Secure Boot Implementation on GNX Pure E600



|  |  |
| --- | --- |
| ***Contributors*** | *Lyubomir Giorgiev*  *Hristo Kolev*  *Yoan-Asen Popov*  *Joep Vinken* |
| ***Assignment*** | *Secure Solution* |
| ***Class*** | *CS4* |
| ***Year*** | *2022-2023* |
| ***Group & Tutor*** | *01-D  Xuemei Pu* |

Contents

[1. Executive Summary 3](#_Toc138004711)

[2. Introduction 3](#_Toc138004712)

[3. Secure Boot Overview 3](#_Toc138004713)

[4. Enabling Secure Boot in Firmware 4](#_Toc138004714)

[4.1 Firmware Compatibility and Support 4](#_Toc138004715)

[4.2 Configuration Steps 4](#_Toc138004716)

[5. Generating and Safeguarding Cryptographic Keys 5](#_Toc138004717)

[5.1 Key Generation 5](#_Toc138004718)

[5.2 Key Storage and Protection 8](#_Toc138004719)

[6 Signing Firmware Components 9](#_Toc138004720)

[6.1 Bootloader 9](#_Toc138004721)

[6.4 Bootloader Security and signing 10](#_Toc138004722)

[6.2 Kernel Signing 13](#_Toc138004723)

[7 Boot Process Validation 15](#_Toc138004724)

[7.1 Signature Verification 15](#_Toc138004725)

[7.2 Integrity Checking 16](#_Toc138004726)

[7.3 Variations in Validation Mechanisms 17](#_Toc138004727)

[8 Plan B: Backup Measures for Secure Boot 18](#_Toc138004728)

[8.1 Corrupted Bootloader Image 18](#_Toc138004729)

[8.2 Invalid or Expired Digital Signature 18](#_Toc138004730)

[8.3 Incompatible or Tampered Kernel Image 19](#_Toc138004731)

[8.4 Rollback Attack 19](#_Toc138004732)

[8.5 Summary 20](#_Toc138004733)

[9 Research Question 21](#_Toc138004734)

[10 Conclusion 22](#_Toc138004735)

[References 23](#_Toc138004736)

# 1. Executive Summary

This report presents a comprehensive manual for implementing Secure Boot on the GNX Pure E600, aiming to enhance its security and safeguard against unauthorized firmware modifications. It covers the technical aspects of enabling Secure Boot within the firmware, including key generation and protection, firmware component signing, and boot process validation. The report provides detailed explanations and practical code snippets to facilitate the implementation process.

This assignment pertains to the Secure Solution project for the current semester. It serves as the second part (2/2) of the group project, which involves the following team members:

* Lyubomir Georgiev (infra)
* Hriste Kolev (software)
* Yoan-Asen Popov (infra)
* Joep Vinken (infra)

This project primarily focuses on the defensive aspects of cybersecurity, in contrast to the first part of the group project that mainly concentrated on the offensive side through pen-testing one of Genexis’ routers.

# 2. Introduction

Modern Wi-Fi routers play a critical role in network infrastructure and require robust security measures to protect against unauthorized access and firmware tampering. Especially with the new law that will categorize these routers as ‘critical infrastructure’.

Secure Boot is a security mechanism that ensures only trusted and signed firmware components are executed during the boot process. This report focuses on implementing Secure Boot on a Linux-based Wi-Fi router, providing in-depth technical insights and practical guidance.

# 3. Secure Boot Overview

Secure Boot is a fundamental security feature that verifies the integrity and authenticity of firmware components before allowing their execution during the boot process. It establishes a trusted boot chain by enforcing the validation of digital signatures for the bootloader, kernel, and other critical components. By ensuring that only trusted code is executed, Secure Boot protects the router's integrity and enhances its overall security.

# 4. Enabling Secure Boot in Firmware

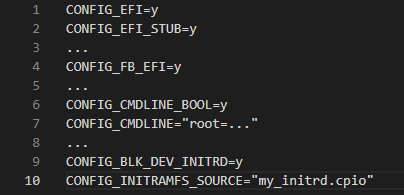
## 4.1 Firmware Compatibility and Support

Secure Boot is a boot integrity feature that is part of the Unified Extensible Firmware Interface (UEFI) industry standard. The extent to which Secure Boot validates the boot process varies based on platform and boot configuration. In general, most enterprise UEFI implementations provide the following options:

* **Thorough or Full Boot** provides the maximum amount of protection by using Secure Boot throughout the boot process. Integrity, signature, and hash checks are performed. All authorized firmware binaries are executed. Alerts may be generated for hardware changes, chassis intrusions, and component states. Thorough Boot prioritizes security over speed. Boot time takes the longest in Thorough Boot.
* **Fast Boot or Minimal Boot** minimizes boot time by skipping numerous checks, which may or may not include Secure Boot checks. Boot speed is prioritized over some security features and/or additional features and peripheral support at boot time. Malware like LoJax can slip by on some systems (Schlej 2018).
* **Automatic Boot** attempts to detect when changes have occurred to the early stages of National Security Agency | Cybersecurity Technical Report UEFI Secure Boot Customization U/OO/168873-20 | PP-23-0464 | Mar 2023 Ver. 1.2 5 UEFI boot. Automatic Boot invokes Fast/Minimal Boot when no changes are detected. Thorough/Full Boot is invoked once after each significant change is detected. Changing firmware, changing hardware, bootloader updates, or toggling options in UEFI configuration may be sufficient to trigger Thorough/Full Mode on the next boot

## 4.2 Configuration Steps

When building the kernal image there are a couple of options that needs to be set in the kernel configuration:



The two variables enable EFI mode, which tells the kernel to build itself as a EFI binary that can be run directly from the UEFI bios. This removes the need of bootloader in the system, the UEFI bios just boots the kernel, no addittional steps are needed.

