

The views, opinions and/or findings contained in this report are those of The MITRE Corporation and should not be construed as an official government position, policy, or decision, unless designated by other documentation.

©2013 The MITRE Corporation.   
All rights reserved.

Copernicus User/Installation Guide

Sam Cornwell, John Butterworth, Corey Kallenberg, Xeno Kovah

August 2014

Executive Summary

Copernicus is a tool to evaluate BIOS security on Intel PCs. A writable BIOS opens a system up to a very stealthy, powerful, and persistent backdoor. It allows the possibility that the attacker can brick the system (make it unbootable) on demand. Unlike attacks on software, this firmware attack cannot be recovered from without physical removal of the BIOS firmware chip.

The Copernicus agent takes two actions to evaluate BIOS vulnerability. First it dumps the contents of the flash chip. Second it checks access control registers that determine whether the BIOS is writable or not. This data can then be analyzed. The BIOS dump can be compared against a known clean copy to check for the presence of implants. The control registers can be be analyzed to determine of the BIOS is writable and therefore vulnerable.

This document includes information about installation, analysis, and the data collected.

Table of Contents

1 Running Copernicus 1

1.1 Obtaining BIOS Dump and BIOS Configuration File 1

1.2 Submitting BIOS Dump and BIOS Configuration Files 1

2 Analysis 2

2.1 Checking BIOS Write-protections 2

2.1.1 Prerequisites 2

2.1.2 Execution 2

2.2 Detecting BIOS changes 3

2.2.1 Prerequisites 3

2.2.2 Single model comparisons 3

2.2.3 Extracting UEFI SHA1 file hashes 4

2.2.4 Other options 4

Appendix A Python Script Prerequisites 6

Appendix B CSV Field Descriptions 7

B.1 General Information 7

B.2 BIOS Information 7

B.3 SMM Information 9

B.4 Trusted Computing Information 10

Appendix C Return Status Codes 11

Appendix D List of Abbreviations 13

List of Tables

Table B-1. General System Identification & Compatibility Information 5

Table B-2. BIOS Access Control Information 5

Table B-3. SMRAM Access Control Information 7

# 

# Running Copernicus

This section describes how to obtain a sample of Copernicus output from a single system in order to perform analysis. Additionally, we have provided a script that you can run if you would like to submit your anonymized data for research.

## Obtaining BIOS Dump and BIOS Configuration File

1. Ensure that copernicus{32/64}.sys, run.bat, and standalone.bat are in the same directory.
2. Run Copernicus by right-clicking standalone.bat and run as Administrator.
3. Allow the tool to run, which typically takes up to 30 seconds.
4. Verify that the output files Copernicus\_Log.txt, Copernicus\_CSV\_Out.csv, and Copernicus\_BIOS.bin appear in C:\.

It is possible that a Copernicus\_BIOS.bin file will not be generated on some machines because it will not run on chipset configurations we have not explicitly tested. If no .bin is generated, send the .txt and .csv files to the authors listed on the cover of this document. See the Analysis section to see how to evaluate the sample.

## Submitting BIOS Dump and BIOS Configuration Files

We have included a script that can be used to submit your BIOS dump and CSV configuration file to a public repository (<https://subzero.io/subzero/>) for research purposes. Prior to being saved in the repository, the hostname and asset tag will be removed from the CSV file to preserve the privacy of the submitter. In addition, the BIOS dump itself will not be publicly available; only certain information about the structure of the BIOS’s flash file system will be available on the public website. Before running this script, ensure that you have already successfully run Copernicus as described in the previous section. Ensure that you have submit.bat, post\_files.exe, and subzerocerts.pem in the same directory before running the following command:

submit.bat

Upon success, you should see output like the following:

{  
  "lookup": "https://subzero.io/subzero/lookup/abc...",  
  "report": "https://subzero.io/copernicus/report/123...",  
  "success": true  
}

The “lookup” is a link to information extracted from the BIOS dump. The “report” is a link to information extracted from the CSV file as well as an analysis of that information performed using the scripts described in the Analysis section of this document.

# Analysis

We have also included some post-processing scripts used to analyze and aggregate the data received from system deployments.

