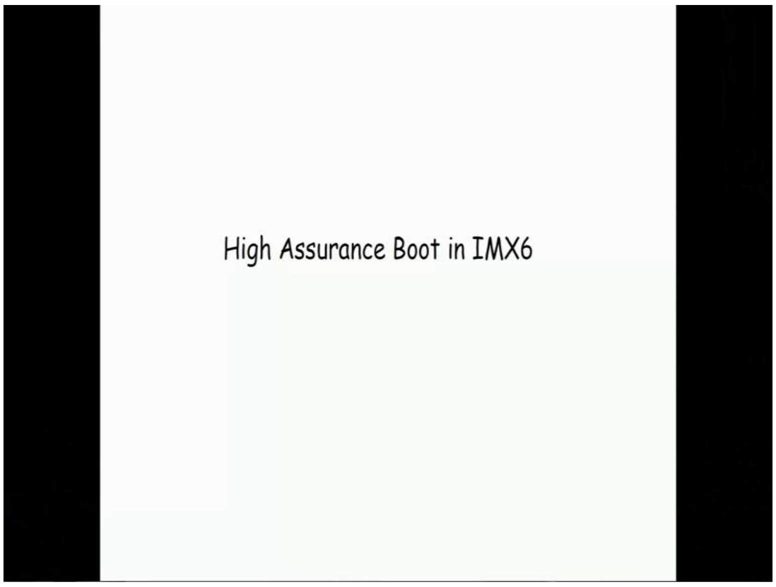
Information Security - II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 38
High Assurance Boot in IMX6

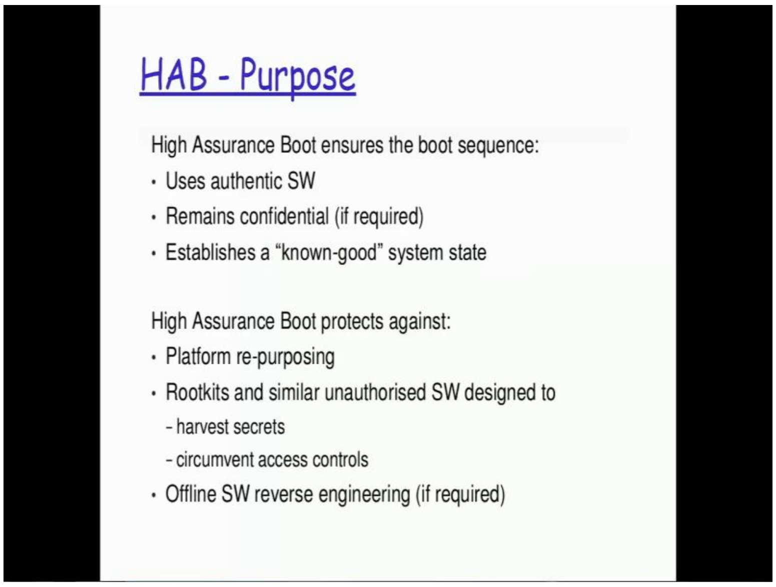
(Refer Slide Time: 00:09)



High Assurance Boot in IMX6

Now, coming into the High Assurance Boot.

(Refer Slide Time: 00:11)



HAB - Purpose

High Assurance Boot ensures the boot sequence:

- Uses authentic SW
- Remains confidential (if required)
- Establishes a "known-good" system state

High Assurance Boot protects against:

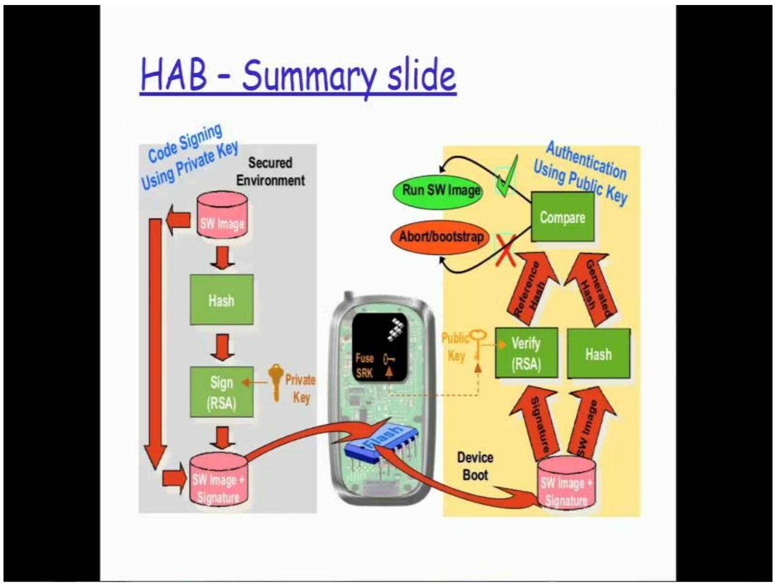
- Platform re-purposing
- Rootkits and similar unauthorised SW designed to
  - harvest secrets
  - circumvent access controls
- Offline SW reverse engineering (if required)

We will have to understand that high assurance boot is basically a boot sequence to ensure that the authentic software is being used. Now, what do we mean by authentic software? When I ship out a device from the factory, I have actually tested particular version software, a particular image to be used in that particular device. As long as my customer is actually making use of that same image, I am perfectly ok because that image is fully tested, found to be secure. So, when a customer using that image I am ok with it but there is also a possibility that a customer could have got compromised and somebody could have actually made use of, dumped another image or trying to give it a new image which is not matching with whatever I had shift the device out with. I have to catch that kind of situations and refuse to boot up in those kinds of scenarios. So, that is basically what is referred to as high assurance boot, establishes a known good system state.

