Boot Sequence

From Vita Development Wiki

The Vita main application processor is a Cortex A9 MPcore (http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407i/DDI0407I_cortex_a9_mpcore_r4p1_trm.pdf). It implements ARM TrustZone for execution in both a non-secure world and a sandboxed Secure World.

Contents

Boot Process

Boot ROM

It is likely the the F00D processor (MeP Core) is the actual secure boot device rather than the ARM CPU. The F00D processor likely is Toshiba MEP based and is the first secure device ("first loader") to start on the Vita. Once it starts it likely maps the eMMC and directly reads in the second_loader.enp or second_loader.enp_ from the eMMC. This is in the native load format of the F00D bootrom. There are likely 2 layers of encryption. First it decrypts the per-console layer that was added during the install. After that it will decrypt the factory-encrypted layer then begin execution.

The second_loader is primarily responsible for preparing the ARM processor. It initializes DRAM and decrypts kernel_boot_loader.self into DRAM. It also writes the ARM exception vector and some boot context information to the 32KB scratch buffer (mirror mapped to 0x00000000 on ARM). The kernel_boot_loader.self contains both the secure world bootloader and kernel, as well as the non-secure boot loader. At this point the kprx_auth_sm.self and prog_rvk.srvk are both loaded into DRAM.

Finally, the second_loader resets itself with a pointer to the secure_kernel.enp or enp_. The F00D processor then restarts and loads the secure_kernel.enp in and again decrypts the per-console layer that was added by the install, and the factory layer. At this point the F00D processor secure kernel is prepared and it resets the ARM CPU at 0x00000000 (F00D scratch buffer). This triggers the ARM secure boot process to begin.

Secure Kernel

The bootloader decompressed the ARZL secure kernel, loads it and sets up the VBAR and MVBAR. It then decompresses the ARZL non-secure bootloader and sets NS in SCR and jumps into non-secure bootloader. See Secure Bootloader for more information.

SceKblForKernel

The non-secure bootloader contains an embedded and likely stripped version of SceSysmem, SceKernelModulemgr, SceSblSmschedProxy, and some core drivers. The non-secure bootloader sets up the eMMC device (again) and loads `os0:psp2bootconfig.skprx`.

ScePsp2BootConfig

This kernel module does not export any libraries. It only has a module init function that has a hard coded list of core kernel modules (ex: sysmem.skprx) which are loaded with calls back into SceKblForKernel. Once the core initialization is done, the next module to run is SceSysStateMgr.

SceSysStateMgr

This kernel module also does not export any libraries. Its init function first loads all the bootfs modules. Then it decrypts `os0:psp2config.skprx` and parses the System Configuration Script to load the remaining modules and finally either SceSafemode or SceShell.

Boot Partition

The boot partition is SLB2 formatted. It contains entries these files:

Name	Earliest Known Version	Comments
kernel_boot_loader.self	1.05	This could be the non-secure bootloader SceKblForKernel
kprx_auth_sm.self	1.05	Used by F00D Processor to decrypt SELFs
prog_rvk.srvk	1.05	SCE encrypted revocation data of some sort.
second_loader.enp	1.05	Possibly the secure bootloader
second_loader.enp_	1.69	Related to second_loader.enp in some way, likely for encryption
secure_kernel.enp	1.05	Possibly the secure kernel ARZL compressed and loaded into memory by ROM.
secure_kernel.enp_	1.05	Related to secure_kernel.enp in some way, likely for encryption

System Configuration Script

`os0:psp2config.skprx` once decryption is a UTF-8 text file that is parsed by SceSysStateMgr. It is a very

simple script format.

Comments

Comments start out with `#`, as an example, here's the header of 1.69 psp2config.skprx

```
#
# PSP2 System Configuration for Release
#
# [NOTICE]
#
# This configuration is only for kernel_boot_loader_release.self.
#
```

Conditionals

Conditionals start with `if` and end with `endif`. There are certain conditional constants defined in SceSysStateMgr. A table of known conditionals is below.

