Code Signing Defensive Systems for IoT Devices

Introduction

As the number of internet-connected devices grow, it is important to defend these devices from malicious actors leveraging compromised devices in large scale cyber-attacks. Mirai is a malware that targets Internet of Things (IoT) devices running Linux into remotely controlled bots [1]. The network of bots become a vehicle to carry out Distributed Denial-of-Service (DDoS) attacks. According to Cisco, the number of of internet connected devices is predicted to reach 30 billion by 2023 [2]. The number of IoT devices has a direct impact on the scale of disruption from botnet attacks. Also, IoT devices are often vulnerable due to limited resources for running security functions, manufacturer default usernames and passwords, and vendors sacrificing security for price-competitiveness or time-to-market.

The Mirai botnet architecture consists of remote servers connected to many Mirai bots [3]. The Mirai attack starts with a scan of IP addresses to attempt to locate IoT devices with open Telnet or SSH ports. If an IoT device is identified, Mirai performs a brute-force attack using a dictionary of common default usernames and passwords. When the brute-force attack succeeds, the malware sends the IP and login credentials of the device to the scan receiver server, where the device information may be stored for later use. The scan receiver server sends the information to the loading server. The loading server checks the device to see if remote file transfer programs like wget or tftp are available. If available, the server will download a Mirai bot binary to the device. If not, the malware will load a binary similar to wget. Once compromised, devices infected with Mirai repeat the previous processes, searching for other vulnerable devices to infect.

The Mirai attack is reliant on the availability of executables like wget for downloading the bot binary from the remote server, and if unavailable, it echo-loads a 1kb binary that functions like wget. Building the loader involves executing a build.sh script which in turn relies on gcc static libraries. Given that the Mirai attack involves downloading and/or executing binaries, libraries and scripts, one way to defend against such an attack involves preventing the unauthorized execution of these files. Additionally, malware may attempt to modify existing binaries or scripts on the target system to achieve the goals of infection. There are tools called file integrity checkers that are used to detect if a file has been altered accidentally or maliciously. The goal of these checkers is to verify that binaries, scripts, and libraries remain unchanged. In this project, we propose a method for protecting Linux-based IoT devices from executing malicious files and tampering with the integrity of existing files.

Our approach is to associate executable binaries and libraries with a digital signature which is checked by the kernel before the execution. Since this approach relies on kernel mechanisms to prevent execution, we assume that the kernel is secure against an attack by the malware. In the past,

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard. Understanding the Mirai Botnet. In *26th USENIX Security Symposium*, 2017.

[2] Cisco Annual Internet Report (2018–2023) White Paper. Available online: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed April 9, 2021

[3] Anna-senpai. [FREE] world's largest net:Mirai botnet, client, echo loader, CNC source code release. https://hackforums.net/ showthread.php?tid=5420472.

this assumption was not accurate, and malware exploitation of the kernel continues to grow [4,5]. However, we concentrate on a narrow subset of malware, namely Mirai and its variants that target IoT devices. The goal of Mirai is to infect as many IoT devices as possible. For Mirai and malware targeting IoT devices, compromising the kernel is not necessary when there are other vulnerable targets that can easily be infected at the user level. Therefore, this assumption that the kernel is secure against IoT botnet malware is sufficient for this project.

There have been efforts to protect the kernel [6,7,8]. One mechanism, the Integrity Measurement Architecture (IMA), was included in the Linux kernel version 2.6.11 and provides an attestation method to detect if a file has been altered [9]. IMA works by appraising files when they are opened, calculating the file's hash value, then storing it in memory. However, hashing alone cannot provide storing integrity guarantees. Code signing is the process of using a digital signature to sign executables and scripts in order to verify the software author and to ensure that the code has not been changed or tampered with since it was signed. Code signing can provide strong integrity guarantees -- the inability to tamper with files undetected, and the inability to add or run unauthorized executables [8].

