Analysis of Secure Boot using Machine Owner Key Technology

Annisa Ayu Pramesti - 13518085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13518085@sted.stei.itb.ac.id

***Abstract*—Secure boot is a security standard to help make sure that a device boots using only software that is trusted. When the PC starts, the firmware checks the signature of each piece of boot software, including UEFI firmware drivers, EFI applications, and the operating system. Machine Owner Key is a technology that can be used to sign a custom efi-software to be recognized as secure by UEFI Secure Boot. The advantages of using MOK technology is that users are able to create custom modules or kernels to the system or install open source efi-software easily. This study's aim is to experiment and analyze MOK secure boot by making a custom kernel signed with MOK. The custom kernel is made by creating a custom system call and tested in the system after it has been signed with MOK technology. After the experiment was conducted, this study analyzed the vulnerabilities formed when the MOK is enabled.**

***Keywords*—Machine Owner Key, system call, UEFI Secure Boot.**

I. Introduction

Secure boot is a technology intended to protect the integrity of the boot process and runtime integrity of the system from adversaries with external physical access to the device. When Secure Boot is enabled, all kernel modules must be signed with a private key and authenticated with the corresponding public key. Secure Boot can be disabled, but a mandatory requirement for changing its state is the physical presence of the user at the computer. In a U.S. government cyber security advisory, the National Security Agency and Federal Bureau of Investigation warn of a previously undisclosed piece of Linux rootkit malware called Drovorub and attribute the threat to malicious actor APT28. Both of them stated that systems with kernel versions of 3.7 or lower are susceptible to this malware due to the absence of adequate kernel signing enforcement.

There was a lot of discussion about UEFI Secure Boot regarding open source issues. An open source OS must get signed to run on computers and that must be paid. This makes the concept of open source less upheld because open source software should be free. Thus, the Machine Owner Key (MOK) concept can be used with a signed shim loader to enable key management at the user/sysadmin level. Machine Owner Key (MOK) Secure Boot is an alternative key management system that allows a target to boot using loaders, kernels, and other binaries signed with user-provided keys. It uses an initial loader called a shim, which is an EFI executable accepted and already signed by a key in the databases. Nevertheless, there are some restrictions to MOK signed kernels such as no direct access to IO port and memory. The custom kernels are also not possible to load unsigned 3rd party modules.

This study's goal is to conduct an experiment using MOK for secure boot and to analyze the security of the custom kernel itself. In this research, we will create a kernel with a custom system call and it will be signed using MOK.

II. UEFI Secure Boot

Secure boot is a technology introduced in 2013 to prevent the execution of unsigned or untrusted program code such as .efi programs, operating system boot loader, and additional hardware firmware like video card and network adapter OPROMs [1]. UEFI Secure Boot (SB) is a secure boot mechanism for computers with UEFI firmware. It is designed to protect a system against malicious code being loaded and executed early in the boot process, before the operating system has been loaded [2]. The assumption is that the attacker can get code execution on the device as the user and as root, but does not have access to the signing keys or the disk encryption key. The goal is to prevent the attacker from exfiltrating data from the device or making persistent changes to the system configuration [3].

The key components of UEFI Secure boot are:

1) UEFI Image Signing. Secure Boot works using cryptographic checksums and signatures. Each program that is loaded by the firmware includes a signature and a checksum, and before allowing execution the firmware will verify that the program is trusted by validating the checksum and the signature. When Secure Boot is enabled on a system, any attempt to execute an untrusted program will not be allowed. This stops unexpected / unauthorised code from running in the UEFI environment. UEFI keys are stored in Secure Boot. The following keys are defined by specification:

- *Platform Key (PK). This is a root key, created for this exact platform. PK Key Owner (usually hardware manufacturer) can modify all other keys. This the top level key in secure boot architecture. PK is self signed i.e. PK is signed by its own private counterpart.*

- *Key Exchange Key (KEK) – The KEK's private counterpart is used to sign db or dbx holding the list of keys marked as secure or unsecure. The owner of this key (usually OS vendor) can update db/dbx keys. KEK is signed using PK's private counterpart.*
- *db – These are the key types that are used to verify the utility that is going to be loaded when the secure boot is enabled on the system. All allowed certificates and hashes are stored here (white list).*
- *dbx – These are anti-db keys i.e. dbx holds the key sources and hashes that are blacklisted from being secure or marked as malware. All forbidden certificates and hashes are stored here (black list).*

2) UEFI Authenticated Variable. It is Time-Based Authenticated Write Access used to update Authenticated variables like keys db. Updated variables also must be signed. PK certification verifies PK/KEK update while KEK verifies db/dbx update. Lastly, certdb verifies general authenticated EFI variable updates.