High assurance boot protects against platform re-purposing. Platform re-purposing here in the sense, I have actually shift the device in the particular platform in the particular image, somebody is actually trying to re-platform that in the sense they are trying to modify that particular image in an unauthenticated manner, which I do not want to accept and allow booting up in that kind of a scenario. So, Rootkits and similar unauthorized software designed to harvest secrets, circumvent access controls. This is an example that

I gave you, somebody puts an modified executable, I run the executable but that executable is compromised. It is doing the job what it is supposed to do but in addition that it is also doing some other security violated jobs also. Offline software reverse engineering, all these to be prevented.

(Refer Slide Time: 02:21)



Now, this is just a summary slide of how the whole thing is actually working. What I will actually do is, this is my processor. I have mentioned that there is something called as a fuse. I am going to dump some value inside that fuse; this is a onetime activity that is why we call it as OTP, one time programmable. So, I am going to fuse some value inside. I will tell you what the value is. Now, on this side, this is basically my image building and image generation part. This is my software image.

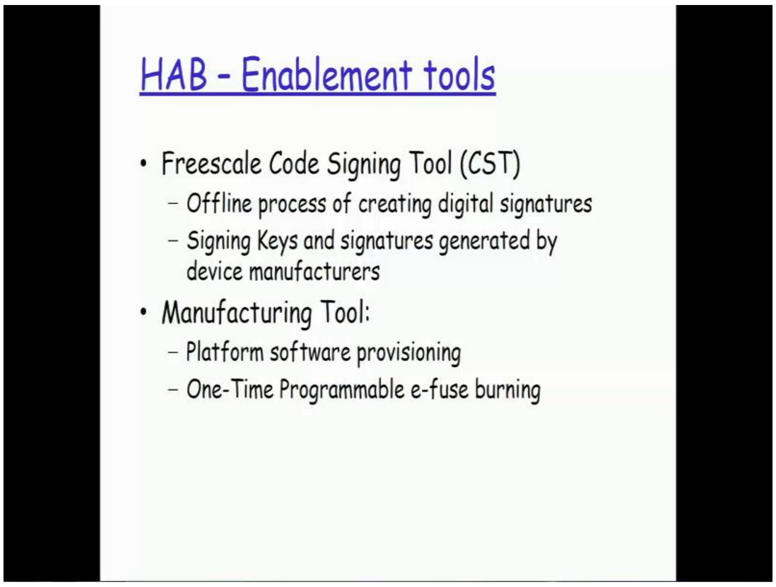
Let us say, this is a u-boot, what is the u-boot? Have you heard of u-boot? It is a boot loader. We told from boot ROM, the next thing will be the boot loader. The boot loader will be to the OS, OS to the root file system. So, the boot loader, let say that this is one image that I actually have, which I want protect because the HAB needs to start protecting right from the boot loader because a boot loader image also I can download it over any of the peripheral. So, I need a protection on that particular boot loader image. This is basically the software image that I want to protect. Any image that I want to protect, I generate a hash value for it, hash we already know, that is the reason why I talked about all that in the morning before lunch. This hash value I basically sign with RSA algorithm. So, I use a private key and then sign, this sign value with the image I basically generate a signature and this complete portion is flashed. When I say flashed, we essentially mean that it is now dumped into my on device storage area. So, it can be a NOR flash, NAN flash, EMMC, whatever.

So, I have different kinds of media that is there, any of the stuff that this particular device is supporting I can dump it on that. Now, this is actually available here, which is basically a signed software image. I am powering it up. The whole concept of security of this high assurance boot, what we are talking of is I want to verify on a power up that whatever I have actually given is the one that is the actually right now trying to get booted up with, that is a requirement. So, now, when I power up I basically have the image with the signature concatenated together, now with the signature, the image I basically split it up into two parts. On the signature, which is basically encrypted with my private key I have to use the public key. So, I use the public key from the header portion that is actually got added here but there is a possibility that this also would have got tempered with by the person who is trying to spoof.

Now, how do I verify that? I take the public key out of that and I told I would have actually got some value fused here, which is not readable by somebody outside, that value is basically a hash value, hash value of my public key that is there as part of this generation. Now, when I extract the public key out of that I have the hash value of the public key generated and compare it with the hash value that fused inside. If they are same, I use that public key and decrypt the sign. Now, I get back the hash value that was encrypted as part of the generation. I have the software image; I send the software image to the same hash algorithm. I got a generated hash; compare these two hashes only after these two are same. It essentially means that there has been no modification on the software image.

So, nobody could have actually modified the public key because if they have modified the public key and then put it, this verification would have failed, the verification or hash of the public key with whatever has been fused inside the processor that verification would have failed. Once that verification is successful, I use that public key and compare it with the hash value that has actually been generated out of this, do this comparison. Only if both of them same, I load the software image and then run it. If they are not same, I basically do abort. This entire process is high assurance boot. This is just a summary, we will actually take a little detail look at this subsequently.

(Refer Slide Time: 08:02)



HAB - Enablement tools

- Freescale Code Signing Tool (CST)
  - Offline process of creating digital signatures
  - Signing Keys and signatures generated by device manufacturers
- Manufacturing Tool:
  - Platform software provisioning
  - One-Time Programmable e-fuse burning

So, what are these enablement tools that are acquired? I need something like code signing tool, this code signing tool is basically going to generate my digital signature and then going to do the signing of the keys and signatures generated by device manufacture. So, this entire part I use code signing tool that freescale is actually providing. So, with this part I will be able to do the signing, I will be able to build up this concatenated image all in one shot. I just run one command, I generate the keys, I edit some files. After these files are edited, I run a command as part of this code signing tool, this entire concatenated image with my original executable contents with the signature all will be generated and that I basically flash it on to my device.

