Vulnerability Analysis and Attacks on Intel Xeon Phi Coprocessor

Seungmin Lee[†*] and Ju-Won Park[†]
[†]Division of Supercomputing, Korea Institute of Science and Technology Information,
Daejeon 34141, Republic of Korea.
[*]Email: smlee76@kisti.re.kr

***Abstract*—The Intel Xeon Phi coprocessor is a PCIe based add-in card. Though it is prone to simple attacks, many high performance computing systems are constructed by combining CPUs and coprocessors. This paper describes two attacks that exploit vulnerabilities related to the boot process of coprocessor and ownership of offload user. Proof of concept codes are developed to demonstrate the problems and we suggest solutions, which adherence to the principle of least privilege, to mitigate problems caused by the characteristics of coprocessor.**

I. Introduction

Intel Xeon Phi coprocessor is becoming widely used in demanding consumer applications and high-performance computing(HPC) in recent years. However, there are vulnerabilities related to the Intel Xeon Phi coprocessor. This is the first report about security problems about the Intel Xeon Phi coprocessor to the best of our knowledge.

II. Background and Vulnerabilities

A. Vulnerability in the Creation of RAM File System

A file system image is pushed to the card as its RAM file system for the root file system, because Intel Xeon Phi coprocessor has no disk storage. The main problem is that all process constructing RAM file system image and transferring it occurs under root privilege. User data is controlled and accessed by each user even though binary executable files and system configuration files are managed by system administrator. Therefore, if the adversary misuses user data, the vulnerability exists in the copy process of user data when creating RAM file system image file.

B. Vulnerability of Supporting Offload Computing Model

User can specify the part of the code to execute in the coprocessor. It can be done at compile time and there is a COI(Coprocessor Offloading Infrastructure) daemon in the coprocessor to serve these request. The COI daemon should support the execution of user code even though the user who want to use resources of coprocessor doesn't exist [1]. It means there is no authentication in the offloading execution model and the user code executes under another special privileges. By default, COI daemon is executed under *micuser* privileges and all user request has the same privilege. Therefore, if adversary knows the execution of another user, s/he can intercept temporary files. In other words, the vulnerability exists in the offloading execution model because daemon inside coprocessor launches user code without user authentication and executes under special and unique privileges.

III. Attacks

We test attacks under Intel Xeon Phi coprocessors 7120 model with 3.6.38.8+mpss3.8.1 embedded linux kernel and MPSS version.

A. Attacking Process of Generating RAM File System

The main source of vulnerability when generating RAM file system image and transferring it occurs under root privilege. User data especially user configuration files, i.e. *.bashrc, .profile*, and ssh authentication files, i.e. *authorized_keys* can be used.

```
[dokto76@localhost dokto76]$ cat mic0.filelist
dir  /home                755  0  0
dir  /home/user1          755  0  0
dir  /home/user1/.ssh     700  0  0
file /home/user1/.ssh/authorized_keys  home/user1/.ssh/authorized_keys  600 500 500
file /home/user1/.ssh/id_rsa  home/user1/.ssh/id_rsa  600 500 500
file /home/user1/.ssh/id_rsa.pub  home/user1/.ssh/id_rsa.pub  600 500 500
......
```

Fig. 1. An Attack by Symbolic Linking User Authentication File to the System File

Figure 1 shows the simple attack method. Public key file for ssh authentication is modified to point out system file */etc/shadow*. This file has very important information about all users including password.

```
[dokto76@localhost ~]# ssh mic0
[dokto76@mic0 dokto76]# ls
[dokto76@mic0 dokto76]# cd .ssh
[dokto76@mic0 .ssh]# ls -l
total 8
-rw-------  1 dokto76  dokto76    411 May 23 19:01 authorized_keys
-rw-------  1 dokto76  dokto76   1675 May 23 19:01 id_rsa
-rw-------  1 dokto76  dokto76   1340 May 23 19:01 id_rsa.pub
```

Fig. 2. A Result of Symbolic Linking Attack

However, as shown in Figure 2 the contents of */etc/shadow* file is copied to the user file which is permitted to access by the user. This is serious problem even though this attack can be possible when the system of coprocessor is starting or restarting because generating RAM file system occurs at booting process for coprocessor and then the copy of user data can be possible.