3) UEFI Secure Update. This component is used to authenticate firmware update images using digital signature verification. In order to enable the Secure Boot mode on a platform one should do the following: Enroll Platform Key (PK) and other keys, if needed. Set value of SetupMode variable to USER_MODE (stored in NVRAM). Adjust SecureBootEnable variable (stored in NVRAM). UEFI variables can be updated using the UEFI runtime service SetVariable. UEFI provides write-protected service for Authenticated variables, based on asymmetric key technology. In order to update variable value, it should be signed with the appropriate key. Disabling of Secure Boot, as well as deploying new PK should be done only in users' physical presence [4].

If UEFI image is not signed by a trusted key, and it's hash is not found in db, the image shall not be loaded. If bootloader signature verification fails, the operating system shall not load.

In the chain of trust, entities are signed in such a way that validation leads to "root of trust". In the case of Secure Boot, platform owner is considered as the root of trust and addition of keys to keystore follows chain of trust. That is, no key can be replaced unless signed by preceding key.

The chain of trust can be explained using the keys defined in UEFI Secure Boot. PK is self signed, KEK is signed by PK and db and dbx are signed by KEK. That means to change anything in db or dbx, it should be signed by KEK's private key, to change KEK itself, it must be signed by PK's private key and to replace PK the new key should be signed by previous PK's private key. Generally replacing PK is done using an empty key certificate signed by the previous platform owner's private key.

Lastly, UEFI Secure Boot is a security measure to protect against malware during early system boot. UEFI Secure Boot is not an attempt by Microsoft to lock Linux out of the PC market. UEFI Secure Boot is also not meant to lock users out of controlling their own systems. Users can enrol extra keys into the system, allowing them to sign programs for their own systems. Many SB-enabled systems also allow users to remove the platform-provided keys altogether, forcing the firmware to only trust user-signed binaries [2].

By customizing computer's Secure Boot keys offers several advantages over these approaches:

1) Locking out threats from the standard keys. In theory, Secure Boot should prevent malware from running. On the other hand, it's always possible that an attacker could trick Microsoft into signing malware. We can use Shim with the default keys, the computer will remain vulnerable to these threats, at least until they're discovered and the blacklist database is updated.

2) Locking out threats from your distribution's keys. Similar to the preceding, it's also possible that the distribution's keys could be compromised, in which case an attacker could distribute malware signed with the compromised keys. Depending on the keys management, this vulnerability can be greatly reduced but the greatest level of protection will require extra effort.

3) Eliminating the need for MOKs. The Shim and PreLoader tools both rely on Machine Owner Keys (MOKs), which are similar to Secure Boot keys but easier to install. Because they can be more easily installed, it's conceivable that they could be more readily abused by social engineering or other means. Eliminating MOKs may therefore slightly increase security, particularly if there is a collection of desktop computers used by other people.

4) Taking philosophical control. Relying on a third party's keys strikes some people as being wrong. Partly this is because of the preceding reasons, but some people object to the dependency on a more philosophical level.

5) Testing and development.. This approach is to develop a custom boot manager by testing a signed version of the customized software in an environment that mimics a "stock" computer. The process for signing binaries with Microsoft's Secure Boot keys is tedious and time-consuming, though, so the developer may need to set the computer up with customized keys to sign the binaries. When the software works as expected, the developer can send it to Microsoft to be signed.

6) Enabling Secure Boot on systems without keys. Some servers ship without Microsoft's Secure Boot keys installed. Using Secure Boot with such a server requires adding keys as described on this page. Note that Linux distributions for exotic platforms, such as ARM64, do not currently (in mid-2018) ship with signed Shim implementations, so the developer will need to sign the boot loader (or add whatever public key matches the private key used to sign the boot loader) and boot it directly.