Now, what do I use for flashing it on to my device that is use for my manufacturing tool? This manufacturing tool will basically flash the signed image that my CST tool has

Student: Excuse me Sir.

Yes.

Student: (Refer Time: 09:13) Actual I have a doubt on this previous slide.

Ok.

Student: In this, how is it taking place, the software image we are sliding it for the first time, is that image continuing boot loader also?

Ok. See.

Student: (Refer Time: 09:26)

So, there is something called as a chain of trust.

Student: Ok.

In the chain of trust, what we are actually doing here is that as I was telling you the boot sequence earlier, you have the boot ROM code, you do not suspect that. So, you always believe that the boot ROM code, once it is being returned there is nobody else who can basically go on and update that part and that boot ROM code is basically going to make use of a boot loader. So, the first image that will be authenticated by this high assurance boot will be the boot loader image. Once the boot loader image is authenticated like your u-boot, for example because most of these devices today actually use u-boot as a boot loader. Once that boot loader is authenticated by this HAB, the boot loader, u-boot will be made to authenticate my kernel image. As long as the kernel image is actually being authenticated by somebody who has already been authenticated before, we are completely ok with it and this basically what we refer to as a chain of trust. So, as long as a person who is authenticating is already authenticated earlier, if he is basically telling another image that is coming in the line, that yes this particular image is fine, it is not compromised and it is successfully authenticated, we can go ahead and believe, that is the underlying assumption. So, right up to your kernel image you can have, you can use HAB to authenticate it from your boot loader.

Student: Right Sir

So, subsequently for our root file system what we have actually done is, we have done the complete encryption the root file system, so that even if somebody tries to access the content and then take it out, they will require the key and that key is actually stored internally as part of our SNVS area. So, with this approach even though we do not have the root file system authentication to be done directly by HAB, we actually have this via media approach, which is also pretty much equally safe to go ahead and authenticate your root file system also completely.

Student: Sir, (Refer Time: 12:13) Sir.

Yes.

Student: Sir, after that you shown in the next slide for generating (Refer Time: 12:40)

Those tool chains are actually provided by the vendor. So, you are basically referring to this code signing tool in the manufacturing tool.

Student: Yes Sir this slide.

Yes, this one, right now.

Student: Yes.

Yes. So, these two tool chains are actually provided by the vendor not in a source code form but in an executable form.

Student: (Refer Time: 13:15)

No, No.

Student: The dependency on this tool.

No, No, No, the trust chain.

Student: Does the chaining.

No, the trust chain that is actually derived is basically only from the point of view of the boot up sequence.

Student: But my question was that you were finding this code using these two region only.

Correct.

Student: If there is some malfunction happens at the time of signing during the entire procedure which you have shown in the previous slide will not that have some other vulnerability because that depends on the signing process itself.

Yes. So, one is, the freescale is actually providing you this executable in an executable form and secondly, we can actually go ahead and request them to have the source code available to us, so that we can do a formal verification of the code.

Student: (Refer Time: 14:26)

Yes, that is right.

Student: Verification of the tool

That is right. So, right now we have actually got a comfort completely on how this whole thing is working. We have implemented it and we have been able to see it happening on our own device also. So, we are basically going to ask freescale to provide us the codes, so that the code of the CST as well as the manufacturing tool can be done of formal verification and verify it that there are no malignancies inside that as well.

Student: And that your estimator would the code size and the properties of the code give you in the current technology for kernel.

Yes, actually the size of that code as I understand is not very big. I guess it should not go beyond very few thousand lines of codes. So, I presume that with the current formal verification methods that we have, we should be able to verify that it is not like really a very huge thing like around tenth of the ten thousand lines of code or whatever it is. So, it should be definitely verifiable.

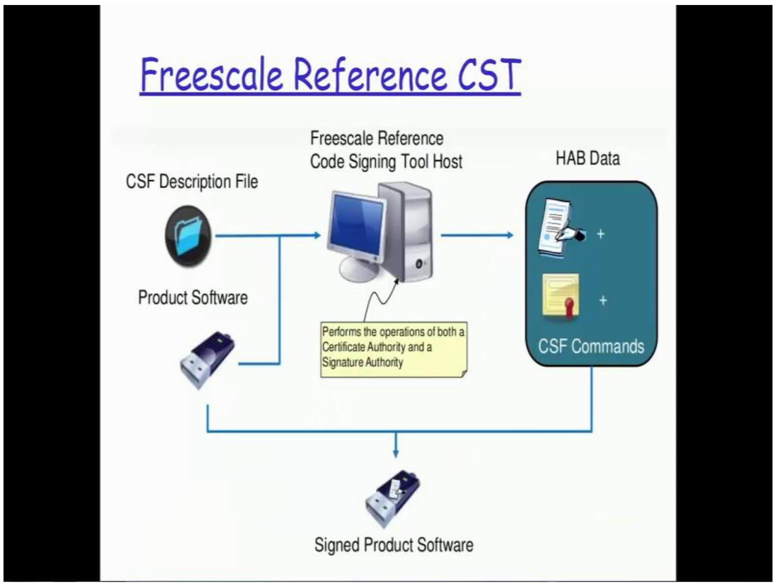
Student: Yes Sir.

So, that is definitely in our plan currently.

Student: Ok Sir. Thanks Sir.

Yes. So, basically the manufacturing tool is sort of in charge of trying to go ahead and dump the signed image on to the flash and also to basically go ahead and write the fuse values inside the appropriate register location because remember the fuse values are only modifiable that is writable only once. Once, you write that values it is no longer no longer programmable or modifiable, that also will be actually getting done as part of the manufacturing tool itself.