Our approach draws inspiration from IMA as well as DigSig, a kernel module that checks RSA signatures of ELF binaries and libraries before executing them. [10]. Our implementation involves kernel modules and patches that add extended security attributes to files and verifies them before execution. [11]

Implementation

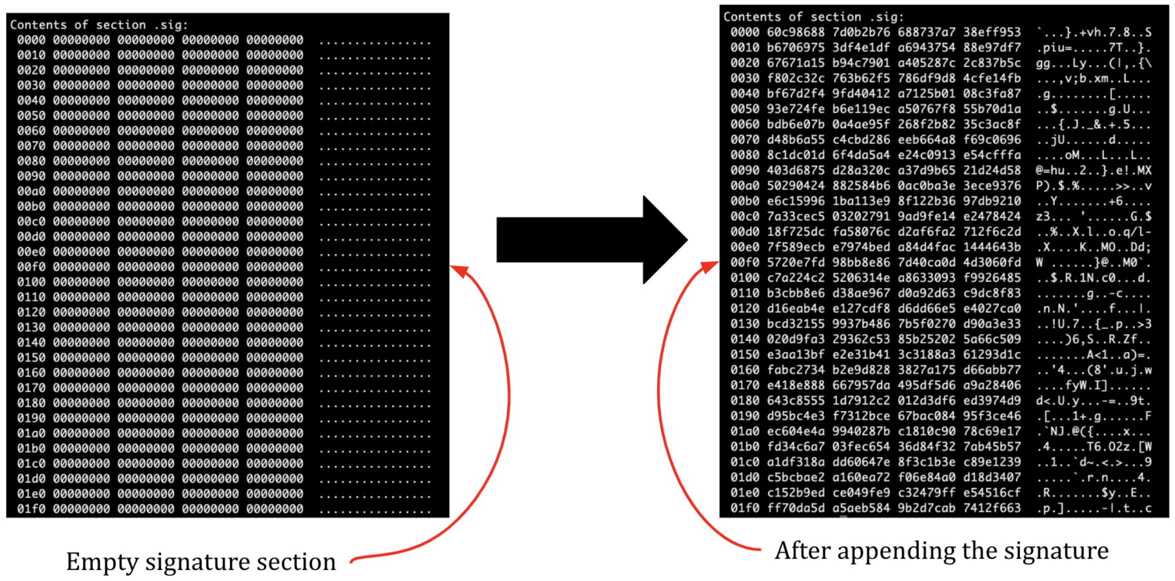
Experimental Setup

We chose OpenWRT, a Linux-based IOT device as our target platform. We used VirtualBox to run the x86-64 OpenWrt images in VM.

Generating signatures

We generated a private and a public key pair using openssl and used sha256 as the encryption algorithm. For the signature segment we computed the MD5 hash of the executable and encrypted the generated hash with the private key. The load_elf_binary() function in fs/binfmt_elf.c is responsible for loading the binaries before execution. We appended the signature to the ELF header.

[4] K. Chiang, L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In *HotBots*, 2007.
[5] J. Wyke. The ZeroAccess rootkit. *Sophos*, 2012.
[6] N. L. Petroni Jr., T. Fraser, J. Molina, W. A. Arbaugh.Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. 13th USENIX Security Symposium*, 2004.
[7] N. L. Petroni Jr., T. Fraser, A. Walters, W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. 15th USENIX Security Symposium*, 2006.
[8] L. van Doorn, G. Ballintign, W. A. Arbaugh. Signed executables for linux. Technical Report CS-TR-4259, University of Maryland, 2001.
[9] J. Corbet. The Integrity Measurement Architecture. Available online https://lwn.net/Articles/137306/. Accessed April 9, 2021.
[10]A Apvrille, D. Gordon. DigSig: Runtime Authentication of Binaries at Kernel Level. In *18th Large Installation System Administration Conference*, 2004.
[11] A Dolding, Peter. Re: [RFC 0/3] WhiteEgret LSM module. Available online https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1409537.html ,2012



