CHIPSEC

version

Platform Security Assesment Framework

December 21, 2023

Contents

CHIPSEC **1**
- Contact 1
- Download CHIPSEC 2
  - GitHub repository 2
  - Releases 2
  - Python 2
- Installation 2
  - Linux Installation 3
    - Creating a Live Linux image 3
    - Installing Kali Linux 3
    - Prerequisites 3
    - Installing CHIPSEC 4
    - Building CHIPSEC 4
    - Run CHIPSEC 4
  - DAL Windows Installation 4
    - Prerequisites 4
    - Building 5
  - Windows Installation 5
    - Install CHIPSEC Dependencies 5
    - Building 6
    - Turn off kernel driver signature checks 6
    - Alternate Build Methods 7
  - Building a Bootable USB drive with UEFI Shell (x64) 8
    - Installing CHIPSEC 8
    - Run CHIPSEC in UEFI Shell 9
    - Building UEFI Python 3.6.8 (optional) 9
- Using CHIPSEC 9
  - Interpreting results 9
    - Results 10
    - Automated Tests 10
    - Tools 13
  - Running CHIPSEC 13
    - Running in Shell 14
    - Using as a Python Package 14
    - chipsec_main options 15
    - chipsec_util options 15
- Module & Command Development 15
  - Architecture Overview 15
    - Core components 16

Commands 16
chipsec.utilcmd package 17
Submodules 17
chipsec.utilcmd.acpi_cmd module 17
chipsec.utilcmd.chipset_cmd module 17
chipsec.utilcmd.cmos_cmd module 17
chipsec.utilcmd.config_cmd module 18
chipsec.utilcmd.cpu_cmd module 19
chipsec.utilcmd.decode_cmd module 19
chipsec.utilcmd.deltas_cmd module 20
chipsec.utilcmd.desc_cmd module 20
chipsec.utilcmd.ec_cmd module 22
chipsec.utilcmd.igd_cmd module 22
chipsec.utilcmd.interrupts_cmd module 23
chipsec.utilcmd.io_cmd module 24
chipsec.utilcmd.iommu_cmd module 24
chipsec.utilcmd.lock_check_cmd module 25
chipsec.utilcmd.mem_cmd module 26
chipsec.utilcmd.mmcfg_base_cmd module 26
chipsec.utilcmd.mmcfg_cmd module 27
chipsec.utilcmd.mmio_cmd module 27
chipsec.utilcmd.msgbus_cmd module 28
chipsec.utilcmd.msr_cmd module 29
chipsec.utilcmd.pci_cmd module 29
chipsec.utilcmd.reg_cmd module 30
chipsec.utilcmd.smbios_cmd module 31
chipsec.utilcmd.smbus_cmd module 31
chipsec.utilcmd.spd_cmd module 31
chipsec.utilcmd.spi_cmd module 32
chipsec.utilcmd.spidesc_cmd module 33
chipsec.utilcmd.tpm_cmd module 33
chipsec.utilcmd.txt_cmd module 34
chipsec.utilcmd.ucode_cmd module 35
chipsec.utilcmd.uefi_cmd module 35
chipsec.utilcmd.vmem_cmd module 36
chipsec.utilcmd.vmm_cmd module 37
Module contents 38
HAL (Hardware Abstraction Layer) 38
chipsec.hal package 38
Submodules 38
chipsec.hal.acpi module 38

chipsec.hal.acpi_tables module 39
chipsec.hal.cmos module 43
chipsec.hal.cpu module 44
chipsec.hal.cpuid module 45
chipsec.hal.ec module 45
chipsec.hal.hal_base module 46
chipsec.hal.igd module 46
chipsec.hal.interrupts module 47
chipsec.hal.io module 47
chipsec.hal.iobar module 48
chipsec.hal.iommu module 48
chipsec.hal.locks module 49
chipsec.hal.mmio module 49
chipsec.hal.msgbus module 51
chipsec.hal.msr module 53
chipsec.hal.paging module 53
chipsec.hal.pci module 56
chipsec.hal.pcidb module 57
chipsec.hal.physmem module 57
chipsec.hal.smbios module 58
chipsec.hal.smbus module 59
chipsec.hal.spd module 59
chipsec.hal.spi module 61
chipsec.hal.spi_descriptor module 62
chipsec.hal.spi_jedec_ids module 62
chipsec.hal.spi_uefi module 63
chipsec.hal.tpm module 64
chipsec.hal.tpm12_commands module 65
chipsec.hal.tpm_eventlog module 65
chipsec.hal.ucode module 66
chipsec.hal.uefi module 66
chipsec.hal.uefi_common module 68
chipsec.hal.uefi_compression module 73
chipsec.hal.uefi_fv module 73
chipsec.hal.uefi_platform module 74
chipsec.hal.uefi_search module 77
chipsec.hal.virtmem module 77
chipsec.hal.vmm module 78
Module contents 78
Fuzzing 78
chipsec.fuzzing package 78

Submodules 78
chipsec.fuzzing.primitives module 78
Module contents 81
CHIPSEC_MAIN Program Flow 81
CHIPSEC_UTIL Program Flow 81
Auxiliary components 81
Executable build scripts 81
Configuration Files 81
Configuration File Example 82
List of Cfg components 82
Writing Your Own Modules 82
OS Helpers and Drivers 83
Mostly invoked by HAL modules 83
Helpers import from BaseHelper 83
Create a New Helper 84
Example 84
Helper components 84
chipsec.helper package 84
Subpackages 84
chipsec.helper.dal package 84
Submodules 84
chipsec.helper.dal.dalhelper module 84
Module contents 86
chipsec.helper.efi package 86
Submodules 86
chipsec.helper.efi.efihelper module 86
Module contents 88
chipsec.helper.linux package 88
Submodules 88
chipsec.helper.linux.linuxhelper module 88
Module contents 91
chipsec.helper.linuxnative package 91
Submodules 91
chipsec.helper.linuxnative.cpuid module 91
chipsec.helper.linuxnative.legacy_pci module 91
chipsec.helper.linuxnative.linuxnativehelper module 92
Module contents 94
chipsec.helper.windows package 94
Submodules 94
chipsec.helper.windows.windowshelper module 94
Module contents 94

Submodules 94
chipsec.helper.basehelper module 94
chipsec.helper.nonehelper module 96
chipsec.helper.oshelper module 97
Module contents 98
Methods for Platform Detection 98
Uses PCI VID and DID to detect processor and PCH 98
Chip information located in `chipsec/chipset.py`. 98
Platform Configuration Options 99
Sample module code template 99
CHIPSEC Modules 100
Modules 105
chipsec.modules package 105
Subpackages 105
chipsec.modules.bdw package 105
Module contents 105
chipsec.modules.byt package 105
Module contents 105
chipsec.modules.common package 105
Subpackages 105
chipsec.modules.common.cpu package 105
Submodules 105
chipsec.modules.common.cpu.cpu_info module 105
chipsec.modules.common.cpu.ia_untrusted module 106
chipsec.modules.common.cpu.spectre_v2 module 106
Module contents 108
chipsec.modules.common.secureboot package 108
Submodules 108
chipsec.modules.common.secureboot.variables module 108
Module contents 109
chipsec.modules.common.uefi package 109
Submodules 109
chipsec.modules.common.uefi.access_uefispec module 109
chipsec.modules.common.uefi.s3bootscript module 110
Module contents 111
Submodules 111
chipsec.modules.common.bios_kbrd_buffer module 111
chipsec.modules.common.bios_smi module 112
chipsec.modules.common.bios_ts module 112
chipsec.modules.common.bios_wp module 113
chipsec.modules.common.debugenabled module 114

chipsec.modules.common.ia32cfg module 115
chipsec.modules.common.me_mfg_mode module 115
chipsec.modules.common.memconfig module 116
chipsec.modules.common.memlock module 117
chipsec.modules.common.remap module 118
chipsec.modules.common.rtclock module 119
chipsec.modules.common.sgx_check module 119
chipsec.modules.common.smm module 120
chipsec.modules.common.smm_code_chk module 120
chipsec.modules.common.smm_dma module 121
chipsec.modules.common.smrr module 122
chipsec.modules.common.spd_wd module 123
chipsec.modules.common.spi_access module 124
chipsec.modules.common.spi_desc module 124
chipsec.modules.common.spi_fdopss module 125
chipsec.modules.common.spi_lock module 125
Module contents 127
chipsec.modules.hsw package 127
Module contents 127
chipsec.modules.ivb package 127
Module contents 127
chipsec.modules.snb package 127
Module contents 127
chipsec.modules.tools package 127
Subpackages 127
chipsec.modules.tools.cpu package 127
Submodules 127
chipsec.modules.tools.cpu.sinkhole module 127
Module contents 128
chipsec.modules.tools.secureboot package 128
Submodules 128
chipsec.modules.tools.secureboot.te module 128
Module contents 130
chipsec.modules.tools.smm package 130
Submodules 130
chipsec.modules.tools.smm.rogue_mmio_bar module 130
chipsec.modules.tools.smm.smm_ptr module 131
Module contents 132
chipsec.modules.tools.uefi package 132
Submodules 132
chipsec.modules.tools.uefi.reputation module 132

chipsec.modules.tools.uefi.s3script_modify module 133
chipsec.modules.tools.uefi.scan_blocked module 135
chipsec.modules.tools.uefi.scan_image module 135
chipsec.modules.tools.uefi.uefivar_fuzz module 136
Module contents 138
chipsec.modules.tools.vmm package 138
Subpackages 138
chipsec.modules.tools.vmm.hv package 138
Submodules 138
chipsec.modules.tools.vmm.hv.define module 138
chipsec.modules.tools.vmm.hv.hypercall module 138
chipsec.modules.tools.vmm.hv.hypercallfuzz module 139
chipsec.modules.tools.vmm.hv.synth_dev module 140
chipsec.modules.tools.vmm.hv.synth_kbd module 140
chipsec.modules.tools.vmm.hv.vmbus module 141
chipsec.modules.tools.vmm.hv.vmbusfuzz module 143
Module contents 144
chipsec.modules.tools.vmm.vbox package 144
Submodules 144
chipsec.modules.tools.vmm.vbox.vbox_crash_apicbase module 144
Module contents 145
chipsec.modules.tools.vmm.xen package 145
Submodules 145
chipsec.modules.tools.vmm.xen.define module 145
chipsec.modules.tools.vmm.xen.hypercall module 145
chipsec.modules.tools.vmm.xen.hypercallfuzz module 146
chipsec.modules.tools.vmm.xen.xsa188 module 146
Module contents 147
Submodules 147
chipsec.modules.tools.vmm.common module 147
chipsec.modules.tools.vmm.cpuid_fuzz module 148
chipsec.modules.tools.vmm.ept_finder module 149
chipsec.modules.tools.vmm.hypercallfuzz module 150
chipsec.modules.tools.vmm.iofuzz module 151
chipsec.modules.tools.vmm.msr_fuzz module 153
chipsec.modules.tools.vmm.pcie_fuzz module 153
chipsec.modules.tools.vmm.pcie_overlap_fuzz module 155
chipsec.modules.tools.vmm.venom module 156
Module contents 157
Submodules 157
chipsec.modules.tools.generate_test_id module 157

chipsec.modules.tools.wsmt module 157
Module contents 158
Module contents 158
Contribution and Style Guides 158
Python Version 158
Python Coding Style Guide 158
f-Strings 161
Type Hints 161
Underscores in Numeric Literals 163
Walrus Operator (:=) 163
Deprecate distutils module support 164

CHIPSEC

CHIPSEC is a framework for analyzing platform level security of hardware, devices, system firmware, low-level protection mechanisms, and the configuration of various platform components.

It contains a set of modules, including simple tests for hardware protections and correct configuration, tests for vulnerabilities in firmware and platform components, security assessment and fuzzing tools for various platform devices and interfaces, and tools acquiring critical firmware and device artifacts.

CHIPSEC can run on *Windows*, *Linux*, *Mac OS* and *UEFI shell*. Mac OS support is Beta.

Warning

Chipsec should only be used on test systems!

It should not be installed/deployed on production end-user systems.

There are multiple reasons for that:

1. Chipsec kernel drivers provide direct access to hardware resources to user-mode applications (for example, access to physical memory). When installed on production systems this could allow malware to access privileged hardware resources.

2. The driver is distributed as source code. In order to load it on Operating System which requires kernel drivers to be signed (for example, 64 bit versions of Microsoft Windows 7 and higher), it is necessary to enable TestSigning (or equivalent) mode and sign the driver executable with test signature. Enabling TestSigning (or equivalent) mode turns off an important OS kernel protection and should not be done on production systems.

3. Due to the nature of access to hardware, if any chipsec module issues incorrect access to hardware resources, Operating System can hang or panic.

Contact

For any questions or suggestions please contact us at: chipsec@intel.com

We also have the issue tracker in our GitHub repo. If you'd like to report a bug or make a request please open an issue.

If you'd like to make a contribution to the code please open a pull request

Mailing list:

- Site: https://lists.linux.dev/
- Name: oe-chipsec
- List Address: oe-chipsec@lists.linux.dev
- Archive Site: https://lore.kernel.org/oe-chipsec/

If you wish to subscribe, please send an email to: oe-chipsec+subscribe@lists.linux.dev

Twitter:

- For CHIPSEC release alerts: Follow CHIPSEC Release
- For general CHIPSEC info: Follow CHIPSEC

2 - Download CHIPSEC

Discord:

- *CHIPSEC Discord Server <https://discord.gg/NvxdPe8RKt>*

> ### Note
>
> For **AMD** related questions or suggestions please contact Gabriel Kerneis at:
> Gabriel.Kerneis@ssi.gouv.fr

Download CHIPSEC

GitHub repository

CHIPSEC source files are maintained in a GitHub repository:

Releases

You can find the latest release here:

Older releases can be found here

After downloading there are some steps to follow to build the driver and run, please refer to Installation and running CHIPSEC

Python

Python downloads:

https://www.python.org/downloads/

Installation

CHIPSEC supports Windows, Linux, Mac OS X, DAL and UEFI shell. Circumstances surrounding the target platform may change which of these environments is most appropriate.

Linux Installation

Tested on:

- Fedora LXDE 64bit
- Ubuntu 64bit
- Debian 64bit and 32bit
- Linux UEFI Validation (LUV)
- ArchStrike Linux
- Kali Linux

Run CHIPSEC on a desired Linux distribution or create a live Linux image on a USB flash drive and boot to it.

Creating a Live Linux image

1. Download things you will need:
   - Desired Linux image (e.g. Fedora LXDE 64bit)
   - Rufus
2. Use Rufus to image a USB stick with the desired Linux image. Include as much persistent storage as possible.
3. Reboot to USB

Installing Kali Linux

Download and install Kali Linux

Prerequisites

Python 3.7 or higher (https://www.python.org/downloads/)

Note

CHIPSEC has deprecated support for Python2 since June 2020

Install or update necessary dependencies before installing CHIPSEC:

```
dnf install kernel kernel-devel-$(uname -r) python3 python3-devel gcc nasm redhat-rpm-config elfutils-libelf-devel git
```

or

```
apt-get install build-essential python3-dev python3 gcc linux-headers-$(uname -r) nasm
```

or

4 - Download CHIPSEC

```
pacman -S python3 python3-setuptools nasm linux-headers
```

To install requirements:

```
pip install -r linux_requirements.txt
```

Installing CHIPSEC

Get latest CHIPSEC release from PyPI repository

```
pip install chipsec
```

> **Note**
>
> Version in PyPI is outdate please refrain from using until further notice

Get CHIPSEC package from latest source code

Download zip from CHIPSEC repo

Download CHIPSEC

or

Clone CHIPSEC source

```
git clone https://github.com/chipsec/chipsec.git
```

Building CHIPSEC

Build the Driver and Compression Tools

```
python setup.py build_ext -i
```

Run CHIPSEC

Follow steps in section “Using as a Python package” of Running CHIPSEC

DAL Windows Installation

Prerequisites

Python 3.7 or higher (https://www.python.org/downloads/)

> **Note**
>
> CHIPSEC has deprecated support for Python2 since June 2020

pywin32: for Windows API support (https://pypi.org/project/pywin32/#files)

Intel System Studio: (https://software.intel.com/en-us/system-studio)

git: open source distributed version control system (https://git-scm.com/)

Building

Clone CHIPSEC source

```
git clone https://github.com/chipsec/chipsec.git
```

Windows Installation

CHIPSEC supports the following versions:
Windows 8, 8.1, 10, 11 - x86 and AMD64
Windows Server 2012, 2016, 2019, 2022 - x86 and AMD64

> **Note**
>
> CHIPSEC has removed support for the RWEverything (https://rweverything.com/) driver due to PCI configuration space access issues.

Install CHIPSEC Dependencies

Python 3.7 or higher (https://www.python.org/downloads/)

> **Note**
>
> CHIPSEC has deprecated support for Python2 since June 2020

To install requirements:

pip install -r windows_requirements.txt

which includes:

- pywin32: for Windows API support (*pip install pywin32*)
- setuptools (*pip install setuptools*)
- WConio2: Optional. For colored console output (*pip install Wconio2*)

To compile the driver:

Visual Studio and WDK: for building the driver.

For best results use the latest available (**VS2022 + SDK/WDK 11** or **VS2019 + SDK/WDK 10 or 11**)

> **Note**
>
> Make sure to install compatible VS/SDK/WDK versions and the spectre mitigation packages

To clone the repo:

git: open source distributed version control system

Building

Clone CHIPSEC source

```
git clone https://github.com/chipsec/chipsec.git
```

Build the Driver and Compression Tools

```
python setup.py build_ext -i
```

> **Note**
>
> If build errors are with signing are encountered, try running as Administrator The .vcxproj file points to the latest SDK, if this is incompatible with the WDK, change the configuration to a compatible SDK within the project properties

Turn off kernel driver signature checks

Enable boot menu

In CMD shell:

```
bcdedit /set {bootmgr} displaybootmenu yes
```

With Secure Boot enabled:

Method 1:

- In CMD shell: `shutdown /r /t 0 /o` or Start button -> Power icon -> SHIFT key + Restart
- Navigate: Troubleshooting -> Advanced Settings -> Startup Settings -> Reboot
- After reset choose F7 or 7 “Disable driver signature checks”

Method 2:

- Disable Secure Boot in the BIOS setup screen then disable driver signature checks as with Secure Boot disabled

With Secure Boot disabled:

7 - Download CHIPSEC

Method 1:

- **Boot in Test mode (allows self-signed certificates)**
  - Start CMD.EXE as Adminstrator `BcdEdit /set TESTSIGNING ON`
  - Reboot
  - **If this doesn't work, run these additional commands:**
    - `BcdEdit /set noIntegrityChecks ON`
    - `BcdEdit /set loadoptions DDISABLE_INTEGRITY_CHECKS`

Method 2:

- Press F8 when booting Windows and choose "No driver signatures enforcement" option to turn off driver signature checks

Alternate Build Methods

Build CHIPSEC kernel driver with Visual Studio

Method 1:

- Open the Visual Studio project file (drivers/windows/chipsec_hlpr.vcxproj) using Visual Studio
- Select Platform and configuration (X86 or x64, Release)
- Go to Build -> Build Solution

Method 2:

- Open a VS developer command prompt
- `> cd <CHIPSEC_ROOT_DIR>\drivers\windows`
- **Build driver using msbuild command:**
  - `> msbuild /p:Platform=x64`

  or
  - `> msbuild /p:Platform=x32`

If build process is completed without any errors, the driver binary will be moved into the chipsec helper directory:

`<CHIPSEC_ROOT_DIR>\chipsec\helper\windows\windows_amd64 (or i386)`

Build the compression tools

Method:

- Navigate to the chipsec_toolscompression directory
- Run *python setup.py build*
- Copy the *EfiCompressor.cp<pyver>-win_<arch>.pyd* file from build/lib.win-<arch>-<pyver> to the root chipsec directory

Alternate Method to load CHIPSEC service/driver

To create and start CHIPSEC service

8 - Download CHIPSEC

```
sc create chipsec binpath="<PATH_TO_SYS>" type= kernel DisplayName="Chipsec driver"
sc start chipsec
```

When finished running CHIPSEC stop/delete service:

```
sc stop chipsec sc delete chipsec
```

Building a Bootable USB drive with UEFI Shell (x64)

1. Format your media as FAT32
2. Create the following directory structure in the root of the new media
    - `/efi/boot`
3. Download the UEFI Shell (Shell.efi) from the following link
    - https://github.com/tianocore/edk2/blob/UDK2018/ShellBinPkg/UefiShell/X64/Shell.efi
4. Rename the UEFI shell file to Bootx64.efi
5. Copy the UEFI shell (now Bootx64.efi) to the /efi/boot directory

Installing CHIPSEC

1. Extract the contents of `__install__/UEFI/chipsec_py368_uefi_x64.zip` to the USB drive, as appropriate.
    - This will create a /efi/Tools directory with Python.efi and /efi/StdLib with subdirectories for dependencies.
2. Copy the contents of CHIPSEC to the USB drive.

   The contents of your drive should look like follows:

```
-  fs0:
   -  efi
      -  boot
         -  bootx64.efi (optional)
      -  StdLib
         -  lib
            -  python36.8
               -  [lots of python files and directories]
      -  Tools
         -  Python.efi
   -  chipsec
      -  chipsec
         -  …
      -  chipsec_main.py
      -  chipsec_util.py
      -  …
```

3. Reboot to the USB drive (this will boot to UEFI shell).
    - You may need to enable booting from USB in BIOS setup.
    - You will need to disable UEFI Secure Boot to boot to the UEFI Shell.

Run CHIPSEC in UEFI Shell

```
fs0:
cd chipsec
python.efi chipsec_main.py or python.efi chipsec_util.py
```

Next follow steps in section “Basic Usage” of Running CHIPSEC

Building UEFI Python 3.6.8 (optional)

1. Start with Py368Readme.txt
    - Latest EDK2, visit Tianocore EDK2 Github (Make sure to update submodules)
    - Latest EDK2-LIBC, visit Tianocore EDK2-LIBC Github
    - Follow setup steps described in the `Py368Readme.txt`
2. Make modifications as needed
    - CPython / C file(s):
        - edk2module.c
    - ASM file(s):
        - cpu.nasm
        - cpu_ia32.nasm
        - cpu_gcc.s
        - cpu_ia32_gcc.s
    - INF file(s):
        - Python368.inf
3. Build and directory creation steps are covered in the Py368Readme.txt
    - MSVS build tools are highly recommended

Using CHIPSEC

CHIPSEC should be launched as Administrator/root

CHIPSEC will automatically attempt to create and start its service, including load its kernel-mode driver. If CHIPSEC service is already running then it will attempt to connect to the existing service.

Use `--no-driver` command-line option to skip loading the kernel module. This option will only work for certain commands or modules.

Interpreting results

> **Note**
>
> DRAFT (work in progress)

In order to improve usability, we are reviewing and improving the messages and meaning of information returned by CHIPSEC.

Results

Generic results meanings

Result	Meaning
PASSED	A **mitigation** to a known vulnerability has been detected
FAILED	A known **vulnerability** has been detected
WARNING	We have detected something that could be a vulnerability but **manual analysis is required** to confirm (inconclusive)
NOT_APPLICABLE	The issue checked by this module is not applicable to this platform. This result can be ignored
INFORMATION	This module does not check for a vulnerability. It just prints information about the system
ERROR	Something went wrong in the execution of CHIPSEC

Automated Tests

Each test module can log additional messaging in addition to the return value. In an effort to standardize and improve the clarity of this messaging, the mapping of result and messages is defined below:

Modules results meanings

Test	PASSED message	FAILED message	WARNING message	Notes
memconfig	All memory map registers seem to be locked down	Not all memory map registers are locked down	N/A	

Remap	Memory Remap is configured correctly and locked	Memory Remap is not properly configured/locked. Remaping attack may be possible.	N/A	
smm_dma	TSEG is properly configured. SMRAM is protected from DMA attacks.	TSEG is properly configured, but the configuration is not locked or TSEG is not properly configured. Portions of SMRAM may be vulnerable to DMA attacks	TSEG is properly configured but can't determine if it covers entire SMRAM	
common.bios_kbrd_buffer	"Keyboard buffer is filled with common fill pattern" or "Keyboard buffer looks empty. Pre-boot passwords don't seem to be exposed	FAILED	Keyboard buffer is not empty. The test cannot determine conclusively if it contains pre-boot passwords.n The contents might have not been cleared by pre-boot firmware or overwritten with garbage.n Visually inspect the contents of keyboard buffer for pre-boot passwords (BIOS, HDD, full-disk encryption).	Also printing a message if size of buffer is revealed. "Was your password %d characters long?"
common.bios_smi	All required SMI sources seem to be enabled and locked	Not all required SMI sources are enabled and locked	Not all required SMI sources are enabled and locked, but SPI flash writes are still restricted to SMM	
common.bios_ts	BIOS Interface is locked (including Top Swap Mode)	BIOS Interface is not locked (including Top Swap Mode)	N/A	

common.bios_wp	BIOS is write protected	BIOS should enable all available SMM based write protection mechanisms or configure SPI protected ranges to protect the entire BIOS region. BIOS is NOT protected completely	N/A	
common.ia32cfg	IA32_FEATURE_CONTROL MSR is locked on all logical CPUs	IA32_FEATURE_CONTROL MSR is not locked on all logical CPUs	N/A	
common.rtclock	Protected locations in RTC memory are locked	N/A	Protected locations in RTC memory are accessible (BIOS may not be using them)	
common.smm	Compatible SMRAM is locked down	Compatible SMRAM is not properly locked. Expected (D_LCK = 1, D_OPEN = 0)	N/A	Should return SKIPPED_NOT_APPLICABLE when compatible SMRAM is not enabled.
common.smrr	SMRR protection against cache attack is properly configured	SMRR protection against cache attack is not configured properly	N/A	
common.spi_access	SPI Flash Region Access Permissions in flash descriptor look ok	SPI Flash Region Access Permissions are not programmed securely in flash descriptor	Software has write access to GBe region in SPI flash" and "Certain SPI flash regions are writeable by software	we have observed production systems reacting badly when GBe was overwritten
common.spi_desc	SPI flash permissions prevent SW from writing to flash descriptor	SPI flash permissions allow SW to write flash descriptor	N/A	we can probably remove this now that we have spi_access
common.spi_fdopss	SPI Flash Descriptor Security Override is disabled	SPI Flash Descriptor Security Override is enabled	N/A	

common.spi_lock	SPI Flash Controller configuration is locked	SPI Flash Controller configuration is not locked	N/A	
common.cpu.spectre_v2	CPU and OS support hardware mitigations (enhanced IBRS and STIBP)	CPU mitigation (IBRS) is missing	CPU supports mitigation (IBRS) but doesn't support enhanced IBRS" or "CPU supports mitigation (enhanced IBRS) but OS is not using it" or "CPU supports mitigation (enhanced IBRS) but STIBP is not supported/enabled	
common.secureboot.variables	All Secure Boot UEFI variables are protected	Not all Secure Boot UEFI variables are protected' (failure when secure boot is enabled)	Not all Secure Boot UEFI variables are protected' (warning when secure boot is disabled)	
common.uefi.access_uefispec	All checked EFI variables are protected according to spec	Some EFI variables were not protected according to spec	Extra/Missing attributes	
common.uefi.s3bootscript	N/A	S3 Boot-Script and Dispatch entry-points do not appear to be protected	S3 Boot-Script is not in SMRAM but Dispatch entry-points appear to be protected. Recommend further testing	unfortunately, if the boot script is well protected (in SMRAM) we cannot find it at all and end up returning warning

Tools

CHIPSEC also contains tools such as fuzzers, which require a knowledgeable user to run. We can examine the usability of these tools as well.