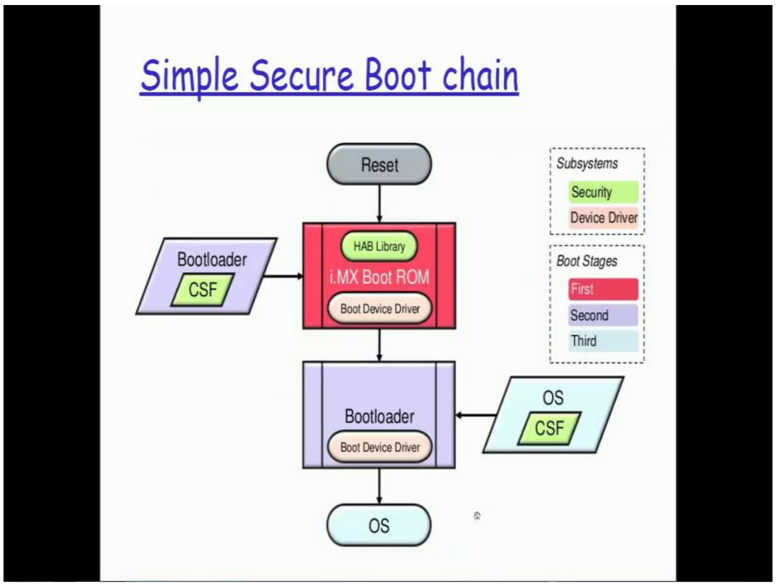
(Refer Slide Time: 16:29)


Freescale Reference CST

CSF Description File
Freescale Reference Code Signing Tool Host
HAB Data
Product Software
Performs the operations of both a Certificate Authority and a Signature Authority
+
+
CSF Commands
Signed Product Software

This slide basically talks about, how the CST tools really work. It basically takes the file called as a CSF file, command sequence file along with the product software. On the code signing tool host, these two files are basically given as input to the CST application. It generates the HAB data, which is basically the signed image with the CSF header what we call as an IVT table, all available as part of the signed image. Then the signed product software is basically what is used by manufacturing tool to dump it on to the storage media on the device.

(Refer Slide Time: 17:09)

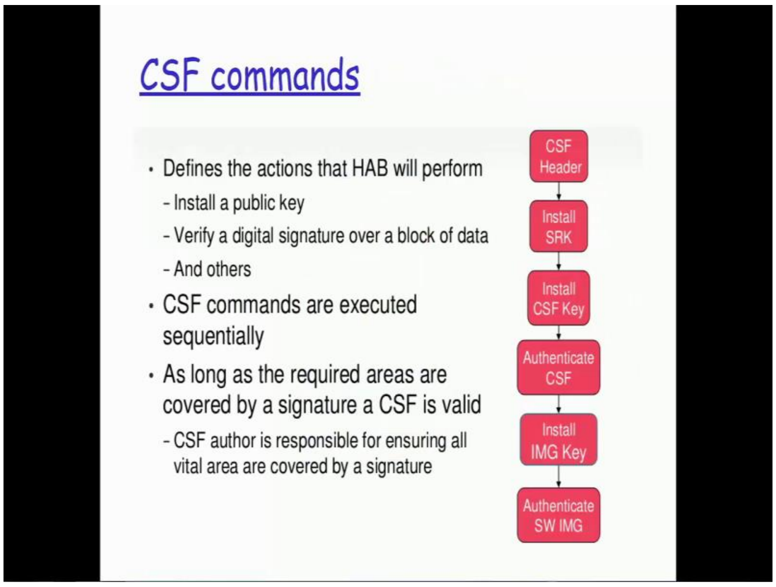


Simple Secure Boot chain

The very flow chart mechanism of how this whole thing is actually working. On a power on reset, the HAB library, if it is configured is going to come in to play, the boot loader image found whatever media that I have basically configured my boot mode register is excepted to come in and that will also have the CSF information with all the details on which part of the entire file is my image file, which part of it is basically pertaining into my signed data and all that stuff and that is actually going to be used by my boot ROM code with the boot device driver inside.

Once, that is authenticated by the HAB library to confirm to this particular checks, my boot loader will be getting executed. So, this boot loader will now take up the OS part, which is the next level of authentication that is the third stage of boot stage. That is basically what I was telling you that in this chain of trust we expect and assume that the boot ROM code is pakka. So, there has been no compromise on that part because that is not possible outside of factory to be modifying that. On the boot ROM code is actually going to looking and validating the boot loader. Once, a boot loader is validated, the boot loader is now going to take look at the OS and validate with the corresponding CSF file. Once that is also validated, now the OS is going to get loaded and then gets start to get executed right, so that basically the chain that it will typically happen.

(Refer Slide Time: 18:52)



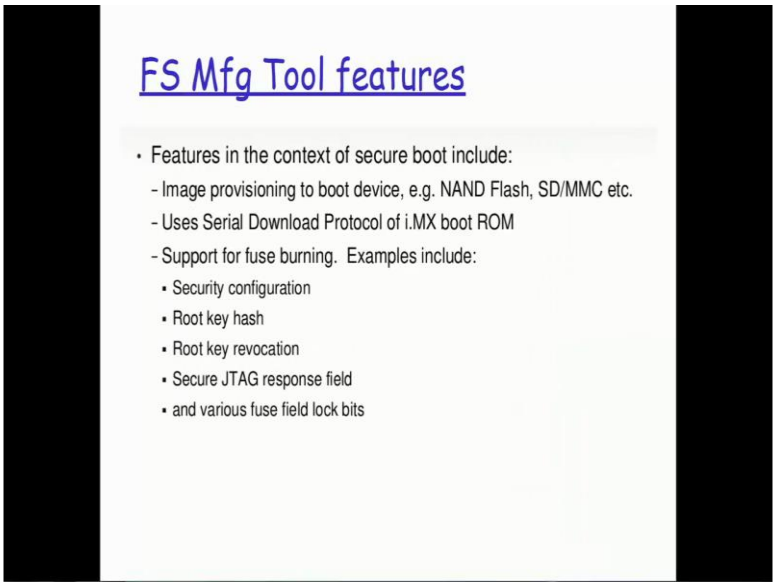
CSF commands

- Defines the actions that HAB will perform
  - Install a public key
  - Verify a digital signature over a block of data
  - And others
- CSF commands are executed sequentially
- As long as the required areas are covered by a signature a CSF is valid
  - CSF author is responsible for ensuring all vital area are covered by a signature