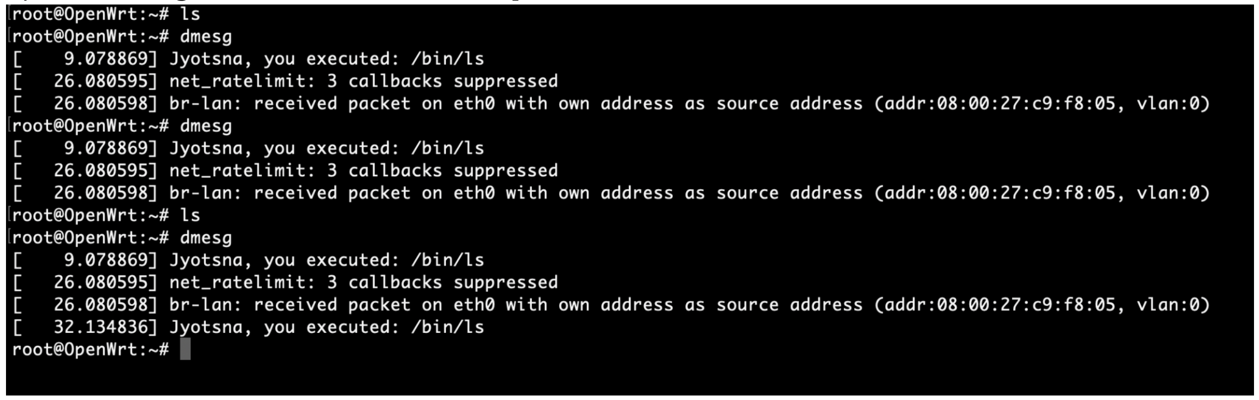
```
Contents of section .sig:
 0000 00000000 00000000 00000000 00000000  ................
 0010 00000000 00000000 00000000 00000000  ................
 0020 00000000 00000000 00000000 00000000  ................
 0030 00000000 00000000 00000000 00000000  ................
 0040 00000000 00000000 00000000 00000000  ................
 0050 00000000 00000000 00000000 00000000  ................
 0060 00000000 00000000 00000000 00000000  ................
 0070 00000000 00000000 00000000 00000000  ................
 0080 00000000 00000000 00000000 00000000  ................
 0090 00000000 00000000 00000000 00000000  ................
 00a0 00000000 00000000 00000000 00000000  ................
 00b0 00000000 00000000 00000000 00000000  ................
 00c0 00000000 00000000 00000000 00000000  ................
 00d0 00000000 00000000 00000000 00000000  ................
 00e0 00000000 00000000 00000000 00000000  ................
 00f0 00000000 00000000 00000000 00000000  ................
 0100 00000000 00000000 00000000 00000000  ................
 0110 00000000 00000000 00000000 00000000  ................
 0120 00000000 00000000 00000000 00000000  ................
 0130 00000000 00000000 00000000 00000000  ................
 0140 00000000 00000000 00000000 00000000  ................
 0150 00000000 00000000 00000000 00000000  ................
 0160 00000000 00000000 00000000 00000000  ................
 0170 00000000 00000000 00000000 00000000  ................
 0180 00000000 00000000 00000000 00000000  ................
 0190 00000000 00000000 00000000 00000000  ................
 01a0 00000000 00000000 00000000 00000000  ................
 01b0 00000000 00000000 00000000 00000000  ................
 01c0 00000000 00000000 00000000 00000000  ................
 01d0 00000000 00000000 00000000 00000000  ................
 01e0 00000000 00000000 00000000 00000000  ................
 01f0 00000000 00000000 00000000 00000000  ................