Running CHIPSEC

CHIPSEC should be launched as Administrator/root.

CHIPSEC will automatically attempt to create and start its service, including load its kernel-mode driver. If CHIPSEC service is already running then it will attempt to connect to the existing service.

Use –no-driver command-line option to skip loading the kernel module. This option will only work for certain commands or modules.

Use `-m --module` to run a specific module (e.g. security check, a tool or a PoC..):

- `# python chipsec_main.py -m common.bios_wp`
- `# python chipsec_main.py -m common.spi_lock`
- `# python chipsec_main.py -m common.smrr`

- You can also use CHIPSEC to access various hardware resources:

  `# python chipsec_util.py`

Running in Shell

Basic usage

```
# python chipsec_main.py
# python chipsec_util.py
```

For help, run

```
# python chipsec_main.py --help
# python chipsec_util.py --help
```

Using as a Python Package

Install CHIPSEC manually or from PyPI. You can then use CHIPSEC from your Python project or from the Python shell:

To install and run CHIPSEC as a package:

```
# python setup.py install
# sudo chipsec_main
```

From the Python shell:

```
>>> import chipsec_main
>>> chipsec_main.run()
>>> chipsec_main.run('-m common.bios_wp')
```

```
>>> import chipsec_util
>>> chipsec_util.run()
>>> chipsec_util.run('spi info')
```

To use CHIPSEC *in place* without installing it:

```
# python setup.py build_ext -i
# sudo python chipsec_main.py
```

chipsec_main options

```
usage: chipsec_main.py [options]

Options:
  -h, --help                          Show this message and exit
  -m, --module _MODULE                Specify module to run (example: -m common.bios_wp)
  -mx, --module_exclude _MODULE1 ...  Specify module(s) to NOT run (example: -mx common.bios_wp common.cpu.cpu_info)
  -a, --module_args _MODULE_ARGV      Additional module arguments
  -v, --verbose                       Verbose logging
  --hal                               HAL logging
  -d, --debug                         Debug logging
  -l, --log  LOG                      Output to log file
  -vv, --vverbose                     Very verbose logging (verbose + HAL + debug)

Advanced Options:
  -p, --platform _PLATFORM            Explicitly specify platform code
  --pch _PCH                          Explicitly specify PCH code
  -n, --no_driver                     Chipsec won't need kernel mode functions so don't load chipsec driver
  -i, --ignore_platform               Run chipsec even if the platform is not recognized (Deprecated)
  -j, --json _JSON_OUT                Specify filename for JSON output
  -x, --xml _XML_OUT                  Specify filename for xml output (JUnit style)
  -k, --markdown                      Specify filename for markdown output
  -t, --moduletype USER_MODULE_TAGS   Run tests of a specific type (tag)
  --list_tags                         List all the available options for -t,--moduletype
  -I, --include IMPORT_PATHS          Specify additional path to load modules from
  --failfast                          Fail on any exception and exit (don't mask exceptions)
  --no_time                           Don't log timestamps
  --deltas _DELTAS_FILE               Specifies a JSON log file to compute result deltas from
  --helper _HELPER                    Specify OS Helper
  -nb, --no_banner                    Chipsec won't display banner information
  --skip_config                       Skip configuration and driver loading
  -nl                                 Chipsec won't save logs automatically
```

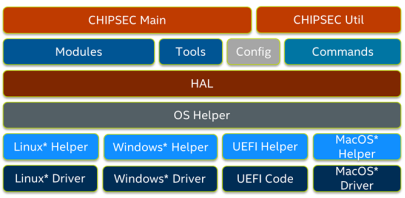
chipsec_util options

```
usage: chipsec_util.py [options] <command> [<args>]

Options:
  -h, --help                  Show this message and exit
  -v, --verbose               Verbose logging
  --hal                       HAL logging
  -d, --debug                 Debug logging
  -vv, --vverbose             Very verbose logging (verbose + HAL + debug)
  -l, --log  LOG              Output to log file
  -p, --platform _PLATFORM    Explicitly specify platform code
  --pch _PCH                  Explicitly specify PCH code
  -n, --no_driver             Chipsec won't need kernel mode functions so don't load chipsec driver
  -i, --ignore_platform       Run chipsec even if the platform is not recognized (Deprecated)
  --helper _HELPER            Specify OS Helper
  -nb, --no_banner            Chipsec won't display banner information
  --skip_config               Skip configuration and driver loading
  -nl                         Chipsec won't save logs automatically
  command                     Util command to run
  args                        Additional arguments for specific command. All numeric values are in hex. <width> is in {1 - byte, 2 - word, 4 - dword}
```

Module & Command Development

Architecture Overview



CHIPSEC Architecture

Core components

`chipsec_main.py`	main application logic and automation functions
`chipsec_util.py`	utility functions (access to various hardware resources)
`chipsec/chipset.py`	chipset detection
`chipsec/command.py`	base class for util commands
`chipsec/defines.py`	common defines
`chipsec/file.py`	reading from/writing to files
`chipsec/logger.py`	logging functions
`chipsec/module.py`	generic functions to import and load modules
`chipsec/module_common.py`	base class for modules
`chipsec/result_deltas.py`	supports checking result deltas between test runs
`chipsec/testcase.py`	support for XML and JSON log file output
`chipsec/helper/helpers.py`	registry of supported OS helpers
`chipsec/helper/oshelper.py`	OS helper: wrapper around platform specific code that invokes kernel driver

Commands

Implement functionality of chipsec_util.

CHIPSEC utilities provide the capability for manual testing and direct hardware access.

Warning

DIRECT HARDWARE ACCESS PROVIDED BY THESE UTILITIES COULD MAKE YOUR SYSTEM UNBOOTABLE. MAKE SURE YOU KNOW WHAT YOU ARE DOING!

Note

All numeric values in the instructions are in hex.

chipsec.utilcmd package

Submodules

chipsec.utilcmd.acpi_cmd module

Command-line utility providing access to ACPI tables

```
>>> chipsec_util acpi list
>>> chipsec_util acpi table <name>|<file_path>
```

Examples:

```
>>> chipsec_util acpi list
>>> chipsec_util acpi table XSDT
>>> chipsec_util acpi table acpi_table.bin
```

class ACPICommand (argv, cs=None)
Bases: BaseCommand

acpi_list () → None

acpi_table () → None

parse_arguments () → None

requirements () → toLoad

set_up () → None

chipsec.utilcmd.chipset_cmd module

usage as a standalone utility:

```
>>> chipsec_util platform
```

class PlatformCommand (argv, cs=None)
Bases: BaseCommand
chipsec_util platform

parse_arguments () → None

requirements () → toLoad

run ()

chipsec.utilcmd.cmos_cmd module

```
>>> chipsec_util cmos dump
>>> chipsec_util cmos readl|writel|readh|writeh <byte_offset> [byte_val]
```

Examples:

```
>>> chipsec_util cmos dump
>>> chipsec_util cmos readl 0x0
>>> chipsec_util cmos writeh 0x0 0xCC
```

class CMOSCommand (argv, cs=None)
Bases: BaseCommand

cmos_dump () → None

cmos_readh () → None

cmos_readl () → None

cmos_writeh () → None

cmos_writel () → None

parse_arguments () → None

requirements () → toLoad

set_up () → None

chipsec.utilcmd.config_cmd module

```
>>> chipsec_util config show [config] <name>
```

Examples:

```
>>> chipsec_util config show ALL
>>> chipsec_util config show MMIO_BARS
>>> chipsec_util config show REGISTERS BC
```

class CONFIGCommand (argv, cs=None)
Bases: BaseCommand

bus_details (regi: str) → str

control_details (regi: Dict[str, Any]) → str

io_details (regi: Dict[str, Any]) → str

lock_details (regi: Dict[str, Any]) → str

mem_details (regi: Dict[str, Any]) → str

mmio_details (regi: Dict[str, Any]) → str

parse_arguments () → None

pci_details (regi: Dict[str, Any]) → str

register_details (regi: Dict[str, Any]) → str

requirements () → toLoad

show () → None

chipsec.utilcmd.cpu_cmd module

```
>>> chipsec_util cpu info
>>> chipsec_util cpu cr <thread> <cr_number> [value]
>>> chipsec_util cpu cpuid <eax> [ecx]
>>> chipsec_util cpu pt [paging_base_cr3]
>>> chipsec_util cpu topology
```

Examples:

```
>>> chipsec_util cpu info
>>> chipsec_util cpu cr 0 0
>>> chipsec_util cpu cr 0 4 0x0
>>> chipsec_util cpu cpuid 0x40000000
>>> chipsec_util cpu pt
>>> chipsec_util cpu topology
```

class CPUCommand (argv, cs=None)

Bases: BaseCommand

cpu_cpuid () → None

cpu_cr () → bool | int | None

cpu_info () → None

cpu_pt () → None

cpu_topology () → Dict[str, Dict[int, List[int]]]

parse_arguments () → None

requirements () → toLoad

chipsec.utilcmd.decode_cmd module

CHIPSEC can parse an image file containing data from the SPI flash (such as the result of chipsec_util spi dump). This can be critical in forensic analysis.

This will create multiple log files, binaries, and directories that correspond to the sections, firmware volumes, files, variables, etc. stored in the SPI flash.

Usage:

```
>>> chipsec_util decode <rom> [fw_type]
```

For a list of fw types run:

```
>>> chipsec_util decode types
```

Examples:

```
>>> chipsec_util decode spi.bin vss
```

Note

- It may be necessary to try various options for fw_type in order to correctly parse NVRAM variables. Currently, CHIPSEC does not autodetect the correct format. If the nvram directory does not appear and the list of nvram variables is empty, try again with another type.

class DecodeCommand (argv, cs=None)
Bases: BaseCommand

decode_rom () → bool

decode_types () → None

parse_arguments () → None

requirements () → toLoad

chipsec.utilcmd.deltas_cmd module

```
>>> chipsec_util deltas <previous> <current> [out-format] [out-name]
```

out-format - JSON | XML out-name - Output file name

Example: >>> chipsec_util deltas run1.json run2.json

class DeltasCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run () → None

chipsec.utilcmd.desc_cmd module

The idt, gdt and ldt commands print the IDT, GDT and LDT, respectively.

IDT command:

```
>>> chipsec_util idt [cpu_id]
```

Examples:

```
>>> chipsec_util idt 0
>>> chipsec_util idt
```

GDT command:

```
>>> chipsec_util gdt [cpu_id]
```

Examples:

```
>>> chipsec_util gdt 0
>>> chipsec_util gdt
```

LDT command:

```
>>> chipsec_util ldt [cpu_id]
```

Examples:

```
>>> chipsec_util ldt 0
>>> chipsec_util ldt
```

class GDTCommand (argv, cs=None)

Bases: BaseCommand

```
>>> chipsec_util gdt [cpu_id]
```

Examples:

```
>>> chipsec_util gdt 0
>>> chipsec_util gdt
```

parse_arguments () → None

requirements () → toLoad

run () → None

class IDTCommand (argv, cs=None)

Bases: BaseCommand

```
>>> chipsec_util idt [cpu_id]
```

Examples:

```
>>> chipsec_util idt 0
>>> chipsec_util idt
```

parse_arguments () → None

requirements () → toLoad

run () → None

class LDTCommand (argv, cs=None)

Bases: BaseCommand

```
>>> chipsec_util ldt [cpu_id]
```

Examples:

```
>>> chipsec_util ldt 0
>>> chipsec_util ldt
```

parse_arguments () → None

requirements () → toLoad

run () → None

chipsec.utilcmd.ec_cmd module

```
>>> chipsec_util ec dump    [<size>]
>>> chipsec_util ec command <command>
>>> chipsec_util ec read    <offset> [<size>]
>>> chipsec_util ec write   <offset> <byte_val>
>>> chipsec_util ec index   [<offset>]
```

Examples:

```
>>> chipsec_util ec dump
>>> chipsec_util ec command 0x001
>>> chipsec_util ec read    0x2F
>>> chipsec_util ec write   0x2F 0x00
>>> chipsec_util ec index
```

class ECCommand (argv, cs=None)

Bases: BaseCommand

command () → None

dump () → None

index () → None

parse_arguments () → None

read () → None

requirements () → toLoad

set_up () → None

write () → None

chipsec.utilcmd.igd_cmd module

The igd command allows memory read/write operations using igd dma.

```
>>> chipsec_util igd
>>> chipsec_util igd dmaread <address> [width] [file_name]
>>> chipsec_util igd dmawrite <address> <width> <value|file_name>
```

Examples:

```
>>> chipsec_util igd dmaread 0x20000000 4
>>> chipsec_util igd dmawrite 0x2217F1000 0x4 deadbeef
```

class IgdCommand (argv, cs=None)

Bases: BaseCommand

parse_arguments () → None

read_dma () → None

requirements () → toLoad

run () → None

write_dma () → None

chipsec.utilcmd.interrupts_cmd module

SMI command:

```
>>> chipsec_util smi count
>>> chipsec_util smi send <thread_id> <SMI_code> <SMI_data> [RAX] [RBX] [RCX] [RDX] [RSI] [RDI]
>>> chipsec_util smi smmc <RT_code_start> <RT_code_end> <GUID> <payload_loc> <payload_file|payload_string> [port]
```

Examples:

```
>>> chipsec_util smi count
>>> chipsec_util smi send 0x0 0xDE 0x0
>>> chipsec_util smi send 0x0 0xDE 0x0 0xAAAAAAAAAAAAAAAA ..
>>> chipsec_util smi smmc 0x79dfe000 0x79efdfff ed32d533-99e6-4209-9cc02d72cdd998a7 0x79dfaaaa payload.bin
```

NMI command:

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

class NMICommand (argv, cs=None)

Bases: BaseCommand

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

parse_arguments () → None

requirements () → toLoad

run () → None

class SMICommand (argv, cs=None)

Bases: BaseCommand

```
>>> chipsec_util smi count
>>> chipsec_util smi send <thread_id> <SMI_code> <SMI_data> [RAX] [RBX] [RCX] [RDX] [RSI] [RDI]
>>> chipsec_util smi smmc <RT_code_start> <RT_code_end> <GUID> <payload_loc> <payload_file|payload_string> [port]
```

Examples:

```
>>> chipsec_util smi count
>>> chipsec_util smi send 0x0 0xDE 0x0
>>> chipsec_util smi send 0x0 0xDE 0x0 0xAAAAAAAAAAAAAAAA ..
>>> chipsec_util smi smmc 0x79dfe000 0x79efdfff ed32d533-99e6-4209-9cc02d72cdd998a7 0x79dfaaaa payload.bin
```

parse_arguments () → None

requirements () → toLoad

run () → None

smi_count () → None

smi_send () → None

smi_smmc () → None

chipsec.utilcmd.io_cmd module

The io command allows direct access to read and write I/O port space.

```
>>> chipsec_util io list
>>> chipsec_util io read  <io_port> <width>
>>> chipsec_util io write <io_port> <width> <value>
```

Examples:

```
>>> chipsec_util io list
>>> chipsec_util io read 0x61 1
>>> chipsec_util io write 0x430 1 0x0
```

class PortIOCommand (argv, cs=None)

Bases: BaseCommand

io_list () → None

io_read () → None

io_write () → None

parse_arguments () → None

requirements () → toLoad

set_up () → None

chipsec.utilcmd.iommu_cmd module

Command-line utility providing access to IOMMU engines

```
>>> chipsec_util iommu list
>>> chipsec_util iommu config [iommu_engine]
>>> chipsec_util iommu status [iommu_engine]
>>> chipsec_util iommu enable|disable <iommu_engine>
>>> chipsec_util iommu pt
```

Examples:

```
>>> chipsec_util iommu list
>>> chipsec_util iommu config VTD
>>> chipsec_util iommu status GFXVTD
>>> chipsec_util iommu enable VTD
>>> chipsec_util iommu pt
```

class IOMMUCommand (argv, cs=None)

Bases: BaseCommand

iommu_config () → None

iommu_disable () → None

iommu_enable () → None

iommu_engine (cmd) → None

iommu_list () → None

iommu_pt () → None

iommu_status () → None

parse_arguments () → None

requirements () → toLoad

run () → None

chipsec.utilcmd.lock_check_cmd module

```
>>> chipsec_util check list
>>> chipsec_util check lock <lockname>
>>> chipsec_util check lock <lockname1, lockname2, ...>
>>> chipsec_util check all
```

Examples:

```
>>> chipsec_util check list
>>> chipsec_util check lock DebugLock
>>> chipsec_util check all
```

KEY:

Lock Name - Name of Lock within configuration file State - Lock Configuration

Undefined - Lock is not defined within configuration Undoc - Lock is missing configuration information Hidden - Lock is in a disabled or hidden state (unable to read the lock) Unlocked - Lock does not match value within configuration Locked - Lock matches value within configuration RW/O - Lock is identified as register is RW/O

class LOCKCHECKCommand (argv, cs=None)

Bases: BaseCommand

check_lock () → None

check_log (lock: str, is_locked: int) → str

checkall_locks () → None

list_locks () → None

log_header () → str

log_key () → None

parse_arguments () → None

requirements () → toLoad

set_up () → None

tear_down () → None

version = *'0.5'*

chipsec.utilcmd.mem_cmd module

The mem command provides direct access to read and write physical memory.

```
>>> chipsec_util mem <op> <physical_address> <length> [value|buffer_file]
>>> <physical_address> : 64-bit physical address
>>> <op>               : read|readval|write|writeval|allocate|pagedump|search
>>> <length>           : byte|word|dword or length of the buffer from <buffer_file>
>>> <value>            : byte, word or dword value to be written to memory at <physical_address>
>>> <buffer_file>      : file with the contents to be written to memory at <physical_address>
```

Examples:

```
>>> chipsec_util mem <op>     <physical_address> <length> [value|file]
>>> chipsec_util mem readval  0xFED40000         dword
>>> chipsec_util mem read     0x41E              0x20     buffer.bin
>>> chipsec_util mem writeval 0xA0000            dword    0x9090CCCC
>>> chipsec_util mem write    0x100000000        0x1000   buffer.bin
>>> chipsec_util mem write    0x100000000        0x10     000102030405060708090A0B0C0D0E0F
>>> chipsec_util mem allocate                    0x1000
>>> chipsec_util mem pagedump 0xFED00000         0x100000
>>> chipsec_util mem search   0xF0000            0x10000  _SM_
```

class MemCommand (argv, cs=None)

Bases: BaseCommand

dump_region_to_path (path: str, pa_start: int, pa_end: int) → None

mem_allocate () → None

mem_pagedump () → None

mem_read () → None

mem_readval () → None

mem_search () → None

mem_write () → None

mem_writeval () → None

parse_arguments () → None

requirements () → toLoad

chipsec.utilcmd.mmcfg_base_cmd module

The mmcfg_base command displays PCIe MMCFG Base/Size.

Usage:

```
>>> chipsec_util mmcfg_base
```

Examples:

```
>>> chipsec_util mmcfg_base
```

class MMCfgBaseCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run () → None

chipsec.utilcmd.mmcfg_cmd module

The mmcfg command allows direct access to memory mapped config space.

```
>>> chipsec_util mmcfg <bus> <device> <function> <offset> <width> [value]
```

Examples:

```
>>> chipsec_util mmcfg 0 0 0 0x88 4
>>> chipsec_util mmcfg 0 0 0 0x88 byte 0x1A
>>> chipsec_util mmcfg 0 0x1F 0 0xDC 1 0x1
>>> chipsec_util mmcfg 0 0 0 0x98 dword 0x004E0040
```

class MMCfgCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run () → None

chipsec.utilcmd.mmio_cmd module

```
>>> chipsec_util mmio list
>>> chipsec_util mmio dump <MMIO_BAR_name> [offset] [length]
>>> chipsec_util mmio dump-abs <MMIO_base_address> [offset] [length]
>>> chipsec_util mmio read <MMIO_BAR_name> <offset> <width>
>>> chipsec_util mmio read-abs <MMIO_base_address> <offset> <width>
>>> chipsec_util mmio write <MMIO_BAR_name> <offset> <width> <value>
>>> chipsec_util mmio write-abs <MMIO_base_address> <offset> <width> <value>
```

Examples:

```
>>> chipsec_util mmio list
>>> chipsec_util mmio dump MCHBAR
>>> chipsec_util mmio dump-abs 0xFE010000 0x70 0x10
>>> chipsec_util mmio read SPIBAR 0x74 0x4
>>> chipsec_util mmio read-abs 0xFE010000 0x74 0x04
>>> chipsec_util mmio write SPIBAR 0x74 0x4 0xFFFF0000
>>> chipsec_util mmio write-abs 0xFE010000 0x74 0x04 0xFFFF0000
```

class MMIOCommand (argv, cs=None)
Bases: BaseCommand

dump_bar ()

dump_bar_abs ()

list_bars ()

parse_arguments () → None

read_abs ()

read_bar ()

requirements () → toLoad

set_up () → None

write_abs ()

write_bar ()

chipsec.utilcmd.msgbus_cmd module

```
>>> chipsec_util msgbus read     <port> <register>
>>> chipsec_util msgbus write    <port> <register> <value>
>>> chipsec_util msgbus mm_read  <port> <register>
>>> chipsec_util msgbus mm_write <port> <register> <value>
>>> chipsec_util msgbus message  <port> <register> <opcode> [value]
>>>
>>> <port>    : message bus port of the target unit
>>> <register>: message bus register/offset in the target unit port
>>> <value>   : value to be written to the message bus register/offset
>>> <opcode>  : opcode of the message on the message bus
```

Examples:

```
>>> chipsec_util msgbus read     0x3 0x2E
>>> chipsec_util msgbus mm_write 0x3 0x27 0xE0000001
>>> chipsec_util msgbus message  0x3 0x2E 0x10
>>> chipsec_util msgbus message  0x3 0x2E 0x11 0x0
```

class MsgBusCommand (argv, cs=None)
Bases: BaseCommand

msgbus_message ()

msgbus_mm_read ()

msgbus_mm_write ()

msgbus_read ()

msgbus_write ()

parse_arguments () → None

requirements () → toLoad

run ()

chipsec.utilcmd.msr_cmd module

The msr command allows direct access to read and write MSRs.

```
>>> chipsec_util msr <msr> [eax] [edx] [thread_id]
```

Examples:

```
>>> chipsec_util msr 0x3A
>>> chipsec_util msr 0x3A 0x0
>>> chipsec_util msr 0x8B 0x0 0x0 0x0
```

class MSRCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

chipsec.utilcmd.pci_cmd module

The pci command can enumerate PCI/PCIe devices, enumerate expansion ROMs and allow direct access to PCI configuration registers via bus/device/function.

```
>>> chipsec_util pci enumerate
>>> chipsec_util pci read <bus> <device> <function> <offset> [width]
>>> chipsec_util pci write <bus> <device> <function> <offset> <width> <value>
>>> chipsec_util pci dump [<bus>] [<device>] [<function>]
>>> chipsec_util pci xrom [<bus>] [<device>] [<function>] [xrom_address]
>>> chipsec_util pci cmd [mask] [class] [subclass]
```

Examples:

```
>>> chipsec_util pci enumerate
>>> chipsec_util pci read 0 0 0 0x00
>>> chipsec_util pci read 0 0 0 0x88 byte
>>> chipsec_util pci write 0 0x1F 0 0xDC 1 0x1
>>> chipsec_util pci write 0 0 0 0x98 dword 0x004E0040
>>> chipsec_util pci dump
>>> chipsec_util pci dump 0 0 0
>>> chipsec_util pci xrom
>>> chipsec_util pci xrom 3 0 0 0xFEDF0000
>>> chipsec_util pci cmd
>>> chipsec_util pci cmd 1
```

class PCICommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

pci_cmd ()

pci_dump ()

pci_enumerate ()

pci_read ()

pci_write ()

pci_xrom ()

requirements () → toLoad

chipsec.utilcmd.reg_cmd module

```
>>> chipsec_util reg read <reg_name> [<field_name>]
>>> chipsec_util reg read_field <reg_name> <field_name>
>>> chipsec_util reg write <reg_name> <value>
>>> chipsec_util reg write_field <reg_name> <field_name> <value>
>>> chipsec_util reg get_control <control_name>
>>> chipsec_util reg set_control <control_name> <value>
```

Examples:

```
>>> chipsec_util reg read SMBUS_VID
>>> chipsec_util reg read HSFC FGO
>>> chipsec_util reg read_field HSFC FGO
>>> chipsec_util reg write SMBUS_VID 0x8088
>>> chipsec_util reg write_field BC BLE 0x1
>>> chipsec_util reg get_control BiosWriteEnable
>>> chipsec_util reg set_control BiosLockEnable 0x1
```

class RegisterCommand (argv, cs=None)

Bases: BaseCommand

parse_arguments () → None

reg_get_control ()

reg_read ()

reg_read_field ()

reg_set_control ()

reg_write ()

reg_write_field ()

requirements () → toLoad

chipsec.utilcmd.smbios_cmd module

```
>>> chipsec_util smbios entrypoint
>>> chipsec_util smbios get [raw|decoded] [type]
```

Examples:

```
>>> chipsec_util smbios entrypoint
>>> chipsec_util smbios get raw
```

class smbios_cmd (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

smbios_ep ()

smbios_get ()

chipsec.utilcmd.smbus_cmd module

```
>>> chipsec_util smbus read <device_addr> <start_offset> [size]
>>> chipsec_util smbus write <device_addr> <offset> <byte_val>
```

Examples:

```
>>> chipsec_util smbus read 0xA0 0x0 0x100
```

class SMBusCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

set_up () → None

smbus_read ()

smbus_write ()

chipsec.utilcmd.spd_cmd module

```
>>> chipsec_util spd detect
>>> chipsec_util spd dump [device_addr]
>>> chipsec_util spd read <device_addr> <offset>
>>> chipsec_util spd write <device_addr> <offset> <byte_val>
```

Examples:

```
>>> chipsec_util spd detect
>>> chipsec_util spd dump DIMM0
>>> chipsec_util spd dump 0xA0
>>> chipsec_util spd read DIMM2 0x0
>>> chipsec_util spd read 0xA0 0x0
>>> chipsec_util spd write 0xA0 0x0 0xAA
```

class SPDCommand (argv, cs=None)

Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

spd_detect ()

spd_dump ()

spd_read ()

spd_write ()

chipsec.utilcmd.spi_cmd module

CHIPSEC includes functionality for reading and writing the SPI flash. When an image file is created from reading the SPI flash, this image can be parsed to reveal sections, files, variables, etc.

> **Warning**
>
> Particular care must be taken when using the SPI write and SPI erase functions. These could make your system unbootable.