CSF Header
Install SRK
Install CSF Key
Authenticate CSF
Install IMG Key
Authenticate SW IMG

So, this is just a slide on how the whole thing is actually working with the CSF. There are different CSF commands that are actually available, which all goes in as part of my header information inside my signed image.

(Refer Slide Time: 19:08)

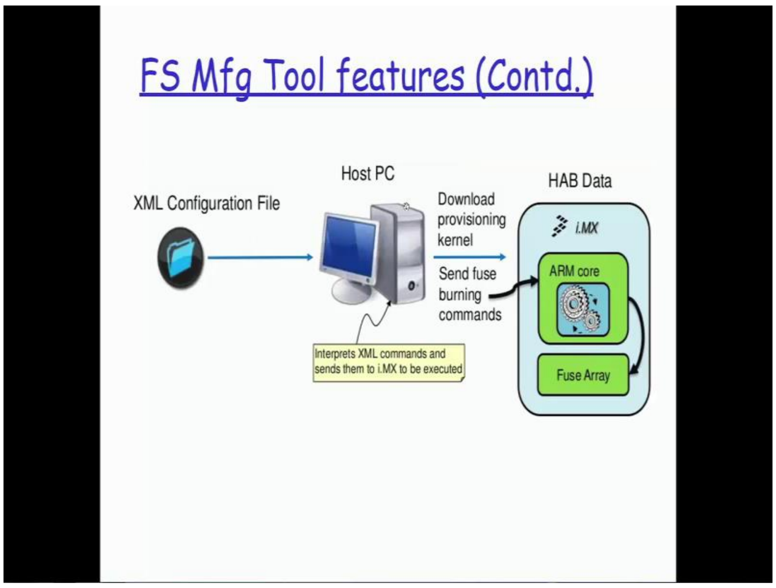


FS Mfg Tool features

- Features in the context of secure boot include:
  - Image provisioning to boot device, e.g. NAND Flash, SD/MMC etc.
  - Uses Serial Download Protocol of i.MX boot ROM
  - Support for fuse burning. Examples include:
    - Security configuration
    - Root key hash
    - Root key revocation
    - Secure JTAG response field
    - and various fuse field lock bits

Manufacturing tool features as I told you image provisioning to boot device. It uses serial download protocol of iMX boot ROM support for fuse burning. These are the things that are actually done by the manufacturing tool. So, basically updates the image into the corresponding storage media on the device and then fuses the value that has to go into the OTP register of my processor.

(Refer Slide Time: 19:30)

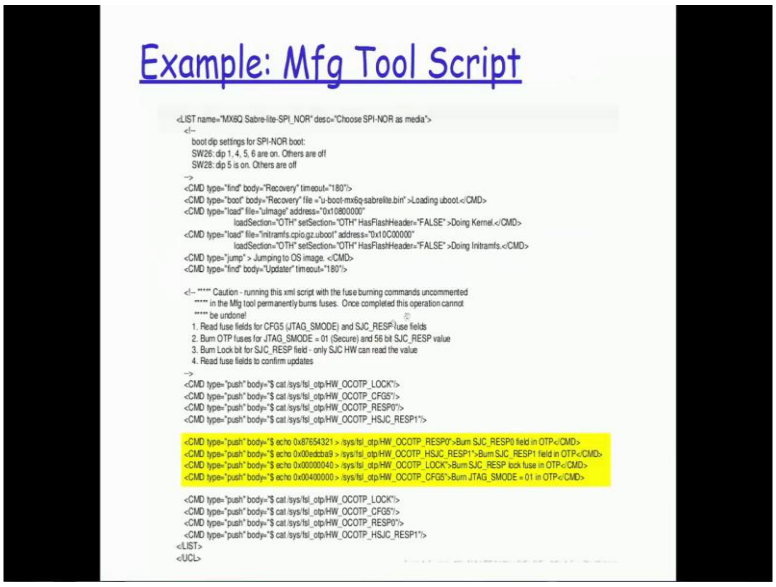


FS Mfg Tool features (Contd.)

XML Configuration File
Host PC
Download provisioning kernel
Send fuse burning commands
Interprets XML commands and sends them to i.MX to be executed
HAB Data
i.MX
ARM core
Fuse Array

So, the XML configuration file is used as an input by my manufacturing tool and using the serial interface into my iMX6 device. The whole thing is actually dumped on to my device. So, the XML file, sample XML file will look something like this.