## Checking BIOS Write-protections

To evaluate the vulnerability of a system, MITRE created a script to determine the vulnerability of the system to BIOS and System Management Random Access Memory (SMRAM or System Management RAM)[[1]](#footnote-1) writes. When a system has a writable BIOS or SMRAM, we described this as “unlocked”, as seen in the counts of unlocked machines at the end of the script’s output. The script is given a directory containing CSV files from Copernicus, attempts to parse them all, and outputs its result. The script will only try to process files with the .csv extension and will ignore files that do not contain Copernicus data. The output is in the form of a table printed to the command line, as well as optionally written to a comma-separated values (CSV) file. It can display the results for each individual CSV file, or it can display results aggregated by version (e.g., manufacturer, model, and BIOS revision).

### Prerequisites

Python 2.7 is required and the Python module “docopt” are required. See Appendix A for installation instructions.

### Execution

An example execution of this script would be as follows:

python protections.py per-version -id -o output.csv <directory with Copernicus data>

The above command will display the results sorted per BIOS version. This command will also output the results to a CSV file called output.csv. An example of the output for the above command is as follows:

COUNT BIOS\_VENDOR PRODUCT\_NAME BIOS\_VERSION SMRAM\_UNLOCKED BIOS\_UNLOCKED

3 Dell Inc. Latitude E6400 A27 0 3

10 Dell Inc. Latitude E6400 A29 0 10

7 Dell Inc. Latitude E6400 A30 0 7

1 Dell Inc. Latitude E6400 A31 0 1

2 Dell Inc. Latitude E6400 A32 0 2

There are also several command line options for displaying more fine-grain results to determine why a BIOS or SMRAM is writable. To see the meaning of the individual command line options, execute the following:

python protections.py -h

## Detecting BIOS changes

In order to detect malicious changes to a BIOS, the MITRE team created a script to compare two BIOSes against each other. **This script is still the subject of active research and should be considered beta!** The script can be used to compare a BIOS with a “known good” image that has been extracted from a vendor update utility. Absent a known good, it can be used to compare two files from two different physical machines that are indicated to be of the same model and BIOS revision (this can be checked in the Copernicus CSV output file).

The script bios\_diff.py displays fine-grain results in order to determine why a BIOS or SMRAM is writable.

### Prerequisites

Python 2.7 and the Python module “docopt” and “pefile” are required. See Appendix A for installation instructions.

These scripts also requires a modified version of the EFIPWN tool[[2]](#footnote-2) which is included in this package. You must also handle EFIPWN prerequisites as described in the EFIPWN/README.txt file. The EFIPWN folder must either be in the same directory as bios\_diff.py, or be specified with the –e option.

### Single model comparisons

The following is an example execution of this script which will diff two BIOS files:

python bios\_diff.py –dapn e6430A03.bin e6430A03\_haxed.bin

Two BIOS that differ based on Copernicus’ examination will produce output like the following:

temp/e6430A03.bin/fv5/e9312938-e56b-4614-a252-cf7d2f377e26/PE32\_944 (AmiTcgPlatformPeiBeforeMem)

1 unique bytes out of 2976

1036,1036

PE Information:

Section .text

RVA 0x40c

VA 0xffe6d090

temp/e6430A03\_haxed.bin/fv5/e9312938-e56b-4614-a252-cf7d2f377e26/PE32\_944 (AmiTcgPlatformPeiBeforeMem)

1 unique bytes out of 2976

1036,1036

PE Information:

Section .text

RVA 0x40c

VA 0xffe6d090

This is indicating that an analyst could take the two PE files found in the two directories, open them in IDA pro, jump to virtual address 0xffe6d090 and see the differences in the assembly between each. (In this case there is a single byte change, making the entry point a return instruction, so that the AmiTcgPlatformPeiBeforeMem file is never able to execute any code to initialize the TPM.)

If no changes are detected, the bios\_diff.py script will produce no output.

The bios\_diff.py script may also accept files in the following forms:

python bios\_diff.py arg1 arg2 ... <directory>

or

python bios\_diff.py arg1 arg2 ... <file> <directory>

In the second case, the <file> is a single BIOS file which is treated as the clean BIOS and is diffed against every file in the given <directory>. In the first case, the first file found in the given <directory> is chosen as the clean BIOS. This file is then diffed against all of the other files in the directory.