```

```
Contents of section .sig:
 0000 60c98688 7d0b2b76 688737a7 38eff953  `...}.+vh.7.8..S
 0010 b6706975 3df4e1df a6943754 88e97df7  .piu=.....7T..}.
 0020 67671a15 b94c7901 a405287c 2c837b5c  gg...Ly...(|,.{\
 0030 f802c32c 763b62f5 786df9d8 4cfe14fb  ...,v;b.xm..L...
 0040 bf67d2f4 9fd40412 a7125b01 08c3fa87  .g........[.....
 0050 93e724fe b6e119ec a50767f8 55b70d1a  ..$.......g.U...
 0060 bdb6e07b 0a4ae95f 268f2b82 35c3ac8f  ...{.J._&.+.5...
 0070 d48b6a55 c4cbd286 eeb664a8 f69c0696  ..jU......d.....
 0080 8c1dc01d 6f4da5a4 e24c0913 e54cfffa  ....oM...L...L..
 0090 403d6875 d28a320c a37d9b65 21d24d58  @=hu..2..}.e!.MX
 00a0 50290424 882584b6 0ac0ba3e 3ece9376  P).$.%.....>>..v
 00b0 e6c15996 1ba113e9 8f122b36 97db9210  ..Y.......+6....
 00c0 7a33cec5 03202791 9ad9fe14 e2478424  z3... '......G.$
 00d0 18f725dc fa58076c d2af6fa2 712f6c2d  ..%..X.l..o.q/l-
 00e0 7f589ecb e7974bed a84d4fac 1444643b  .X....K..MO..Dd;
 00f0 5720e7fd 98bb8e86 7d40ca0d 4d3060fd  W ......}@..M0`.
 0100 c7a224c2 5206314e a8633093 f9926485  ..$.R.1N.c0...d.
 0110 b3cbb8e6 d38ae967 d0a92d63 c9dc8f83  .......g..-c....
 0120 d16eab4e e127cdf8 d6dd66e5 e4027ca0  .n.N.'....f...|.
 0130 bcd32155 9937b486 7b5f0270 d90a3e33  ..!U.7..{_.p..>3
 0140 020d9fa3 29362c53 85b25202 5a66c509  ....)6,S..R.Zf..
 0150 e3aa13bf e2e31b41 3c3188a3 61293d1c  .......A<1..a)=.
 0160 fabc2734 b2e9d828 3827a175 d66abb77  ..'4...(8'.u.j.w
 0170 e418e888 667957da 495df5d6 a9a28406  ....fyW.I]......
 0180 643c8555 1d7912c2 012d3df6 ed3974d9  d<.U.y...-=..9t.
 0190 d95bc4e3 f7312bce 67bac084 95f3ce46  .[...1+.g......F
 01a0 ec604e4a 9940287b c1810c90 78c69e17  .`NJ.@({....x...
 01b0 fd34c6a7 03fec654 36d84f32 7ab45b57  .4.....T6.O2z.[W
 01c0 a1df318a dd60647e 8f3c1b3e c89e1239  ..1..`d~.<.>...9
 01d0 c5bcbae2 a160ea72 f06e84a0 d18d3407  .....`.r.n....4.
 01e0 c152b9ed ce049fe9 c32479ff e54516cf  .R.......$y..E..
 01f0 ff70da5d a5aeb584 9b2d7cab 7412f663  .p.].....-|.t..c
```