A basic forensic operation might be to dump the entire SPI flash to a file. This is accomplished as follows:

```
# python chipsec_util.py spi dump rom.bin
```

The file rom.bin will contain the full binary of the SPI flash. It can then be parsed using the decode util command.

```
>>> chipsec_util spi info|dump|read|write|erase|disable-wp [flash_address] [length] [file]
```

Examples:

```
>>> chipsec_util spi info
>>> chipsec_util spi dump rom.bin
>>> chipsec_util spi read 0x700000 0x100000 bios.bin
>>> chipsec_util spi write 0x0 flash_descriptor.bin
>>> chipsec_util spi disable-wp
>>> chipsec_util spi sfdp
>>> chipsec_util spi jedec
>>> chipsec_util spi jedec decode
```

class SPICommand (argv, cs=None)

Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

set_up () → None

spi_disable_wp ()

spi_dump ()

spi_erase ()

spi_info ()

spi_jedec ()

spi_read ()

spi_sfdp ()

spi_write ()

chipsec.utilcmd.spidesc_cmd module

```
>>> chipsec_util spidesc <rom>
```

Examples:

```
>>> chipsec_util spidesc spi.bin
```

class SPIDescCommand (argv, cs=None)

Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

chipsec.utilcmd.tpm_cmd module

```
>>> chipsec_util tpm parse_log <file>
>>> chipsec_util tpm state <locality>
>>> chipsec_util tpm command <commandName> <locality> <command_parameters>
```

locality: 0 | 1 | 2 | 3 | 4 commands - parameters: pccrread - pcr number (0 - 23) nvread - Index, Offset, Size startup - startup type (1 - 3) continueselftest getcap - Capabilities Area, Size of Sub-capabilities, Sub-capabilities forceclear

Examples:

```
>>> chipsec_util tpm parse_log binary_bios_measurements
>>> chipsec_util tpm state 0
>>> chipsec_util tpm command pcrread 0 17
>>> chipsec_util tpm command continueselftest 0
```

class TPMCommand (argv, cs=None)
Bases: BaseCommand

no_driver_cmd = *['parse_log']*

parse_arguments () → None

requirements () → toLoad

run ()

set_up ()

tpm_command ()

tpm_parse ()

tpm_state ()

chipsec.utilcmd.txt_cmd module

Command-line utility providing access to Intel TXT (Trusted Execution Technology) registers

Usage:

```
>>> chipsec_util txt dump
>>> chipsec_util txt state
```

class TXTCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

txt_dump ()

txt_state ()
Dump Intel TXT state
This is similar to command “txt-stat” from Trusted Boot project https://sourceforge.net/p/tboot/code/ci/v2.0.0/tree/utils/txt-stat.c which was documented on https://www.intel.com/content/dam/www/public/us/en/documents/guides/dell-one-stop-txt-activation-guide.pdf and it is also similar to command “sl-stat” from TrenchBoot project https://github.com/TrenchBoot/sltools/blob/842cfd041b7454727b363b72b6d4dcca9c00daca/sl-stat/sl-stat.c

chipsec.utilcmd.ucode_cmd module

```
>>> chipsec_util ucode id|load|decode [ucode_update_file (in .PDB or .BIN format)] [cpu_id]
```

Examples:

```
>>> chipsec_util ucode id
>>> chipsec_util ucode load ucode.bin 0
>>> chipsec_util ucode decode ucode.pdb
```

class UCodeCommand (argv, cs=None)

Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

ucode_decode ()

ucode_id ()

ucode_load ()

chipsec.utilcmd.uefi_cmd module

The uefi command provides access to UEFI variables, both on the live system and in a SPI flash image file.

```
>>> chipsec_util uefi types
>>> chipsec_util uefi var-list
>>> chipsec_util uefi var-find <name>|<GUID>
>>> chipsec_util uefi var-read|var-write|var-delete <name> <GUID> <efi_variable_file>
>>> chipsec_util uefi decode <rom_file> [filetypes]
>>> chipsec_util uefi nvram[-auth] <rom_file> [fwtype]
>>> chipsec_util uefi keys <keyvar_file>
>>> chipsec_util uefi tables
>>> chipsec_util uefi s3bootscript [script_address]
>>> chipsec_util uefi assemble <GUID> freeform none|lzma|tiano <raw_file> <uefi_file>
>>> chipsec_util uefi insert_before|insert_after|replace|remove <GUID> <rom> <new_rom> <uefi_file>
```

Examples:

```
>>> chipsec_util uefi types
>>> chipsec_util uefi var-list
>>> chipsec_util uefi var-find PK
>>> chipsec_util uefi var-read db D719B2CB-3D3A-4596-A3BC-DAD00E67656F db.bin
>>> chipsec_util uefi var-write db D719B2CB-3D3A-4596-A3BC-DAD00E67656F db.bin
>>> chipsec_util uefi var-delete db D719B2CB-3D3A-4596-A3BC-DAD00E67656F
>>> chipsec_util uefi decode uefi.rom
>>> chipsec_util uefi decode uefi.rom FV_MM
>>> chipsec_util uefi nvram uefi.rom vss_auth
>>> chipsec_util uefi keys db.bin
>>> chipsec_util uefi tables
>>> chipsec_util uefi s3bootscript
>>> chipsec_util uefi assemble AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE freeform lzma uefi.raw mydriver.efi
>>> chipsec_util uefi replace  AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE bios.bin new_bios.bin mydriver.efi
```

class UEFICommand (argv, cs=None)

Bases: BaseCommand

assemble ()

decode ()

insert_after ()

insert_before ()

keys ()

nvram ()

nvram_auth ()

parse_arguments () → None

remove ()

replace ()

requirements () → toLoad

s3bootscript ()

set_up () → None

tables ()

var_delete ()

var_find ()

var_list ()

var_read ()

var_write ()

chipsec.utilcmd.vmem_cmd module

The vmem command provides direct access to read and write virtual memory.

```
>>> chipsec_util vmem <op> <physical_address> <length> [value|buffer_file]
>>>
>>> <physical_address> : 64-bit physical address
>>> <op>               : read|readval|write|writeval|allocate|pagedump|search|getphys
>>> <length>           : byte|word|dword or length of the buffer from <buffer_file>
>>> <value>            : byte, word or dword value to be written to memory at <physical_address>
>>> <buffer_file>      : file with the contents to be written to memory at <physical_address>
```

Examples:

```
>>> chipsec_util vmem <op>     <virtual_address>  <length> [value|file]
>>> chipsec_util vmem readval  0xFED40000         dword
>>> chipsec_util vmem read     0x41E              0x20     buffer.bin
>>> chipsec_util vmem writeval 0xA0000            dword    0x9090CCCC
>>> chipsec_util vmem write    0x100000000        0x1000   buffer.bin
>>> chipsec_util vmem write    0x100000000        0x10     000102030405060708090A0B0C0D0E0F
>>> chipsec_util vmem allocate                    0x1000
>>> chipsec_util vmem search   0xF0000            0x10000  _SM_
>>> chipsec_util vmem getphys  0xFED00000
```

class VMemCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

set_up () → None

vmem_allocate ()

vmem_getphys ()

vmem_read ()

vmem_readval ()

vmem_search ()

vmem_write ()

vmem_writeval ()

chipsec.utilcmd.vmm_cmd module

```
>>> chipsec_util vmm hypercall <rax> <rbx> <rcx> <rdx> <rdi> <rsi> [r8] [r9] [r10] [r11]
>>> chipsec_util vmm hypercall <eax> <ebx> <ecx> <edx> <edi> <esi>
>>> chipsec_util vmm pt|ept <ept_pointer>
>>> chipsec_util vmm virtio [<bus>:<device>.<function>]
```

Examples:

```
>>> chipsec_util vmm hypercall 32 0 0 0 0 0
>>> chipsec_util vmm pt 0x524B01E
>>> chipsec_util vmm virtio
>>> chipsec_util vmm virtio 0:6.0
```

class VMMCommand (argv, cs=None)
Bases: BaseCommand

parse_arguments () → None

requirements () → toLoad

run ()

vmm_hypercall ()

vmm_pt ()

vmm_virtio ()

Module contents

HAL (Hardware Abstraction Layer)

Useful abstractions for common tasks such as accessing the SPI.

chipsec.hal package

Submodules

chipsec.hal.acpi module

HAL component providing access to and decoding of ACPI tables

class ACPI (cs)
Bases: HALBase

ParseTable
alias of Tuple[ACPI_TABLE_HEADER, Optional[ACPI_TABLE], bytes, bytes]

RsdtXsdt
alias of Union[RSDT, XSDT]

dump_ACPI_table (name: str, isfile: bool = False) → None

find_RSDP () → Tuple[int | None, RSDP | None]

get_ACPI_table (name: str, isfile: bool = False) → List[Tuple[bytes, bytes]]

get_ACPI_table_list () → Dict[str, List[int]]

get_DSDT_from_FADT () → None

get_SDT (search_rsdp: bool = True) → Tuple[bool, int | None, RSDT | XSDT | None, ACPI_TABLE_HEADER | None]

get_parse_ACPI_table (name: str, isfile: bool = False) → List[ParseTable]

get_table_list_from_SDT (sdt: RSDT | XSDT, is_xsdt: bool) → None

is_ACPI_table_present (name: str) → bool

print_ACPI_table_list () → None

read_RSDP (rsdp_pa: int) → RSDP

class ACPI_TABLE_HEADER (Signature, Length, Revision, Checksum, OEMID, OEMTableID, OEMRevision, CreatorID, CreatorRevision)

Bases: ACPI_TABLE_HEADER

chipsec.hal.acpi_tables module

HAL component decoding various ACPI tables

class ACPI_TABLE
Bases: object

parse (table_content: bytes) → None

class ACPI_TABLE_APIC_GICC_CPU (Type, Length, Reserved, CPUIntNumber, ACPIProcUID, Flags, ParkingProtocolVersion, PerformanceInterruptGSIV, ParkedAddress, PhysicalAddress, GICV, GICH, VGICMaintenanceINterrupt, GICRBaseAddress, MPIDR)
Bases: ACPI_TABLE_APIC_GICC_CPU

class ACPI_TABLE_APIC_GIC_DISTRIBUTOR (Type, Length, Reserved, GICID, PhysicalBaseAddress, SystemVectorBase, Reserved2)
Bases: ACPI_TABLE_APIC_GIC_DISTRIBUTOR

class ACPI_TABLE_APIC_GIC_MSI (Type, Length, Reserved, GICMSIFrameID, PhysicalBaseAddress, Flags, SPICount, SPIBase)
Bases: ACPI_TABLE_APIC_GIC_MSI

class ACPI_TABLE_APIC_GIC_REDISTRIBUTOR (Type, Length, Reserved, DiscoverRangeBaseAdd, DiscoverRangeLength)
Bases: ACPI_TABLE_APIC_GIC_REDISTRIBUTOR

class ACPI_TABLE_APIC_INTERRUPT_SOURSE_OVERRIDE (Type, Length, Bus, Source, GlobalSysIntBase, Flags)
Bases: ACPI_TABLE_APIC_INTERRUPT_SOURSE_OVERRIDE

class ACPI_TABLE_APIC_IOAPIC (Type, Length, IOAPICID, Reserved, IOAPICAddr, GlobalSysIntBase)
Bases: ACPI_TABLE_APIC_IOAPIC

class ACPI_TABLE_APIC_IOSAPIC (Type, Length, IOAPICID, Reserved, GlobalSysIntBase, IOSAPICAddress)
Bases: ACPI_TABLE_APIC_IOSAPIC

class ACPI_TABLE_APIC_LAPIC_ADDRESS_OVERRIDE (Type, Length, Reserved, LocalAPICAddress)
Bases: ACPI_TABLE_APIC_LAPIC_ADDRESS_OVERRIDE

class ACPI_TABLE_APIC_LAPIC_NMI (Type, Length, ACPIProcessorID, Flags, LocalAPICLINT)
Bases: ACPI_TABLE_APIC_LAPIC_NMI

class ACPI_TABLE_APIC_Lx2APIC_NMI (Type, Length, Flags, ACPIProcUID, Localx2APICLINT, Reserved)
Bases: ACPI_TABLE_APIC_Lx2APIC_NMI

class ACPI_TABLE_APIC_NMI_SOURCE (Type, Length, Flags, GlobalSysIntBase)
Bases: ACPI_TABLE_APIC_NMI_SOURCE

class ACPI_TABLE_APIC_PLATFORM_INTERRUPT_SOURCES (Type, Length, Flags, InterruptType, ProcID, ProcEID, IOSAPICVector, GlobalSystemInterrupt, PlatIntSourceFlags)

Bases: ACPI_TABLE_APIC_PLATFORM_INTERRUPT_SOURCES

class ACPI_TABLE_APIC_PROCESSOR_LAPIC (Type, Length, ACPIProcID, APICID, Flags)
Bases: ACPI_TABLE_APIC_PROCESSOR_LAPIC

class ACPI_TABLE_APIC_PROCESSOR_LSAPIC (Type, Length, ACPIProcID, LocalSAPICID, LocalSAPICEID, Reserved, Flags, ACPIProcUIDValue, ACPIProcUIDString)
Bases: ACPI_TABLE_APIC_PROCESSOR_LSAPIC

class ACPI_TABLE_APIC_PROCESSOR_Lx2APIC (Type, Length, Reserved, x2APICID, Flags, ACPIProcUID)
Bases: ACPI_TABLE_APIC_PROCESSOR_Lx2APIC

class ACPI_TABLE_DMAR_ANDD (Type, Length, Reserved, ACPIDevNum, ACPIObjectName)
Bases: ACPI_TABLE_DMAR_ANDD

class ACPI_TABLE_DMAR_ATSR (Type, Length, Flags, Reserved, SegmentNumber, DeviceScope)
Bases: ACPI_TABLE_DMAR_ATSR

class ACPI_TABLE_DMAR_DRHD (Type, Length, Flags, Reserved, SegmentNumber, RegisterBaseAddr, DeviceScope)
Bases: ACPI_TABLE_DMAR_DRHD

class ACPI_TABLE_DMAR_DeviceScope (Type, Length, Flags, Reserved, EnumerationID, StartBusNum, Path)
Bases: ACPI_TABLE_DMAR_DeviceScope

class ACPI_TABLE_DMAR_RHSA (Type, Length, Reserved, RegisterBaseAddr, ProximityDomain)
Bases: ACPI_TABLE_DMAR_RHSA

class ACPI_TABLE_DMAR_RMRR (Type, Length, Reserved, SegmentNumber, RMRBaseAddr, RMRLimitAddr, DeviceScope)
Bases: ACPI_TABLE_DMAR_RMRR

class ACPI_TABLE_DMAR_SATC (Type, Length, Flags, Reserved, SegmentNumber, DeviceScope)
Bases: ACPI_TABLE_DMAR_SATC

class ACPI_TABLE_DMAR_SIDP (Type, Length, Reserved, SegmentNumber, DeviceScope)
Bases: ACPI_TABLE_DMAR_SIDP

class APIC
Bases: ACPI_TABLE

get_structure_APIC (value: int, DataStructure: bytes) → str

parse (table_content: bytes) → None

class BERT (bootRegion: bytes)
Bases: ACPI_TABLE

parse (table_content: bytes) → None

parseErrorBlock (table_content: bytes) → None

parseGenErrorEntries (table_content: bytes) → str

parseSectionType (table_content: bytes) → str

parseTime (table_content: bytes) → str

class BGRT
Bases: ACPI_TABLE

parse (table_content: bytes) → None

class DMAR
Bases: ACPI_TABLE

parse (table_content: bytes) → None

class EINJ
Bases: ACPI_TABLE

parse (table_content: bytes) → None

parseAddress (table_content: bytes) → str

parseInjection (table_content: bytes) → None

parseInjectionActionTable (table_contents: bytes, numInjections: int) → None

class ERST
Bases: ACPI_TABLE

parse (table_content: bytes) → None

parseActionTable (table_content: bytes, instrCountEntry: int) → None

parseAddress (table_content: bytes) → str

parseInstructionEntry (table_content: bytes) → None

class FADT
Bases: ACPI_TABLE

get_DSDT_address_to_use () → int | None

parse (table_content: bytes) → None

class GAS (table_content: bytes)
Bases: object

get_info () → Tuple[int, int, int, int, int]

class HEST
Bases: ACPI_TABLE

machineBankParser (table_content: bytes) → None

parse (table_content: bytes) → None

parseAMCES (table_content: bytes) → int

parseAMCS (table_content: bytes, _type: int) → int

parseAddress (table_content: bytes) → str

parseErrEntry (table_content: bytes) → int | None

parseGHESS (table_content: bytes, _type: int) → int

parseNMIStructure (table_content: bytes) → int

parseNotify (table_content: bytes) → str

parsePCIe (table_content: bytes, _type: int) → int

class MSCT
Bases: ACPI_TABLE

parse (table_content: bytes) → None

parseProx (table_content: bytes, val: int) → str

parseProxDomInfoStruct (table_contents: bytes, num: int) → str

class NFIT (header)
Bases: ACPI_TABLE

flushHintAddrStruct (tableLen: int, table_content: bytes) → Tuple[int, str]

interleave (tableLen: int, table_content: bytes) → Tuple[int, str]

nvdimmBlockDataWindowsRegionStruct (tableLen: int, table_content: bytes) → str

nvdimmControlRegionStructMark (tableLen: int, table_content: bytes) → str

parse (table_content: bytes) → None

parseMAP (tableLen: int, table_content: bytes) → str

parseSPA (tableLen: int, table_content: bytes) → str

parseStructures (table_content: bytes) → str

platCapStruct (tableLen: int, table_content: bytes) → str

smbiosManagementInfo (tableLen: int, table_content: bytes) → str

class RASF
Bases: ACPI_TABLE

parse (table_content: bytes) → None

class RSDP
Bases: ACPI_TABLE

is_RSDP_valid () → bool

parse (table_content: bytes) → None

class RSDT
Bases: ACPI_TABLE

parse (table_content: bytes) → None

class SPMI
Bases: ACPI_TABLE

parse (table_content: bytes) → None

parseAddress (table_content: bytes) → str

parseNonUID (table_content: bytes) → str

parseUID (table_content: bytes) → str

class UEFI_TABLE
Bases: ACPI_TABLE

CommBuffInfo
alias of Tuple[int, int, Optional[GAS]]

get_commbuf_info () → Tuple[int, int, GAS | None]

parse (table_content: bytes) → None

class WSMT
Bases: ACPI_TABLE

COMM_BUFFER_NESTED_PTR_PROTECTION = *2*

FIXED_COMM_BUFFERS = *1*

SYSTEM_RESOURCE_PROTECTION = *4*

parse (table_content: bytes) → None

class XSDT
Bases: ACPI_TABLE

parse (table_content: bytes) → None

chipsec.hal.cmos module

CMOS memory specific functions (dump, read/write)

usage:

```
>>> cmos.dump_low()
>>> cmos.dump_high()
>>> cmos.dump()
>>> cmos.read_cmos_low( offset )
>>> cmos.write_cmos_low( offset, value )
```

```
>>> cmos.read_cmos_high( offset )
>>> cmos.write_cmos_high( offset, value )
```

class CMOS (cs)
Bases: HALBase

dump () → None

dump_high () → List[int]

dump_low () → List[int]

read_cmos_high (offset: int) → int

read_cmos_low (offset: int) → int

write_cmos_high (offset: int, value: int) → None

write_cmos_low (offset: int, value: int) → None

chipsec.hal.cpu module

CPU related functionality

class CPU (cs)
Bases: HALBase

check_SMRR_supported () → bool

check_vmm () → int

cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

dump_page_tables (cr3: int, pt_fname: str | None = None) → None

dump_page_tables_all () → None

get_SMRAM () → Tuple[int, int, int]

get_SMRR () → Tuple[int, int]

get_SMRR_SMRAM () → Tuple[int, int, int]

get_TSEG () → Tuple[int, int, int]

get_cpu_topology () → Dict[str, Dict[int, List[int]]]

get_number_logical_processor_per_core () → int

get_number_logical_processor_per_package () → int

get_number_physical_processor_per_package () → int

get_number_sockets_from_APIC_table () → int

get_number_threads_from_APIC_table () → int

is_HT_active () → bool

read_cr (cpu_thread_id: int, cr_number: int) → int

write_cr (cpu_thread_id: int, cr_number: int, value: int) → int

chipsec.hal.cpuid module

CPUID information

usage:

```
>>> cpuid(0)
```

class CpuID (cs)

Bases: HALBase

cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

get_proc_info ()

chipsec.hal.ec module

Access to Embedded Controller (EC)

Usage:

```
>>> write_command( command )
>>> write_data( data )
>>> read_data()
>>> read_memory( offset )
>>> write_memory( offset, data )
>>> read_memory_extended( word_offset )
>>> write_memory_extended( word_offset, data )
>>> read_range( start_offset, size )
>>> write_range( start_offset, buffer )
```

class EC (cs)

Bases: HALBase

read_data () → int | None

read_idx (offset: int) → int

read_memory (offset: int) → int | None

read_memory_extended (word_offset: int) → int | None

read_range (start_offset: int, size: int) → bytes

write_command (command: int) → None

write_data (data: int) → None

write_idx (offset: int, value: int) → bool

write_memory (offset: int, data: int) → None

write_memory_extended (word_offset: int, data: int) → None

write_range (start_offset: int, buffer: bytes) → bool

chipsec.hal.hal_base module

Base for HAL Components

class HALBase (cs)
Bases: object

chipsec.hal.igd module

Working with Intel processor Integrated Graphics Device (IGD)

usage:

```
>>> gfx_aperture_dma_read(0x80000000, 0x100)
```

class IGD (cs)
Bases: HALBase

dump_GGTT_PTEs (num: int) → None

get_GGTT_PTE_from_PA (pa: int) → int

get_GGTT_PTE_from_PA_gen8 (pa: int) → int

get_GGTT_PTE_from_PA_legacy (pa: int) → int

get_GGTT_base () → int

get_GMADR () → int

get_GTTMMADR () → int

get_PA_from_PTE (pte: int) → int

get_PA_from_PTE_gen8 (pte: int) → int

get_PA_from_PTE_legacy (pte: int) → int

get_PTE_size () → int

gfx_aperture_dma_read_write (address: int, size: int = 4, value: bytes | None = None, pte_num: int = 0) → bytes

is_device_enabled () → bool

is_enabled () → bool

is_legacy_gen () → bool

read_GGTT_PTE (pte_num: int) → int

write_GGTT_PTE (pte_num: int, pte: int) → int

write_GGTT_PTE_from_PA (pte_num: int, pa: int) → int

chipsec.hal.interrupts module

Functionality encapsulating interrupt generation CPU Interrupts specific functions (SMI, NMI)

usage:

```
>>> send_SMI_APMC( 0xDE )
>>> send_NMI()
```

class Interrupts (cs)
Bases: HALBase

find_ACPI_SMI_Buffer () → Tuple[int, int, GAS | None] | None

find_smmc (start: int, end: int) → int

send_ACPI_SMI (thread_id: int, smi_num: int, buf_addr: int, invoc_reg: GAS, guid: str, data: bytes) → int | None

send_NMI () → None

send_SMI_APMC (SMI_code_port_value: int, SMI_data_port_value: int) → None

send_SW_SMI (thread_id: int, SMI_code_port_value: int, SMI_data_port_value: int, _rax: int, _rbx: int, _rcx: int, _rdx: int, _rsi: int, _rdi: int) → Tuple[int, int, int, int, int, int, int] | None

send_smmc_SMI (smmc: int, guid: str, payload: bytes, payload_loc: int, CommandPort: int = 0, DataPort: int = 0) → int

chipsec.hal.io module

Access to Port I/O

usage:

```
>>> read_port_byte( 0x61 )
>>> read_port_word( 0x61 )
>>> read_port_dword( 0x61 )
>>> write_port_byte( 0x71, 0 )
>>> write_port_word( 0x71, 0 )
>>> write_port_dword( 0x71, 0 )
```

class PortIO (cs)
Bases: object

dump_IO (range_base: int, range_size: int, size: int = 1) → None

read_IO (range_base: int, range_size: int, size: int = 1) → List[int]

read_port_byte (io_port: int) → int

read_port_dword (io_port: int) → int

read_port_word (io_port: int) → int

write_port_byte (io_port: int, value: int) → None

write_port_dword (io_port: int, value: int) → None

write_port_word (io_port: int, value: int) → None

chipsec.hal.iobar module

I/O BAR access (dump, read/write)

usage:

```
>>> get_IO_BAR_base_address( bar_name )
>>> read_IO_BAR_reg( bar_name, offset, size )
>>> write_IO_BAR_reg( bar_name, offset, size, value )
>>> dump_IO_BAR( bar_name )
```

class IOBAR (cs)
Bases: HALBase

dump_IO_BAR (bar_name: str, size: int = 1) → None

get_IO_BAR_base_address (bar_name: str) → Tuple[int, int]

is_IO_BAR_defined (bar_name: str) → bool

is_IO_BAR_enabled (bar_name: str) → bool

list_IO_BARs () → None

read_IO_BAR (bar_name: str, size: int = 1) → List[int]

read_IO_BAR_reg (bar_name: str, offset: int, size: int) → int

write_IO_BAR_reg (bar_name: str, offset: int, size: int, value: int) → int

chipsec.hal.iommu module

Access to IOMMU engines

class IOMMU (cs)
Bases: HALBase

dump_IOMMU_configuration (iommu_engine: str) → None

dump_IOMMU_page_tables (iommu_engine: str) → None

dump_IOMMU_status (iommu_engine: str) → None

get_IOMMU_Base_Address (iommu_engine: str) → int

is_IOMMU_Engine_Enabled (iommu_engine: str) → bool

is_IOMMU_Translation_Enabled (iommu_engine: str) → bool

set_IOMMU_Translation (iommu_engine: str, te: int) → bool

chipsec.hal.locks module

class LockResult
Bases: object

CAN_READ = *8*

DEFINED = *1*

HAS_CONFIG = *2*

INCONSISTENT = *16*

LOCKED = *4*

class locks (cs)
Bases: HALBase

get_locks () → List[str]
Return a list of locks defined within the configuration file

is_locked (lock_name: str, bus: int | None = None) → int
Return whether the lock has the value setting

lock_valid (lock_name: str, bus: int | None = None) → int

chipsec.hal.mmio module

Access to MMIO (Memory Mapped IO) BARs and Memory-Mapped PCI Configuration Space (MMCFG)

usage:

```
>>> read_MMIO_reg(cs, bar_base, 0x0, 4)
>>> write_MMIO_reg(cs, bar_base, 0x0, 0xFFFFFFFF, 4)
>>> read_MMIO(cs, bar_base, 0x1000)
>>> dump_MMIO(cs, bar_base, 0x1000)
```

Access MMIO by BAR name:

```
>>> read_MMIO_BAR_reg(cs, 'MCHBAR', 0x0, 4)
>>> write_MMIO_BAR_reg(cs, 'MCHBAR', 0x0, 0xFFFFFFFF, 4)
>>> get_MMIO_BAR_base_address(cs, 'MCHBAR')
>>> is_MMIO_BAR_enabled(cs, 'MCHBAR')
>>> is_MMIO_BAR_programmed(cs, 'MCHBAR')
>>> dump_MMIO_BAR(cs, 'MCHBAR')
>>> list_MMIO_BARs(cs)
```