For integrity-checking purposes, it is recommended that all BIOS files tested using this script are of the same vendor, model, and BIOS revision.

### Extracting UEFI SHA1 file hashes

The following are examples of recommended usages of this script:

python bios\_diff.py -s <file>

This will produce output like the following (where the bold text are the SHA1 hashes):

temp/e6430A03.bin/fv2/ae717c2f-1a42-4f2b-8861-78b79ca07e07/csc/fvsc/fv3/bdfcc092-36a4-4668-bafe-ec8f1b02a28a/csc/PE32\_94

**ef361e5b0e37720d8d11eb6adfebd264302deec8**

temp/e6430A03.bin/fv2/ae717c2f-1a42-4f2b-8861-78b79ca07e07/csc/fvsc/fv3/878ac2cc-5343-46f2-b563-51f89daf56ba/rfc

**c952d953d11def92ef74405c84671ff2ff709efa**

temp/e6430A03.bin/fv2/ae717c2f-1a42-4f2b-8861-78b79ca07e07/csc/fvsc/fv3/bc3245bd-b982-4f55-9f79-056ad7e987c5/DXE\_DEPEX\_243

**27622185e51f6b8bf1f3360ecbee31c4647616aa**

This can potentially be used for quick pre-processing on endpoints if you do not want to bring the entire BIOS dump back to a server for diffing. Additionally, <file> may be either a file or a directory. If it is a directory, the script will process all BIOS files in that directory.

### Other options

To see the current meaning of the individual command line options, execute:

python bios\_diff.py –h

###### Python Script Prerequisites

The following is a table of the various prerequisites for running our analysis scripts, which are all written in Python.

|  |  |
| --- | --- |
| Name | Python 2.7.x |
| Website | <https://www.python.org/downloads/> |
| Description | Our scripts must be run with Python 2.7.x. Any version of Python 2.7 should work, but as of this writing, Python 2.7.8 is available and is known to work. The installation package appropriate for your OS should be available at the above website. |

|  |  |
| --- | --- |
| Name | Docopt |
| Website | <https://github.com/docopt/docopt> |
| Description | The Python module “docopt” is required for all of our scripts as of this writing. You can install it using pip or easy\_install, or by following the installation instructions on the above website. |

|  |  |
| --- | --- |
| Name | Pefile |
| Website | <https://code.google.com/p/pefile> |
| Description | The python module “pefile” python module is required for our BIOS diffing scripts. It can be installed using pip or easy\_install, or by downloading the most recent zip file from the above website and following the included README. |

###### CSV Field Descriptions

The following are descriptions of fields reported by Copernicus.

General Information

For stability reasons, we do not allow Copernicus to run on chipsets that we have not tested on. Therefore this information is used to identify whether a system has been tested on, as well as to give a human-readable sense of what family a chipset belongs to.

Table B-. General System Identification & Compatibility Information

|  |  |
| --- | --- |
| Field Name | Field Description |
| ICH\_VENDOR\_ID, ICH\_DEVICE\_ID | These are the PCI Vendor and Device IDs used to identify the ICH or PCH on the system. |
| MCH\_VENDOR\_ID, MCH\_DEVICE\_ID | These are the PCI Vendor and Device IDs used to identify the MCH on the system. |
| ICH\_FAMILY | The ICH/PCH families supported by Copernicus are ICH7, ICH8, ICH9, ICH10, and PCH-based chipsets. It is determined from the ICH\_DEVICE\_ID. It is included for ease of quickly determining which other Copernicus fields are relevant to a particular machine. |
| MCH\_FAMILY | This can be used to determine the memory controller (MCH) which on the system. In modern "Core i" processors, this is tied to the generation of the processor since the memory controller is located in the CPU in these models. |
| COPERNICUS\_TIMEDATESTAMP | This value is the TimeDateStamp which is extracted from the PE headers of the Copernicus binary. It can be used to determine the version of Copernicus run on the system. |
| HOSTNAME | The hostname according to the Windows registry. |
| BIOS\_VENDOR | The BIOS vendor. Usually set to things like “Dell Inc.”, “Hewlett-Packard”, or “LENOVO”. But in some cases, the vendor leaves the 3rd party BIOS maker’s designation there like “American Megatrends Inc.” or “Phoenix Technologies Ltd.” |
| PRODUCT\_NAME | Often an easily understandable string corresponding to the hardware model, such as “Latitude E6430” or “HP EliteBook 2760p”. But from some vendors like Lenovo, a less useful designation like “3448AVU” which requires Googling to look up. |
| BIOS\_VERSION | The revision number of the BIOS according to the manufacturer. |
| ASSET\_TAG | The SMBIOS asset tag, if set. Some organizations set this value as part of their system deployment and asset management process. |
| COP\_RETURN\_STATUS | Indicates whether any information was not able to be collected due to being unsupported or due to errors. |

