "Symbol Address Table", always begins with two pointers to the kernel's virtual load address (which can be easily calculated from the kernel's physical load address since there's no KASLR). Moreover, the symbol addresses in the table are monotonically nondecreasing addresses within the kernel's virtual address range - a fact which we can use to confirm our suspicion whenever we find two such consecutive pointers to the kernel's virtual load address.

```
00 80 00 C0 00 80 00 C0 4C 80 00 C0 00 81 00 C0
0C 81 00 C0 00 00 10 C0 00 00 10 C0 00 00 10 C0
20 02 10 C0 FC 03 10 C0 A4 04 10 C0 A8 04 10 C0
A8 04 10 C0 C4 04 10 C0 E0 04 10 C0 D4 05 10 C0
D8 05 10 C0 EC 05 10 C0 04 06 10 C0 48 06 10 C0
```

Symbol Address Table

Now that we can find the symbol table within the kernel's image, all we need to do is implement the decompression scheme in order to be able to iterate over it and lookup any symbol. Great!

Using the method above to find the kernel's symbol table, we can now locate and hijack any kernel function from QSEE. Following the tradition from the previous kernel exploits, let's hijack an easily accessible function pointer from a very rarely-used network protocol - PPPOLAC.

The function pointers relating to this protocol are stored in the following kernel structure:

```
static struct proto_ops pppolac_proto_ops = {
    .family = PF_PPPOX,
    .owner = THIS_MODULE,
    .release = pppolac_release,
    .bind = sock_no_bind,
    .connect = pppolac_connect,
    .socketpair = sock_no_socketpair,
    .accept = sock_no_accept,
    .getname = sock_no_getname,
    .poll = sock_no_poll,
    .ioctl = pppox_ioctl,
    .listen = sock_no_listen,
    .shutdown = sock_no_shutdown,
    .setsockopt = sock_no_setsockopt,
    .getsockopt = sock_no_getsockopt,
    .sendmsg = sock_no_sendmsg,
    .recvmsg = sock_no_recvmsg,
    .mmap = sock_no_mmap,
};
```

Overwriting the "release" pointer in this structure would cause the kernel to execute our crafted function pointer whenever a PPPOLAC socket is closed.

Putting it all together

Now that we have all the pieces, all we need to do to gain code execution within the Linux Kernel is to:

- Achieve QSEE code execution
- Map-in all the kernel's memory in QSEE using "qsee_register_shared_buffer"
- Find the kernel's symbol table
- Lookup the "pppolac_proto_ops" symbol in the symbol table
- Overwrite any function pointer to our user-supplied function address
- Flush the changes made in QSEE using "qsee_prepare_shared_buf_for_nosecure_read"
- Cause the kernel to call our user-supplied function by using a PPPOLAC socket

I've written some QSEE code which performs all of these steps and exports an easy-to-use interface to allow kernel code execution, like so:

```
/**
 * Executes the given function within the kernel, using the supplied arguments.
 * @param func_addr The address of the function to execute.
 * @param arg1 The first argument.
 * @param arg2 The second argument.
 * @param arg3 The third argument.
 * @param arg4 The fourth argument.
 * @return The result of the function's execution.
 */
uint32_t execute_in_kernel(uint32_t func_addr, uint32_t arg1, uint32_t arg2,
                           uint32_t arg3, uint32_t arg4) {
```

As always, you can find the full code here:

https://github.com/laginimaineb/WarOfTheWorlds

I should note that the code currently only reads memory one DWORD at a time, making it quite slow. I didn't bother to speed it up, but any and all improvements are more than welcome (for example, reading large chunks of memory at a time would be much faster).

In the next blog post, we'll continue our journey from zero-to-TrustZone, and attempt to gain code execution within the TrustZone kernel.