ISBN: 1-60132-467-7, CSREA Press ©

B. Attacking Proxy for Offload Computing

As seen in the previous section, the offloading execution model has a vulnerability. Figure 3 shows an example code to verify the proof of concept. This code has a problem in the for loop because it misses the increasing of the loop index *i*.

```
#include <stdio.h>
#define N 10
int main()
{
      int i,j=0;
#pragma offload target (mic)
      for(i=0; i<N; )
            j++;
}
[dokto76@localhost mic_test]$ ./infini.x
offload error: process on the device 0 was unexpectedly terminated
```

Fig. 3. A Simple Infinite Loop Code and Termination by Attack

The line 6 *#pragma offload target (mic)* specifies that loop iterations will be executed in the coprocessor. As a result, the process forked in to coprocessor executes the loop part infinitely.

```
[dokto76@mic0 dokto76]$ ps -ef | grep mic
 3125 root      0:00 [kmicevent]
 3127 root      0:00 [kmicidle]
 4435 micuser  114:21 /bin/coi_daemon --coiuser=micuser
26096 micuser   0:14 /tmp/coi_procs/1/26096/offload_main
26107 dokto76   0:00 grep mic
```

Fig. 4. Process List in Coprocessor under micuser Privilege

Figure 4 shows the process list in the coprocessor. The process ID(*pid*) that executing previous code is 26096 and the owner of this process is *micuser*. The daemon which handles offloading execution request is also owned by the *micuser*. When the attack occurred to the process spawned in to coprocessor and terminated abnormally, the process on the host also terminated as shown in the last line of Figure 3.

IV. Solutions

The attack using symbolic link is possible when user data is handled under root privilege and when the system of coprocessor is starting or restarting. Therefore, the attack is impossible as we check the type of file before accessing it and isolate the scope of user files inside of one's home directory.

```
#!/bin/sh
cat $1 |\
while read line
do
   type=`echo $line | cut -d' ' -f1`      # parsing each line
   if [ $type == "file" ]; then
      file=`echo $line | cut -d' ' -f2`   # get file path information
      if [ -h $file ]; then               # check whether file type is symbolic
         sym=`readlink $file`             # get original file path
         if [ $sym != "" ]; then
            realpath=`dirname $sym`
            check_inside_home($HOME, $realpath)
         fi
      else
         echo "$line"
      fi
   fi
done
```

Fig. 5. Sanitizing filelist Configuration File

Figure 5 shows the implementation of this solution. It sanitizes the filelist configuration file because boot process copies data which is necessary when generating RAM file system image based on the filelist configuration file. This script should carry out before the generating process of RAM file system.

The second solution is applying the principle of least privileges, the privileges assigned to a process should be no more than the privileges required to perform the designed task [2]. This concept is applied to mitigate the attack.

```
#!/bin/sh
cat $1 |\
while read line
do
   type=`echo $line | cut -d' ' -f1`      # parsing each line
   if [ $type == "file" ]; then
      file=`echo $line | cut -d' ' -f2`   # get source file path
      target=`echo $line | cut -d' ' -f3`    # get target file
      user = `echo $line | cut -d' ' -f4`  # get user information
      exec = `su –session-command="cp $file $target"`
      #ignore error case
   fi
done
```

Fig. 6. Changing User Privileges

Figure 6 shows the implementation of this concept. As there is user information in the filelist configuration file, the process can determine the owner of each file line by line in the configuration file. Therefore it is possible to copy files under one's own privilege.

V. Conclusion

T. Shinagawa[3] presents a scheme called privileged code minimization that reduces the risk to setuid programs. And recent research [4] shows that GPU vulnerability can leak GPU data of other applications. However there is no security issue about Intel Xeon Phi coprocessor to the best of our knowledge.

There exists a vulnerable process when booting Intel Xeon Phi coprocessor and the attacker can acquire significant information easily. However, suggested solutions can mitigate problems caused by mis-configuration and characteristics in booting process of coprocessor.

References

[1] J. Jeffers and J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann, Walthman, MA, USA, 2013

[2] Mayfield, T., Roskos, J. E., Welke, S. R., and Boone, J. M., Integrity in automated information systems. Tech. Rep. 79-91, National Computer Security Center, September, 1991

[3] T. Shinagawa and K. Kono, Implementing a Secure Setuid Program, Parallel and Distributed Computing and Networks 2004

[4] S. Lee, Y. Kim, J. Kim, and J. Kim, Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014, pages 19-33, 2014