As no bootloader is going to be involved in the boot process, you need to ensure that the kernel knows where the root partition is, what init is going to be run, and anything else that the bootloader normally passes to the kernel image. The option listed above, CONFIG\_CMDLINE should be set to whatever you want the kernel to use as the command line.

Also, as we don’t have an initrd passed by the bootloader to the kernel, if you want to use one, you need to build it into the kernel itself. The option CONFIG\_INITRAMFS\_SOURCE should be set to your pre-built cpio initramfs image you wish to use.

After setting these options, the kernel can be build and installed on the boot partition. In order to be booted directly by EUFI, it needs to be placed on the following path: /boot/EFI/boot/bootx64.efi. This way the code will be treated as the default bootloader.

To test the kernel image that you have installed on the boot partition, we need to go to the BIOS. Find the location where the UEFI boot mode is indicated in the BIOS settings, then disable the "Secure Boot" option there. When you reboot after saving the option, the BIOS should automatically boot the kernel from boot/EFI/boot/bootx64.efi.

Only after seuccessful boot of the kernel through the BIOS, in insecure mode we can create the neccessary secure boot keys and sign the kernel.

# 5. Generating and Safeguarding Cryptographic Keys

## 5.1 Key Generation

In cryptography, a key is a string of characters used within an encryption algorithm for altering data so that it appears random. Like a physical key, it locks (encrypts) data so that only someone with the right key can unlock (decrypt) it.

The choice of cryptographic key for the bootloader depends on several factors, including the specific security requirements, the cryptographic algorithms supported by the bootloader, and industry best practices. In the modern cryptography there are two ways of encryption mechanism for data security.

* Symmetric key encryption
* Asymmetric key encryption

**Symmetric key encryption** is a simple method of encryption that uses only one secret key, known as the "Symmetric Key", which remains to both parties. The key is used by the sender prior to sending the message, and on the receiver side, it is used to decipher the encoded message.

Symmetric encryption algorithms like **Advanced Encryption Standard (AES**) are commonly used for encrypting bootloader communications or encrypting sensitive bootloader data. A secure symmetric encryption key is generated and shared among authorized entities.

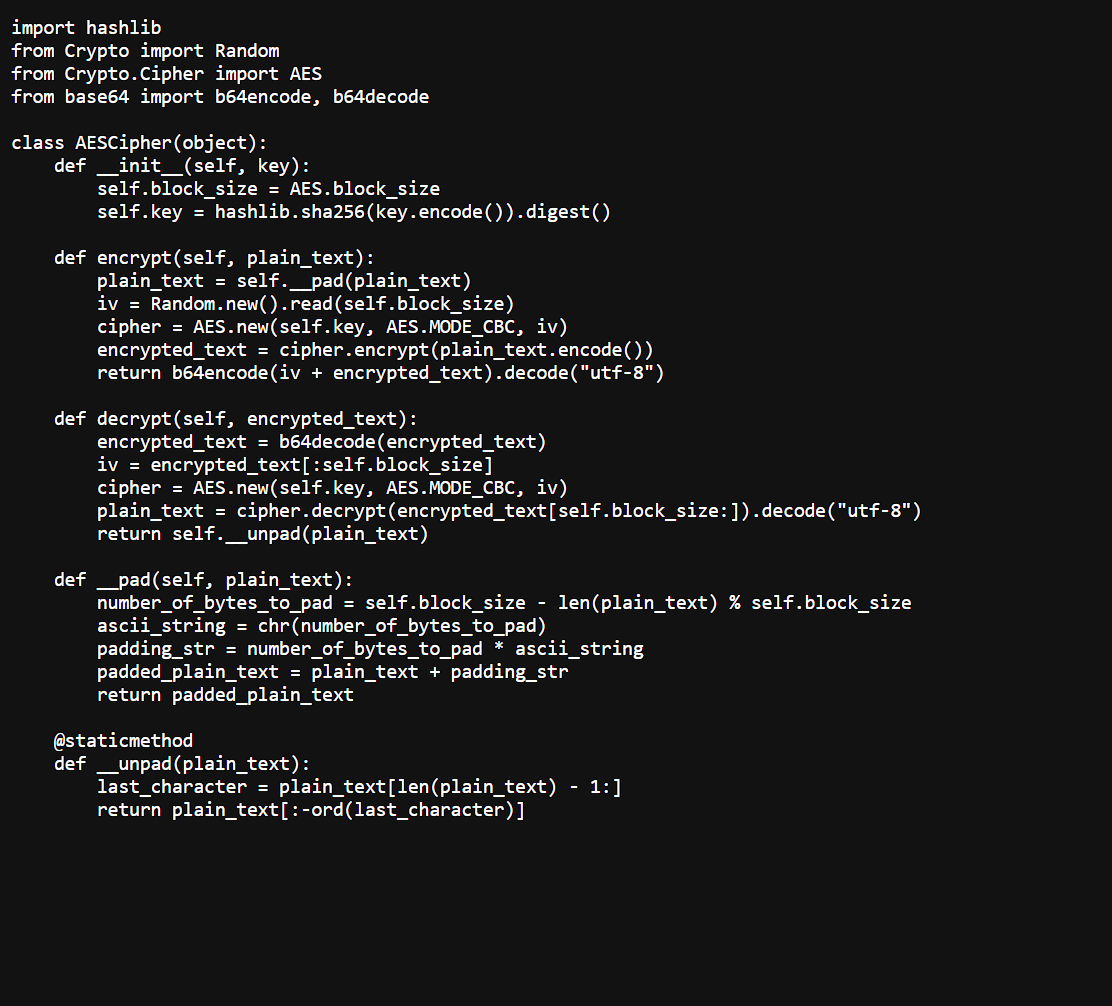
**Asymmetric Encryption** is a new and sophisticated encryption technique that uses two keys: a Public Key and a Private Key. The Public Key is accessible to all who want to send an encrypted message, while the Private Key is kept secure by the owner of the public key or the one who is encrypting. Encryption of information is done through a public key first, with the help of a particular algorithm, then the receiver uses the private key to decrypt that information. The same algorithm is used in both encodings and decoding.