Access Memory Mapped Config Space:

```
>>> get_MMCFG_base_address(cs)
>>> read_mmcfg_reg(cs, 0, 0, 0, 0x10, 4)
>>> read_mmcfg_reg(cs, 0, 0, 0, 0x10, 4, 0xFFFFFFFF)
```

class ECEntry (bus: int, dev: int, fun: int, off: int, value: int)
Bases: object

class MMIO (cs)
Bases: HALBase

dump_MMIO (bar_base: int, size: int) → None

dump_MMIO_BAR (bar_name: str) → None

enable_cache_address_resolution (enable: bool) → None

flush_bar_address_cache () → None

get_MMCFG_base_address () → Tuple[int, int]

get_MMIO_BAR_base_address (bar_name: str, bus: int | None = None) → Tuple[int, int]

get_extended_capabilities (bus: int, dev: int, fun: int) → List[ECEntry]

get_vsec (bus: int, dev: int, fun: int, ecoff: int) → VSECEntry

is_MMIO_BAR_defined (bar_name: str) → bool

is_MMIO_BAR_enabled (bar_name: str, bus: int | None = None) → bool

is_MMIO_BAR_programmed (bar_name: str) → bool

list_MMIO_BARs () → None

read_MMIO (bar_base: int, size: int) → List[int]

read_MMIO_BAR (bar_name: str, bus: int | None = None) → List[int]

read_MMIO_BAR_reg (bar_name: str, offset: int, size: int = 4, bus: int | None = None) → int

read_MMIO_reg (bar_base: int, offset: int, size: int = 4, bar_size: int | None = None) → int

read_MMIO_reg_byte (bar_base: int, offset: int) → int

read_MMIO_reg_dword (bar_base: int, offset: int) → int

read_MMIO_reg_word (bar_base: int, offset: int) → int

read_mmcfg_reg (bus: int, dev: int, fun: int, off: int, size: int) → int

write_MMIO_BAR_reg (bar_name: str, offset: int, value: int, size: int = 4, bus: int | None = None) → int | None

write_MMIO_reg (bar_base: int, offset: int, value: int, size: int = 4) → int

write_MMIO_reg_byte (bar_base: int, offset: int, value: int) → int

write_MMIO_reg_dword (bar_base: int, offset: int, value: int) → int

write_MMIO_reg_word (bar_base: int, offset: int, value: int) → int

write_mmcfg_reg (bus: int, dev: int, fun: int, off: int, size: int, value: int) → bool

class VSECEntry (value: int)
Bases: object

print_pci_extended_capability (ecentries: List[ECEntry]) → None

chipsec.hal.msgbus module

Access to message bus (IOSF sideband) interface registers on Intel SoCs

References:

- Intel(R) Atom(TM) Processor D2000 and N2000 Series Datasheet, Volume 2, July 2012, Revision 003 http://www.intel.com/content/dam/doc/datasheet/atom-d2000-n2000-vol-2-datasheet.pdf (section 1.10.2)

usage:

```
>>> msgbus_reg_read( port, register )
>>> msgbus_reg_write( port, register, data )
>>> msgbus_read_message( port, register, opcode )
>>> msgbus_write_message( port, register, opcode, data )
>>> msgbus_send_message( port, register, opcode, data )
```

class MessageBusOpcode
Bases: object

MB_OPCODE_CFG_READ = *4*

MB_OPCODE_CFG_WRITE = *5*

MB_OPCODE_CR_READ = *6*

MB_OPCODE_CR_WRITE = *7*

MB_OPCODE_ESRAM_READ = *18*

MB_OPCODE_ESRAM_WRITE = *19*

MB_OPCODE_IO_READ = *2*

MB_OPCODE_IO_WRITE = *3*

MB_OPCODE_MMIO_READ = *0*

MB_OPCODE_MMIO_WRITE = *1*

MB_OPCODE_REG_READ = *16*

MB_OPCODE_REG_WRITE = *17*

class MessageBusPort_Atom
Bases: object

UNIT_AUNIT = *0*

UNIT_BUNIT = *3*

UNIT_CPU = *2*

UNIT_GFX = *6*

UNIT_PCIE = *166*

UNIT_PMC = *4*

UNIT_SATA = *163*

UNIT_SMC = *1*

UNIT_SMI = *12*

UNIT_USB = *67*

class MessageBusPort_Quark
Bases: object

UNIT_HB = *3*

UNIT_HBA = *0*

UNIT_MM = *5*

UNIT_RMU = *4*

UNIT_SOC = *49*

class MsgBus (cs)
Bases: HALBase

mm_msgbus_reg_read (port: int, register: int) → int

mm_msgbus_reg_write (port: int, register: int, data: int) → int | None

msgbus_read_message (port: int, register: int, opcode: int) → int | None

msgbus_reg_read (port: int, register: int) → int | None

msgbus_reg_write (port: int, register: int, data: int) → None

msgbus_send_message (port: int, register: int, opcode: int, data: int | None = None) → int | None

msgbus_write_message (port: int, register: int, opcode: int, data: int) → None

chipsec.hal.msr module

Access to CPU resources (for each CPU thread): Model Specific Registers (MSR), IDT/GDT

usage:

```
>>> read_msr( 0x8B )
>>> write_msr( 0x79, 0x12345678 )
>>> get_IDTR( 0 )
>>> get_GDTR( 0 )
>>> dump_Descriptor_Table( 0, DESCRIPTOR_TABLE_CODE_IDTR )
>>> IDT( 0 )
>>> GDT( 0 )
>>> IDT_all()
>>> GDT_all()
```

class Msr (cs)
Bases: object

GDT (cpu_thread_id: int, num_entries: int | None = None) → Tuple[int, int]

GDT_all (num_entries: int | None = None) → None

IDT (cpu_thread_id: int, num_entries: int | None = None) → Tuple[int, int]

IDT_all (num_entries: int | None = None) → None

dump_Descriptor_Table (cpu_thread_id: int, code: int, num_entries: int | None = None) → Tuple[int, int]

get_Desc_Table_Register (cpu_thread_id: int, code: int) → Tuple[int, int, int]

get_GDTR (cpu_thread_id: int) → Tuple[int, int, int]

get_IDTR (cpu_thread_id: int) → Tuple[int, int, int]

get_LDTR (cpu_thread_id: int) → Tuple[int, int, int]

get_cpu_core_count () → int

get_cpu_thread_count () → int

read_msr (cpu_thread_id: int, msr_addr: int) → Tuple[int, int]

write_msr (cpu_thread_id: int, msr_addr: int, eax: int, edx: int) → None

chipsec.hal.paging module

x64/IA-64 Paging functionality including x86 page tables, Extended Page Tables (EPT) and VT-d page tables

class c_4level_page_tables (cs)
Bases: c_paging

get_attr (entry: int) → str

get_virt_addr (pml4e_index: int, pdpte_index: int = 0, pde_index: int = 0, pte_index: int = 0) → int

is_bigpage (entry: int) → int

is_present (entry: int) → int

print_entry (lvl: int, pa: int, va: int = 0, perm: str = '') → None

read_entry_by_virt_addr (virt: int) → Dict[str, Any]

read_page_tables (ptr: int) → None

read_pd (addr: int, pml4e_index: int, pdpte_index: int) → None

read_pdpt (addr: int, pml4e_index: int) → None

read_pml4 (addr: int) → None

read_pt (addr: int, pml4e_index: int, pdpte_index: int, pde_index: int) → None

class c_extended_page_tables (cs)
Bases: c_4level_page_tables

get_attr (entry: int) → str

is_bigpage (entry: int) → bool

is_present (entry: int) → bool

map_bigpage_1G (virt: int, i: int) → None

read_pt_and_show_status (path: str, name: str, ptr: int) → None

class c_ia32e_page_tables (cs)
Bases: c_4level_page_tables

get_attr (entry: int) → str

is_bigpage (entry: int) → bool

is_present (entry: int) → bool

class c_pae_page_tables (cs)
Bases: c_ia32e_page_tables

read_page_tables (ptr: int) → None

read_pdpt (addr: int, pml4e_index: int | None = None) → None

read_pml4 (addr: int)

class c_paging (cs)
Bases: c_paging_with_2nd_level_translation, c_translation

check_misconfig (addr_list: List[int]) → None

get_canonical (va: int) → int

get_field (entry: int, desc: Dict[str, int]) → int

load_configuration (path: str) → None

print_info (name: str) → None

read_entries (info: str, addr: int, size: int = 8) → List[Any]

read_page_tables (entry: int)

read_pt_and_show_status (path: str, name: str, ptr: int) → None

save_configuration (path: str) → None

set_field (value: int, desc: Dict[str, int]) → int

class c_paging_memory_access (cs)
Bases: object

readmem (name: str, addr: int, size: int = 4096) → bytes

class c_paging_with_2nd_level_translation (cs)
Bases: c_paging_memory_access

readmem (name: str, addr: int, size: int = 4096) → bytes

class c_reverse_translation (translation: Dict[int, Dict[str, Any]])
Bases: object

get_reverse_translation (addr: int) → List[Dict[str, Any]]

class c_translation
Bases: object

add_page (virt: int, phys: int, size: str, attr: str) → None

del_page (addr: int) → None

expand_pages (exp_size: str) → None

get_address_space () → int

get_mem_range (noattr: bool = False) → List[List[int]]

get_pages_by_physaddr (addr: int) → List[Dict[str, int]]

get_translation (addr: int) → int | None

is_translation_exist (addr: int, mask: int, size: str) → bool

class c_vtd_page_tables (cs)
Bases: c_extended_page_tables

print_context_entry (source_id: int, cee: Dict[int, int]) → None

read_ce (addr: int, ree_index: int) → None

read_page_tables (ptr: int) → None

read_pt_and_show_status (path: str, name: str, ptr: int) → None

read_re (addr: int) → None

read_vtd_context (path: str, ptr: int) → None

chipsec.hal.pci module

Access to of PCI/PCIe device hierarchy - enumerating PCI/PCIe devices - read/write access to PCI configuration headers/registers - enumerating PCI expansion (option) ROMs - identifying PCI/PCIe devices MMIO and I/O ranges (BARs)

usage:

```
>>> self.cs.pci.read_byte( 0, 0, 0, 0x88 )
>>> self.cs.pci.write_byte( 0, 0, 0, 0x88, 0x1A )
>>> self.cs.pci.enumerate_devices()
>>> self.cs.pci.enumerate_xroms()
>>> self.cs.pci.find_XROM( 2, 0, 0, True, True, 0xFED00000 )
>>> self.cs.pci.get_device_bars( 2, 0, 0 )
>>> self.cs.pci.get_DIDVID( 2, 0, 0 )
>>> self.cs.pci.is_enabled( 2, 0, 0 )
```

class EFI_XROM_HEADER (Signature, InitSize, EfiSignature, EfiSubsystem, EfiMachineType, CompressType, Reserved, EfiImageHeaderOffset, PCIROffset)
Bases: EFI_XROM_HEADER

class PCI_XROM_HEADER (Signature, ArchSpecific, PCIROffset)
Bases: PCI_XROM_HEADER

class Pci (cs)
Bases: object

calc_bar_size (bus: int, dev: int, fun: int, off: int, reg) → int

dump_pci_config (bus: int, device: int, function: int) → List[int]

enumerate_devices (bus: int | None = None, device: int | None = None, function: int | None = None, spec: bool | None = True) → List[Tuple[int, int, int, int, int, int]]

enumerate_xroms (try_init: bool = False, xrom_dump: bool = False, xrom_addr: int | None = None) → List[XROM | None]

find_XROM (bus: int, dev: int, fun: int, try_init: bool = False, xrom_dump: bool = False, xrom_addr: int | None = None) → Tuple[bool, XROM | None]

get_DIDVID (bus: int, dev: int, fun: int) → Tuple[int, int]

get_device_bars (bus: int, dev: int, fun: int, bCalcSize: bool = False) → List[Tuple[int, bool, bool, int, int, int]]

is_enabled (bus: int, dev: int, fun: int) → bool

parse_XROM (xrom: XROM, xrom_dump: bool = False) → PCI_XROM_HEADER | None

print_pci_config_all () → None

read_byte (bus: int, device: int, function: int, address: int) → int

read_dword (bus: int, device: int, function: int, address: int) → int

read_word (bus: int, device: int, function: int, address: int) → int

write_byte (bus: int, device: int, function: int, address: int, byte_value: int) → None

write_dword (bus: int, device: int, function: int, address: int, dword_value: int) → None

write_word (bus: int, device: int, function: int, address: int, word_value: int) → None

class XROM (bus, dev, fun, en, base, size)
Bases: object

class XROM_HEADER (Signature, InitSize, InitEP, Reserved, PCIROffset)
Bases: XROM_HEADER

get_device_name_by_didvid (vid: int, did: int) → str

get_vendor_name_by_vid (vid: int) → str

print_pci_XROMs (_xroms: List[XROM]) → None

print_pci_devices (_devices: List[Tuple[int, int, int, int, int]]) → None

chipsec.hal.pcidb module

PCI Vendor & Device ID data.

Note

THIS FILE WAS GENERATED

Auto generated from:

https://github.com/pciutils/pciids

chipsec.hal.physmem module

Access to physical memory

usage:

```
>>> read_physical_mem( 0xf0000, 0x100 )
>>> write_physical_mem( 0xf0000, 0x100, buffer )
>>> write_physical_mem_dowrd( 0xf0000, 0xdeadbeef )
>>> read_physical_mem_dowrd( 0xfed40000 )
```

class Memory (cs)
Bases: HALBase

alloc_physical_mem (length: int, max_phys_address: int = 18446744073709551615) → Tuple[int, int]

free_physical_mem (pa: int) → bool

map_io_space (pa: int, length: int, cache_type: int) → int

read_physical_mem (phys_address: int, length: int) → bytes

read_physical_mem_byte (phys_address: int) → int

read_physical_mem_dword (phys_address: int) → int

read_physical_mem_qword (phys_address: int) → int

read_physical_mem_word (phys_address: int) → int

set_mem_bit (addr: int, bit: int) → int

va2pa (va: int) → int | None

write_physical_mem (phys_address: int, length: int, buf: bytes) → int

write_physical_mem_byte (phys_address: int, byte_value: int) → int

write_physical_mem_dword (phys_address: int, dword_value: int) → int

write_physical_mem_word (phys_address: int, word_value: int) → int

chipsec.hal.smbios module

HAL component providing access to and decoding of SMBIOS structures

class SMBIOS (cs)
Bases: HALBase

find_smbios_table () → bool

get_decoded_structs (struct_type: int | None = None, force_32bit: bool = False) → List[Type[SMBIOS_BIOS_INFO_2_0 | SMBIOS_SYSTEM_INFO_2_0]] | None

get_header (raw_data: bytes) → SMBIOS_STRUCT_HEADER | None

get_raw_structs (struct_type: int | None, force_32bit: bool)
Returns a list of raw data blobs for each SMBIOS structure. The default is to process the 64bit entries if available unless specifically specified.
Error: None

get_string_list (raw_data: bytes) → List[str] | None

class SMBIOS_2_x_ENTRY_POINT (Anchor, EntryCs, EntryLen, MajorVer, MinorVer, MaxSize, EntryRev, FormatArea0, FormatArea1, FormatArea2, FormatArea3, FormatArea4, IntAnchor, IntCs, TableLen, TableAddr, NumStructures, BcdRev)
Bases: SMBIOS_2_x_ENTRY_POINT

class SMBIOS_3_x_ENTRY_POINT (Anchor, EntryCs, EntryLen, MajorVer, MinorVer, Docrev, EntryRev, Reserved, MaxSize, TableAddr)
Bases: SMBIOS_3_x_ENTRY_POINT

class SMBIOS_BIOS_INFO_2_0 (type, length, handle, vendor_str, version_str, segment, release_str, rom_sz, bios_char, strings)
Bases: SMBIOS_BIOS_INFO_2_0_ENTRY

class SMBIOS_STRUCT_HEADER (Type, Length, Handle)
Bases: SMBIOS_STRUCT_HEADER

class SMBIOS_SYSTEM_INFO_2_0 (type, length, handle, manufacturer_str, product_str, version_str, serial_str, strings)
Bases: SMBIOS_SYSTEM_INFO_2_0_ENTRY

chipsec.hal.smbus module

Access to SMBus Controller

class SMBus (cs)
Bases: HALBase

display_SMBus_info () → None

enable_SMBus_host_controller () → None

get_SMBus_Base_Address () → int

get_SMBus_HCFG () → int

is_SMBus_enabled () → bool

is_SMBus_host_controller_enabled () → int

is_SMBus_supported () → bool

read_byte (target_address: int, offset: int) → int

read_range (target_address: int, start_offset: int, size: int) → bytes

reset_SMBus_controller () → bool

write_byte (target_address: int, offset: int, value: int) → bool

write_range (target_address: int, start_offset: int, buffer: bytes) → bool

chipsec.hal.spd module

Access to Memory (DRAM) Serial Presence Detect (SPD) EEPROM

References:

http://www.jedec.org/sites/default/files/docs/4_01_02R19.pdf
http://www.jedec.org/sites/default/files/docs/4_01_02_10R17.pdf

http://www.jedec.org/sites/default/files/docs/4_01_02_11R24.pdf
http://www.jedec.org/sites/default/files/docs/4_01_02_12R23A.pdf
https://www.simmtester.com/News/PublicationArticle/184
https://www.simmtester.com/News/PublicationArticle/153
https://www.simmtester.com/News/PublicationArticle/101 http://en.wikipedia.org/wiki/Serial_presence_detect

class SPD (smbus)
Bases: object

decode (device: int = 160) → None

detect () → List[int]

dump_spd_rom (device: int = 160) → bytes

getDRAMDeviceType (device: int = 160) → int

getModuleType (device: int = 160) → int

isECC (device: int = 160) → bool

isSPDPresent (device: int = 160) → bool

read_byte (offset: int, device: int = 160) → int

read_range (start_offset: int, size: int, device: int = 160) → bytes

write_byte (offset: int, value: int, device: int = 160) → bool

write_range (start_offset: int, buffer: bytes, device: int = 160) → bool

class SPD_DDR (SPDBytes, TotalBytes, DeviceType, RowAddressCount)
Bases: SPD_DDR

class SPD_DDR2 (SPDBytes, TotalBytes, DeviceType, RowAddressCount)
Bases: SPD_DDR2

class SPD_DDR3 (SPDBytes, Revision, DeviceType, ModuleType, ChipSize, Addressing, Voltages, ModuleOrg, BusWidthECC, FTB, MTBDivident, MTBDivisor, tCKMin, RsvdD, CASLo, CASHi)
Bases: SPD_DDR3

class SPD_DDR4 (SPDBytes, Revision, DeviceType, ModuleType, Density, Addressing, PackageType, OptFeatures, ThermalRefresh, OptFeatures1, ReservedA, VDD, ModuleOrg, BusWidthECC, ThermSensor, ModuleTypeExt)
Bases: SPD_DDR4

SPD_REVISION (revision: int) → str

dram_device_type_name (dram_type: int) → str

module_type_name (module_type: int) → str

chipsec.hal.spi module

Access to SPI Flash parts

usage:

```
>>> read_spi( spi_fla, length )
>>> write_spi( spi_fla, buf )
>>> erase_spi_block( spi_fla )
>>> get_SPI_JEDEC_ID()
>>> get_SPI_JEDEC_ID_decoded()
```

Note

!! IMPORTANT: Size of the data chunk used in SPI read cycle (in bytes) default = maximum 64 bytes (remainder is read in 4 byte chunks)

If you want to change logic to read SPI Flash in 4 byte chunks: SPI_READ_WRITE_MAX_DBC = 4

@TBD: SPI write cycles operate on 4 byte chunks (not optimized yet)

Approximate performance (on 2-core SMT Intel Core i5-4300U (Haswell) CPU 1.9GHz): SPI read: ~7 sec per 1MB (with DBC=64)

class SPI (cs)
Bases: HALBase

SpiRegions
alias of Dict[int, Tuple[int, int, int, str, int]]

check_hardware_sequencing () → None

disable_BIOS_write_protection () → bool

display_BIOS_region () → None

display_BIOS_write_protection () → None

display_SPI_Flash_Descriptor () → None

display_SPI_Flash_Regions () → None

display_SPI_Protected_Ranges () → None

display_SPI_Ranges_Access_Permissions () → None

display_SPI_map () → None

display_SPI_opcode_info () → None

erase_spi_block (spi_fla: int) → bool

get_SPI_JEDEC_ID () → int

get_SPI_JEDEC_ID_decoded () → Tuple[int, str, str]

get_SPI_MMIO_base () → int

get_SPI_Protected_Range (pr_num: int) → Tuple[int, int, int, int, int, int]

get_SPI_SFDP () → bool

get_SPI_region (spi_region_id: int) → Tuple[int, int, int]

get_SPI_regions (all_regions: bool = True) → Dict[int, Tuple[int, int, int, str, int]]

ptmesg (offset: int) → int

read_spi (spi_fla: int, data_byte_count: int) → bytes

read_spi_to_file (spi_fla: int, data_byte_count: int, filename: str) → bytes

spi_reg_read (reg: int, size: int = 4) → int

spi_reg_write (reg: int, value: int, size: int = 4) → int | None

write_spi (spi_fla: int, buf: bytes) → bool

write_spi_from_file (spi_fla: int, filename: str) → bool

get_SPI_region (flreg: int) → Tuple[int, int]

chipsec.hal.spi_descriptor module

SPI Flash Descriptor binary parsing functionality

usage:

```
>>> fd = read_file( fd_file )
>>> parse_spi_flash_descriptor( fd )
```

get_SPI_master (flmstr: int) → Tuple[int, int, int]

get_spi_flash_descriptor (rom: bytes) → Tuple[int, bytes]

get_spi_regions (fd: bytes) → List[Tuple[int, str, int, int, int, bool]] | None

parse_spi_flash_descriptor (cs, rom: bytes) → None

chipsec.hal.spi_jedec_ids module

JEDED ID : Manufacturers and Device IDs

class JEDEC_ID
Bases: object

DEVICE: *Dict[int, str] = {12722199: 'MX25L6408', 12722200: 'MX25L12805', 15679511: 'W25Q64FV (SPI)', 15679512: 'W25Q128 (SPI)', 15679513: 'W25Q256', 15687703: 'W25Q64FV (QPI)', 15687704: 'W25Q128 (QPI)', 15691798: 'W25Q32JV'}*

MANUFACTURER: *Dict[int, str] = {194: 'Macronix', 239: 'Winbond'}*

chipsec.hal.spi_uefi module

UEFI firmware image parsing and manipulation functionality

usage:

```
>>> parse_uefi_region_from_file(_uefi, filename, fwtype, outpath):
```

class EFIModuleType
Bases: object

FILE = *4*

FV = *2*

SECTION = *1*

SECTION_EXE = *0*

FILENAME (mod: EFI_FILE | EFI_SECTION, parent: EFI_MODULE | None, modn: int) → str

class UUIDEncoder (*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)
Bases: JSONEncoder

default (obj: Any)
Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a `TypeError`).
For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

build_efi_file_tree (fv_img: bytes, fwtype: str) → List[EFI_FILE]

build_efi_model (data: bytes, fwtype: str) → List[EFI_MODULE]

build_efi_modules_tree (fwtype: str, data: bytes, Size: int, offset: int, polarity: bool) → List[EFI_SECTION]

build_efi_tree (data: bytes, fwtype: str) → List[EFI_MODULE]

compress_image (image: bytes, compression_type: int) → bytes

decode_uefi_region (pth: str, fname: str, fwtype: str, filetype: List[int] = []) → None

decompress_section_data (section_dir_path: str, sec_fs_name: str, compressed_data: bytes, compression_type: int) → bytes

dump_efi_module (mod, parent: EFI_MODULE | None, modn: int, path: str) → str

efi_data_search (data: bytes, fwtype: str, polarity: bool)

```
modify_uefi_region (data: bytes, command: int, guid: UUID, uefi_file: bytes = b'') → bytes

parse_uefi_region_from_file (filename: str, fwtype: str, outpath: str | None = None, filetype: List[int]
= []) → None

save_efi_tree (modules: List[EFI_MODULE], parent: EFI_MODULE | None = None, save_modules:
bool = True, path: str = '', save_log: bool = True, lvl: int = 0) → List[Dict[str, Any]]

save_efi_tree_filetype (modules: List[EFI_MODULE], parent: EFI_MODULE | None = None, path:
str = '', lvl: int = 0, filetype: List[int] = [], save: bool = False) → List[Dict[str, Any]]

search_efi_tree (modules: List[EFI_MODULE], search_callback: Callable | None,
match_module_types: int = 0, findall: bool = True) → List[EFI_MODULE]

update_efi_tree (modules: List[EFI_MODULE], parent_guid: UUID | None = None) → str
```

chipsec.hal.tpm module

Trusted Platform Module (TPM) HAL component

https://trustedcomputinggroup.org

class TPM (cs)
Bases: HALBase

command (commandName: str, locality: str, *command_argv: str) → None
Send command to the TPM and receive data

dump_access (locality: str) → None
View the contents of the register used to gain ownership of the TPM

dump_didvid (locality: str) → None
TPM's Vendor and Device ID

dump_intcap (locality: str) → None
Provides information of which interrupts that particular TPM supports

dump_intenable (locality: str) → None
View the contents of the register used to enable specific interrupts

dump_register (register_name: str, locality: str) → None

dump_rid (locality: str) → None
TPM's Revision ID

dump_status (locality: str) → None
View general status details

log_register_header (register_name: str, locality: str) → None

class TPM_RESPONSE_HEADER (ResponseTag, DataSize, ReturnCode)
Bases: TPM_RESPONSE_HEADER

chipsec.hal.tpm12_commands module

Definition for TPMv1.2 commands to use with TPM HAL

TCG PC Client TPM Specification TCG TPM v1.2 Specification

continueselftest (*command_argv: str) → Tuple[bytes, int]
TPM_ContinueSelfTest informs the TPM that it should complete self-test of all TPM functions. The TPM may return success immediately and then perform the self-test, or it may perform the self-test and then return success or failure.

forceclear (*command_argv: str) → Tuple[bytes, int]

getcap (*command_argv: str) → Tuple[bytes, int]
Returns current information regarding the TPM CapArea - Capabilities Area SubCapSize - Size of SubCapabilities SubCap - Subcapabilities

nvread (*command_argv: str) → Tuple[bytes, int]
Read a value from the NV store Index, Offset, Size

pcrread (*command_argv: str) → Tuple[bytes, int]
The TPM_PCRRead operation provides non-cryptographic reporting of the contents of a named PCR

startup (*command_argv: str) → Tuple[bytes, int]
Execute a tpm_startup command. TPM_Startup is always preceded by TPM_Init, which is the physical indication (a system wide reset) that TPM initialization is necessary Type of Startup to be used: 1: TPM_ST_CLEAR 2: TPM_ST_STATE 3: TPM_ST_DEACTIVATED

chipsec.hal.tpm_eventlog module

Trusted Platform Module Event Log

Based on the following specifications:

TCG EFI Platform Specification For TPM Family 1.1 or 1.2

TCG PC Client Specific Implementation Specification for Conventional BIOS", version 1.21

TCG EFI Protocol Specification, Family "2.0"

TCG PC Client Platform Firmware Profile Specification

class EFIFirmwareBlob (*args: Any)
Bases: TcgPcrEvent

class PcrLogParser (log: BinaryIO)
Bases: object
Iterator over the events of a log.

next () → TcgPcrEvent

class SCRTMVersion (*args: Any)
Bases: TcgPcrEvent

class TcgPcrEvent (pcr_index: int, event_type: int, digest: bytes, event_size: int, event: Any)
Bases: object
An Event (TPM 1.2 format) as recorded in the SML.

classmethod parse (log: BinaryIO) → EventType | None
Try to read an event from the log.

Args:

log (file-like): Log where the event is stored.