(Refer Slide Time: 19:53)

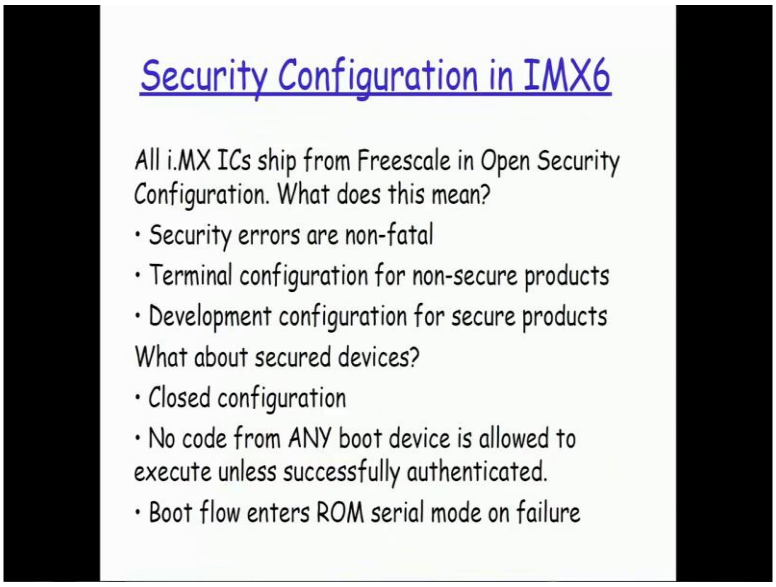


Example: Mfg Tool Script

```
<LIST name="MX6Q Sabre-lite-SPI_NOR" desc="Choose SPI-NOR as media">
  <!--
    boot dip settings for SPI-NOR boot:
    SW26: dip 1, 4, 5, 6 are on. Others are off
    SW28: dip 5 is on. Others are off
  -->
  <CMD type="find" body="Recovery" timeout="180"/>
  <CMD type="boot" body="Recovery" file ="u-boot-mx6q-sabrelite.bin" >Loading uboot.</CMD>
  <CMD type="load" file="uImage" address="0x10800000"
                loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Kernel.</CMD>
  <CMD type="load" file="initramfs.cpio.gz.uboot" address="0x10C00000"
                loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Initramfs.</CMD>
  <CMD type="jump" > Jumping to OS image. </CMD>
  <CMD type="find" body="Updater" timeout="180"/>

  <!-- **** Caution - running this xml script with the fuse burning commands uncommented
     **** in the Mfg tool permanently burns fuses.  Once completed this operation cannot
     **** be undone!
    1. Read fuse fields for CFG5 (JTAG_SMODE) and SJC_RESP fuse fields
    2. Burn OTP fuses for JTAG_SMODE = 01 (Secure) and 56 bit SJC_RESP value
    3. Burn Lock bit for SJC_RESP field - only SJC HW can read the value
    4. Read fuse fields to confirm updates
  -->
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_LOCK"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_CFG5"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_RESP0"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_HSJC_RESP1"/>

  <CMD type="push" body="$ echo 0x87654321 > /sys/fsl_otp/HW_OCOTP_RESP0">Burn SJC_RESP0 field in OTP</CMD>
  <CMD type="push" body="$ echo 0x00edcba9 > /sys/fsl_otp/HW_OCOTP_HSJC_RESP1">Burn SJC_RESP1 field in OTP</CMD>
  <CMD type="push" body="$ echo 0x00000040 > /sys/fsl_otp/HW_OCOTP_LOCK">Burn SJC_RESP lock fuse in OTP</CMD>
  <CMD type="push" body="$ echo 0x00400000 > /sys/fsl_otp/HW_OCOTP_CFG5">Burn JTAG_SMODE = 01 in OTP</CMD>

  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_LOCK"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_CFG5"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_RESP0"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_HSJC_RESP1"/>
</LIST>
</UCL>
```

For example, look at it here, these are the commands that are used, the XML commands they are actually used to fuse the values in the corresponding locations. If, I basically write it into this particular file, this particular file is memory mapped into my corresponding OTP register. So, if I echo a value into this file that will go and get burned into my particular location and then once I do the complete verification of this entire fuse values, I basically change the configuration into a closed configuration.

(Refer Slide Time: 20:36)



Security Configuration in IMX6

All i.MX ICs ship from Freescale in Open Security Configuration. What does this mean?

- Security errors are non-fatal
- Terminal configuration for non-secure products
- Development configuration for secure products

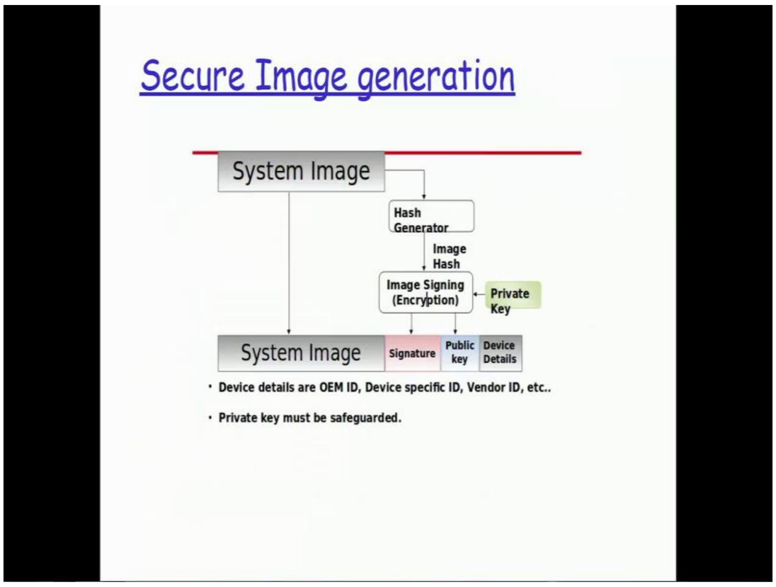
What about secured devices?