BIOS Information

The BIOS\_CNTL bits provide the broadest and highest level of protection of the flash chip. When properly set, the entire flash chip is write-protected. The PR (protected region) registers can offer the same functionality, but also provide read-protection and finer granularity while still being able to protect the entire flash chip. The FLOCKDN must be set in order to properly enforce PR protections.

Table B-. BIOS Access Control Information

|  |  |
| --- | --- |
| Field Name | Field Description |
| FREGx | These registers are only applicable in descriptor mode. They contain the base and limit addresses for the five possible regions in the flash chip. A chip may not implement all regions, but the supported regions (0 through 4) are: Flash Descriptor, BIOS, Intel ME, Gigabit Ethernet, and Platform Data. MITRE is current primarily focused on the BIOS region. |
| FLOCKDN | Once set, certain configuration registers become "locked down" (read only). This is primarily used for locking down security configurations. Once set, this bit cannot be cleared without a system reset. This is a bit is the SPI HSFS register. |
| FLASH\_DESCRIPTOR\_MODE | This parameter indicates whether the flash chip is running in descriptor mode. In descriptor mode, the flash chip is segmented into regions described by the FREG registers. In non-descriptor mode, the chip is considered to be one monolithic region. It is determined from a bit in the HSFS register. |
| BIOSWE\_LOCK | When this is True, it indicates that the BIOSWE bit is disabled and locked down by SMM mode. This means that the BLE bit is set and SMM is blocking any attempts to turn on BIOSWE. BIOSWE and BLE are found in the BIOS\_CNTL register, which is described below. |
| FLASH\_DESCRIPTOR\_OVERRIDE | This is true whenever the flash descriptor override is enabled. While this is enabled, the FLMSTR1 protections are overridden with the FRAP protections. Note that this override is a physical pin that overrides the FLMSTR1 protections and it cannot be set by software. |
| PRx | These registers allow for hardware-enforced read or write protection of arbitrary regions of the flash chip. On ICH7-based chipsets, PR3 and PR4 are not applicable. These registers are read-only when FLOCKDN is set. |
| PREOP, OPMENU, OPTYPE, UVSCC\_ERASE, and LVSCC\_ERASE | Collectively, these registers specify the operations that may be performed on the flash chip using the SPI software sequencing functionality. These operations are various types of read and write operations. Flash chips in non-descriptor mode only support software sequencing, whereas flash chips in descriptor mode support both hardware sequencing and software sequencing functionality. These registers are only interesting to us on (usually older) chips that are running in non-descriptor mode. This is because the chip cannot be written if these registers do declare any write operations, therefore the register *can* provide write-protection functionality. A chip operating in descriptor mode, however, will always have write operations available to use (barring any other write-protection mechanisms). PREOP, OPMENU, and OPTYPE become read-only when FLOCKDN is set. |
| FRAP | This register is only applicable in descriptor mode. When FLASH\_DESCRIPTOR\_OVERRIDE is enabled, the permissions in FLMSTR1 are overridden by the permissions in this register. |
| FLMSTRx | These registers are only applicable in descriptor mode. There are three of these registers, one each for flash regions 1, 2, and 3. The bits in these registers control the ability of code in these regions to read or write to any of the other regions. Each of these registers provides enough bits to individually control readability and writability for each of the five regions. FLMSTR1 specifies permissions for the Host CPU as well as the BIOS. |
| BIOS\_CNTL | This contains the BIOSWE and BLE bits that are used to determine and protect writability of the entire flash chip. BIOSWE enables writability to the flash chip. When BLE is set, then SMM receives an interrupt whenever there is an attempt to enable BIOSWE, and SMM then has the ability to block the change (this is up to the vendor who provides the SMM code). |
| FLASH\_CHIP\_SIZE | This is the size of the flash chip which contains the BIOS. |