Returns:

An instance of the created event. If a subclass exists for such event_type, an object of this class is returned. Otherwise, a TcgPcrEvent is returned.

parse (log: BinaryIO) → None
Simple wrapper around PcrLogParser.

chipsec.hal.ucode module

Microcode update specific functionality (for each CPU thread)

usage:

```
>>> ucode_update_id( 0 )
>>> load_ucode_update( 0, ucode_buf )
>>> update_ucode_all_cpus( 'ucode.pdb' )
>>> dump_ucode_update_header( 'ucode.pdb' )
```

class Ucode (cs)
Bases: object

get_cpu_thread_count () → int

load_ucode_update (cpu_thread_id: int, ucode_buf: AnyStr) → int

ucode_update_id (cpu_thread_id: int) → int

update_ucode (cpu_thread_id: int, ucode_file: str) → int

update_ucode_all_cpus (ucode_file: str) → bool

class UcodeUpdateHeader (header_version, update_revision, date, processor_signature, checksum, loader_revision, processor_flags, data_size, total_size, reserved1, reserved2, reserved3)
Bases: UcodeUpdateHeader

dump_ucode_update_header (pdb_ucode_buffer: bytes) → UcodeUpdateHeader

read_ucode_file (ucode_filename: str) → bytes

chipsec.hal.uefi module

Main UEFI component using platform specific and common UEFI functionality

class UEFI (cs)
Bases: HALBase

EfiTable
alias of Tuple[bool, int, Optional[EFI_TABLE_HEADER], Optional[EFI_SYSTEM_TABLE], bytes]

delete_EFI_variable (name: str, guid: str) → int | None

dump_EFI_tables () → None

dump_EFI_variables_from_SPI () → bytes

find_EFI_BootServices_Table () → Tuple[bool, int, EFI_TABLE_HEADER | None, EFI_SYSTEM_TABLE | None, bytes]

find_EFI_Configuration_Table () → Tuple[bool, int, EFI_CONFIGURATION_TABLE | None, bytes]

find_EFI_DXEServices_Table () → Tuple[bool, int, EFI_TABLE_HEADER | None, EFI_SYSTEM_TABLE | None, bytes]

find_EFI_RuntimeServices_Table () → Tuple[bool, int, EFI_TABLE_HEADER | None, EFI_SYSTEM_TABLE | None, bytes]

find_EFI_System_Table () → Tuple[bool, int, EFI_TABLE_HEADER | None, EFI_SYSTEM_TABLE | None, bytes]

find_EFI_Table (table_sig: str) → Tuple[bool, int, EFI_TABLE_HEADER | None, EFI_SYSTEM_TABLE | None, bytes]

find_s3_bootscript () → Tuple[bool, List[int]]

get_EFI_variable (name: str, guid: str, filename: str | None = None) → bytes | None

get_s3_bootscript (log_script: bool = False) → Tuple[List[int], Dict[int, List[S3BOOTSCRIPT_ENTRY]] | None]

list_EFI_variables () → Dict[str, List[Tuple[int, bytes, int, bytes, str, int]]] | None

read_EFI_variables (efi_var_store: bytes | None, authvars: bool) → Dict[str, List[EfiVariableType]]

read_EFI_variables_from_SPI (BIOS_region_base: int, BIOS_region_size: int) → bytes

read_EFI_variables_from_file (filename: str) → bytes

set_EFI_variable (name: str, guid: str, var: bytes, datasize: int | None = None, attrs: int | None = None) → int | None

set_EFI_variable_from_file (name: str, guid: str, filename: str, datasize: int | None = None, attrs: int | None = None) → int | None

set_FWType (efi_nvram_format: str) → None

decode_EFI_variables (efi_vars: Dict[str, List[EfiVariableType]], nvram_pth: str) → None

find_EFI_variable_store (rom_buffer: bytes | None, _FWType: str | None) → bytes

get_attr_string (attr: int) → str

get_auth_attr_string (attr: int) → str

identify_EFI_NVRAM (buffer: bytes) → str

parse_EFI_variables (fname: str, rom: bytes, authvars: bool, _fw_type: str | None = None) → bool

parse_script (script: bytes, log_script: bool = False) → List[S3BOOTSCRIPT_ENTRY]

print_efi_variable (offset: int, var_buf: bytes, var_header: EfiTableType, var_name: str, var_data: bytes, var_guid: str, var_attrib: int) → None

print_sorted_EFI_variables (variables: Dict[str, List[EfiVariableType]]) → None

chipsec.hal.uefi_common module

Common UEFI/EFI functionality including UEFI variables, Firmware Volumes, Secure Boot variables, S3 boot-script, UEFI tables, etc.

class EFI_BOOT_SERVICES_TABLE (RaiseTPL, RestoreTPL, AllocatePages, FreePages, GetMemoryMap, AllocatePool, FreePool, CreateEvent, SetTimer, WaitForEvent, SignalEvent, CloseEvent, CheckEvent, InstallProtocolInterface, ReinstallProtocolInterface, UninstallProtocolInterface, HandleProtocol, Reserved, RegisterProtocolNotify, LocateHandle, LocateDevicePath, InstallConfigurationTable, LoadImage, StartImage, Exit, UnloadImage, ExitBootServices, GetNextMonotonicCount, Stall, SetWatchdogTimer, ConnectController, DisconnectController, OpenProtocol, CloseProtocol, OpenProtocolInformation, ProtocolsPerHandle, LocateHandleBuffer, LocateProtocol, InstallMultipleProtocolInterfaces, UninstallMultipleProtocolInterfaces, CalculateCrc32, CopyMem, SetMem, CreateEventEx)
Bases: EFI_BOOT_SERVICES_TABLE

class EFI_CONFIGURATION_TABLE
Bases: object

class EFI_DXE_SERVICES_TABLE (AddMemorySpace, AllocateMemorySpace, FreeMemorySpace, RemoveMemorySpace, GetMemorySpaceDescriptor, SetMemorySpaceAttributes, GetMemorySpaceMap, AddIoSpace, AllocateIoSpace, FreeIoSpace, RemoveIoSpace, GetIoSpaceDescriptor, GetIoSpaceMap, Dispatch, Schedule, Trust, ProcessFirmwareVolume)
Bases: EFI_DXE_SERVICES_TABLE

EFI_ERROR_STR (error: int) → str
Translates an EFI_STATUS value into its corresponding textual representation.

EFI_GUID_STR (guid: bytes) → str

class EFI_RUNTIME_SERVICES_TABLE (GetTime, SetTime, GetWakeupTime, SetWakeupTime, SetVirtualAddressMap, ConvertPointer, GetVariable, GetNextVariableName, SetVariable, GetNextHighMonotonicCount, ResetSystem, UpdateCapsule, QueryCapsuleCapabilities, QueryVariableInfo)
Bases: EFI_RUNTIME_SERVICES_TABLE

class EFI_SYSTEM_TABLE (FirmwareVendor, FirmwareRevision, ConsoleInHandle, ConIn, ConsoleOutHandle, ConOut, StandardErrorHandle, StdErr, RuntimeServices, BootServices, NumberOfTableEntries, ConfigurationTable)
Bases: EFI_SYSTEM_TABLE

EFI_SYSTEM_TABLE_REVISION (revision: int) → str

class EFI_TABLE_HEADER (Signature, Revision, HeaderSize, CRC32, Reserved)
Bases: EFI_TABLE_HEADER

class EFI_VENDOR_TABLE (VendorGuidData, VendorTable)

Bases: EFI_VENDOR_TABLE

VendorGuid () → str

IS_EFI_VARIABLE_AUTHENTICATED (attr: int) → bool

IS_VARIABLE_ATTRIBUTE (_c: int, _Mask: int) → bool

class S3BOOTSCRIPT_ENTRY (script_type: int, index: int | None, offset_in_script: int, length: int, data: bytes | None = None)
Bases: object

class S3BootScriptOpcode
Bases: object

EFI_BOOT_SCRIPT_DISPATCH_OPCODE = *8*

EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE = *1*

EFI_BOOT_SCRIPT_IO_WRITE_OPCODE = *0*

EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE = *3*

EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE = *2*

EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE = *5*

EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE = *4*

EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE = *6*

EFI_BOOT_SCRIPT_STALL_OPCODE = *7*

EFI_BOOT_SCRIPT_TERMINATE_OPCODE = *255*

class S3BootScriptOpcode_EdkCompat
Bases: S3BootScriptOpcode

EFI_BOOT_SCRIPT_INFORMATION_OPCODE = *10*

EFI_BOOT_SCRIPT_MEM_POLL_OPCODE = *9*

EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE = *12*

EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE = *11*

EFI_BOOT_SCRIPT_TABLE_OPCODE = *170*

class S3BootScriptOpcode_MDE
Bases: S3BootScriptOpcode

EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE = *9*

EFI_BOOT_SCRIPT_INFORMATION_OPCODE = *10*

EFI_BOOT_SCRIPT_IO_POLL_OPCODE = *13*

EFI_BOOT_SCRIPT_MEM_POLL_OPCODE = *14*

EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE = *16*

EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE = *12*

EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE = *11*

EFI_BOOT_SCRIPT_PCI_CONFIG_POLL_OPCODE = *15*

class S3BootScriptSmbusOperation
Bases: object

BWBR_PROCESS_CALL = *11*

PROCESS_CALL = *10*

QUICK_READ = *0*

QUICK_WRITE = *1*

READ_BLOCK = *8*

READ_BYTE = *4*

READ_WORD = *6*

RECEIVE_BYTE = *2*

SEND_BYTE = *3*

WRITE_BLOCK = *9*

WRITE_BYTE = *5*

WRITE_WORD = *7*

class S3BootScriptWidth
Bases: object

EFI_BOOT_SCRIPT_WIDTH_UINT16 = *1*

EFI_BOOT_SCRIPT_WIDTH_UINT32 = *2*

EFI_BOOT_SCRIPT_WIDTH_UINT64 = *3*

EFI_BOOT_SCRIPT_WIDTH_UINT8 = *0*

class StatusCode
Bases: object

EFI_ABORTED = *21*

EFI_ACCESS_DENIED = *15*

EFI_ALREADY_STARTED = *20*

EFI_BAD_BUFFER_SIZE = *4*

EFI_BUFFER_TOO_SMALL = *5*

EFI_COMPROMISED_DATA = *33*

EFI_CRC_ERROR = *27*

EFI_DEVICE_ERROR = *7*

EFI_END_OF_FILE = *31*

EFI_END_OF_MEDIA = *28*

EFI_HTTP_ERROR = *35*
EFI_WARN_UNKNOWN_GLYPH = 1 EFI_WARN_DELETE_FAILURE = 2
EFI_WARN_WRITE_FAILURE = 3 EFI_WARN_BUFFER_TOO_SMALL = 4 EFI_WARN_STALE_DATA = 5 EFI_WARN_FILE_SYSTEM = 6

EFI_ICMP_ERROR = *22*

EFI_INCOMPATIBLE_VERSION = *25*

EFI_INVALID_LANGUAGE = *32*

EFI_INVALID_PARAMETER = *2*

EFI_LOAD_ERROR = *1*

EFI_MEDIA_CHANGED = *13*

EFI_NOT_FOUND = *14*

EFI_NOT_READY = *6*

EFI_NOT_STARTED = *19*

EFI_NO_MAPPING = *17*

EFI_NO_MEDIA = *12*

EFI_NO_RESPONSE = *16*

EFI_OUT_OF_RESOURCES = *9*

EFI_PROTOCOL_ERROR = *24*

EFI_SECURITY_VIOLATION = *26*

EFI_SUCCESS = *0*

EFI_TFTP_ERROR = *23*

EFI_TIMEOUT = *18*

EFI_UNSUPPORTED = *3*

EFI_VOLUME_CORRUPTED = *10*

EFI_VOLUME_FULL = *11*

EFI_WRITE_PROTECTED = *8*

align (of: int, size: int) → int

bit_set (value: int, mask: int, polarity: bool = False) → bool

get_3b_size (s_data: bytes) → int

get_nvar_name (nvram: bytes, name_offset: int, isAscii: bool)

class op_dispatch (opcode: int, size: int, entrypoint: int, context: int | None = None)
Bases: object

class op_io_pci_mem (opcode: int, size: int, width: int, address: int, unknown: int | None, count: int | None, buffer: bytes | None, value: int | None = None, mask: int | None = None)
Bases: object

class op_mem_poll (opcode: int, size: int, width: int, address: int, duration: int, looptimes: int)
Bases: object

class op_smbus_execute (opcode: int, size: int, address: int, command: int, operation: int, peccheck: int)
Bases: object

class op_stall (opcode: int, size: int, duration: int)
Bases: object

class op_terminate (opcode: int, size: int)
Bases: object

class op_unknown (opcode: int, size: int)
Bases: object

parse_auth_var (db: bytes, decode_dir: str) → List[bytes]

parse_efivar_file (fname: str, var: bytes | None = None, var_type: int = 1) → None

parse_esal_var (db: bytes, decode_dir: str) → List[bytes]

parse_external (data)

parse_pkcs7 (data)

parse_rsa2048 (data)

parse_rsa2048_sha1 (data)

parse_rsa2048_sha256 (data)

parse_sb_db (db: bytes, decode_dir: str) → List[bytes]

parse_sha1 (data)

parse_sha224 (data)

parse_sha256 (data)

parse_sha384 (data)

parse_sha512 (data)

parse_x509 (data)

parse_x509_sha256 (data)

parse_x509_sha384 (data)

parse_x509_sha512 (data)

chipsec.hal.uefi_compression module

class UEFICompression
Bases: object

compress_EFI_binary (uncompressed_data: bytes, compression_type: int) → bytes

decompress_EFI_binary (compressed_data: bytes, compression_type: int) → bytes

decompression_oder_type1: *List*[*int*] = *[1, 2]*

decompression_oder_type2: *List*[*int*] = *[1, 2, 3, 4]*

rotate_list (rot_list: List[Any], n: int) → List[Any]

unknown_decompress (compressed_data: bytes) → bytes

unknown_efi_decompress (compressed_data: bytes) → bytes

chipsec.hal.uefi_fv module

UEFI Firmware Volume Parsing/Modification Functionality

DecodeSection (SecType, SecBody, SecHeaderSize) → None

class EFI_FILE (Offset: int, Guid: UUID, Type: int, Attributes: int, State: int, Checksum: int, Size: int, Image: bytes, HeaderSize: int, UD: bool, CalcSum: int)
Bases: EFI_MODULE

class EFI_FV (Offset: int, Guid: UUID, Size: int, Attributes: int, HeaderSize: int, Checksum: int, ExtHeaderOffset: int, Image: bytes, CalcSum: int)
Bases: EFI_MODULE

class EFI_MODULE (Offset: int, Guid: UUID | None, HeaderSize: int, Attributes: int, Image: bytes)
Bases: object

calc_hashes (off: int = 0) → None

name () → str

class EFI_SECTION (Offset: int, Name: str, Type: int, Image: bytes, HeaderSize: int, Size: int)
Bases: EFI_MODULE

name () → str

FvChecksum16 (buffer: bytes) → int

FvChecksum8 (buffer: bytes) → int

FvSum16 (buffer: bytes) → int

FvSum8 (buffer: bytes) → int

GetFvHeader (buffer: bytes, off: int = 0) → Tuple[int, int, int]

NextFwFile (FvImage: bytes, FvLength: int, fof: int, polarity: bool) → EFI_FILE | None

NextFwFileSection (sections: bytes, ssize: int, sof: int, polarity: bool) → EFI_SECTION | None

NextFwVolume (buffer: bytes, off: int = 0, last_fv_size: int = 0) → EFI_FV | None

ValidateFwVolumeHeader (ZeroVector: str, FsGuid: UUID, FvLength: int, HeaderLength: int, ExtHeaderOffset: int, Reserved: int, size: int, Calcsum: int, Checksum: int) → bool

align_image (image: bytes, size: int = 8, fill: bytes = b'\x00') → bytes

assemble_uefi_file (guid: UUID, image: bytes) → bytes

assemble_uefi_raw (image: bytes) → bytes

assemble_uefi_section (image: bytes, uncomressed_size: int, compression_type: int) → bytes

get_guid_bin (guid: UUID) → bytes

chipsec.hal.uefi_platform module

Platform specific UEFI functionality (parsing platform specific EFI NVRAM, capsules, etc.)

class EFI_HDR_NVAR1 (StartId, TotalSize, Reserved1, Reserved2, Reserved3, Attributes, State)
Bases: EFI_HDR_NVAR1

class EFI_HDR_VSS (StartId, State, Reserved, Attributes, NameSize, DataSize, guid)
Bases: EFI_HDR_VSS

class EFI_HDR_VSS_APPLE (StartId, State, Reserved, Attributes, NameSize, DataSize, guid, unknown)
Bases: EFI_HDR_VSS_APPLE

class EFI_HDR_VSS_AUTH (StartId, State, Reserved, Attributes, MonotonicCount, TimeStamp1, TimeStamp2, PubKeyIndex, NameSize, DataSize, guid)
Bases: EFI_HDR_VSS_AUTH

EFIvar_EVSA (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

class FWType
Bases: object

EFI_FW_TYPE_EVSA = *'evsa'*

EFI_FW_TYPE_NVAR = *'nvar'*

EFI_FW_TYPE_UEFI = *'uefi'*

EFI_FW_TYPE_UEFI_AUTH = *'uefi_auth'*

EFI_FW_TYPE_VSS = *'vss'*

EFI_FW_TYPE_VSS2 = *'vss2'*

EFI_FW_TYPE_VSS2_AUTH = *'vss2_auth'*

EFI_FW_TYPE_VSS_APPLE = *'vss_apple'*

EFI_FW_TYPE_VSS_AUTH = *'vss_auth'*

IS_VARIABLE_STATE (_c: int, _Mask: int) → bool

class S3BootScriptType
Bases: object

EFI_BOOT_SCRIPT_TYPE_DEFAULT = *0*

EFI_BOOT_SCRIPT_TYPE_EDKCOMPAT = *170*

class UEFI_VARIABLE_HEADER (StartId, State, Reserved, Attributes, NameSize, DataSize, VendorGuid0, VendorGuid1, VendorGuid2, VendorGuid3)
Bases: UEFI_VARIABLE_HEADER

UEFI_VARIABLE_STORE_HEADER_SIZE = *28*
EFI_VARIABLE_HEADER_AUTH = “<HBBI28sIIIHH8s” EFI_VARIABLE_HEADER_AUTH_SIZE = struct.calcsize(EFI_VARIABLE_HEADER_AUTH)
EFI_VARIABLE_HEADER = “<HBBIIIIHH8s” EFI_VARIABLE_HEADER_SIZE = struct.calcsize(EFI_VARIABLE_HEADER)

class VARIABLE_STORE_HEADER_VSS (Signature, Size, Format, State, Reserved, Reserved1)
Bases: VARIABLE_STORE_HEADER_VSS

class VARIABLE_STORE_HEADER_VSS2 (Signature, Size, Format, State, Reserved, Reserved1)
Bases: VARIABLE_STORE_HEADER_VSS2

create_s3bootscript_entry_buffer (script_type: int, op, index=None) → bytes

decode_s3bs_opcode (s3bootscript_type, script_data)

decode_s3bs_opcode_def (data)

decode_s3bs_opcode_edkcompat (data: bytes)

encode_s3bootscript_entry (entry) → bytes | None

encode_s3bs_opcode (s3bootscript_type: int, op: S3BOOTSCRIPT_ENTRY) → bytes

encode_s3bs_opcode_def (op) → bytes

encode_s3bs_opcode_edkcompat (op: S3BOOTSCRIPT_ENTRY) → bytes

getEFIvariables_NVAR (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_NVAR_simple (nvram_buf: bytes) → Dict[str, Tuple[int, bytes, bytes, int, str, int]]

getEFIvariables_UEFI (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_UEFI_AUTH (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_VSS (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_VSS2 (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_VSS2_AUTH (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_VSS_APPLE (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getEFIvariables_VSS_AUTH (nvram_buf: bytes) → Dict[str, List[Tuple[int, bytes, EFI_HDR_VSS | EFI_HDR_VSS_AUTH | EFI_HDR_VSS_APPLE | None, bytes, str, int]]]

getNVstore_EFI (nvram_buf: bytes) → Tuple[int, int, None]

getNVstore_EFI_AUTH (nvram_buf: bytes) → Tuple[int, int, None]

getNVstore_EVSA (nvram_buf: bytes) → Tuple[int, int, None]

getNVstore_NVAR (nvram_buf: bytes) → Tuple[int, int, None]

getNVstore_NVAR_simple (nvram_buf: bytes) → Tuple[int | None, int, None]

getNVstore_VSS (nvram_buf: bytes)

getNVstore_VSS2 (nvram_buf: bytes)

getNVstore_VSS2_AUTH (nvram_buf: bytes)

getNVstore_VSS_APPLE (nvram_buf: bytes)

getNVstore_VSS_AUTH (nvram_buf: bytes)

id_s3bootscript_type (script: bytes, log_script: bool = False) → Tuple[int, int]

isCorrectVSStype (nvram_buf: bytes, vss_type: str)

parse_s3bootscript_entry (s3bootscript_type: int, script: bytes, off: int, log_script: bool = False)

chipsec.hal.uefi_search module

UEFI image search auxillary functionality

usage:

```
>>> chipsec.hal.uefi_search.check_match_criteria(efi_module, match_criteria, self.logger)
```

check_match_criteria (efi: EFI_SECTION, criteria: Dict[str, Dict[str, Dict[str, str]]], _log: Callable, cpuid: str | None = None) → bool

check_rules (efi: EFI_SECTION, rules: Dict[str, Any], entry_name: str, _log: Callable, bLog: bool = True, cpuid: str | None = None) → bool

chipsec.hal.virtmem module

Access to virtual memory

usage:

```
>>> read_virtual_mem( 0xf0000, 0x100 )
>>> write_virtual_mem( 0xf0000, 0x100, buffer )
>>> write_virtual_mem_dowrd( 0xf0000, 0xdeadbeef )
>>> read_virtual_mem_dowrd( 0xfed40000 )
```

class VirtMemory (cs)

Bases: HALBase

alloc_virtual_mem (length: int, max_phys_address: int = 18446744073709551615) → Tuple[int, int]

free_virtual_mem (virt_address: int) → bool

read_virtual_mem (virt_address: int, length: int) → int

read_virtual_mem_byte (virt_address: int) → int

read_virtual_mem_dword (virt_address: int) → int

read_virtual_mem_word (virt_address: int) → int

va2pa (va: int) → int

write_virtual_mem (virt_address: int, length: int, buf: bytes) → int

write_virtual_mem_byte (virt_address: int, byte_value: int) → int

write_virtual_mem_dword (virt_address: int, dword_value: int) → int

write_virtual_mem_word (virt_address: int, word_value: int) → int

chipsec.hal.vmm module

VMM specific functionality 1. Hypervisor hypercall interfaces 2. Second-level Address Translation (SLAT) 3. VirtIO devices 4. …

class VMM (cs)
Bases: object

dump_EPT_page_tables (eptp: str, pt_fname: str | None = None) → None

hypercall (rax: int, rbx: int, rcx: int, rdx: int, rdi: int, rsi: int, r8: int = 0, r9: int = 0, r10: int = 0, r11: int = 0, xmm_buffer: int = 0) → int

hypercall64_extended_fast (hypervisor_input_value: int, parameter_block: bytes) → int

hypercall64_fast (hypervisor_input_value: int, param0: int = 0, param1: int = 0) → int

hypercall64_five_args (vector: int, arg1: int = 0, arg2: int = 0, arg3: int = 0, arg4: int = 0, arg5: int = 0) → int

hypercall64_memory_based (hypervisor_input_value: int, parameters: AnyStr, size: int = 0) → int

init () → None

class VirtIO_Device (cs, b, d, f)
Bases: object

dump_device () → None

get_virtio_devices (devices: List[Tuple[int, int, int, int, int]]) → List[Tuple[int, int, int, int, int]]

Module contents

Fuzzing

chipsec.fuzzing package

Submodules

chipsec.fuzzing.primitives module

class base_primitive
Bases: object
The primitive base class implements common functionality shared across most primitives.

exhaust ()
Exhaust the possible mutations for this primitive.
@rtype: Integer @return: The number of mutations to reach exhaustion

mutate ()
Mutate the primitive by stepping through the fuzz library, return False on completion.
@rtype: Boolean @return: True on success, False otherwise.

num_mutations ()
Calculate and return the total number of mutations for this individual primitive.
@rtype: Integer @return: Number of mutated forms this primitive can take

render ()
Nothing fancy on render, simply return the value.

reset ()
Reset this primitive to the starting mutation state.

class bit_field (value, width, max_num=None, endian='<', format='binary', signed=False, full_range=False, fuzzable=True, name=None)
Bases: base_primitive

add_integer_boundaries (integer)
Add the supplied integer and border cases to the integer fuzz heuristics library.
@type integer: Int @param integer: Integer to append to fuzz heuristics

render ()
Render the primitive.

to_binary (number=None, bit_count=None)
Convert a number to a binary string.
@type number: Integer @param number: (Optional, def=self.value) Number to convert @type bit_count: Integer @param bit_count: (Optional, def=self.width) Width of bit string
@rtype: String @return: Bit string

to_decimal (binary)
Convert a binary string to a decimal number.
@type binary: String @param binary: Binary string
@rtype: Integer @return: Converted bit string

class byte (value, endian='<', format='binary', signed=False, full_range=False, fuzzable=True, name=None)
Bases: bit_field

class delim (value, fuzzable=True, name=None)
Bases: base_primitive

class dword (value, endian='<', format='binary', signed=False, full_range=False, fuzzable=True, name=None)
Bases: bit_field

class group (name, values)
Bases: base_primitive

mutate ()
Move to the next item in the values list.
@rtype: False @return: False

num_mutations ()
Number of values in this primitive.
@rtype: Integer @return: Number of values in this primitive.

class qword (value, endian='<', format='binary', signed=False, full_range=False, fuzzable=True, name=None)
Bases: bit_field

class random_data (value, min_length, max_length, max_mutations=25, fuzzable=True, step=None, name=None)
Bases: base_primitive

mutate ()
Mutate the primitive value returning False on completion.
@rtype: Boolean @return: True on success, False otherwise.

num_mutations ()
Calculate and return the total number of mutations for this individual primitive.
@rtype: Integer @return: Number of mutated forms this primitive can take

class static (value, name=None)
Bases: base_primitive

mutate ()
Do nothing.
@rtype: False @return: False

num_mutations ()
Return 0.
@rtype: 0 @return: 0

class string (value, size=-1, padding='\x00', encoding='ascii', fuzzable=True, max_len=0, name=None)
Bases: base_primitive

add_long_strings (sequence)
Given a sequence, generate a number of selectively chosen strings lengths of the given sequence and add to the string heuristic library.
@type sequence: String @param sequence: Sequence to repeat for creation of fuzz strings.

fuzz_library = *[]*

mutate ()
Mutate the primitive by stepping through the fuzz library extended with the “this” library, return False on completion.
@rtype: Boolean @return: True on success, False otherwise.

num_mutations ()
Calculate and return the total number of mutations for this individual primitive.
@rtype: Integer @return: Number of mutated forms this primitive can take

render ()
Render the primitive, encode the string according to the specified encoding.

class word (value, endian='<', format='binary', signed=False, full_range=False, fuzzable=True, name=None)
Bases: bit_field

Module contents

CHIPSEC_MAIN Program Flow

1. Select OS Helpers and Drivers
    - Load Driver (optional)
2. Detect Platform
3. Load Configuration Files
4. Load Modules
5. Run Loaded Modules
6. Report Results
7. Cleanup

CHIPSEC_UTIL Program Flow

1. Select OS Helpers and Drivers
    - Load Driver (optional)
2. Detect Platform
3. Load Configuration Files
4. Load Utility Commands
5. Run Selected Command
6. Cleanup

Auxiliary components

- `setup.py` setup script to install CHIPSEC as a package

Executable build scripts

- `<CHIPSEC_ROOT>/scripts/build_exe_*.py` make files to build Windows executables

Configuration Files

Provide a human readable abstraction for registers in the system

`chipsec/cfg/8086`	platform specific configuration xml files
`chipsec/cfg/8086/common.xml`	common configuration
`chipsec/cfg/8086/<platform>.xml`	configuration for a specific <platform>

Broken into common and platform specific configuration files.