- Closed configuration
- No code from ANY boot device is allowed to execute unless successfully authenticated.
- Boot flow enters ROM serial mode on failure

So, after which it will work only when I have the authenticated image alone loaded and it will fail when I do not have the authenticated information. All iMX ICs ship from freescale in open security configuration. When I say open security, security errors are non fatal. So, even if I have in open configuration, non authenticated image it will still go through fine without any issues. Terminal configuration for non-secure products when I really have the device required to be working in a non-secure manner, this will be my final configuration which I keep it as a open configuration and for secure products this development configuration which I used purely for development and before I ship it, if the product is to be a secure product I go ahead and do a closed configuration.

What is this closed configuration, no code from any boot device is allowed to execute unless successfully authenticated and boot flow enters a ROM serial mode on failure waiting for the authenticated image to be coming in over that serial mode. Suppose, if the image that I are actually stored internally has not been successfully authenticated in a closed configuration, the boot flow will enter the ROM serial mode to look at to see if I can have an image coming into that, which could be successfully authenticated. So, if there is no image coming in on that also which could be successfully authenticated then it will just refuse to boot as if the device itself is completely failed.

(Refer Slide Time: 22:17)

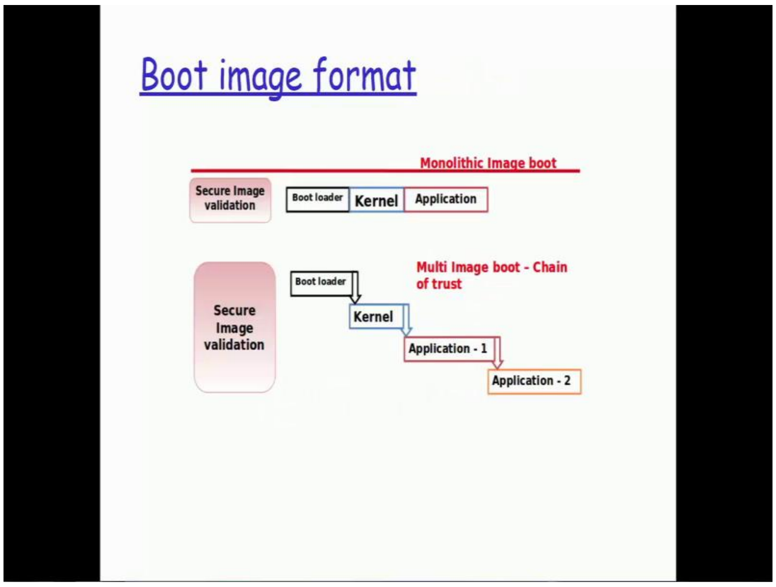


Secure Image generation

- Device details are OEM ID, Device specific ID, Vendor ID, etc..
- Private key must be safeguarded.

The way, the secure image is actually generated we just gone through this part right now.

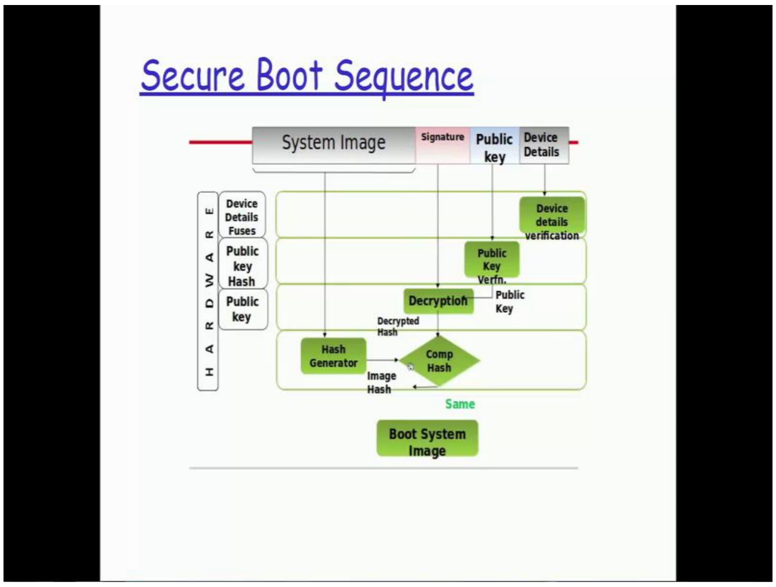
(Refer Slide Time: 22:22)



Boot image format

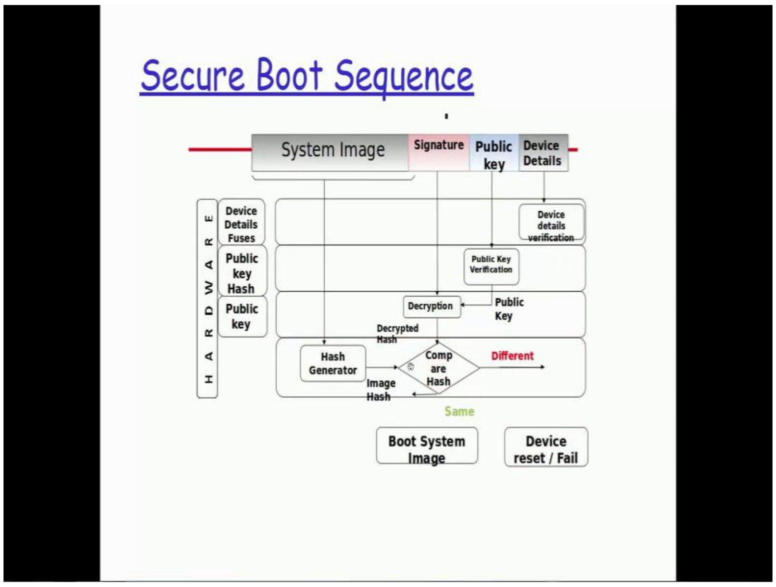
This is basically the chain of trust that I was actually trying to talk. The boot loader getting validated by the ROM code, boot loader then validating the kernel, kernel validating the individual applications and so on. On the iMX6 processor with the high assurance boot all the applications right from the root file system, we do not have the inherent capability to do it with HAB itself. So, we are implementing the encryption part of it here, to have the trust chain implemented.