SMM Information

There are three potential physical memory regions that can to be used as SMM RAM (SMRAM). These are the compatibility segment (CSEG), high memory segment (HSEG), and top of memory segment (TSEG). The enabled regions are determined by reading the SMRAMC and ESMRAMC registers. SMM runs at a higher privilege level than even ring 0 and so these regions should be protected from reads and writes when the CPU is not running in SMM mode. These protections are provided by setting the D\_LCK bit and using the SMRR registers. Due to caching features on modern CPUs, the SMRAM may be vulnerable even if D\_LCK protections are enabled (e.g. through CPU cache poisoning[[3]](#footnote-3)). Therefore the SMRR registers should be used, if they are available, to lock down the entire SMRAM region. The implementation of the SMRR registers in the CPU, as opposed to the chipset, allows caching vulnerabilities to be mitigated.

Table B-. SMRAM Access Control Information

|  |  |
| --- | --- |
| Field Name | Field Description |
| SMRAMC | This register allows SMRAM to be enabled. It also features the D\_LCK bit. When set, reads and writes to the SMRAM regions are blocked by the chipset. In addition, most of SMRAMC and ESMRAMC become read-only when D\_LCK is set, and D\_LCK can only be cleared again by a system reset. |
| ESMRAMC | This register allows HSEG, TSEG, or both to be enabled. When HSEG is enabled, CSEG is effectively disabled by the chipset because HSEG memory addresses map to the same DRAM region that CSEG would have mapped to. This register also features the bits which can be set to enable caching of HSEG and TSEG. Cacheability of HSEG is the advantage of using it rather than CSEG. |
| SMRR\_PHYSMASK, SMRR\_PHYSBASE | These registers combined specify a single contiguous region of physical memory, which is only readable and writable in SMM mode. They also specify cacheability options, which override any other system caching settings. These registers are only writable in SMM mode. |
| SMRR\_SUPPORTED | Indicates whether this machine supports System Management Range Registers. If this is False, then the SMRR\_PHYSMASK and SMRR\_PHYSBASE field values are meaningless. |
| SMI\_EN | There are many different types of events that can generate SMIs. This register allows for disabling/enabling different types of SMIs. The bit of this register that we are particularly interested in is the GBL\_SMI\_EN bit. This bit gives the capability to disable all types of SMIs. |
| GEN\_PMCON\_1 | This register contains the SMI\_LOCK bit, which is used to lock down the GBL\_SMI\_EN bit in the SMI\_EN register. |

Trusted Computing Information

Copernicus 1 is a “best effort” system for BIOS inspection. It is possible for attackers at a variety of vantage points to subvert Copernicus’ measurement, just as it is possible against all other currently deployed security software. By contrast, Copernicus 2 utilizes Intel Trusted Execution Technology (TXT) in order to provide measurements that are resistant to tampering by a wide range of adversaries. All high-end Intel chips support TXT, but some low-end ones do not. Therefore the following information is to check whether the system will support TXT. Additionally, when the chip supports TXT it is often configured to be off by default, and must be explicitly turned on in the BIOS configuration. The information will indicate whether the TXT support is both present and turned on.

TXT requires the presence of an activated Trusted Platform Module (TPM). Any system that has a BIOS option to enable TXT will also have a BIOS option to enable the TPM. Often this will be split into two options for the TPM being enabled, and the TPM being activated. Often you will need to first enable the TPM, then reboot, and activate it. Once the TPM is enabled & activated, and TXT is supported and activated, Copernicus 2 can potentially be used on the system. (Note: some older systems, which ostensibly support TXT, have bugs that prevent it from working. We are creating a list of such systems as we find them.)