Figure 1: Signature section in the elf header of an executable

Getting access to the file contents

We patched the OpenWRT kernel such that we can detect if the executed program was /bin/ls. To identify our target binary "/bin/ls", we accessed "linux_binprm → file_name". Since linux_binprm structure holds the arguments that are used for loading binaries and "linux_binprm →file" holds the binary file content, we deduced that we could perform the signature check in any function inside fs/exec.c that gives us access to linux_binprm struct before execution.

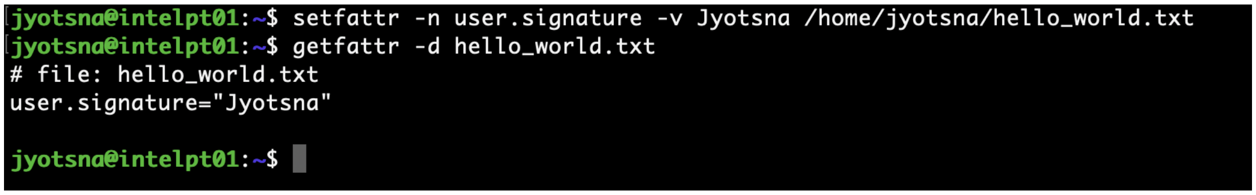


```
root@OpenWrt:~# ls
root@OpenWrt:~# dmesg
[    9.078869] Jyotsna, you executed: /bin/ls
[   26.080595] net_ratelimit: 3 callbacks suppressed
[   26.080598] br-lan: received packet on eth0 with own address as source address (addr:08:00:27:c9:f8:05, vlan:0)
root@OpenWrt:~# dmesg
[    9.078869] Jyotsna, you executed: /bin/ls
[   26.080595] net_ratelimit: 3 callbacks suppressed
[   26.080598] br-lan: received packet on eth0 with own address as source address (addr:08:00:27:c9:f8:05, vlan:0)
root@OpenWrt:~# ls
root@OpenWrt:~# dmesg
[    9.078869] Jyotsna, you executed: /bin/ls
[   26.080595] net_ratelimit: 3 callbacks suppressed
[   26.080598] br-lan: received packet on eth0 with own address as source address (addr:08:00:27:c9:f8:05, vlan:0)
[   32.134836] Jyotsna, you executed: /bin/ls
root@OpenWrt:~#
```

Figure 2: Kernel patch that identifies if the current executable is "/bin/ls"

Discovering Attributes

Using the ELF header to store the signature restricted our mechanism to ELF executables only. We discovered an alternative to store the files's signature, extended file attributes. Using extended file attributes we could expand our integrity check to libraries and config files, since extended attributes can be added to all types of files and not just ELF executables. Moreover, extended attributes can be accessed from the kernel using the "__vfs_getxattr" function.



```
jyotsna@intelpt01:~$ setfattr -n user.signature -v Jyotsna /home/jyotsna/hello_world.txt
jyotsna@intelpt01:~$ getfattr -d hello_world.txt
# file: hello_world.txt
user.signature="Jyotsna"

jyotsna@intelpt01:~$ 
```

Figure 3: Extended file attributes

Securing binaries with a key:

We modified the OpenWRT kernel to allow execution only if the key attribute of the executable matched the expected value "__EC700". We perform the key-check inside "bprm_execve". Inorder to get access to the binary file's key attribute we use "__vfs_getxattr" function, that takes the inode, dentry, attribute's name as its arguments. We extracted the dentry of the current executable from "bprm → file → f_path.dentry" and the inode struct from the dentry using "d_backing_inode" function. We then perform the check to see if the extracted key is the same as the expected key, if it is then we allow the execution otherwise we return an error.

Securing binaries with Hash over the file contents:

After validating our approach to whitelist binaries, we made our mechanism more robust by replacing the key with the sha1 hash of the binary file contents.

Allowing root to execute all files

We wanted to allow root to be able to execute all the binaries, so we extracted the uid of the process trying to execute the binary from the task_struct → cred → uid. If the uid is 0, we allow the execution irrespective of whether the executable has the correct hash-attribute or not.

Hash-attribute

We calculate the sha1 hash of the file contents. To make our mechanism more secure we enabled SELinux, that allowed us to use the security attributes, that can only be added/modified by the root user. We added the computed hash as a security attribute to the executable. Extracting the hash-attribute was similar to extracting the key attribute in the key approach.

Calculating the Hash in the kernel

We used the sha1 cipher algorithm, we first allocated a cipher handle using crypto_alloc_shash(), then we initialized the hash using crypto_shash_init(), we then read the file contents stored in "linux_binprm → file" in page-size chucks, and recalculated the hash with the data we just read, using crypto_shash_update().