(Refer Slide Time: 22:55)



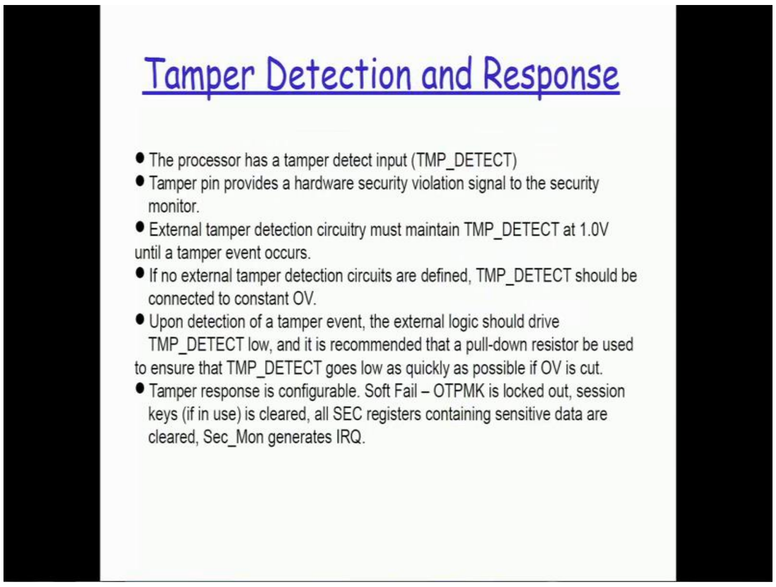
This is basically what actually happens. The device details part of it is also fused. As part of my image, I will have the device details also available. As the first phase, those will be verified. Once, they are successful my public key point of it will be extracted, the public key verification will happen. So, the hash of this public key and what has been fused earlier as a public key hash will be compared. Once, they are compared I know that I have got the correct public key. I will use that public key and then decrypt the hash that has been available for this system image. So, that decrypted hash value and the hash that I have generated right now on the device for my system image portion, pure system image portion that comparison will be made.

(Refer Slide Time: 23:50)



If it is same, the system will be booted. If it is a failure, the device will go with the reset.

(Refer Slide Time: 23:52)

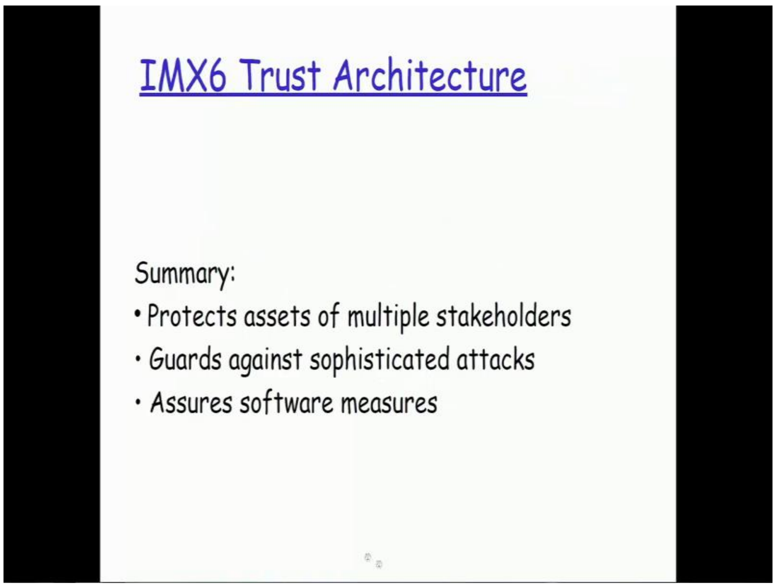


Tamper Detection and Response

- The processor has a tamper detect input (TMP_DETECT)
- Tamper pin provides a hardware security violation signal to the security monitor.
- External tamper detection circuitry must maintain TMP_DETECT at 1.0V until a tamper event occurs.
- If no external tamper detection circuits are defined, TMP_DETECT should be connected to constant OV.
- Upon detection of a tamper event, the external logic should drive TMP_DETECT low, and it is recommended that a pull-down resistor be used to ensure that TMP_DETECT goes low as quickly as possible if OV is cut.
- Tamper response is configurable. Soft Fail – OTPMK is locked out, session keys (if in use) is cleared, all SEC registers containing sensitive data are cleared, Sec_Mon generates IRQ.

The tamper detection part of it, I have a tamper detect pin that is there. So, pin is actually exposed out of my processor. That pin basically will have a one volt continuously coming in until a tamper event occurs. When a tamper event is occurring, the TMP detect pin will be given zero volt and what needs to be done when it detects a voltage difference is actually configurable. So, whether I have to go ahead and delete certain particular memory areas or certain files whatever it is, all these are configurable, which I can just configure and then get those things actioned upon whenever tamper is deducted.

(Refer Slide Time: 24:41)



IMX6 Trust Architecture

Summary:
- Protects assets of multiple stakeholders
- Guards against sophisticated attacks
- Assures software measures

The trust architecture, as the summery protects assets of multiple stakeholders. We saw what those different assets are and for each of those assets, what kind of threats come in, guards against sophisticated attacks with the tamper deduction, you will be able to detect different kinds of attacks physical as well and assures software measures by ensuring that whatever is authenticated image is what is actually getting booted up every time.