7) Overcoming default boot-hogs. This one is admittedly speculative. Some people have reported difficulty getting their computers to boot anything but Windows by default; they can boot to Linux temporarily, use efibootmgr to set Linux as the default boot loader, but then find themselves booting back to Windows because the firmware keeps setting Windows as the default. If the Linux boot entry remains in place but is "demoted," setting customized own boot keys might enable to control this problem by removing Microsoft's key from the regular Secure Boot list and adding it to your MOK list. However, some computers hang upon encountering the first Secure Boot error; and if the problem is caused by a computer "forgetting" its entire boot list, this solution will do no good whatsoever.

III. Machine Owner Key

A. Shim

Shim is a simple software package developed by a group of Linux developers from various distros that is designed to work as a first-stage bootloader on UEFI systems [2]. The main purpose of the shim is to relocate and execute code from a second-stage boot loader after it has been cryptographically validated using user-provided keys. Once the second-stage boot loader is running, it loads the Linux kernel, which in turn triggers a cryptographic validation of the kernel image [5]. This allows Microsoft (or other potential firmware CA providers) only to worry about signing the shim, and not all of the other programs that distro vendors might want to support.

Shim then becomes the root of trust for all the other distro-provided UEFI programs. It embeds a further distro-specific CA key that is itself used for signing further programs (e.g. Linux, GRUB, fwupdate). This allows for a clean delegation of trust - the distros are then responsible for signing the rest of their packages. Shim itself should ideally not need to be updated very often, reducing the workload on the central auditing and CA teams [2].

B. Machine Owner Key

MOK (Machine Owner Key) is about securing the boot process by only allowing approved OS components and drivers to run. The main idea is that only code which is signed is allowed to run while loading the operating system (OS). Once that is booted, the OS can take over responsibility for securing the system from the BIOS.

The MOK system uses public key cryptography, which means that the user can create a key pair, then sign, with private/secret key, all components that are allowed to run. This includes the GRUB boot loader itself. Then the BIOS uses the corresponding public key to check signatures before running the code.

Keys can be added and removed in the MOK list by the user, entirely separate from the distro Certification Authority (CA) key. The mokutil utility can be used to help manage the keys here from Linux userland, but changes to the MOK keys may only be confirmed directly from the console at boot time. This removes the risk of userland malware potentially enrolling new keys and therefore bypassing the entire point of Secure Boot.

IV. Experiment

A. Creating a custom kernel

In this research, a custom kernel will be made by making a simple system call in a Linux system. The following steps are compatible with the 5.3.18 version of the Linux kernel. The system call increments a global integer by 1 and returns it.

1) Implement the system call. At the linux source, make a folder to store the source code of the new system call. Below is the code of the system call implemented in C with file name mycall.c.

Algorithm 1 System call implementation

```
#include <linux/kernel.h> /* required */
#include <linux/module.h> /* for EXPORT_SYMBOL */
#include <linux/init.h> /* for init */

int mycall_global = 0;
EXPORT_SYMBOL(mycall_global); /* required to make it global */

asmlinkage long sys_mycall(void) /* the system call */
{
        printk("running sys_mycall\n");
        mycall_global++;
        return mycall_global;
}

void __init mycall_init(void) /* init mycall_global */
{
        mycall_global = 0;
}
```

2) Add a makefile. To compile the source code, we have to add a makefile and include the source code as built in. The makefile is placed in the same folder as the source code.

```
obj-y := mycall.o
```

We also have to edit the one in the root directory of linux source. Add /mycall at the end of the line below.

```
core-y += kernel/ …
```

3) Register the system call. From the source root folder, we then go to arch/x86/entry/syscalls and add the custom system call at the end of the file syscall_64.tbl.

```
548     common  mycall          sys_mycall
```

We also have to add the system call at the end of syscall_32.tbl.

```
437     i386    mycall          sys_mycall
```

4) Compile and install the kernel. In the source root folder, we make a compile configuration file then compile the kernel. After we compile the kernel, we can install the kernel to be able to run it on the computer.

If the steps are completed successfully, there will be image files in the bootloader directory.

E. Signing the custom kernel using MOK

After we made a custom kernel, it won't be able to be booted because of the Secure Boot. As explained above, the Secure Boot forbids unsigned modules as well as unsigned kernels to be booted in the system to prevent malware.