**RSA (Rivest-Shamir-Adleman)** is a widely used asymmetric encryption algorithm. RSA key pairs, consisting of a public key and a private key, can be utilized for bootloader authentication and secure communication.

**Elliptic Curve Cryptography (ECC) Keys:** ECC is an alternative to RSA that offers strong security with shorter key lengths, making it more efficient for resource-constrained devices like routers. ECC keys, such as those based on curves like secp256r1 or secp384r1, can be employed for bootloader security.

### 5.1.1 Example

**Symmetric Key Encryption example:**



Example code snippet of AES in Python

The first step we need is to install the pycrypto library.

The constructor receives a key of any length and generates a 256 bit hash from it. This hash is a unique identifier of a given length, 32 characters in this case. The block\_size is set to 128, which is the block size of AES. This enables the constructor to generate a unique 256 bit key for a cipher.

**The “\_\_pad” method** receives the “plain\_text” to be encrypted and adds a number bytes for the text to be a multiple of 128 bits. This number is stored in “number\_of\_bytes\_to\_pad”. Then in “ascii\_string” we generate our padding character, and “padding\_str” will contain that character times “number\_of\_bytes\_to\_pad”. So we only have to add “padding\_str” at the end of our “plain\_text” so that it is now a multiple of 128 bits.

In an opposite manner, **“\_\_unpad” method** will receive the decrypted text, also known as “plain\_text” and will remove all the extra added characters in the “\_\_pad” method. For that we first must identify the last character and store in “bytes\_to\_remove” how many bytes we need to trim of the end of “plain\_text” in order to unpad it.

**The “encrypt” method** receives the “plain\_text” to be encrypted. First we pad that “plain\_text” in order to be able to encrypt it. After we generate a new random “iv” with the size of an AES block, 128bits. We now create our AES cipher with “AES.new” with our key, in mode CBC and with our just generated “iv”. We now invoke the encrypt function of our “cipher”, passing it our “plain\_text” converted to bits. The encrypted output is then placed after our “iv” and converted back from bits to readable characters.

In order to **decrypt**, we must backtrack all the steps done in the encrypt method. First we convert our “encrypted\_text” to bits and extract the “iv”, which will be the first 128 bits of our “encrypted\_text”. Much like before, we now create a new AES cipher with our key, in mode CBC and with the extracted “iv”. We now invoke the decrypt method of our “cipher” and convert it to text from bits. We remove the padding with “\_\_unpad” and that’s it!

## 5.2 Key Storage and Protection

To ensure the safety of cryptographic keys used for bootloader security, it is important to store them in a secure manner. Some of the best practices for key storage are using hardware security module(HSM), Trusted Platform Module(TPM), Offline Storage

**Hardware Security Modules (HSM)**: HSMs are dedicated hardware devices specifically designed to securely store and manage cryptographic keys. They provide strong physical and logical protection against key theft or unauthorized access. Storing keys in an HSM is considered one of the most secure options.

**Trusted Platform Module (TPM)**: If your router or device has a TPM chip, you can leverage its capabilities to store cryptographic keys securely. TPMs provide hardware-based security features, including key storage, cryptographic operations, and attestation of the system's integrity.

**Offline Storage**: Store the keys offline, such as on removable storage devices like smart cards, USB tokens, or hardware security dongles. These devices can be physically secured and stored in a safe or a secure location.

# 6 Signing Firmware Components

## 6.1 Bootloader

A bootloader serves several important purposes in the operation of a computer or device. Here are some reasons why a bootloader is used:

1. **Initialization and Hardware Configuration**: The bootloader is responsible for initializing and configuring various hardware components, such as the CPU, memory, and peripherals. It sets up the environment for the operating system to run efficiently and interact with the hardware.
2. **Loading the Operating System**: The primary function of a bootloader is to locate the operating system and load it into the computer's memory. It reads the necessary files from storage, such as the OS kernel or boot files, and transfers control to the operating system, allowing it to take over the execution.
3. **Booting from Different Sources**: Bootloaders provide the flexibility to boot from different sources, such as the computer's hard drive, solid-state drive (SSD), optical disc, USB drive, or network. The bootloader can detect and select the appropriate source based on the configuration or user input.
4. **Multiple Operating Systems or Boot Options**: Bootloaders enable the selection of different operating systems or boot options during startup. This is particularly useful in systems with dual-boot configurations, where multiple operating systems are installed on the same computer. The bootloader presents a menu to choose the desired operating system or boot option.
5. **Firmware Upgrades and Recovery**: Bootloaders are involved in firmware upgrades or recovery processes. They can provide mechanisms to update or recover the firmware of the device, ensuring its stability and security.
6. **Security and Authentication**: Bootloaders can implement security measures such as verifying the integrity and authenticity of the operating system before loading it. This helps prevent unauthorized or tampered software from being executed, enhancing the overall security of the system.