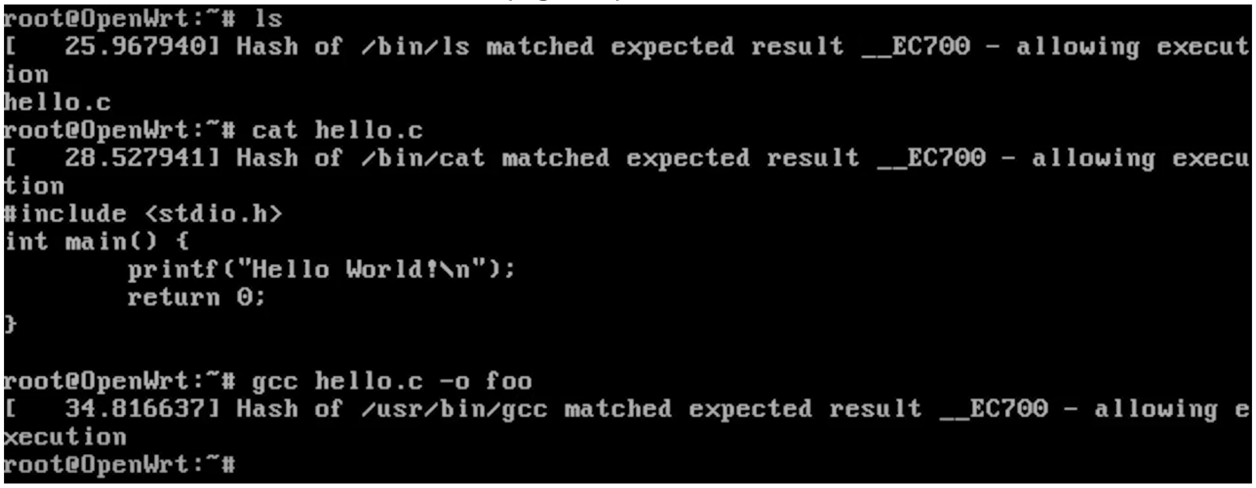
Comparison

After reading all of the file contents, we compare the calculated hash with the extracted hash-attribute. If the hash matches, we allow execution otherwise we terminate the process.

Evaluation

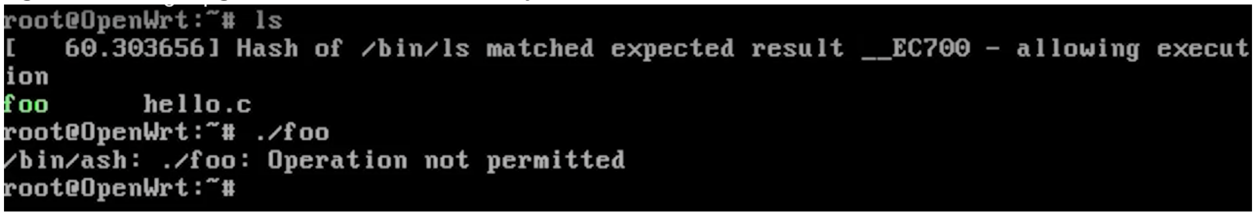
Evaluating the key approach in OpenWrt

Securing binaries using a key is a whitelisting approach. After compiling the modified OpenWrt kernel, we added a key attribute to a few executables (like /bin/ls, /bin/cat etc.). From Figure 3 we find that we can successfully execute executables that have the correct key as its attribute. If the executable has a missing attribute (Figure 4) or the attribute has an incorrect value the execution is terminated and an error is returned see (Figure 4)