To complete this experiment we have to sign the custom kernel using Machine Owner Key. There are some required utilities such as openssl to generate public and private keys, mokutil to manually enroll the public key, and sbsign to sign the custom kernel [7]. Below are the following steps to sign the custom kernel.

1) Create signing keys. First we have to create a config file to create the signing key and save the file with .cnf extension.

```
 HOME                       = .
RANDFILE                    = $ENV::HOME/.rnd
[ req ]
distinguished_name          = req_distinguished_name
x509_extensions             = v3
string_mask                 = utf8only
prompt                      = no


 [ req_distinguished_name ]
 countryName                = <country code>
 stateOrProvinceName        = <state>
 localityName               = <city>
 0.organizationName         = <organization>
 commonName                 = Secure Boot Signing Key
 emailAddress               = <email>

 [ v3 ]
 subjectKeyIdentifier       = hash
 authorityKeyIdentifier     = keyid:always,issuer
 basicConstraints           = critical,CA:FALSE
 extendedKeyUsage           = codeSigning,
1.3.6.1.4.1.311.10.3.6
 nsComment                  = "OpenSSL Generated
Certificate"
```

Using openssl, we then generate the public and private keys in the format .der to be used by mokutil and .pem to be used by sbsign [8].

```
openssl req -config ./mokconfig.cnf \
        -new -x509 -newkey rsa:2048 \
        -nodes -days 36500 -outform DER \
        -keyout "MOK.priv" \
        -out "MOK.der"

openssl x509 -in MOK.der -inform DER -outform PEM -out
MOK.pem
```

2) Enroll MOK key. Using mokutil, we enroll the MOK key to the shim installation by giving a password. After that, the system has to be restarted to continue the enrolling process by entering the password.

```
sudo mokutil --import MOK.der
```

3) Sign the custom kernel. Using sbsign, we can sign the kernel using the MOK key in the .pem format. The kernel is signed and authorized by Secure Boot to be booted in the system.

```
sudo sbsign --key MOK.priv
            --cert MOK.pem
            /boot/vmlinuz-5.3.18-generic
            --output
            /boot/vmlinuz-5.3.18-generic.signed

sudo cp /boot/initrd.img-5.3.18-generic{,.signed}
```

After all of the steps are finished, the kernel can be booted in the system by updating the grub-config.

```
sudo update-grub
```

.

IV. Testing

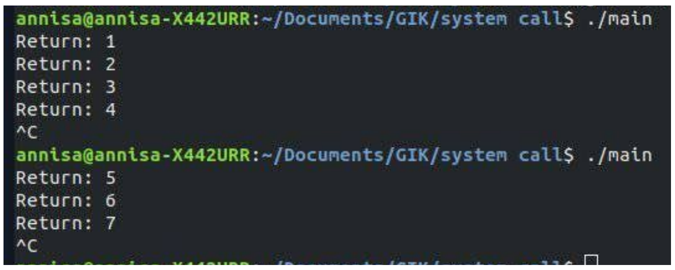
To determine the success of the experiment, here is the source code used to test the system call in C.

Algorithm 2 System call testing

```
#include <stdio.h>
#include <unistd.h>
#include <linux/kernel.h>
#define __NR_mycall 549


int main(int argc, char *argv[])
{
        while(1)
        {
                int ret;
                ret = syscall(__NR_mycall);
                printf("Return: %d\n", ret);
                sleep(1);
        }
        return 0;
}
```

Below is the output after the code above is executed.



```
annisa@annisa-X442URR:~/Documents/GIK/system call$ ./main
Return: 1
Return: 2
Return: 3
Return: 4
^C
annisa@annisa-X442URR:~/Documents/GIK/system call$ ./main
Return: 5
Return: 6
Return: 7
^C
```

Figure 1. Testing result of the system call

The system call that is made runs successfully because it returns the value of a global variable in the kernel which is always incremented

V. Security Analysis

Based on the experiment, we can conclude that the custom kernel can run perfectly in the system even though it is signed by a user-defined MOK key. By using Machine Owner Key technology to sign the custom kernel, there are some risks or vulnerabilities that possibly formed in the system.