In summary, bootloader is a program or set of instructions that runs when a computer or device is powered on or restarted. It is responsible for initializing the system and loading the operating system (OS) into the computer's memory.

The bootloader is typically stored in the firmware of the device, such as the In this case) router’s BIOS (Basic Input/Output System) or the router's bootloader partition. When the router is turned on, the bootloader is executed, and it performs a series of tasks, including hardware initialization, self-tests, and/or locating the operating system.

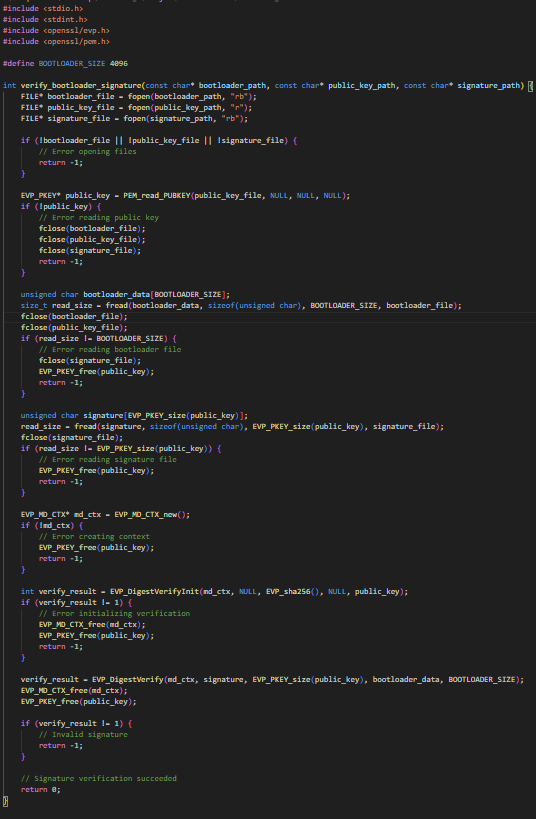
The main function of the bootloader is to locate and load the operating system into the router's memory so that it can start executing. The bootloader usually contains information about the file system and location of the OS kernel or boot files. It loads these files into memory and transfers control to the operating system, allowing it to take over the execution of the computer.

Bootloaders are crucial for the boot process and are specific to each device architecture. They provide an interface between the hardware and the operating system, ensuring a smooth startup process and enabling the loading of the necessary software components to initiate the system.

## 6.4 Bootloader Security and signing

To ensure the integrity and authenticity of the bootloader, it should be securely signed and verified during the boot process. This prevents unauthorized modifications to the bootloader and protects against the execution of malicious code.

In the screenshot below, you can see and exmaple code snippet for the bootloader signature verification:



Example code snipper for Bootloader Signature Verification

### 6.4.1 Code explained

As we don’t have access to the firmware of the router, nor time to implement actual secure boot, this code is just a general example.

The above code snippet demonstrates an example implementation of bootloader signature verification using the OpenSSL library in C. It reads the bootloader binary from a specified path, along with the public key and signature files. The bootloader data is compared with the signature using the provided public key, verifying its integrity and authenticity.

This code assumes that the bootloader binary, public key, and signature files are available and correctly formatted. Additionally, it uses the SHA-256 hashing algorithm for verification. Adjustments may be necessary based on the specific bootloader and signature format used in the router implementation.

It is important to note that this is a simplified example, and actual implementation on the router requires additional error handling, file validation, and integration with the router's boot process.

By performing bootloader signature verification, you establish trust in the integrity and authenticity of the bootloader code, mitigating the risk of unauthorized modifications and ensuring a secure boot process.

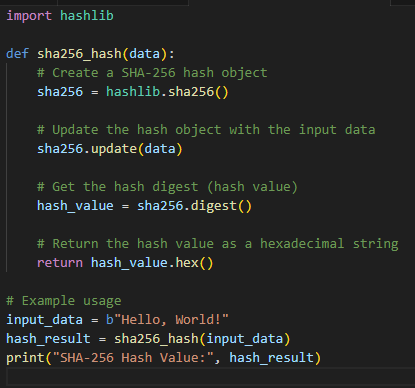
#### 6.4.1.1 SHA hashing

SHA (Secure Hash Algorithm) is a family of cryptographic hash functions that generate a fixed-size output, typically represented as a hash value or hash digest, from input data of any size. These hash functions are widely used in various security applications, including data integrity verification, password storage, digital signatures, and secure boot processes.