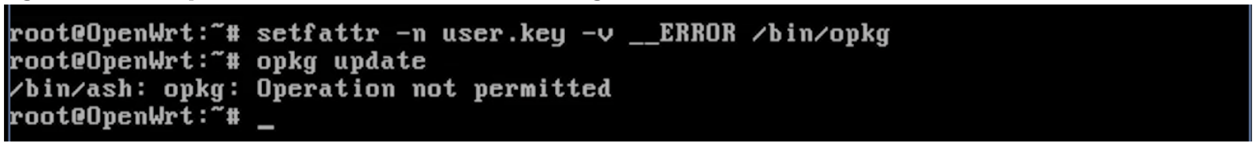
```
root@OpenWrt:~# ls
[   25.967940] Hash of /bin/ls matched expected result __EC700 - allowing execut
ion
hello.c
root@OpenWrt:~# cat hello.c
[   28.527941] Hash of /bin/cat matched expected result __EC700 - allowing execu
tion
#include <stdio.h>
int main() {
        printf("Hello World!\n");
        return 0;
}

root@OpenWrt:~# gcc hello.c -o foo
[   34.816637] Hash of /usr/bin/gcc matched expected result __EC700 - allowing e
xecution
root@OpenWrt:~#
```

Figure 3: Executing binaries with correct key attribute



```
root@OpenWrt:~# ls
[   60.303656] Hash of /bin/ls matched expected result __EC700 - allowing execut
ion
foo      hello.c
root@OpenWrt:~# ./foo
/bin/ash: ./foo: Operation not permitted
root@OpenWrt:~#
```

Figure 4: Attempt to execute binaries with missing attribute



```
root@OpenWrt:~# setfattr -n user.key -v __ERROR /bin/opkg
root@OpenWrt:~# opkg update
/bin/ash: opkg: Operation not permitted
root@OpenWrt:~# _
```

Figure 5: Attempt to execute binaries with an incorrect attribute

Limitations

There are few drawbacks with this approach. The attacker might change the contents of the file, since the attribute doesn't change after file modifications, he/she will still be able to execute his/her binary, since the file has the correct signature. Having said that, our intention was to use the key approach only as an intermediate step, to test our approach of attribute-extraction and its validation.

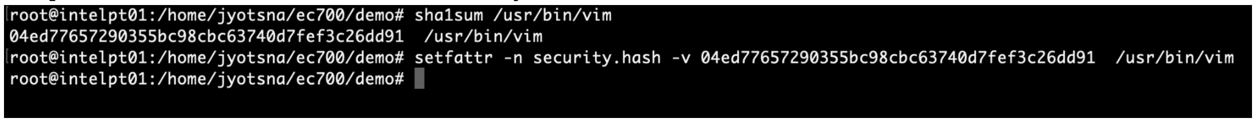
Evaluating the hashing approach

In OpenWRT

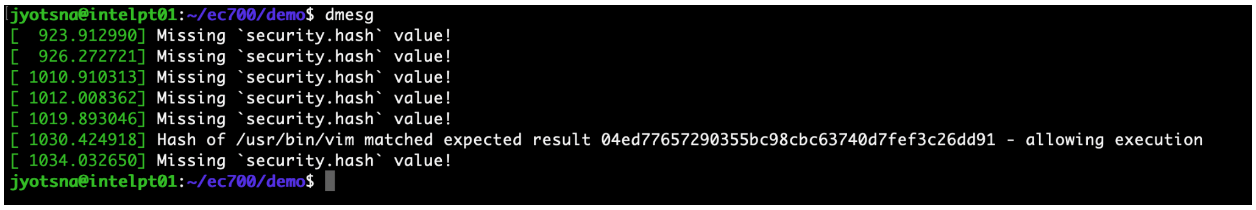
We compiled the patched kernel, and added the hash of the file contents as the attribute to the "/bin/ls". Executing "/bin/ls" resulted in a kernel panic. We were able to identify the root cause of kernel panic, by removing chunks of code from our file, the first time we make a call to the function from the crypto api, the system starts panicking.