There are 2 conditions that can trigger vulnerabilities in the system. First is when the user is enabling MOK keys provided by malware authors. This can be done by enrolling the keys into the MOK database so it can be recognized in the system and the UEFI firmware will treat the efi file as secure. Second is when the MOK key is stolen. If in some conditions, a hacker can have access to the system and get the MOK keys, the risks triggered will be even greater. Below are possible risks of enabling MOK to the Secure Boot.

1) Exploitation using third-party software. By enabling MOK, any third-party efi-software that is signed using MOK key can be run in the system and will be recognized as secure by UEFI secure boot. Anyone with access to MOK utils can also register their keys/hashes and mark their utilities as secure. If the module is malicious, it can be used to exploit the system.

2) Embedded third-party unauthorized MOK key. Hidden embedding of a third-party unauthorized Machine Owner Key into the system for subsequent running of any unauthorized efi-software on the infected platform.

All of the vulnerabilities above can occur only if the hackers or attackers have access to the MOK utility or if they can create their public and private keys and register them to the system. From the other side if the MOK key is stolen by an attacker then it could be used to sign and boot any efi-application. The problem is even worse because MOK–key can be added to the MOK DB from the level of the operating system (e.g. etc/secureboot in Ubuntu) and UEFI Runtimeservices.

VI. Conclusion

Machine Owner Key is a technology that can be used to sign a custom efi-software to be recognized as secure by UEFI Secure Boot. The advantages of using MOK technology is that users are able to create custom modules or kernels to the system or install open source efi-software easily. However, by enabling MOK, the system is also exposed to some threats. A variety of platform manufacturers and drivers developers, each of them implementing the specifications requirements on their own, may result in vulnerabilities in particular UEFI implementations. When the MOK access is stolen by attackers or hackers such as exploitation using third-party software and embedded third-party unauthorized MOK key.

VII. Future Research

There are many researches that can be developed based on this proposition such as creating a system above MOK system to ensure authentication of the keys enrolled in the MOK management system. We can also conduct research that focuses on analyzing the threats deeper such as making an attack simulation to the system and find a way to minimize the effect of the attack.

VIII. Acknowledgment

The author would like to express her gratitude to God Almighty for his guidance, to her parents for their eternal support, love and education, and to Mr. Rinaldi Munir for his teachings of Cryptography to complete this paper.

References

[1] ValdikSS, "Exploiting signed bootloaders to circumvent UEFI Secure Boot," 2019. [Online]. Available: https://habr.com/en/post/446238/. [Accessed: 19-Dec-2020].

[2] Debian, "SecureBoot," 2020. [Online]. Available: https://wiki.debian.org/SecureBoot. [Accessed: 19-Dec-2020].

[3] Safeboot, "Threat Model," 2020. [Online]. Available: https://safeboot.dev/threats/. [Accessed: 19-Dec-2020].

[4] V. Bashun, A. Sergeev, V. Minchenkov, and A. Yakovlev, "Too young to be secure: Analysis of UEFI threats and vulnerabilities," Conf. Open Innov. Assoc. Fruct, no. March 2015, pp. 16–24, 2013.

[5] windriver, "EFI Secure Boot Using Machine Owner Keys," 2019. [Online]. Available: https://docs.windriver.com/bundle/Wind_River_Linux_Security_Features_Guide_9_1/page/qii1503681802496.html. [Accessed: 20-Dec-2020].

[6] D. Glover, "Signing a Linux Kernel for Secure Boot." [Online]. Available: https://gloveboxes.github.io/Ubuntu-for-Azure-Developers/docs/signing-kernel-for-secure-boot.html. [Accessed: 20-Dec-2020].

[7] R. Hat, "CHAPTER 3. MANAGING KERNEL MODULES," 2020. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_monitoring_and_updating_the_kernel/managing-kernel-modules_managing-monitoring-and-updating-the-kernel#signing-kernel-modules-for-secure-boot_managing-kernel-modules. [Accessed: 20-Dec-2020].

[8] R. Smith, "Managing EFI Boot Loaders for Linux: Controlling Secure Boot," 2018. [Online]. Available: https://www.rodsbooks.com/efi-bootloaders/controlling-sb.html. [Accessed: 21-Dec-2020].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2020

[illegible]

Annisa Ayu Pramesti
13518085