SHA-256 is a specific variant of the SHA-2 family and is commonly used due to its strong cryptographic properties and wide support. It produces a 256-bit (32-byte) hash value. The SHA-256 algorithm performs a series of logical operations, such as bitwise operations and modular arithmetic, on the input data to generate the hash value. The primary purposes of SHA-256 hashing are:

1. **Data Integrity**: Hashing allows you to verify the integrity of data. By comparing the hash values of the original and received data, you can determine if the data has been altered during transmission or storage. Even a small change in the input data will result in a significantly different hash value, making it highly improbable for two different inputs to produce the same hash value.
2. **Password Storage**: Hashing is used to securely store passwords. Instead of storing the actual passwords, their hash values are stored. During authentication, the entered password is hashed and compared with the stored hash value. This ensures that even if the password database is compromised, the original passwords are not easily obtained.
3. **Digital Signatures**: SHA-256 hashing is an integral part of digital signatures. It is used to generate the hash value of the message or document being signed. The hash value is then encrypted with the sender's private key to create the digital signature. The recipient can verify the signature by decrypting it with the sender's public key and comparing it with the computed hash value of the received message.
4. **Secure Boot**: SHA-256 hashing is commonly used in the secure boot process of devices, including routers. The bootloader, kernel, and other critical components are hashed using SHA-256 to generate their unique hash values. These hash values are then digitally signed using cryptographic keys. During the boot process, the firmware verifies the integrity of these components by comparing their computed hash values with the signed hash values.

Down below, you can find an example of performaning a SHA-256 hash in Python. Again, this is an example and not the actual code for the router itself.



Example SHA-256 hash in python

In this example, the ‘sha256\_hash’ function takes an input data (in bytes) and performs SHA-256 hashing using the ‘hashlib’ module in Python. It returns the hash value as a hexadecimal string. The resulting hash value can be used for various security applications, such as data integrity verification or digital signatures.

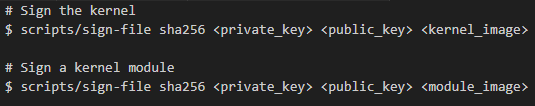
## 6.2 Kernel Signing

Kernel signing is a process of digitally signing the kernel or kernel modules in an operating system to ensure their integrity and authenticity. It helps protect against the execution of malicious or unauthorized code by verifying the identity of the kernel or modules before they are loaded into memory.

The specific steps and methods for kernel signing can vary depending on the operating system and the signing infrastructure used. Here, I'll provide a general overview of kernel signing and an example code snippet using the Linux kernel's signing mechanism.

1. **Generate signing keys**: The first step is to generate a pair of cryptographic keys—a private key and a corresponding public key. The private key is kept secure and used for signing, while the public key is distributed and used for verification.
2. **Configure the kernel**: The kernel needs to be configured to enable the signing mechanism and provide the necessary information for signing and verification. This involves setting up the appropriate configuration options and specifying the location of the signing keys.
3. **Sign the kernel/module**: Once the kernel is built, you can sign it or individual kernel modules using the private key generated earlier. The signing process generates a cryptographic signature that is embedded in the kernel/module.
4. **Verify the signature**: During the boot process, the bootloader or the kernel itself verifies the signature of the kernel/modules using the corresponding public key. If the signature is valid, the kernel/module is loaded; otherwise, it is rejected.

Below is an example code snippet demonstrating kernel signing using the Linux kernel's signing mechanism. This example assumes you have already set up the necessary tools and keys for signing.

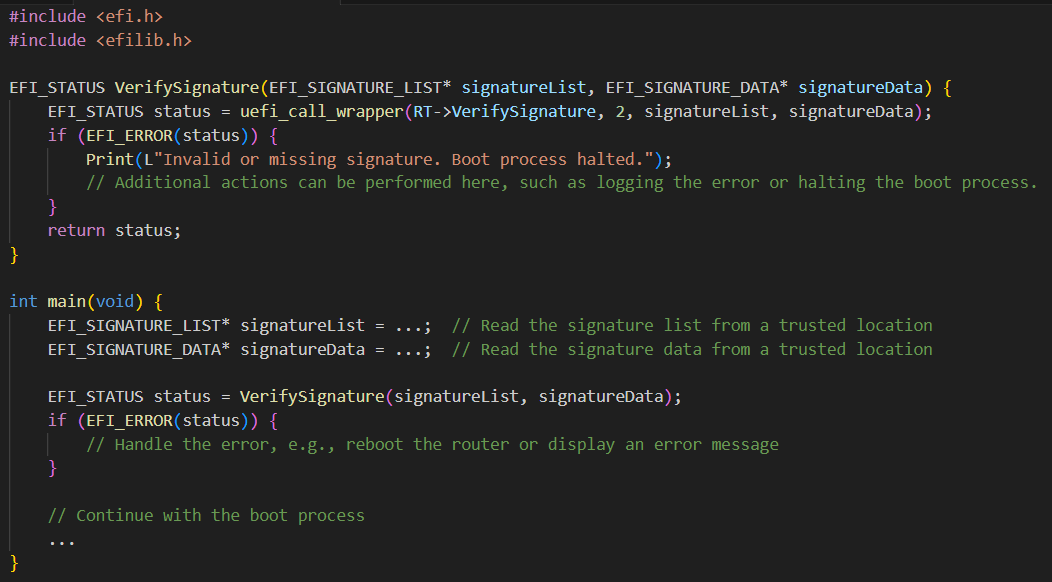


In the above code, ‘<private\_key>’ represents the path to your private key file, ‘<public\_key>’ is the path to the corresponding public key file, ‘<kernel\_image>’ is the path to the kernel image file, and ‘<module\_image>’ is the path to the module image file. The ‘sign-file’ script is provided by the Linux kernel source code in the ‘scripts’ directory.