Used to define controls, registers and bit fields.

Common files always loaded first so the platform files can override values.

Correct platform configuration files loaded based off of platform detection.

Configuration File Example

```
<mmio>
<bar name="SPIBAR" bus="0" dev="0x1F" fun="5" reg="0x10" width="4" mask="0xFFFFF000" size="0x1000" desc="SPI Controller Register Range" offset="0x0"/>
</mmio>
<registers>
<register name="BC" type="pcicfg" bus="0" dev="0x1F" fun="5" offset="0xDC" size="4" desc="BIOS Control">
<field name="BIOSWE" bit="0" size="1" desc="BIOS Write Enable" />
…
<field name="BILD" bit="7" size="1" desc="BIOS Interface Lock Down"/>
</register>
</registers>
<controls>
<control name="BiosInterfaceLockDown" register="BC" field="BILD" desc="BIOS Interface Lock-Down"/>
</controls>
```

List of Cfg components

Writing Your Own Modules

Your module class should subclass BaseModule and implement at least the methods named `is_supported` and `run`. When chipsec_main runs, it will first run `is_supported` and if that returns true, then it will call `run`.

As of CHIPSEC version 1.2.0, CHIPSEC implements an abstract name for platform *controls*. Module authors are encouraged to create controls in the XML configuration files for important platform configuration information and then use `get_control` and `set_control` within modules. This abstraction allows modules to test for the abstract control without knowing which register provides it. (This is especially important for test reuse across platform generations.)

Most modules read some platform configuration and then pass or fail based on the result. For example:

1. Define the control in the platform XML file (in `chipsec/cfg`):

```
<control name="BiosLockEnable"     register="BC"    field="BLE"    desc="BIOS Lock Enable"/>
```

2. Get the current status of the control:

```
ble = self.cs.get_control('BiosLockEnable')
```

3. React based on the status of the control:

```
if ble:
    self.logger.log_passed("BIOS Lock is set.")
else:
    self.logger.log_failed("BIOS Lock is not set.")
```

4. Set and Return `self.res`:

```
if ble:
    self.res = ModuleResult.PASSED
else:
    self.res = ModuleResult.FAILED
return self.res
```

The CHIPSEC HAL and other APIs are also available within these modules. See the next sections for details about the available functionality.

Copy your module into the `chipsec/modules/` directory structure.

- Modules specific to a certain platform should implement `is_supported` function which returns `True` for the platforms the module is applicable to.
- Modules specific to a certain platform can also be located in `chipsec/modules/<platform_code>` directory, for example `chipsec/modules/hsw`. Supported platforms and their code can be found by running `chipsec_main.py --help`.
- Modules common to all platform which CHIPSEC supports can be located in `chipsec/modules/common` directory.

If a new platform needs to be added:

- Review the platform datasheet and include appropriate information in an XML configuration file for the platform. Place this file in chipsec/cfg/8086. Registers that are correctly defined in `common.xml` will be inherited and do not need to be added. Use `common.xml` as an example. It is based on the 4th Generation Intel Core platform (Haswell).

Seealso

Creating CHIPSEC modules and commands

OS Helpers and Drivers

Provide common interfaces to interact with system drivers/commands

Mostly invoked by HAL modules

Directly invoking helpers from modules should be minimized

Helpers import from BaseHelper

Override applicable functions – default is to generate exception

I/O, PCI, MSR, UEFI Variables, etc.

Create a New Helper

Helper needs to be added into the helper folder with a folder structure like:
`chipsec/helper/<type>/<type>helper.py`

Example

The new helper should be added to `chipsec/helper/new/newhelper.py` and should import from Helper Base Class

```
from chipsec.helper.basehelper import Helper
class NewHelper(Helper):

    def __init__(self):
        super(NewHelper, self).__init__()
        self.name = "NewHelper"
```

Helper components

chipsec.helper package

Subpackages

chipsec.helper.dal package

Submodules

chipsec.helper.dal.dalhelper module

Intel DFx Abstraction Layer (DAL) helper

From the Intel(R) DFx Abstraction Layer Python* Command Line Interface User Guide

class DALHelper
Bases: Helper

EFI_supported () → bool

alloc_phys_mem (length, max_phys_address)

cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

create (start_driver: bool) → bool

```
dal_version () → str

delete () → bool

delete_EFI_variable (name, guid)

find_thread () → int

free_phys_mem (physical_address)

get_ACPI_SDT ()

get_ACPI_table (table_name)

get_EFI_variable (name, guid, attrs)

get_affinity ()

get_descriptor_table (cpu_thread_id, desc_table_code)

get_threads_count () → int

get_tool_info (tool_type: str) → Tuple[str, str]

hypercall (rcx, rdx, r8, r9, r10, r11, rax, rbx, rdi, rsi, xmm_buffer)

list_EFI_variables ()

load_ucode_update (core_id, ucode_update_buf)

map_io_space (physical_address: int, length: int, cache_type: int) → int

msgbus_send_message (mcr, mcrx, mdr)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

pci_addr (bus: int, device: int, function: int, offset: int) → int

read_cr (cpu_thread_id: int, cr_number: int) → int

read_io_port (io_port: int, size: int) → int

read_mmio_reg (phys_address: int, size: int) → int

read_msr (thread: int, msr_addr: int) → Tuple[int, int]

read_pci_reg (bus: int, device: int, function: int, address: int, size: int) → int

read_phys_mem (phys_address: int, length: int, bytewise: bool = False) → bytes

retpoline_enabled () → bool

send_sw_smi (cpu_thread_id, SMI_code_data, _rax, _rbx, _rcx, _rdx, _rsi, _rdi)
```

```
    set_EFI_variable (name, guid, buffer, buffer_size, attrs)
    set_affinity (value)
    start (start_driver: bool, driver_exhists: bool = False) → bool
    stop () → bool
    target_machine () → str
    va2pa (va)
    write_cr (cpu_thread_id: int, cr_number: int, value: int) → int
    write_io_port (io_port: int, value: int, size: int) → int
    write_mmio_reg (phys_address: int, size: int, value: int) → int
    write_msr (thread: int, msr_addr: int, eax: int, edx: int) → int
    write_pci_reg (bus: int, device: int, function: int, address: int, dword_value: int, size: int) → int
    write_phys_mem (phys_address: int, length: int, buf: bytes, bytewise: bool = False) → int
get_helper () → DALHelper
```

Module contents

chipsec.helper.efi package

Submodules

chipsec.helper.efi.efihelper module

On UEFI use the efi package functions

```
class EfiHelper
  Bases: Helper
    EFI_supported () → bool
    alloc_phys_mem (length: int, max_pa: int) → Tuple[int, int]
    cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]
    create () → bool
    delete () → bool
```

```
delete_EFI_variable (name: str, guid: str) → int

free_phys_mem (physical_address)

get_ACPI_SDT () → Tuple[None, bool]

get_ACPI_table (table_name)

get_EFI_variable (name: str, guidstr: str) → bytes | None

get_EFI_variable_full (name: str, guidstr: str) → Tuple[int, bytes | None, int]

get_affinity ()

get_descriptor_table (cpu_thread_id: int, desc_table_code: int) → None

get_threads_count () → int

get_tool_info (tool_type: str) → Tuple[str, str]

hypercall (rcx, rdx, r8, r9, r10, r11, rax, rbx, rdi, rsi, xmm_buffer)

list_EFI_variables () → Dict[str, List[EfiVariableType]] | None

load_ucode_update (cpu_thread_id: int, ucode_update_buf: int) → bool

map_io_space (physical_address: int, length: int, cache_type: int) → int

msgbus_send_message (mcr: int, mcrx: int, mdr: int | None = None) → None

msgbus_send_read_message (mcr: int, mcrx: int) → None

msgbus_send_write_message (mcr: int, mcrx: int, mdr: int) → None

pa2va (pa: int) → int

read_cr (cpu_thread_id: int, cr_number: int) → int

read_io_port (io_port: int, size: int) → int

read_mmio_reg (phys_address: int, size: int) → int

read_msr (cpu_thread_id: int, msr_addr: int) → Tuple[int, int]

read_pci_reg (bus: int, device: int, function: int, address: int, size: int) → int

read_phys_mem (phys_address: int, length: int) → bytes

retpoline_enabled () → bool

send_sw_smi (cpu_thread_id: int, SMI_code_data: int, _rax: int, _rbx: int, _rcx: int, _rdx: int, _rsi:
int, _rdi: int) → None

set_EFI_variable (name: str, guidstr: str, buffer: bytes, buffer_size: int | None = None, attrs: int
| None = 7) → int
```

set_affinity (value: int) → None

split_address (pa: int) → Tuple[int, int]

start () → bool

stop () → bool

va2pa (va: int) → Tuple[int, int]

write_cr (cpu_thread_id: int, cr_number: int, value: int) → int

write_io_port (io_port: int, value: int, size: int) → int

write_mmio_reg (phys_address: int, size: int, value: int) → int

write_msr (cpu_thread_id: int, msr_addr: int, eax: int, edx: int) → int

write_pci_reg (bus: int, device: int, function: int, address: int, value: int, size: int) → int

write_phys_mem (phys_address: int, length: int, buf: bytes) → int

get_helper () → EfiHelper

Module contents

chipsec.helper.linux package

Submodules

chipsec.helper.linux.linuxhelper module

Linux helper

class LinuxHelper
Bases: Helper

DEVICE_NAME = *'/dev/chipsec'*

DEV_MEM = *'/dev/mem'*

DEV_PORT = *'/dev/port'*

DKMS_DIR = *'/var/lib/dkms/'*

EFI_supported () → bool

MODULE_NAME = *'chipsec'*

SUPPORT_KERNEL26_GET_PAGE_IS_RAM = *False*

SUPPORT_KERNEL26_GET_PHYS_MEM_ACCESS_PROT = *False*

alloc_phys_mem (num_bytes: int, max_addr: int)

close () → None

compute_ioctlbase (itype: str = 'C') → int

cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

create ()

delete () → bool

delete_EFI_variable (name: str, guid: str) → int

free_phys_mem (physmem: int)

get_ACPI_SDT ()

get_ACPI_table (table_name)

get_EFI_variable (name: str, guid: str, attrs: int | None = None) → bytes

get_affinity () → int | None

get_descriptor_table (cpu_thread_id: int, desc_table_code: int) → Tuple[int, int, int]

get_dkms_module_location () → str

get_page_is_ram () → bytes | None

get_phys_mem_access_prot () → bytes | None

get_threads_count () → int

get_tool_info (tool_type: str) → Tuple[str | None, str]

hypercall (rcx: int, rdx: int, r8: int, r9: int, r10: int, r11: int, rax: int, rbx: int, rdi: int, rsi: int, xmm_buffer: int) → int

init () → None

ioctl (nr: int, args: Iterable, *mutate_flag: bool) → bytes

kern_get_EFI_variable (name: str, guid: str) → bytes

kern_get_EFI_variable_full (name: str, guid: str) → EfiVariableType

kern_list_EFI_variables () → Dict[str, List[EfiVariableType]] | None

kern_set_EFI_variable (name: str, guid: str, value: bytes, attr: int = 7) → int

list_EFI_variables () → Dict[str, List[EfiVariableType]] | None

load_chipsec_module ()

load_ucode_update (cpu_thread_id: int, ucode_update_buf: bytes) → bool

map_io_space (base: int, size: int, cache_type: int) → None

msgbus_send_message (mcr: int, mcrx: int, mdr: int | None = None) → int

msgbus_send_read_message (mcr: int, mcrx: int) → int | None

msgbus_send_write_message (mcr: int, mcrx: int, mdr: int) → None

read_cr (cpu_thread_id: int, cr_number: int) → int

read_io_port (io_port: int, size: int) → int

read_mmio_reg (phys_address: int, size: int) → int

read_msr (thread_id: int, msr_addr: int) → Tuple[int, int]

read_pci_reg (bus: int, device: int, function: int, offset: int, size: int = 4) → int

read_phys_mem (phys_address: int, length: int) → bytes

retpoline_enabled ()

send_sw_smi (cpu_thread_id: int, SMI_code_data: int, _rax: int, _rbx: int, _rcx: int, _rdx: int, _rsi: int, _rdi: int) → Tuple[int, int, int, int, int, int, int] | None

set_EFI_variable (name: str, guid: str, buffer: bytes, buffer_size: int, attrs: int | None = None) → int

set_affinity (thread_id: int) → int | None

start () → bool

stop () → bool

unload_chipsec_module () → None

va2pa (va: int) → Tuple[int | None, int]

write_cr (cpu_thread_id: int, cr_number: int, value: int)

write_io_port (io_port: int, value: int, size: int) → bytes

write_mmio_reg (phys_address: int, size: int, value: int)

write_msr (thread_id: int, msr_addr: int, eax: int, edx: int)

write_pci_reg (bus: int, device: int, function: int, offset: int, value: int, size: int = 4) → int

write_phys_mem (phys_address: int, length: int, newval: bytes) → int

get_helper ()

Module contents

chipsec.helper.linuxnative package

Submodules

chipsec.helper.linuxnative.cpuid module

class CPUID
Bases: object

class CPUID_struct
Bases: Structure

eax
Structure/Union member

ebx
Structure/Union member

ecx
Structure/Union member

edx
Structure/Union member

chipsec.helper.linuxnative.legacy_pci module

class LegacyPci
Bases: object

static read_pci_config (bus: int, dev: int, func: int, offset: int) → int

static write_pci_config (bus: int, dev: int, func: int, offset: int, value: int) → None

class Ports
Bases: object

classmethod get_instance () → Ports

inl (port: int) → int

instance = *None*

outl (value: int, port: int) → None

chipsec.helper.linuxnative.linuxnativehelper module

Native Linux helper

class LinuxNativeHelper
Bases: Helper

DEV_MEM = *'/dev/mem'*

DEV_PORT = *'/dev/port'*

EFI_supported ()

alloc_phys_mem (length, max_phys_address)

close ()

cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

create () → bool

delete () → bool

delete_EFI_variable (name, guid)

devmem_available () → bool
Check if /dev/mem is usable. In case the driver is not loaded, we might be able to perform the requested operation via /dev/mem. Returns True if /dev/mem is accessible.

devmsr_available () → bool
Check if /dev/cpu/CPUNUM/msr is usable. In case the driver is not loaded, we might be able to perform the requested operation via /dev/cpu/CPUNUM/msr. This requires loading the (more standard) msr driver. Returns True if /dev/cpu/CPUNUM/msr is accessible.

devport_available () → bool
Check if /dev/port is usable. In case the driver is not loaded, we might be able to perform the requested operation via /dev/port. Returns True if /dev/port is accessible.

free_phys_mem (physical_address)

get_ACPI_SDT ()

get_ACPI_table (table_name)

get_EFI_variable (name, guid)

get_affinity () → int | None

get_bios_version () → str

get_descriptor_table (cpu_thread_id, desc_table_code)

get_threads_count () → int

hypercall (rcx=0, rdx=0, r8=0, r9=0, r10=0, r11=0, rax=0, rbx=0, rdi=0, rsi=0, xmm_buffer=0)

init ()

list_EFI_variables ()

load_ucode_update (cpu_thread_id, ucode_update_buf)

map_io_space (base: int, size: int, cache_type: int) → None
Map to memory a specific region.

memory_mapping (base: int, size: int) → MemoryMapping | None
Returns the mmap region that fully encompasses this area. Returns None if no region matches.

msgbus_send_message (mcr, mcrx, mdr)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

read_cr (cpu_thread_id, cr_number)

read_io_port (io_port: int, size: int) → int

read_mmio_reg (phys_address: int, size: int) → int

read_msr (thread_id: int, msr_addr: int) → Tuple[int, int]

read_pci_reg (bus: int, device: int, function: int, offset: int, size: int, domain: int = 0) → int

read_phys_mem (phys_address, length: int) → bytes

retpoline_enabled ()

send_sw_smi (cpu_thread_id, SMI_code_data, _rax, _rbx, _rcx, _rdx, _rsi, _rdi)

set_EFI_variable (name, guid, buffer, buffer_size=None, attrs=None)

set_affinity (thread_id: int) → int | None

start () → bool

stop () → bool

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port: int, value: int, size: int) → bool

write_mmio_reg (phys_address: int, size: int, value: int) → None

write_msr (thread_id: int, msr_addr: int, eax: int, edx: int) → int

write_pci_reg (bus: int, device: int, function: int, offset: int, value: int, size: int = 4, domain: int = 0) → int

write_phys_mem (phys_address, length: int, newval: bytes) → int

class MemoryMapping (fileno, length, flags, prot, offset)
Bases: mmap
Memory mapping based on Python's mmap. This subclass keeps tracks of the start and end of the mapping.

get_helper ()

Module contents

chipsec.helper.windows package

Submodules

chipsec.helper.windows.windowshelper module

Module contents

Submodules

chipsec.helper.basehelper module

class Helper
Bases: ABC

abstract EFI_supported () → bool

abstract alloc_phys_mem (size: int, max_phys_address: int) → Tuple[int, int]

abstract cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

abstract create () → bool

abstract delete () → bool

abstract delete_EFI_variable (name: str, guid: str) → int | None

abstract free_phys_mem (phys_address: int)

abstract get_ACPI_SDT () → Tuple[Array | None, bool]

abstract get_ACPI_table (table_name: str) → Array | None

abstract get_EFI_variable (name: str, guid: str) → bytes | None

abstract get_affinity () → int | None

abstract get_descriptor_table (cpu_thread_id: int, desc_table_code: int) → Tuple[int, int, int] | None

get_info () → Tuple[str, str]

abstract get_threads_count () → int

abstract hypercall (rcx: int, rdx: int, r8: int, r9: int, r10: int, r11: int, rax: int, rbx: int, rdi: int, rsi: int, xmm_buffer: int) → int

abstract list_EFI_variables () → Dict[str, List[EfiVariableType]] | None

abstract load_ucode_update (cpu_thread_id: int, ucode_update_buffer: bytes) → bool

abstract map_io_space (phys_address: int, size: int, cache_type: int) → int

abstract msgbus_send_message (mcr: int, mcrx: int, mdr: int | None) → int | None

abstract msgbus_send_read_message (mcr: int, mcrx: int) → int | None

abstract msgbus_send_write_message (mcr: int, mcrx: int, mdr: int) → None

abstract read_cr (cpu_thread_id: int, cr_number: int) → int

abstract read_io_port (io_port: int, size: int) → int

abstract read_mmio_reg (phys_address: int, size: int) → int

abstract read_msr (cpu_thread_id: int, msr_addr: int) → Tuple[int, int]

abstract read_pci_reg (bus: int, device: int, function: int, address: int, size: int) → int

abstract read_phys_mem (phys_address: int, size: int) → bytes

abstract retpoline_enabled () → bool

abstract send_sw_smi (cpu_thread_id: int, SMI_code_data: int, _rax: int, _rbx: int, _rcx: int, _rdx: int, _rsi: int, _rdi: int) → int | None

abstract set_EFI_variable (name: str, guid: str, buffer: bytes, buffer_size: int | None, attrs: int | None) → int | None

abstract set_affinity (value: int) → int | None

abstract start () → bool

abstract stop () → bool

abstract va2pa (virtual_address: int) → Tuple[int, int]

abstract write_cr (cpu_thread_id: int, cr_number: int, value: int) → int

abstract write_io_port (io_port: int, value: int, size: int) → int

abstract write_mmio_reg (phys_address: int, size: int, value: int) → int

abstract write_msr (cpu_thread_id: int, msr_addr: int, eax: int, edx: int) → int

abstract write_pci_reg (bus: int, device: int, function: int, address: int, value: int, size: int) → int

abstract write_phys_mem (phys_address: int, size: int, buffer: bytes) → int

chipsec.helper.nonehelper module

class NoneHelper
Bases: Helper

EFI_supported () → bool

alloc_phys_mem (length: int, max_phys_address: int) → Tuple[int, int]

cpuid (eax: int, ecx: int) → Tuple[int, int, int, int]

create () → bool

delete () → bool

delete_EFI_variable (name: str, guid: str) → int | None

free_phys_mem (physical_address: int)

get_ACPI_SDT () → Tuple[Array | None, bool]

get_ACPI_table (table_name: str) → Array | None

get_EFI_variable (name: str, guid: str) → bytes | None

get_affinity () → int | None

get_descriptor_table (cpu_thread_id: int, desc_table_code: int) → Tuple[int, int, int] | None

get_info () → Tuple[str, str]

get_threads_count () → int

hypercall (rcx: int, rdx: int, r8: int, r9: int, r10: int, r11: int, rax: int, rbx: int, rdi: int, rsi: int, xmm_buffer: int) → int

list_EFI_variables () → Dict[str, List[EfiVariableType]] | None

load_ucode_update (cpu_thread_id: int, ucode_update_buf: bytes) → bool

map_io_space (physical_address: int, length: int, cache_type: int) → int

msgbus_send_message (mcr: int, mcrx: int, mdr: int | None) → int | None

msgbus_send_read_message (mcr: int, mcrx: int) → int | None

msgbus_send_write_message (mcr: int, mcrx: int, mdr: int) → None

read_cr (cpu_thread_id: int, cr_number: int) → int

read_io_port (io_port: int, size: int) → int

read_mmio_reg (phys_address: int, size: int) → int

read_msr (cpu_thread_id: int, msr_addr: int) → Tuple[int, int]

read_pci_reg (bus: int, device: int, function: int, address: int, size: int) → int

read_phys_mem (phys_address: int, length: int) → bytes

retpoline_enabled () → bool

send_sw_smi (cpu_thread_id: int, SMI_code_data: int, _rax: int, _rbx: int, _rcx: int, _rdx: int, _rsi: int, _rdi: int) → int | None

set_EFI_variable (name: str, guid: str, data: bytes, datasize: int | None, attrs: int | None) → int | None

set_affinity (value: int) → int | None

start () → bool

stop () → bool

va2pa (va: int) → Tuple[int, int]

write_cr (cpu_thread_id: int, cr_number: int, value: int) → int

write_io_port (io_port: int, value: int, size: int) → int

write_mmio_reg (phys_address: int, size: int, value: int) → int

write_msr (cpu_thread_id: int, msr_addr: int, eax: int, edx: int) → int

write_pci_reg (bus: int, device: int, function: int, address: int, value: int, size: int) → int

write_phys_mem (phys_address: int, length: int, buf: bytes) → int

chipsec.helper.oshelper module

Abstracts support for various OS/environments, wrapper around platform specific code that invokes kernel driver

class OsHelper
Bases: object

get_available_helpers () → List[str]

get_base_helper ()

get_default_helper ()

get_helper (name: str) → Any

getcwd () → str

is_dal () → bool

is_efi () → bool

is_linux () → bool

is_macos () → bool

is_win8_or_greater () → bool

is_windows () → bool

load_helpers () → None

get_tools_path () → str

helper ()

Module contents

Methods for Platform Detection

Uses PCI VID and DID to detect processor and PCH

Processor: 0:0.0

PCH: Scans enumerated PCI Devices for corresponding VID/DID per configurations.

Chip information located in `chipsec/chipset.py`.

Currently requires VID of 0x8086 (Intel) or 0x1022 (AMD).

DID is used as the lookup key.

If there are matching DID, will fall back to cpuid check for CPU.

Platform Configuration Options

Select a specific platform using the `-p` flag.

Specify PCH using the `--pch` flag.

~Ignore the platform specific registers using the `-i` flag.~ The `-i` flag has been depricated and should not be used.

Sample module code template

```
from chipsec.module_common import BaseModule, ModuleResult

class ModuleClass(BaseModule):
    """Class name aligns with file name, eg ModuleClass.py"""
    def __init__(self):
        BaseModule.__init__(self)

    def is_supported(self):
        """Module prerequisite checks"""
        if some_module_requirement():
            return True  # Module is applicable
        self.res = ModuleResult.NOTAPPLICABLE
        return False  # Module is not applicable

    def action(self):
        """Module test logic and methods as needed"""
        self.logger.log_passed('Module was successful!')
        return ModuleResult.PASSED

    def run(self, module_argv):
        """Primary module execution and result handling"""
        self.logger.start_test('Module Description')
        self.res = self.action()
        return self.res
```

Util Command

```
# chipsec_util.py commands live in chipsec/utilcmd/
# Example file name: <command_display_name>_cmd.py

from argparse import ArgumentParser

from chipsec.command import BaseCommand, toLoad

class CommandClass(BaseCommand):
    """
        >>> chipsec_util command_display_name action
    """
    def requirements(self) -> toLoad:
        return toLoad.All

    def parse_arguments(self):
        parser = ArgumentParser(prog='chipsec_util command_display_name', usage=CommandClass.__doc__)
        subparsers = parser.add_subparsers()
        parser_entrypoint = subparsers.add_parser('action')
        parser_entrypoint.set_defaults(func=self.action)
        parser.parse_args(self.argv, namespace=self)

    def action(self):
        return
```

```
    def run(self):
        self.func()

commands = {'command_display_name': CommandClass}
```

CHIPSEC Modules

A CHIPSEC module is just a python class that inherits from BaseModule and implements `is_supported` and `run`. Modules are stored under the chipsec installation directory in a subdirectory “modules”. The “modules” directory contains one subdirectory for each chipset that chipsec supports. There is also a directory for common modules that should apply to every platform.

`chipsec/modules/`	modules including tests or tools (that’s where most of the chipsec functionality is)
`chipsec/modules/common/`	modules common to all platforms
`chipsec/modules/<platform>/`	modules specific to <platform>
`chipsec/modules/tools/`	security tools based on CHIPSEC framework (fuzzers, etc.)

Internally the chipsec application uses the concept of a module name, which is a string of the form: `common.bios_wp`. This means module `common.bios_wp` is a python script called `bios_wp.py` that is stored at `<ROOT_DIR>\chipsec\modules\common\`.

Modules can be mapped to one or more security vulnerabilities being checked. More information also found in the documentation for any individual module.