In the Linux Kernel

We built a linux kernel from source, added our patch and recompiled the kernel to confirm our hunch that OpenWRT doesn't support crypto libraries. As we expected, our patched linux kernel did not panic and we were able to successfully test our modification.



```
root@intelpt01:/home/jyotsna/ec700/demo# sha1sum /usr/bin/vim
04ed77657290355bc98cbc63740d7fef3c26dd91  /usr/bin/vim
root@intelpt01:/home/jyotsna/ec700/demo# setfattr -n security.hash -v 04ed77657290355bc98cbc63740d7fef3c26dd91  /usr/bin/vim
root@intelpt01:/home/jyotsna/ec700/demo#
```



```
jyotsna@intelpt01:~/ec700/demo$ dmesg
[  923.912990] Missing `security.hash` value!
[  926.272721] Missing `security.hash` value!
[ 1010.910313] Missing `security.hash` value!
[ 1012.008362] Missing `security.hash` value!
[ 1019.893046] Missing `security.hash` value!
[ 1030.424918] Hash of /usr/bin/vim matched expected result 04ed77657290355bc98cbc63740d7fef3c26dd91 - allowing execution
[ 1034.032650] Missing `security.hash` value!
jyotsna@intelpt01:~/ec700/demo$
```

Limitations

There is a bug in the methodology we use to compute the hash over the file contents. Sometimes, we get a warning about a missing hash, even though we add an attribute with the correct hash to the executable. For testing purposes, we chose not to terminate the program in case the hash-attribute was missing, instead we print a "Missing `security.hash` value" message from the kernel.

Lessons Learned

We were able to

- Whitelist binaries in OpenWRT, by only allowing the execution of the executables with a valid key.
- Get familiar with OpenWRT build process and the generate custom images
- Patch OpenWRT Linux Kernel
- Learn how binaries get executed in the Linux kernel.
- Get familiar with Crypto API functions
- Calculate hash of the file contents in the kernel

Due to insufficient knowledge about OpenWRT at the early stages of the project, we couldn't get to debug the hashing approach. After our final presentation, we learned that Linux security modules must have a GPL license in order to use cryptographic algorithms in the Linux Kernel. The function that we relied on, crypto_alloc_shash(), is exported for use in kernel modules using the EXPORT_SYMBOL_GPL macro. If the kernel module is not GPL licensed, then the Linux kernel will assume that the module is proprietary, and will not permit it to access kernel functionalities. We initially aimed to develop a digital signature verification method that encrypts the computed hash with a generated private key. A public key would be used to decrypt hash extracted from the attribute in the kernel. Additionally, using a Linux Security Module enables extended security attributes which prevents unprivileged modifications. However, the difficulties we encountered working with the Linux kernel, and mismanagement of task/time prioritization resulted in falling short of our final goals.

References:

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard. Understanding the Mirai Botnet. In *26th USENIX Security Symposium*, 2017.

[2] Cisco Annual Internet Report (2018–2023) White Paper. Available online: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Accessed April 9, 2021
[3] Anna-senpai. [FREE] world's largest net:Mirai botnet, client, echo loader, CNC source code release. https://hackforums.net/ showthread.php?tid=5420472.
[4] K. Chiang, L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In HotBots, 2007.
[5] J. Wyke. The ZeroAccess rootkit. Sophos, 2012.
[6] N. L. Petroni Jr., T. Fraser, J. Molina, W. A. Arbaugh.Copilot - a coprocessor-based kernel runtime integrity monitor. In Proc. 13th USENIX Security Symposium, 2004.
[7] N. L. Petroni Jr., T. Fraser, A. Walters, W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In Proc. 15th USENIX Security Symposium, 2006.
[8] L. van Doorn, G. Ballintign, W. A. Arbaugh. Signed executables for linux. Technical Report CS-TR-4259, University of Maryland, 2001.
[9] J. Corbet. The Integrity Measurement Architecture. Available online https://lwn.net/Articles/137306/. Accessed April 9, 2001.
[10] Apvrille, D. Gordon. DigSig: Runtime Authentication of Binaries at Kernel Level. In *18th Large Installation System Administration Conference*, 2004.
[11] Dolding, Peter. Re: [RFC 0/3] WhiteEgret LSM module. Available online https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1409537.html ,2012
[12]