# 7 Boot Process Validation

## 7.1 Signature Verification

During the boot process, Secure Boot validates the digital signatures of the bootloader, kernel, and other critical components. This ensures that only trusted and signed code is executed. The validation process involves verifying the signature against the corresponding public key and checking the integrity of the signed components. The specific mechanisms and steps for boot process validation may vary depending on the router's firmware.



Example code snippet - Boot Process Validation

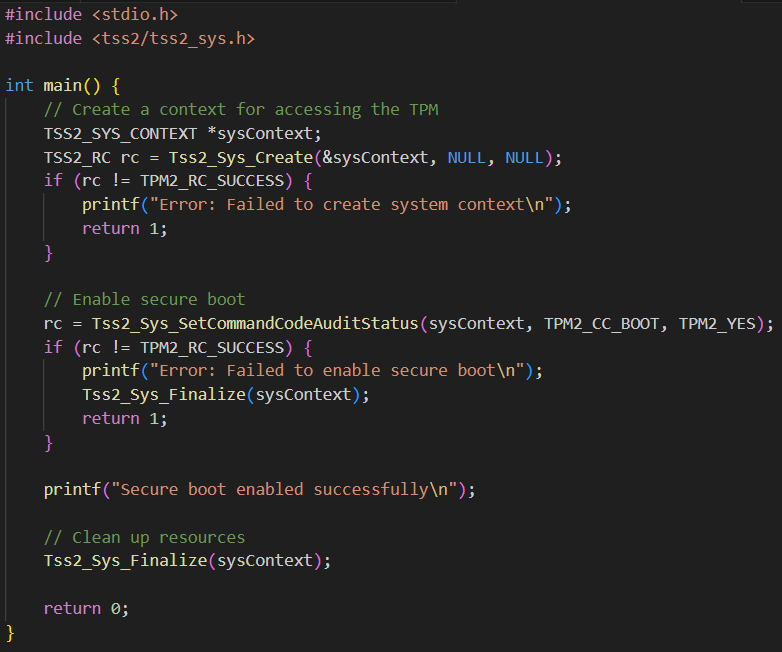
Secure Boot begins by validating the digital signatures of the bootloader, kernel, and other critical components. Digital signatures are generated using asymmetric encryption, where a component is signed with a private key and the corresponding public key is embedded in the system's firmware or trusted platform module (TPM). During the boot process, the signature is verified against the corresponding public key. If the signature is valid, it confirms the authenticity and integrity of the component. This process ensures that only components from trusted sources are executed, preventing the execution of unauthorized or tampered code.

### 7.1.1 TPM

TPM, which stands for Trusted Platform Module, is a dedicated hardware component that provides a secure foundation for various security-related tasks on a computer system. It is designed to store cryptographic keys, securely generate random numbers, and perform cryptographic operations. TPM helps establish and maintain the integrity of a system by ensuring that only authorized software and firmware can be executed during the boot process.

One of the features supported by TPM is secure boot, which ensures that only trusted software components, such as the operating system and bootloader, are loaded and executed during the boot process. Secure boot relies on digital signatures to verify the integrity and authenticity of the software components before allowing them to run.

Here's an example code snippet in C using the TSS (TPM Software Stack) library to enable secure boot using TPM:



Example code snippet in C using TSS to enable secure boot

#### Code explained

In this code snippet, we first create a system context to interact with the TPM using the Tss2\_Sys\_Create function. Then, we use the Tss2\_Sys\_SetCommandCodeAuditStatus function to enable secure boot by specifying TPM2\_CC\_BOOT as the command code and TPM2\_YES as the audit status. If the function call is successful, secure boot is enabled, and a success message is displayed. Finally, we clean up the resources using Tss2\_Sys\_Finalize.

## 7.2 Integrity Checking

In addition to signature verification, the boot process validation involves checking the integrity of the signed components. This step ensures that the components have not been modified or corrupted since they were signed. Integrity checking is typically performed by calculating a cryptographic hash of the component and comparing it with the stored hash value. If the calculated hash matches the stored value, it indicates that the component has remained unchanged and can be trusted for execution. Any discrepancy in the hash value suggests that the component has been tampered with or corrupted, raising a red flag for potential security risks.

## Variations in Validation Mechanisms

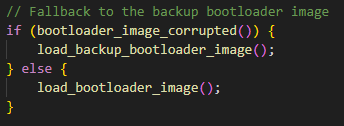
The specific mechanisms and steps for boot process validation may vary depending on the router's firmware. Different routers may have different implementations of Secure Boot, incorporating variations in cryptographic algorithms, key management, and validation procedures. It is crucial for system administrators to understand the firmware specifications and follow the recommended guidelines provided by the router manufacturer. By adhering to these guidelines, administrators can ensure the effectiveness of boot process validation and mitigate the risk of unauthorized code execution.

# 8 Plan B: Backup Measures for Secure Boot

Secure boot is, as mentioned thoughout this report, a critical process that ensures the integrity and authenticity of the software running on a device. However, despite the robustness of secure boot mechanisms, unforeseen issues may still arise. To mitigate potential problems, it is important to have a plan B in place. Here are some examples of what could go wrong during secure boot, how they can occur and corresponding solutions, including code snippets. These snippets are written in pseudocode, a high-level, informal description version of actual code.