Name	Description
MANUFACTURING_MODE	Unknown. Depends on some condition set on boot by some bootloader
EXTERNAL_BOOT_MODE	Unknown. Depends on some condition set on boot by some bootloader
UPDATE_MODE	Set by SceSyscon when an update is about to be performed.
USB_ENUM_WAKEUP	Unknown. Could be CMA connection while device is turned off.
KERMIT_REV_ES1_X	Unknown. Likely PSP Emulator related.
KERMIT_REV_ES2_X	Unknown. Likely PSP Emulator related.
KERMIT_REV_ES3_X	Unknown. Likely PSP Emulator related.
KERMIT_REV_ES4_X	Unknown. Likely PSP Emulator related.
UD0_EXIST	Does the ud0: Partition exist?
DEMO_MODE	Is the Vita a IDU flagged?
SAFE_MODE	Some flag from boot indicates device should enter safe mode.
DEVELOPMENT_MODE	SceSblACMgr is called to check if device is a PDEL unit.

Example:

```
if SAFE_MODE
spawn   os0:ue/safemode.self
end
endif
```

Load

`load path` will load the kernel module at `path`.

`tload path` possibly stands for "test load." Possibly used in development units to load to module to dedicated devkit RAM.

Example:

```
load    os0:kd/ngs.skprx
```

Spawn

`spawn path` will spawn an app and continue processing the script in the background.

`spawnwait path` will spawn an app and wait for it to exit before continuing processing the script.

`appspawn path param` is used to spawn `vs0:vsh/shell/shell.self`. Specify a paramater to pass.

Known param constants:

Name	Description
SHELL_BUDGET_ID	Unknown.

Example:

```
if UPDATE_MODE
if UD0_EXIST
spawn ud0:PSP2UPDATE/psp2swu.self
else
spawn ur0:PSP2UPDATE/psp2swu.self
endif
end
endif
```

End

`end` will end script processing

Boot Debug Checkpoint Codes

During the boot sequence, the various bootloaders will update a GPIO register specifying the progress into boot. This can be used to debug where in the boot process something fails.

GPIO

The GPIO registers are registered at `0xE20A000C` (turn off bits) and `0xE20A0008` (turn on bits). On PDEL units, this maps to the LED lights.

Known Codes

Code	Location	Description
72	?	?
84	?	?
85	?	?
86	?	?
96	?	?
129	Secure Kernel Loader	Core 0 (secure world) pre-init complete
130	Secure Kernel Loader	Secure world interrupts registered (?)
131	Secure Kernel Loader	Serial console ready, boot message printed
132	Secure Kernel Loader	Some device init
133	Secure Kernel Loader	Some co-processor init. Starting point for other cores.
134	Secure Kernel Loader	MMU enabled, VBAR/MVBAR set up
135	Secure Kernel Loader	Nothing since 134
136	Secure Kernel Loader	Boot setup complete, secure kernel loading begin
137	Secure Kernel Loader	Secure kernel loaded. About to load NS KBL at `0x51000000`
138	Secure Kernel Loader	Secure kernel loaded. About to resume context at `0x1F000000`. Or undefined instruction exception.
139	Secure Kernel Loader	SVC exception (should not happen, error)
140	Secure Kernel Loader	Prefetch abort exception
141	Secure Kernel Loader	Data abort exception
142	Secure Kernel Loader	IRQ exception (should not happen, error)
143	Secure Kernel Loader	FIQ exception (should not happen, error)
161	NS Kernel Loader	Core 0 (non-secure world) pre-init complete
162	NS Kernel Loader	Some interrupts registered (?)
163	NS Kernel Loader	Serial console ready, boot message printed (if enabled)
164	NS Kernel Loader	Some buffer is initialized to device addresses
165	NS Kernel Loader	Some co-processor init. Starting point for other cores.
166	NS Kernel Loader	MMU enabled, VBAR set up
167	NS Kernel Loader	Nothing since 166
168	NS Kernel Loader	Boot setup complete, NS kernel loading begin

169	NS Kernel Loader	Kernel pre-init (setup stacks, interrupts, etc) done. Right before first external loading.
170	NS Kernel Loader	Undefined instruction exception
171	NS Kernel Loader	SVC exception (should not happen, error)
172	NS Kernel Loader	Prefetch abort exception
173	NS Kernel Loader	Data abort exception
174	NS Kernel Loader	IRQ exception (should not happen, error)
175	NS Kernel Loader	FIQ exception (should not happen, error)

Suspend and Resume

Upon suspension, context is written to memory and a syscon command is issued to save the context pointer as well as other information (for example, if it should restart into update mode). When resuming, the boot process is the same as cold boot up until the secure kernel loader. After secure kernel loads, instead of decompressing and jumping to the non-secure kernel loader, it restores the saved context and returns to the kernel resume code.

Categories: Startup | Loaders | Kernel