Table B-. Trusted Computing Information

|  |  |
| --- | --- |
| Field Name | Field Description |
| CPUID\_1\_ECX | The Intel “CPUID” instruction can query information about the CPU. In this case we return the value of register ECX when CPUID is called with register EAX set to 1. Bit 6 of this field indicates support for “Safer Mode Extensions” (SMX), which is a term for the TXT instruction set. |
| IA32\_FEATURE\_CONTROL\_MSR | This is Intel Model Specific Register (MSR) number 0x3A. Bit 15 indicates whether the TXT “SENTER” instruction necessary to start TXT is available. Bits 14:8 indicate support for sub-capabilities of SENTER. |
| TPM\_PRESENT | Indicates if any TPM was found on the system |
| TPM\_ENABLED | Indicates whether the TPM is in the “enabled” state. (Both enabled & activated are needed to use the TPM.) |
| TPM\_ACTIVATED | Indicates whether the TPM is in the “activated” state. (Both enabled & activated are needed to use the TPM.) |
| TPM\_VENDORID | Indicates the manufacturer of the TPM. Can be looked up by Googling “TCG Vendor ID Registry” and selecting the result from http://www.trustedcomputinggroup.org/ |

###### Return Status Codes

The following are the status codes that Copernicus can return upon exit. They are used to provide information on what parts (if any) of the application failed to execute. Several of these status codes can be bitwise ORed together. The exit status code is printed to both the log file and the CSV file.

Return Status 0x00000000

All Copernicus measurements ran without error (exit success).

Return Status 0x00000001

Copernicus was unable to detect or identify the system’s Platform/IO controller hub.

Return Status 0x00000002

Copernicus was unable to detect or identify the system’s memory controller.

Return Status 0x00000004

Copernicus was unable to map the SPI Configuration space (SPIBAR) to memory.

Return Status 0x00000008

Copernicus was unable to determine the size of the BIOS flash chip.

Return Status 0x00000010

Copernicus was unable to directly read the contents of the BIOS flash chip. If there is a .bin file, this flag indicates that it will only contain the BIOS portion of the flash chip that is mapped into memory.

Return Status 0x00000020

Copernicus was unable to read the portion of the BIOS flash that is mapped to memory. This only applies if it was also unable to read directly from BIOS flash.

Return Status 0x00000040

Copernicus was unable to write one of the csv, log, or bin files.

Return Status 0x00000080

Copernicus was unable to create one of the csv, log, or bin files.

Return Status 0x00000100

Copernicus was unable to find the SMBIOS entry point to collect SMBIOS information. Currently, this error will typically be seen on systems running Windows 8 (until the issue is resolved).

Return Status 0x00000200

Copernicus was unable to map the SMBIOS information into memory.

Return Status 0x00000400

This status code is deprecated.

Return Status 0x00000800

Copernicus failed to collect the host data due to memory allocation failure.

Return Status 0x00001000

Deprecated. System Management Range Register (SMRR) is unsupported on this system. This common on older systems where the hardware did not yet have SMRRs.

Return Status 0x00002000

Copernicus was unable to retrieve the hostname of the system.

Return Status 0x00004000

Copernicus was unable to map the Option ROM memory space.

Return Status 0x00008000

This status code is reserved.

Return Status 0x00010000

Copernicus was unable to collect the CMOS data.

Return Status 0x00020000

Copernicus was unable to locate the SMRAMC and ESMRAMC fields.

Return Status 0x00040000

General memory allocation failure.

Return Status 0x80000000

Generic failure case.

###### List of Abbreviations

BIOS Basic Input/Output System

PCI Peripheral Component Interconnect

RAM Random Access Memory

ROM Read Only Memory

SMM System Management Mode

SMRAM System Management RAM

SMRR System Management Range Register

SPI SCSI Parallel Interface

URL Universal Resource Locator

VM Virtual Machine

1. SMRAM is a portion of the systems memory used by the processor to store System Management Mode (SMM) code and data. SMM is an Intel CPU execution mode that the BIOS sets up. Code running in SMM is capable of hiding its contents from code running in other CPU modes, while simultaneously having access to all RAM. It is therefore an execution mode that can inspect all other code, but no other code can inspect it. [↑](#footnote-ref-1)
2. The original version of the EFIPWN tool can be obtained from <https://github.com/G33KatWork/EFIPWN>. [↑](#footnote-ref-2)
3. http://conference.hitb.org/hitbsecconf2013kul/materials/D1T1%20-%20Kallenberg,%20Kovah,%20Butterworth%20-%20Defeating%20Signed%20BIOS%20Enforcement.pdf [↑](#footnote-ref-3)