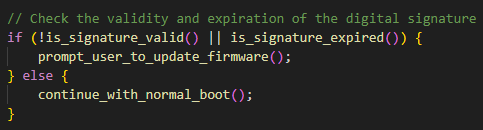
## 8.1 Corrupted Bootloader Image

* **Issue**: The bootloader image used for secure boot becomes corrupted or compromised, preventing successful booting of the router.
* **Possible causes**:
  + During the bootloader image transfer from a development environment to the production environment, file corruption may occur due to transmission errors or storage malfunctions.
  + Malware or malicious actors may attempt to modify the bootloader image during the production process or while the router is in transit, leading to a corrupted image.
  + Environmental factors such as power surges or hardware failures during the firmware update process can result in a corrupted bootloader image.
* **Solution**: Provide a failsafe option to fall back to a known good bootloader image stored in a separate location. This can be implemented by adding a redundant copy of the bootloader image in a backup partition.



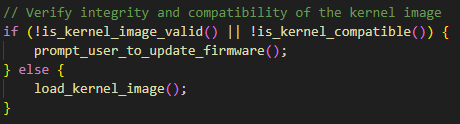
## 8.2 Invalid or Expired Digital Signature

* **Issue**: The digital signature of the firmware or bootloader becomes invalid or expired, causing the secure boot process to fail.
* **Possible causes**
  + The private key used for signing the firmware or bootloader may become compromised, leading to the creation of invalid signatures.
  + The digital certificate used to verify the firmware or bootloader signature may expire, rendering the signature invalid.
  + Inadequate certificate management practices, such as failing to update or renew certificates in a timely manner, can result in expired certificates.
* **Solution**: Implement a mechanism to check the validity and expiration date of the digital signature during secure boot. If the signature is invalid or expired, provide an option to boot with a limited functionality mode or prompt the user to update the firmware.



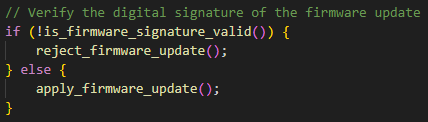
## 8.3 Incompatible or Tampered Kernel Image

* **Issue**: The kernel image used for secure boot is incompatible with the hardware or has been tampered with, leading to boot failures.
* **Possible causes**
  + Development or integration teams may accidentally introduce incompatible changes to the kernel image, leading to compatibility issues with specific hardware components or device configurations.
  + Malicious actors may attempt to modify the kernel image during production or during the update process, introducing code that is incompatible with the router's hardware or compromising its integrity.
  + Lack of rigorous (pen-)testing and validation processes during the kernel image integration can result in undetected incompatibilities or tampering.
* **Solution**: Employ a mechanism to verify the integrity and compatibility of the kernel image before loading it. If the image is found to be incompatible or tampered with, switch to a known working kernel image or prompt the user to update the firmware.



## 8.4 Rollback Attack

* **Issue**: An attacker attempts to downgrade the firmware or bootloader to a vulnerable version by exploiting weaknesses in the secure boot process.
* **Possible causes**
  + Insufficient secure boot mechanisms may fail to detect or prevent unauthorized firmware or bootloader downgrades, allowing attackers to exploit vulnerabilities in older versions.
  + Attackers may exploit weaknesses in the firmware update process, such as man-in-the-middle attacks or intercepted update packages, to inject older or vulnerable versions of the firmware or bootloader.
  + Inadequate validation and enforcement of digital signatures during the firmware update process can enable attackers to forge or bypass signature checks, facilitating the rollback attack.
* **Solution**: Implement a secure firmware update mechanism that ensures only authorized firmware versions can be installed. This can include enforcing digital signature checks on firmware updates and storing a secure version history to prevent rollback attacks.



## 

By incorporating these backup measures into the secure boot process of the router, you can enhance the overall robustness and reliability of the device, mitigating potential issues that may arise during secure boot.

## 8.5 Summary

unforeseen issues can arise during secure boot, such as corrupted bootloader images, invalid or expired digital signatures, incompatible or tampered kernel images, and rollback attacks. To mitigate these problems, it is essential to have a plan B in place. By implementing backup measures like fallback bootloader images, signature checks, compatibility verification, and secure firmware updates, the overall robustness and reliability of the router can be enhanced. Understanding the specific ways in which these issues can occur enables the development of targeted solutions to address them effectively.

# 9 Research Question

The research question was as follows:

*How to implement secure boot on the GNC Pure E600 network router?*

*Sub-Questions:*

1. *What is secure boot and why is it important for network routers?*
2. *How can the integrity and authenticity of the firmware and boot components be verified during the secure boot process?*
3. *How can secure boot be effectively integrated into the boot process of the GNC Pure E600 router?*

|  |  |  |  |
| --- | --- | --- | --- |
| **Research Questions** | **Strategies** | **Methods** | **Outcome** |
| How to implement secure boot on the GNC Pure E600 network router? | Determine the specific requirements and capabilities of the GNC Pure E600 router for implementing secure boot. | - Study the GNC Pure E600 router's documentation and specifications  - Research existing secure boot implementation strategies  - Identify appropriate tools and software for implementing secure boot. | Develop a step-by-step implementation guide for secure boot on the GNC Pure E600 network router, considering the router's requirements, capabilities, and utilizing established secure boot implementation strategies. |
| What is secure boot and why is it important for network routers? | Define and explain the concept of secure boot. | - Conduct literature review on secure boot for network routers. | Gain a clear understanding of secure boot and its significance in network routers. |
| How can the integrity and authenticity of the firmware and boot components be verified during the secure boot process? | Explore techniques and mechanisms for verifying firmware and boot component integrity and authenticity. | - Research cryptographic algorithms for secure boot verification. | Identify suitable methods for ensuring the integrity and authenticity of firmware and boot components during the secure boot process. |
| How can secure boot be effectively integrated into the boot process of the GNC Pure E600 router? | Investigate the boot process of the GNC Pure E600 router and determine integration possibilities. | - Analyze the GNC Pure E600 router's boot process and firmware architecture. | Develop a plan to integrate secure boot into the boot process of the GNC Pure E600 router. |

Unfortunately, the research question cannot be adequately answered in a brief response due to the extensive nature of the subject matter. The scope of this project is focused solely on the implementation of secure boot on routers, which is just one aspect of the broader topic of secure boot. Secure boot encompasses a wide range of services, topics, and hardware/software considerations, necessitating a comprehensive report that provides all the necessary advice for implementing secure boot.

# 10 Conclusion

In conclusion, implementing Secure Boot on a Linux-based Wi-Fi router involves a series of configuration steps, cryptographic key generation and storage, and bootloader signature verification. By adopting these robust security measures, router manufacturers and network administrators can effectively mitigate various risks associated with unauthorized access and tampering. The implementation of Secure Boot strengthens the security of critical network infrastructure, safeguarding against potential attacks that may compromise the router's functionality or expose sensitive data. This report has provided detailed technical insights, along with example code snippets, to guide the implementation of Secure Boot on a Linux-based Wi-Fi router.

# References

Booting a Self Signed Linux Kernel. Linux Kernel Monkey Log. Retrieved June 18, 2023, from <http://www.kroah.com/log/blog/2013/09/02/booting-a-self-signed-linux-kernel/>

N.d. Modern Encryption (Cryptography). W3schools  
<https://www.w3schools.in/cyber-security/modern-encryption>

N.d. What is a cryptograpic key? Cloudflare.  
<https://www.cloudflare.com/en-gb/learning/ssl/what-is-a-cryptographic-key/>

Campos. P. (2020, April 27). AES Implementation in Python. Medium.  
<https://medium.com/quick-code/aes-implementation-in-python-a82f582f51c2>

N.d. What types of encryption are there? Ico.  
<https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/security/encryption/what-types-of-encryption-are-there/#:~:text=There%20are%20two%20types%20of,used%20for%20encryption%20and%20decryption>.

Cobb, M (2021, November). RSA Algorithm (Rivest-Shamir-Adleman). Techtarget.  
<https://www.techtarget.com/searchsecurity/definition/RSA#:~:text=RSA%20is%20a%20type%20of,is%20used%20to%20decrypt%20it>.

N.d. (2022, November 05). What is a bootloader and how does it work? Ionos.  
<https://www.ionos.com/digitalguide/server/configuration/what-is-a-bootloader/>

Ebu-Amu, C (2021, April 20). What is a Bootloader? How Does a Bootloader Work?  
<https://www.makeuseof.com/what-is-a-bootloader/>

Trudel-Lapierre, M. (2017, Agusut 11). How to sign things for Secure Boot. Ubuntu.  
<https://ubuntu.com/blog/how-to-sign-things-for-secure-boot>

N.d. (2021, February 04). How to sign your kernel and bootloader? Askubuntu.  
<https://askubuntu.com/questions/1313033/how-to-sign-your-kernel-and-bootloader>

ValdikSS (2019, April 01). Exploiting signed bootloaders in circumvent UEFI Secure Boot. Habr.  
<https://habr.com/en/articles/446238/>

N.d. (2022, June 25). Secure Hash Algorithms. Wikipedia.  
<https://en.wikipedia.org/wiki/Secure_Hash_Algorithms>

N.d. What is SHA? What is SHA used for?  
<https://www.encryptionconsulting.com/education-center/what-is-sha>

N.d. (2021, February 01). Understanding Digital Signatures. Cisa.  
<https://www.cisa.gov/news-events/news/understanding-digital-signatures#:~:text=What%20is%20a%20digital%20signature,%2C%20or%20a%20digital%20document>).

N.d. Kernel module signing facility. Kernel.  
<https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html#:~:text=The%20kernel%20module%20signing%20facility,signed%20with%20an%20invalid%20key>.

N.d. (2022, September 23). Signed kernel modules. Archlinux.  
<https://wiki.archlinux.org/title/Signed_kernel_modules>

Nakamoto, R (2016, October). Secure Boot and Image Authentication. Qualcomm.  
<https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/secure-boot-image-authentication_11.30.16.pdf>

Tashkinov, A. (2021, February 25). How to verify UEFI binaries? Stack Exchange  
<https://unix.stackexchange.com/questions/636263/how-to-verify-signed-uefi-binaries>

Gillis, A. (2022, February). Trusted Platform Module (TPM). TechTarget.  
<https://www.techtarget.com/whatis/definition/trusted-platform-module-TPM>

N.d. Trusted Platform Module (TPM) Summary. Trusted Computing Group.  
<https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/>

N.d. TPM Software Stack (TSS). Trusted Copmuting Group.  
<https://trustedcomputinggroup.org/work-groups/software-stack/#:~:text=The%20TPM%20Software%20Stack%20(TSS,for%20more%20tamper-resistant%20computing>.

Dock, D. The TPM2 Software Stack: Introducing a Major Open Source Release. Intel.  
<https://www.intel.com/content/www/us/en/developer/articles/technical/tpm2-software-stack-open-source.html>