Known vulnerabilities can be mapped to CHIPSEC modules as follows:

Attack Surface/Vector: Firmware protections in ROM

Vulnerability Description	CHIPSEC Module	Example
SMI event configuration is not locked	common.bios_smi	
SPI flash descriptor is not protected	common.spi_desc	
SPI controller security override is enabled	common.spi_fdopss	
SPI flash controller is not locked	common.spi_lock	
Device-specific SPI flash protection is not used	chipsec_util spi write (manual analysis)	
SMM BIOS write protection is not correctly used	common.bios_wp	
Flash protected ranges do not protect bios region	common.bios_wp	
BIOS interface is not locked	common.bios_ts	

Attack Surface/Vector: Runtime protection of SMRAM

Vulnerability Description	CHIPSEC Module	Example
Compatibility SMRAM is not locked	common.smm	
SMM cache attack	common.smrr	
Memory remapping vulnerability in SMM protection	remap	
DMA protections of SMRAM are not in use	smm_dma	
Graphics aperture redirection of SMRAM	chipsec_util memconfig remap	
Memory sinkhole vulnerability	tools.cpu.sinkhole	

Attack Surface/Vector: Secure boot - Incorrect protection of secure boot configuration

Vulnerability Description	CHIPSEC Module	Example
Root certificate	common.bios_wp, common.secureboot.variables	
Key exchange keys	common.secureboot.variables	
Controls in setup variable (CSM enable/disable, image verification policies, secure boot enable/disable, clear/restore keys)	chipsec_util uefi var-find Setup	
TE header confusion	tools.secureboot.te	
UEFI NVRAM is not write protected	common.bios_wp	
Insecure handling of secure boot disable	chipsec_util uefi var-list	

Attack Surface/Vector: Persistent firmware configuration

Vulnerability Description	CHIPSEC Module	Example
Secure boot configuration is stored in unprotected variable	common.secureboot.variables, chipsec_util uefi var-list	
Variable permissions are not set according to specification	common.uefi.access_uefispec	
Sensitive data (like passwords) are stored in uefi variables	chipsec_util uefi var-list (manual analysis)	

Firmware doesn't sanitize pointers/addresses stored in variables	chipsec_util uefi var-list (manual analysis)	
Firmware hangs on invalid variable content	chipsec_util uefi var-write, chipsec_util uefi var-delete (manual analysis)	
Hardware configuration stored in unprotected variables	chipsec_util uefi var-list (manual analysis)	
Re-creating variables with less restrictive permissions	chipsec_util uefi var-write (manual analysis)	
Variable NVRAM overflow	chipsec_util uefi var-write (manual analysis)	
Critical configuration is stored in unprotected CMOS	chipsec_util cmos, common.rtclock	

Attack Surface/Vector: Platform hardware configuration

Vulnerability Description	CHIPSEC Module	Example
Boot block top-swap mode is not locked	common.bios_ts	
Architectural features not locked	common.ia32cfg	
Memory map is not locked	memconfig	
IOMMU usage	chipsec_util iommu	
Memory remapping is not locked	remap	

Attack Surface/Vector: Runtime firmware (eg. SMI handlers)

Vulnerability Description	CHIPSEC Module	Example
SMI handlers use pointers/addresses from OS without validation	tools.smm.smm_ptr	
Legacy SMI handlers call legacy BIOS outside SMRAM		
INT15 in legacy SMI handlers		
UEFI SMI handlers call UEFI services outside SMRAM		
Malicious CommBuffer pointer and contents		
Race condition during SMI handler		

Authenticated variables SMI handler is not implemented	chipsec_util uefi var-write	
SmmRuntime vulnerability	tools.uefi.scan_blocked	

Attack Surface/Vector: Boot time firmware

Vulnerability Description	CHIPSEC Module	Example
Software vulnerabilities when parsing, decompressing, and loading data from ROM		
Software vulnerabilities in implementation of digital signature verification		
Pointers stored in UEFI variables and used during boot	chipsec_util uefi var-write	
Loading unsigned PCI option ROMs	chipsec_util pci xrom	
Boot hangs due to error condition (eg. ASSERT)		

Attack Surface/Vector: Power state transitions (eg. resume from sleep)

Vulnerability Description	CHIPSEC Module	Example
Insufficient protection of S3 boot script table	common.uefi.s3bootscript, tools.uefi.s3script_modify	
Dispatch opcodes in S3 boot script call functions in unprotected memory	common.uefi.s3bootscript, tools.uefi.s3script_modify	
S3 boot script interpreter stored in unprotected memory		
Pointer to S3 boot script table in unprotected UEFI variable	common.uefi.s3bootscript, tools.uefi.s3script_modify	
Critical setting not recorded in S3 boot script table	chipsec_util uefi s3bootscript (manual analysis)	
OS waking vector in ACPI tables can be modified	chipsec_util acpi dump (manual analysis)	
Using pointers on S3 resume stored in unprotected UEFI variables	chipsec_util uefi var-write	

Attack Surface/Vector: Firmware update

Vulnerability Description	CHIPSEC Module	Example
Software vulnerabilities when parsing firmware updates		
Unauthenticated firmware updates		
Runtime firmware update that can be interrupted		
Signature not checked on capsule update executable		

Attack Surface/Vector: Network interfaces

Vulnerability Description	CHIPSEC Module	Example
Software vulnerabilities when handling messages over network interfaces		
Booting unauthenticated firmware over unprotected network interfaces		

Attack Surface/Vector: Misc

Vulnerability Description	CHIPSEC Module	Example
BIOS keyboard buffer is not cleared during boot	common.bios_kbrd_buffer	
DMA attack from devices during firmware execution		

Modules

chipsec.modules package

Subpackages

chipsec.modules.bdw package

Module contents

chipsec.modules.byt package

Module contents

chipsec.modules.common package

Subpackages

chipsec.modules.common.cpu package

Submodules

chipsec.modules.common.cpu.cpu_info module

Displays CPU information

Reference:

- **Intel 64 and IA-32 Architectures Software Developer Manual (SDM)**
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Usage:

```
chipsec_main -m common.cpu.cpu_info
```

Examples:

```
>>> chipsec_main.py -m common.cpu.cpu_info
```

Registers used:

- IA32_BIOS_SIGN_ID.Microcode

class cpu_info
Bases: BaseModule

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.cpu.ia_untrusted module

IA Untrusted checks

Usage:

```
chipsec_main -m common.cpu.ia_untrusted
```

Examples:

```
>>> chipsec_main.py -m common.cpu.ia_untrusted
```

Registers used:

- MSR_BIOS_DONE.IA_UNTRUSTED
- MSR_BIOS_DONE.SoC_BIOS_DONE

class ia_untrusted
Bases: BaseModule

check_untrusted () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.cpu.spectre_v2 module

The module checks if system includes hardware mitigations for Speculative Execution Side Channel. Specifically, it verifies that the system supports CPU mitigations for Branch Target Injection vulnerability a.k.a. Spectre Variant 2 (CVE-2017-5715)

The module checks if the following hardware mitigations are supported by the CPU and enabled by the OS/software:

1. Indirect Branch Restricted Speculation (IBRS) and Indirect Branch Predictor Barrier (IBPB): CPUID.(EAX=7H,ECX=0):EDX[26] == 1

2. Single Thread Indirect Branch Predictors (STIBP): CPUID.(EAX=7H,ECX=0):EDX[27] == 1 IA32_SPEC_CTRL[STIBP] == 1
3. Enhanced IBRS: CPUID.(EAX=7H,ECX=0):EDX[29] == 1 IA32_ARCH_CAPABILITIES[IBRS_ALL] == 1 IA32_SPEC_CTRL[IBRS] == 1
4. @TODO: Mitigation for Rogue Data Cache Load (RDCL): CPUID.(EAX=7H,ECX=0):EDX[29] == 1 IA32_ARCH_CAPABILITIES[RDCL_NO] == 1

In addition to checking if CPU supports and OS enables all mitigations, we need to check that relevant MSR bits are set consistently on all logical processors (CPU threads).

The module returns the following results:

FAILED:

IBRS/IBPB is not supported

WARNING:

IBRS/IBPB is supported

Enhanced IBRS is not supported

WARNING:

IBRS/IBPB is supported

Enhanced IBRS is supported

Enhanced IBRS is not enabled by the OS

WARNING:

IBRS/IBPB is supported

STIBP is not supported or not enabled by the OS

PASSED:

IBRS/IBPB is supported

Enhanced IBRS is supported

Enhanced IBRS is enabled by the OS

STIBP is supported

Notes:

- The module returns WARNING when CPU doesn't support enhanced IBRS Even though OS/software may use basic IBRS by setting IA32_SPEC_CTRL[IBRS] when necessary, we have no way to verify this
- The module returns WARNING when CPU supports enhanced IBRS but OS doesn't set IA32_SPEC_CTRL[IBRS] Under enhanced IBRS, OS can set IA32_SPEC_CTRL[IBRS] once to take advantage of IBRS protection
- The module returns WARNING when CPU doesn't support STIBP or OS doesn't enable it Per Speculative Execution Side Channel Mitigations: "enabling IBRS prevents software operating on one logical processor from controlling the predicted targets of indirect branches executed on another logical processor. For that reason, it is not necessary to enable STIBP when IBRS is enabled"
- OS/software may implement "retpoline" mitigation for Spectre variant 2 instead of using CPU hardware IBRS/IBPB

@TODO: we should verify CPUID.07H:EDX on all logical CPUs as well because it may differ if ucode update wasn't loaded on all CPU cores

Hardware registers used:

- CPUID.(EAX=7H,ECX=0):EDX[26] - enumerates support for IBRS and IBPB

- CPUID.(EAX=7H,ECX=0):EDX[27] - enumerates support for STIBP
- CPUID.(EAX=7H,ECX=0):EDX[29] - enumerates support for the IA32_ARCH_CAPABILITIES MSR
- IA32_ARCH_CAPABILITIES[IBRS_ALL] - enumerates support for enhanced IBRS
- IA32_ARCH_CAPABILITIES[RCDL_NO] - enumerates support RCDL mitigation
- IA32_SPEC_CTRL[IBRS] - enable control for enhanced IBRS by the software/OS
- IA32_SPEC_CTRL[STIBP] - enable control for STIBP by the software/OS

References:

- Reading privileged memory with a side-channel by Jann Horn, Google Project Zero: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- Spectre: https://spectreattack.com/spectre.pdf
- Meltdown: https://meltdownattack.com/meltdown.pdf

- Speculative Execution Side Channel Mitigations: https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf
  - Retpoline: a software construct for preventing branch-target-injection: https://support.google.com/faqs/answer/7625886

class spectre_v2
Bases: BaseModule

check_spectre_mitigations () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

Module contents

chipsec.modules.common.secureboot package

Submodules

chipsec.modules.common.secureboot.variables module

Verify that all Secure Boot key UEFI variables are authenticated (BS+RT+AT) and protected from unauthorized modification.

Reference:

- UEFI 2.4 spec Section 28

Usage:

`chipsec_main -m common.secureboot.variables [-a modify]` - `-a` : modify = will try to write/corrupt the variables

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -m common.secureboot.variables
>>> chipsec_main.py -m common.secureboot.variables -a modify
```

Note

- Module is not supported in all environments.

class variables
Bases: BaseModule

can_modify (name: str, guid: AnyStr | None, data: bytes | None) → bool

check_secureboot_variable_attributes (do_modify: bool) → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

Module contents

chipsec.modules.common.uefi package

Submodules

chipsec.modules.common.uefi.access_uefispec module

Checks protection of UEFI variables defined in the UEFI spec to have certain permissions.

Returns failure if variable attributes are not as defined in table 11 “Global Variables” of the UEFI spec.

usage:

`chipsec_main -m common.uefi.access_uefispec [-a modify]`

- `-a modify`: Attempt to modify each variable in addition to checking attributes

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -m common.uefi.access_uefispec
>>> chipsec_main.py -m common.uefi.access_uefispec -a modify
```

NOTE: Requires an OS with UEFI Runtime API support.

class access_uefispec
Bases: BaseModule

can_modify (name: str, guid: str, data: bytes) → bool

check_vars (do_modify: bool) → int

diff_var (data1: int, data2: int) → bool

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.uefi.s3bootscript module

Checks protections of the S3 resume boot-script implemented by the UEFI based firmware

References:

VU#976132 UEFI implementations do not properly secure the EFI S3 Resume Boot Path boot script

Technical Details of the S3 Resume Boot Script Vulnerability by Intel Security's Advanced Threat Research team.

Attacks on UEFI Security by Rafal Wojtczuk and Corey Kallenberg.

Attacking UEFI Boot Script by Rafal Wojtczuk and Corey Kallenberg.

Exploiting UEFI boot script table vulnerability by Dmytro Oleksiuk.

Usage:

```
chipsec_main.py -m common.uefi.s3bootscript [-a <script_address>]
```

- `-a <script_address>`: Specify the bootscript address

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -m common.uefi.s3bootscript
>>> chipsec_main.py -m common.uefi.s3bootscript -a 0x00000000BDE10000
```

Note

Requires an OS with UEFI Runtime API support.

class s3bootscript
Bases: BaseModule

check_dispatch_opcodes (bootscript_entries: List[S3BOOTSCRIPT_ENTRY]) → bool

check_s3_bootscript (bootscript_pa: int) → int

check_s3_bootscripts (bsaddress=None) → int

is_inside_SMRAM (pa: int) → bool

is_inside_SPI (pa: int) → bool

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

Module contents

Submodules

chipsec.modules.common.bios_kbrd_buffer module

Checks for exposure of pre-boot passwords (BIOS/HDD/pre-bot authentication SW) in the BIOS keyboard buffer.

Reference:

- DEFCON 16: Bypassing Pre-boot Authentication Passwords by Instrumenting the BIOS Keyboard Buffer by Jonathan Brossard

Usage:

```
chipsec_main -m common.bios_kbrd_buffer
```

Examples:

```
>>> chipsec_main.py -m common.bios_kbrd_buffer
```

class bios_kbrd_buffer
Bases: BaseModule

check_BIOS_keyboard_buffer () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.bios_smi module

The module checks that SMI events configuration is locked down - Global SMI Enable/SMI Lock - TCO SMI Enable/TCO Lock

References:

- Setup for Failure: Defeating SecureBoot by Corey Kallenberg, Xeno Kovah, John Butterworth, Sam Cornwell
- Summary of Attacks Against BIOS and Secure Boot

Usage:

```
chipsec_main -m common.bios_smi
```

Examples:

```
>>> chipsec_main.py -m common.bios_smi
```

Registers used:

- SmmBiosWriteProtection (Control)
- TCOSMILock (Control)
- SMILock (Control)
- BiosWriteEnable (Control)

class bios_smi
Bases: BaseModule

check_SMI_locks () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.bios_ts module

Checks for BIOS Interface Lock including Top Swap Mode

References:

- BIOS Boot Hijacking and VMware Vulnerabilities Digging by Bing Sun

Usage:

```
chipsec_main -m common.bios_ts
```

Examples:

```
>>> chipsec_main.py -m common.bios_ts
```

Registers used:

- BiosInterfaceLockDown (control)
- TopSwapStatus (control)
- TopSwap (control)

class bios_ts

Bases: BaseModule

check_bios_iface_lock () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.bios_wp module

The BIOS region in flash can be protected either using SMM-based protection or using configuration in the SPI controller. However, the SPI controller configuration is set once and locked, which would prevent writes later.

This module checks both mechanisms. In order to pass this test using SPI controller configuration, the SPI Protected Range registers (PR0-4) will need to cover the entire BIOS region. Often, if this configuration is used at all, it is used only to protect part of the BIOS region (usually the boot block). If other important data (eg. NVRAM) is not protected, however, some vulnerabilities may be possible.

A Tale of One Software Bypass of Windows 8 Secure Boot In a system where certain BIOS data was not protected, malware may be able to write to the Platform Key stored on the flash, thereby disabling secure boot.

SMM based write protection is controlled from the BIOS Control Register. When the BIOS Write Protect Disable bit is set (sometimes called BIOSWE or BIOS Write Enable), then writes are allowed. When cleared, it can also be locked with the BIOS Lock Enable (BLE) bit. When locked, attempts to change the WPD bit will result in generation of an SMI. This way, the SMI handler can decide whether to perform the write.

As demonstrated in the Speed Racer issue, a race condition may exist between the outstanding write and processing of the SMI that is generated. For this reason, the EISS bit (sometimes called SMM_BWP or SMM BIOS Write Protection) must be set to ensure that only SMM can write to the SPI flash.

References:

- A Tale of One Software Bypass of Windows 8 Secure Boot
- Speed Racer

Usage:

```
chipsec_main -m common.bios_wp
```

Examples:

```
>>> chipsec_main.py -m common.bios_wp
```

Registers used: (n = 0,1,2,3,4)

- BiosLockEnable (Control)
- BiosWriteEnable (Control)
- SmmBiosWriteProtection (Control)
- PRn.PRB
- PRn.RPE
- PRn.PRL
- PRn.WPE

Note

- Module will fail if SMM-based protection is not correctly configured and SPI protected ranges (PR registers) do not protect the entire BIOS region.

class bios_wp
Bases: BaseModule

check_BIOS_write_protection () → int

check_SPI_protected_ranges () → bool

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.debugenabled module

This module checks if the system has debug features turned on, specifically the Direct Connect Interface (DCI).

This module checks the following bits: 1. HDCIEN bit in the DCI Control Register 2. Debug enable bit in the IA32_DEBUG_INTERFACE MSR 3. Debug lock bit in the IA32_DEBUG_INTERFACE MSR 4. Debug occurred bit in the IA32_DEBUG_INTERFACE MSR

Usage:

```
chipsec_main -m common.debugenabled
```

Examples:

```
>>> chipsec_main.py -m common.debugenabled
```

The module returns the following results:

- **FAILED** : Any one of the debug features is enabled or unlocked.
- **PASSED** : All debug feature are disabled and locked.

Registers used:

- IA32_DEBUG_INTERFACE[DEBUGENABLE]
- IA32_DEBUG_INTERFACE[DEBUGELOCK]
- IA32_DEBUG_INTERFACE[DEBUGEOCCURED]
- P2SB_DCI.DCI_CONTROL_REG[HDCIEN]

class debugenabled
Bases: BaseModule

check_cpu_debug_enable () → int

check_dci () → int

is_supported () → bool

This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.ia32cfg module

Tests that IA-32/IA-64 architectural features are configured and locked, including IA32 Model Specific Registers (MSRs)

Reference:

- **Intel 64 and IA-32 Architectures Software Developer Manual (SDM)**
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Usage:

```
chipsec_main -m common.ia32cfg
```

Examples:

```
>>> chipsec_main.py -m common.ia32cfg
```

Registers used:

- IA32_FEATURE_CONTROL
- Ia32FeatureControlLock (control)

class ia32cfg
Bases: BaseModule

check_ia32feature_control () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.me_mfg_mode module

This module checks that ME Manufacturing mode is not enabled.

References:

https://blog.ptsecurity.com/2018/10/intel-me-manufacturing-mode-macbook.html

PCI_DEVS.H

```
#define PCH_DEV_SLOT_CSE        0x16
#define  PCH_DEVFN_CSE          _PCH_DEVFN(CSE, 0)
#define  PCH_DEV_CSE            _PCH_DEV(CSE, 0)
```

https://github.com/coreboot/coreboot/blob/master/src/soc/intel/apollolake/cse.c

```
fwsts1 = dump_status(1, PCI_ME_HFSTS1);
# Minimal decoding is done here in order to call out most important
# pieces. Manufacturing mode needs to be locked down prior to shipping
# the product so it's called out explicitly.
printk(BIOS_DEBUG, "ME: Manufacturing Mode      : %s", (fwsts1 & (1 << 0x4)) ? "YES" : "NO");
```

PCH.H

```
#define PCH_ME_DEV                PCI_DEV(0, 0x16, 0)
```

ME.H

```
struct me_hfs {
        u32 working_state: 4;
        u32 mfg_mode: 1;
        u32 fpt_bad: 1;
        u32 operation_state: 3;
        u32 fw_init_complete: 1;
        u32 ft_bup_ld_flr: 1;
        u32 update_in_progress: 1;
        u32 error_code: 4;
        u32 operation_mode: 4;
        u32 reserved: 4;
        u32 boot_options_present: 1;
        u32 ack_data: 3;
        u32 bios_msg_ack: 4;
} __packed;
```

ME_STATUS.C

```
printk(BIOS_DEBUG, "ME: Manufacturing Mode      : %s", hfs->mfg_mode ? "YES" : "NO");
```

This module checks the following:

`HFS.MFG_MODE BDF: 0:22:0 offset 0x40 - Bit [4]`

Usage:

`chipsec_main -m common.me_mfg_mode`

Examples:

```
>>> chipsec_main.py -m common.me_mfg_mode
```

The module returns the following results:

FAILED : HFS.MFG_MODE is set

PASSED : HFS.MFG_MODE is not set.

Hardware registers used:

- HFS.MFG_MODE

class me_mfg_mode
Bases: BaseModule

check_me_mfg_mode () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.memconfig module

This module verifies memory map secure configuration, that memory map registers are correctly configured and locked down.

Usage:

`chipsec_main -m common.memconfig`

Example:

```
>>> chipsec_main.py -m common.memconfig
```

Note

- This module will only run on Core (client) platforms.

class memconfig
Bases: BaseModule

check_memmap_locks () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.memlock module

This module checks if memory configuration is locked to protect SMM

Reference:

- https://github.com/coreboot/coreboot/blob/master/src/cpu/intel/model_206ax/finalize.c
- https://github.com/coreboot/coreboot/blob/master/src/soc/intel/broadwell/include/soc/msr.h

This module checks the following: - MSR_LT_LOCK_MEMORY MSR (0x2E7) - Bit [0]

The module returns the following results:

- **FAILED** : MSR_LT_LOCK_MEMORY[0] is not set
- **PASSED** : MSR_LT_LOCK_MEMORY[0] is set
- **ERROR** : Problem reading MSR_LT_LOCK_MEMORY values

Usage:

```
chipsec_main -m common.memlock
```

Example:

```
>>> chipsec_main.py -m common.memlock
```

Registers used:

- MSR_LT_LOCK_MEMORY

Note

- This module will not run on Atom based platforms.

class memlock
Bases: BaseModule

check_MSR_LT_LOCK_MEMORY () → bool

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.remap module

Check Memory Remapping Configuration

Reference:

- Preventing & Detecting Xen Hypervisor Subversions by Joanna Rutkowska & Rafal Wojtczuk

Usage:

```
chipsec_main -m common.remap
```

Example:

```
>>> chipsec_main.py -m common.remap
```

Registers used:

- PCI0.0.0_REMAPBASE
- PCI0.0.0_REMAPLIMIT
- PCI0.0.0_TOUUD
- PCI0.0.0_TOLUD
- PCI0.0.0_TSEGMB

Note

- This module will only run on Core platforms.

class remap
Bases: BaseModule

check_remap_config () → int

is_ibecc_enabled () → bool

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (_) → int

chipsec.modules.common.rtclock module

chipsec.modules.common.sgx_check module

Check SGX related configuration

Reference:

- SGX BWG, CDI/IBP#: 565432

Usage:

```
chipsec_main -m common.sgx_check
```

Examples:

```
>>> chipsec_main.py -m common.sgx_check
```

Registers used:

- IA32_FEATURE_CONTROL.SGX_GLOBAL_EN
- IA32_FEATURE_CONTROL.LOCK
- IA32_DEBUG_INTERFACE.ENABLE
- IA32_DEBUG_INTERFACE.LOCK
- MTRRCAP.PRMRR
- PRMRR_VALID_CONFIG
- PRMRR_PHYBASE.PRMRR_base_address_fields
- PRMRR_PHYBASE.PRMRR_MEMTYPE
- PRMRR_MASK.PRMRR_mask_bits
- PRMRR_MASK.PRMRR_VLD
- PRMRR_MASK.PRMRR_LOCK
- PRMRR_UNCORE_PHYBASE.PRMRR_base_address_fields
- PRMRR_UNCORE_MASK.PRMRR_mask_bits
- PRMRR_UNCORE_MASK.PRMRR_VLD
- PRMRR_UNCORE_MASK.PRMRR_LOCK
- BIOS_SE_SVN.PFAT_SE_SVN
- BIOS_SE_SVN.ANC_SE_SVN
- BIOS_SE_SVN.SCLEAN_SE_SVN
- BIOS_SE_SVN.SINIT_SE_SVN
- BIOS_SE_SVN_STATUS.LOCK
- SGX_DEBUG_MODE.SGX_DEBUG_MODE_STATUS_BIT

Note

- Will not run within the EFI Shell

class sgx_check
Bases: BaseModule

class PRMRR (logger, cs)
Bases: object

reset_variables () → None

check_prmrr_values () → None

check_sgx_config () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (_) → int

chipsec.modules.common.smm module

Compatible SMM memory (SMRAM) Protection check module This CHIPSEC module simply reads SMRAMC and checks that D_LCK is set.

Reference: In 2006, Security Issues Related to Pentium System Management Mode outlined a configuration issue where compatibility SMRAM was not locked on some platforms. This means that ring 0 software was able to modify System Management Mode (SMM) code and data that should have been protected.

In Compatability SMRAM (CSEG), access to memory is defined by the SMRAMC register. When SMRAMC[D_LCK] is not set by the BIOS, SMRAM can be accessed even when the CPU is not in SMM. Such attacks were also described in Using CPU SMM to Circumvent OS Security Functions and Using SMM for Other Purposes.

usage:

```
chipsec_main -m common.smm
```

Examples:

```
>>> chipsec_main.py -m common.smm
```

This module will only run on client (core) platforms that have PCI0.0.0_SMRAMC defined.

class smm
Bases: BaseModule

check_SMRAMC () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.smm_code_chk module

SMM_Code_Chk_En (SMM Call-Out) Protection check

SMM_Code_Chk_En is a bit found in the MSR_SMM_FEATURE_CONTROL register. Once set to '1', any CPU that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE. As such, enabling and locking this bit is an important step in mitigating SMM call-out vulnerabilities. This CHIPSEC module simply reads the register and checks that SMM_Code_Chk_En is set and locked.

Reference:

- **Intel 64 and IA-32 Architectures Software Developer Manual (SDM)**
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Usage:

```
chipsec_main -m common.smm_code_chk
```

Examples:

```
>>> chipsec_main.py -m common.smm_code_chk
```

Registers used:

- MSR_SMM_FEATURE_CONTROL.LOCK
- MSR_SMM_FEATURE_CONTROL.SMM_Code_Chk_En

> **Note**
>
> - MSR_SMM_FEATURE_CONTROL may not be defined or readable on all platforms.

class smm_code_chk
Bases: BaseModule

check_SMM_Code_Chk_En () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.smm_dma module

SMM TSEG Range Configuration Checks

This module examines the configuration and locking of SMRAM range configuration protecting from DMA attacks. If it fails, then DMA protection may not be securely configured to protect SMRAM.

Just like SMRAM needs to be protected from software executing on the CPU, it also needs to be protected from devices that have direct access to DRAM (DMA). Protection from DMA is configured through proper programming of SMRAM memory range. If BIOS does not correctly configure and lock the configuration, then malware could reprogram configuration and open SMRAM area to DMA access, allowing manipulation of memory that should have been protected.

References:

- System Management Mode Design and Security Issues
- Summary of Attack against BIOS and Secure Boot

Usage:

```
chipsec_main -m smm_dma
```

Examples:

```
>>> chipsec_main.py -m smm_dma
```

Registers used:

- TSEGBaseLock (control)
- TSEGLimitLock (control)
- MSR_BIOS_DONE.IA_UNTRUSTED
- PCI0.0.0_TSEGMB.TSEGMB
- PCI0.0.0_BGSM.BGSM
- IA32_SMRR_PHYSBASE.PhysBase
- IA32_SMRR_PHYSMASK.PhysMask

Supported Platforms:

- Core (client)

class smm_dma
Bases: BaseModule

check_tseg_config () → int

check_tseg_locks () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.smrr module

CPU SMM Cache Poisoning / System Management Range Registers check

This module checks to see that SMRRs are enabled and configured.

Reference:

Researchers demonstrated a way to use CPU cache to effectively change values in SMRAM in Attacking SMM Memory via Intel CPU Cache Poisoning and Getting into the SMRAM: SMM Reloaded . If ring 0 software can make SMRAM cacheable and then populate cache lines at SMBASE with exploit code, then when an SMI is triggered, the CPU could execute the exploit code from cache. System Management Mode Range Registers (SMRRs) force non-cachable behavior and block access to SMRAM when the CPU is not in SMM. These registers need to be enabled/configured by the BIOS.

Usage:

```
chipsec_main -m common.smrr [-a modify]
```

- `-a modify`: Attempt to modify memory at SMRR base

Examples:

```
>>> chipsec_main.py -m common.smrr
>>> chipsec_main.py -m common.smrr -a modify
```

Registers used:

- IA32_SMRR_PHYSBASE.PhysBase
- IA32_SMRR_PHYSBASE.Type
- IA32_SMRR_PHYSMASK.PhysMask
- IA32_SMRR_PHYSMASK.Valid

class smrr
Bases: BaseModule

check_SMRR (do_modify: bool) → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.spd_wd module

This module checks that SPD Write Disable bit in SMBus controller has been set

References:

Intel 8 Series/C220 Series Chipset Family Platform Controller Hub datasheet Intel 300 Series Chipset Families Platform Controller Hub datasheet

This module checks the following:

SMBUS_HCFG.SPD_WD

The module returns the following results:

PASSED : SMBUS_HCFG.SPD_WD is set

FAILED : SMBUS_HCFG.SPD_WD is not set and SPDs were detected

INFORMATION: SMBUS_HCFG.SPD_WD is not set, but no SPDs were detected

Hardware registers used:

SMBUS_HCFG

Usage:

```
chipsec_main -m common.spd_wd
```

Examples:

```
>>> chipsec_main.py -m common.spd_wd
```

Note

This module will only run if:

- SMBUS device is enabled
- SMBUS_HCFG.SPD_WD is defined for the platform

class spd_wd
Bases: BaseModule

check_spd_wd () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.spi_access module

SPI Flash Region Access Control

Checks SPI Flash Region Access Permissions programmed in the Flash Descriptor

Usage:

```
chipsec_main -m common.spi_access
```

Examples:

```
>>> chipsec_main.py -m common.spi_access
```

Registers used:

- HSFS.FDV
- FRAP.BRWA

> **Important**
>
> - Some platforms may use alternate means of protecting these regions. Consider this when assessing results.

class spi_access
Bases: BaseModule

check_flash_access_permissions () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.spi_desc module

The SPI Flash Descriptor indicates read/write permissions for devices to access regions of the flash memory. This module simply reads the Flash Descriptor and checks that software cannot modify the Flash Descriptor itself. If software can write to the Flash Descriptor, then software could bypass any protection defined by it. While often used for debugging, this should not be the case on production systems.

This module checks that software cannot write to the flash descriptor.

Usage:

```
chipsec_main -m common.spi_desc
```

Examples:

```
>>> chipsec_main.py -m common.spi_desc
```

Registers used:

- FRAP.BRRA
- FRAP.BRWA

class spi_desc
Bases: BaseModule

check_flash_access_permissions () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.spi_fdopss module

Checks for SPI Controller Flash Descriptor Security Override Pin Strap (FDOPSS). On some systems, this may be routed to a jumper on the motherboard.

Usage:

```
chipsec_main -m common.spi_fdopss
```

Examples:

```
>>> chipsec_main.py -m common.spi_fdopss
```

Registers used:

- HSFS.FDOPSS

class spi_fdopss
Bases: BaseModule

check_fd_security_override_strap () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

chipsec.modules.common.spi_lock module

The configuration of the SPI controller, including protected ranges (PR0-PR4), is locked by HSFS[FLOCKDN] until reset. If not locked, the controller configuration may be bypassed by reprogramming these registers.

This vulnerability (not setting FLOCKDN) is also checked by other tools, including flashrom and Copernicus by MITRE.

This module checks that the SPI Flash Controller configuration is locked.

Reference:

- flashrom
- Copernicus: Question Your Assumptions about BIOS Security

Usage:

```
chipsec_main -m common.spi_lock
```

Examples:

```
>>> chipsec_main.py -m common.spi_lock
```

Registers used:

- FlashLockDown (control)
- SpiWriteStatusDis (control)

class spi_lock
Bases: BaseModule

check_spi_lock () → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

Module contents

chipsec.modules.hsw package

Module contents

chipsec.modules.ivb package

Module contents

chipsec.modules.snb package

Module contents

chipsec.modules.tools package

Subpackages

chipsec.modules.tools.cpu package

Submodules

chipsec.modules.tools.cpu.sinkhole module

This module checks if CPU is affected by 'The SMM memory sinkhole' vulnerability

References:

- Memory Sinkhole presentation by Christopher Domas
- Memory Sinkhole whitepaper

Usage:

```
chipsec_main -m tools.cpu.sinkhole
```

Examples:

```
>>> chipsec_main.py -m tools.cpu.sinkhole
```

Registers used:

- IA32_APIC_BASE.APICBase
- IA32_SMRR_PHYSBASE.PhysBase
- IA32_SMRR_PHYSMASK

Note

- Supported OS: Windows or Linux

Warning

- The system may hang when running this test.
- In that case, the mitigation to this issue is likely working but we may not be handling the exception generated.

class sinkhole
Bases: BaseModule

check_LAPIC_SMRR_overlap ()

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

Module contents

chipsec.modules.tools.secureboot package

Submodules

chipsec.modules.tools.secureboot.te module

Tool to test for 'TE Header' vulnerability in Secure Boot implementations as described in All Your Boot Are Belong To Us

Usage:

```
chipsec_main.py -m tools.secureboot.te [-a <mode>,<cfg_file>,<efi_file>]
```

- `<mode>`
  - `generate_te` (default) convert PE EFI binary `<efi_file>` to TE binary
  - `replace_bootloader` replace bootloader files listed in `<cfg_file>` on ESP with modified `<efi_file>`
  - `restore_bootloader` restore original bootloader files from `.bak` files
- `<cfg_file>` path to config file listing paths to bootloader files to replace
- `<efi_file>` path to EFI binary to convert to TE binary. If no file path is provided, the tool will look for Shell.efi

Examples:

Convert Shell.efi PE/COFF EFI executable to TE executable:

```
chipsec_main.py -m tools.secureboot.te -a generate_te,Shell.efi
```

Replace bootloaders listed in te.cfg file with TE version of Shell.efi executable:

```
chipsec_main.py -m tools.secureboot.te -a replace_bootloader,te.cfg,Shell.efi
```

Restore bootloaders listed in te.cfg file:

```
chipsec_main.py -m tools.secureboot.te -a restore_bootloader,te.cfg
```

IsValidPEHeader (data)

confirm ()

get_bootloader_paths (cfg_file)

get_efi_mount ()

produce_te (fname, outfname)

replace_bootloader (bootloader_paths, new_bootloader_file, do_mount=True)

replace_efi_binary (orig_efi_binary, new_efi_binary)

replace_header (data)

restore_bootloader (bootloader_paths, do_mount=True)

restore_efi_binary (orig_efi_binary)

class te
Bases: BaseModule

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

umount (drive)

usage ()

Module contents

chipsec.modules.tools.smm package

Submodules

chipsec.modules.tools.smm.rogue_mmio_bar module

Experimental module that may help checking SMM firmware for MMIO BAR hijacking vulnerabilities described in the following presentation:

BARing the System: New vulnerabilities in Coreboot & UEFI based systems by Intel Advanced Threat Research team at RECon Brussels 2017

Usage:

```
chipsec_main -m tools.smm.rogue_mmio_bar [-a <smi_start:smi_end>,<b:d.f>]
```

- `smi_start:smi_end`: range of SMI codes (written to IO port 0xB2)
- `b:d.f`: PCIe bus/device/function in b:d.f format (in hex)

Example:

```
>>> chipsec_main.py -m tools.smm.rogue_mmio_bar -a 0x00:0x80
>>> chipsec_main.py -m tools.smm.rogue_mmio_bar -a 0x00:0xFF,0:1C.0
```

Note

Look for 'changes found' messages for items that should be further investigated.

Warning

When running this test, system may freeze, reboot, etc. This is not unexpected behavior and not generally considered a failure.

DIFF (s, t, sz)

class rogue_mmio_bar
Bases: BaseModule

copy_bar (bar_base, bar_base_mem, size)

modify_bar (b, d, f, off, is64bit, bar, new_bar)

restore_bar (b, d, f, off, is64bit, bar)

run (module_argv)

smi_mmio_range_fuzz (thread_id, b, d, f, bar_off, is64bit, bar, new_bar, base, size)

chipsec.modules.tools.smm.smm_ptr module

A tool to test SMI handlers for pointer validation vulnerabilities

Reference:

- **Presented in CanSecWest 2015:**
  - c7zero.info: A New Class of Vulnerability in SMI Handlers of BIOS/UEFI Firmware

Usage: `chipsec_main -m tools.smm.smm_ptr -l log.txt \`
`[-a <mode>,<config_file>|<smic_start:smic_end>,<size>,<address>]`

- `mode`: SMI fuzzing mode
  - `config` = use SMI configuration file <config_file>
  - `fuzz` = fuzz all SMI handlers with code in the range <smic_start:smic_end>
  - `fuzzmore` = fuzz mode + pass 2nd-order pointers within buffer to SMI handlers
- `size`: size of the memory buffer (in Hex)
- `address`: physical address of memory buffer to pass in GP regs to SMI handlers (in Hex)
  - `smram` = option passes address of SMRAM base (system may hang in this mode!)

In `config` mode, SMI configuration file should have the following format

```
SMI_code=<SMI code> or *
SMI_data=<SMI data> or *
RAX=<value of RAX> or * or PTR or VAL
RBX=<value of RBX> or * or PTR or VAL
RCX=<value of RCX> or * or PTR or VAL
RDX=<value of RDX> or * or PTR or VAL
RSI=<value of RSI> or * or PTR or VAL
RDI=<value of RDI> or * or PTR or VAL
[PTR_OFFSET=<offset to pointer in the buffer>]
[SIG=<signature>]
[SIG_OFFSET=<offset to signature in the buffer>]
[Name=<SMI name>]
[Desc=<SMI description>]
```

Where:

- `[]`: optional line
- *: Don't Care (the module will replace * with 0x0)
- PTR: Physical address SMI handler will write to (the module will replace PTR with physical address provided as a command-line argument)
- VAL: Value SMI handler will write to PTR address (the module will replace VAL with hardcoded _FILL_VALUE_xx)

Examples:

```
>>> chipsec_main.py -m tools.smm.smm_ptr
>>> chipsec_main.py -m tools.smm.smm_ptr -a fuzzmore,0x0:0xFF -l smm.log
```

Warning

- This is a potentially destructive test

DIFF (s, t, sz)

FILL_BUFFER (_fill_byte, _fill_size, _ptr_in_buffer, _ptr, _ptr_offset, _sig, _sig_offset)

class smi_desc
Bases: object

class smm_ptr
Bases: BaseModule

check_memory (_addr, _smi_desc, fn, restore_contents=False)

fill_memory (_addr, is_ptr_in_buffer, _ptr, _ptr_offset, _sig, _sig_offset)

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

send_smi (thread_id, smi_code, smi_data, name, desc, rax, rbx, rcx, rdx, rsi, rdi)

smi_fuzz_iter (thread_id, _addr, _smi_desc, fill_contents=True, restore_contents=False)

test_config (thread_id, _smi_config_fname, _addr, _addr1)

test_fuzz (thread_id, smic_start, smic_end, _addr, _addr1)

Module contents

chipsec.modules.tools.uefi package

Submodules

chipsec.modules.tools.uefi.reputation module

This module checks current contents of UEFI firmware ROM or specified firmware image for bad EFI binaries as per the VirusTotal API. These can be EFI firmware volumes, EFI executable binaries (PEI modules, DXE drivers..) or EFI sections. The module can find EFI binaries by their UI names, EFI GUIDs, MD5/SHA-1/SHA-256 hashes or contents matching specified regular expressions.

Important! This module can only detect bad or vulnerable EFI modules based on the file's reputation on VT.

Usage:

chipsec_main.py -i -m tools.uefi.reputation -a <vt_api_key>[,<vt_threshold>,<fw_image>]

vt_api_key : *API key to VirusTotal. Can be obtained by visting https://www.virustotal.com/gui/join-us.*

This argument must be specified.

vt_threshold : *The minimal number of different AV vendors on VT which must claim an EFI module is malicious*

before failing the test. Defaults to 10.

fw_image : *Full file path to UEFI firmware image*

If not specified, the module will dump firmware image directly from ROM

Note

- Requires virustotal-api

class reputation
Bases: BaseModule

check_reputation ()

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

reputation_callback (efi_module)

run (module_argv)

usage ()

chipsec.modules.tools.uefi.s3script_modify module

This module will attempt to modify the S3 Boot Script on the platform. Doing this could cause the platform to malfunction. Use with care!

Usage:

Replacing existing opcode:

```
chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,<reg_opcode>,<address>,<value>
    <reg_opcode> = pci_wr|mmio_wr|io_wr|pci_rw|mmio_rw|io_rw

chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,mem[,<address>,<value>]

chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,dispatch``

chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,dispatch_ep``
```

Adding new opcode:

```
chipsec_main.py -m tools.uefi.s3script_modify -a add_op,<reg_opcode>,<address>,<value>,<width>
    <reg_opcode> = pci_wr|mmio_wr|io_wr

chipsec_main.py -m tools.uefi.s3script_modify -a add_op,dispatch[,<entrypoint>]
```

Examples:

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,<reg_opcode>,<address>,<value>
>>>   <reg_opcode> = pci_wr|mmio_wr|io_wr|pci_rw|mmio_rw|io_rw
```

The option will look for a script opcode that writes to PCI config, MMIO or I/O registers and modify the opcode to write the given value to the register with the given address.

After executing this, if the system is vulnerable to boot script modification, the hardware configuration will have changed according to given <reg_opcode>.

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,mem
```

The option will look for a script opcode that writes to memory and modify the opcode to write the given value to the given address.

By default this test will allocate memory and write write `0xB007B007` that location.

After executing this, if the system is vulnerable to boot script modification, you should find the given value in the allocated memory location.

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,dispatch
```

The option will look for a dispatch opcode in the script and modify the opcode to point to a different entry point. The new entry point will contain a HLT instruction.

After executing this, if the system is vulnerable to boot script modification, the system should hang on resume from S3.

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,dispatch_ep
```

The option will look for a dispatch opcode in the script and will modify memory at the entry point for that opcode. The modified instructions will contain a HLT instruction.

After executing this, if the system is vulnerable to dispatch opcode entry point modification, the system should hang on resume from S3.

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a add_op,<reg_opcode>,<address>,<value>,<width>
>>>   <reg_opcode> = pci_wr|mmio_wr|io_wr
```

The option will add a new opcode which writes to PCI config, MMIO or I/O registers with specified values.

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a add_op,dispatch
```

The option will add a new DISPATCH opcode to the script with entry point to either existing or newly allocated memory.

class s3script_modify
Bases: BaseModule

DISPATCH_ENTRYPOINT_INSTR = *'\x90\x90ôô'*

get_bootscript ()

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

modify_s3_add (new_opcode)

modify_s3_dispatch ()

modify_s3_dispatch_ep ()

modify_s3_mem (address, new_value)

modify_s3_reg (opcode, address, new_value)

run (module_argv)

chipsec.modules.tools.uefi.scan_blocked module

This module checks current contents of UEFI firmware ROM or specified firmware image for blocked EFI binaries which can be EFI firmware volumes, EFI executable binaries (PEI modules, DXE drivers..) or EFI sections. The module can find EFI binaries by their UI names, EFI GUIDs, MD5/SHA-1/SHA-256 hashes or contents matching specified regular expressions.

Important! This module can only detect what it knows about from its config file. If a bad or vulnerable binary is not detected then its 'signature' needs to be added to the config.

Usage:

```
chipsec_main.py -i -m tools.uefi.scan_blocked [-a <fw_image>,<blockedlist>]
```

- `fw_image` Full file path to UEFI firmware image. If not specified, the module will dump firmware image directly from ROM
- `blockedlist` JSON file with configuration of blocked EFI binaries (default = `blockedlist.json`). Config file should be located in the same directory as this module

Examples:

```
>>> chipsec_main.py -m tools.uefi.scan_blocked
```

Dumps UEFI firmware image from flash memory device, decodes it and checks for blocked EFI modules defined in the default config `blockedlist.json`

```
>>> chipsec_main.py -i --no_driver -m tools.uefi.scan_blocked -a uefi.rom,blockedlist.json
```

Decodes `uefi.rom` binary with UEFI firmware image and checks for blocked EFI modules defined in `blockedlist.json` config

Note

- `-i` and `--no_driver` arguments can be used in this case because the test does not depend on the platform and no kernel driver is required when firmware image is specified

class scan_blocked
Bases: BaseModule

blockedlist_callback (efi_module)

check_blockedlist ()

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

usage ()

chipsec.modules.tools.uefi.scan_image module

The module can generate a list of EFI executables from (U)EFI firmware file or extracted from flash ROM, and then later check firmware image in flash ROM or file against this list of expected executables

Usage:

```
chipsec_main -m tools.uefi.scan_image [-a generate|check,<json>,<fw_image>]
```

- `generate` **Generates a list of EFI executable binaries from the UEFI** firmware image (default)
- `check` **Decodes UEFI firmware image and checks all EFI executable** binaries against a specified list
- `json` **JSON file with configuration of allowed list EFI** executables (default = `efilist.json`)
- `fw_image` **Full file path to UEFI firmware image. If not specified,** the module will dump firmware image directly from ROM

Examples:

```
>>> chipsec_main -m tools.uefi.scan_image
```

Creates a list of EFI executable binaries in `efilist.json` from the firmware image extracted from ROM

```
>>> chipsec_main -i -n -m tools.uefi.scan_image -a generate,efilist.json,uefi.rom
```

Creates a list of EFI executable binaries in `efilist.json` from `uefi.rom` firmware binary

```
>>> chipsec_main -i -n -m tools.uefi.scan_image -a check,efilist.json,uefi.rom
```

Decodes `uefi.rom` UEFI firmware image binary and checks all EFI executables in it against a list defined in `efilist.json`

Note

- `-i` and `-n` arguments can be used when specifying firmware file because the module doesn't depend on the platform and doesn't need kernel driver

class scan_image
Bases: BaseModule

check_list (json_pth: str) → int

generate_efilist (json_pth: str) → int

genlist_callback (efi_module: EFI_MODULE) → None

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

usage ()

chipsec.modules.tools.uefi.uefivar_fuzz module

The module is fuzzing UEFI Variable interface.

The module is using UEFI SetVariable interface to write new UEFI variables to SPI flash NVRAM with randomized name/attributes/GUID/data/size.

Usage:

```
chipsec_main -m tools.uefi.uefivar_fuzz [-a <options>]
```

Options:

```
[-a <test>,<iterations>,<seed>,<test_case>]
```

- `test` : UEFI variable interface to fuzz (all, name, guid, attrib, data, size)
- `iterations` : Number of tests to perform (default = 1000)
- `seed` : RNG seed to use
- `test_case` : Test case # to skip to (combined with seed, can be used to skip to failing test)

All module arguments are optional

Examples::

```
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz -a all,100000
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz -a data,1000,123456789
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz -a name,1,123456789,94
```

Note

- This module returns a WARNING by default to indicate that a manual review is needed.
- Writes may generate 'ERROR's, this can be expected behavior if the environment rejects them.

Warning

- This module modifies contents of non-volatile SPI flash memory (UEFI Variable NVRAM).
- This may render system UNBOOTABLE if firmware doesn't properly handle variable update/delete operations.

Important

- Evaluate the platform for expected behavior to determine PASS/FAIL.
- Behavior can include platform stability and retaining protections.

class uefivar_fuzz
Bases: BaseModule

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

rnd (n=1)

run (module_argv)

usage ()

Module contents

chipsec.modules.tools.vmm package

Subpackages

chipsec.modules.tools.vmm.hv package

Submodules

chipsec.modules.tools.vmm.hv.define module

Hyper-V specific defines

get_hypercall_name (code, defvalue='')

get_hypercall_status (code, defvalue='')

get_msr_name (code, defvalue='')

set_variables (varlist)

chipsec.modules.tools.vmm.hv.hypercall module

Hyper-V specific hypercall functionality

class HyperVHypercall
Bases: BaseModuleHwAccess

custom_fuzzing (call_code, total_tests)

input_parameters_fuzzing (i, maxlen, status_list, total_tests)

print_connectionid (status_list)

print_hypercall_status ()

print_hypervisor_cpuid (cpuid_eax, cpuid_ecx=0)

print_hypervisor_info ()

print_input_parameters (i, maxlen, status_list)

print_partition_properties ()

print_partitionid ()

print_synthetic_msrs ()

scan_connectionid (id_list)

scan_for_success_status (i, total_tests)

scan_hypercalls (code_list)

scan_input_parameters (i, maxlen)

scan_partitionid (id_list)

set_partition_property (part, prop, value)

getrandbits (k) → x. Generates an int with k random bits.

chipsec.modules.tools.vmm.hv.hypercallfuzz module

Hyper-V hypercall fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.hv.hypercall -a <mode>[,<vector>,<iterations>] -l log.txt
```

- `mode` fuzzing mode
  - = `status-fuzzing` finding parameters with hypercall success status
  - = `params-info` shows input parameters valid ranges
  - = `params-fuzzing` parameters fuzzing based on their valid ranges
  - = `custom-fuzzing` fuzzing of known hypercalls
- `vector` hypercall vector
- `iterations` number of hypercall iterations

Note: the fuzzer is incompatible with native VMBus driver (`vmbus.sys`). To use it, remove `vmbus.sys`

class HypercallFuzz
Bases: BaseModule

run (module_argv)

usage ()

getrandbits (k) → x. Generates an int with k random bits.

chipsec.modules.tools.vmm.hv.synth_dev module

Hyper-V VMBus synthetic device generic fuzzer

Usage:

Print channel offers:

```
chipsec_main.py -i -m tools.vmm.hv.synth_dev -a info
```

Fuzzing device with specified relid:

```
chipsec_main.py -i -m tools.vmm.hv.synth_dev -a fuzz,<relid> -l log.txt
```

Note: the fuzzer is incompatible with native VMBus driver (`vmbus.sys`). To use it, remove `vmbus.sys`

class VMBusDeviceFuzzer
Bases: VMBusDiscovery

device_fuzzing (relid)

print_1 (info, indent=0)

print_statistics ()

send_1 (relid, messages, info, order)

getrandbits (k) → x. Generates an int with k random bits.

class synth_dev
Bases: BaseModule

run (module_argv)

usage ()

chipsec.modules.tools.vmm.hv.synth_kbd module

Hyper-V VMBus synthetic keyboard fuzzer. Fuzzes inbound ring buffer in VMBus virtual keyboard device.

Usage:

```
chipsec_main.py -i -m tools.vmm.hv.synth_kbd -a fuzz -l log.txt
```

Note: the fuzzer is incompatible with native VMBus driver (`vmbus.sys`). To use it, remove `vmbus.sys`

class RingBufferFuzzer
Bases: RingBuffer

ringbuffer_read ()

getrandbits (k) → x. Generates an int with k random bits.

class synth_kbd
Bases: BaseModule

run (module_argv)

usage ()

chipsec.modules.tools.vmm.hv.vmbus module

Hyper-V VMBus functionality

class HyperV
Bases: BaseModuleDebug

hv_init ()

hv_post_msg (message)

hv_recv_events (sint)

hv_recv_msg (sint)

hv_signal_event (connection_id, flag_number)

class RingBuffer
Bases: BaseModuleDebug

ringbuffer_alloc (pages=4)

ringbuffer_copyfrom (index, total)

ringbuffer_copyto (index, data)

ringbuffer_init ()

ringbuffer_read ()

ringbuffer_read_with_timeout (timeout=0)

ringbuffer_write (data)

ringbuffer_write_with_timeout (message, timeout=0)

class VMBus
Bases: HyperV

vmbus_clear ()

vmbus_close (child_relid)

vmbus_connect (vmbus_version=131076, target_vcpu=0)

vmbus_disconnect ()

vmbus_establish_gpadl (child_relid, gpadl, pfn)

vmbus_get_next_gpadl ()

vmbus_get_next_version (current_version)

vmbus_init ()

vmbus_onmessage ()

vmbus_open (child_relid, gpadl, pageoffset=2, userdata='\x00')

vmbus_post_msg (message)

vmbus_process_rescind_offer (child_relid)

vmbus_recv_events ()

vmbus_recv_msg (timeout=0)

vmbus_recvpacket (child_relid)

vmbus_recvpacket_raw ()

vmbus_request_offers ()

vmbus_sendpacket (child_relid, data, requestid, packet_type, flags)

vmbus_sendpacket_multipagebuffer ()

vmbus_sendpacket_pagebuffer ()

vmbus_setevent (child_relid)

vmbus_teardown_gpadl (child_relid, gpadl)

class VMBusDiscovery
Bases: VMBus

get_relid_by_guid (guid)

print_created_gpadl ()

print_events ()

print_offer_channels ()

print_open_channels ()

print_supported_versions ()

scan_physical_addresses (version)

scan_supported_versions (mask=983055)

vmbus_rescind_all_offers ()

getrandbits (k) → x. Generates an int with k random bits.

chipsec.modules.tools.vmm.hv.vmbusfuzz module

Hyper-V VMBus generic fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.hv.vmbusfuzz -a fuzz,<parameters>
```

Parameters:

- `all` : Fuzzing all bytes
- `hv` : Fuzzing HyperV message header
- `vmbus` : Fuzzing HyperV message body / VMBUS message
- `<pos>,<size>` : Fuzzing number of bytes at specific position

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.hv.vmbusfuzz -a fuzz,all -l log.txt
```

Note

- The fuzzer is incompatible with native VMBus driver (`vmbus.sys`). To use it, remove `vmbus.sys`
- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class VMBusFuzz

Bases: VMBusDiscovery

hv_post_msg (message)

run (module_argv)

vmbus_test1_run ()

getrandbits (k) → x. Generates an int with k random bits.

Module contents

chipsec.modules.tools.vmm.vbox package

Submodules

chipsec.modules.tools.vmm.vbox.vbox_crash_apicbase module

Oracle VirtualBox CVE-2015-0377 check

Reference:

- PoC test for Host OS Crash when writing to IA32_APIC_BASE MSR (Oracle VirtualBox CVE-2015-0377)
- http://www.oracle.com/technetwork/topics/security/cpujan2015-1972971.html

Usage:

```
chipsec_main.py -i -m tools.vmm.vbox_crash_apicbase
```

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.vbox_crash_apicbase
```

Registers used:

- IA32_APIC_BASE

Warning

- Module can cause VMM/Host OS to crash; if so, this is a FAILURE

class vbox_crash_apicbase
Bases: BaseModule

run (module_argv)

Module contents

chipsec.modules.tools.vmm.xen package

Submodules

chipsec.modules.tools.vmm.xen.define module

Xen specific defines

get_hypercall_name (vector, defvalue='')

get_hypercall_status (code, brief=False)

get_hypercall_status_extended (code)

get_invalid_hypercall_code ()

get_iverr (status, bits=64)

set_variables (varlist)

chipsec.modules.tools.vmm.xen.hypercall module

Xen specific hypercall functionality

class XenHypercall
Bases: BaseModuleHwAccess

fuzz_hypercall (code, iterations)

fuzz_hypercalls_randomly (codes, iterations)

get_hypervisor_info ()

get_value (arg)

hypercall (args, size=0, data='')

print_hypercall_status ()

print_hypervisor_info (info)

scan_hypercalls (vector_list)

xen_version (cmd, size=0, data='')

getrandbits (k) → x. Generates an int with k random bits.

chipsec.modules.tools.vmm.xen.hypercallfuzz module

Xen hypercall fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a <mode>[,<vector>,<iterations>]
```

- `mode` : fuzzing mode
  - `help` : Prints this help
  - `info` : Hypervisor information
  - `fuzzing` : Fuzzing specified hypercall
  - `fuzzing-all` : Fuzzing all hypercalls
  - `fuzzing-all-randomly` : Fuzzing random hypercalls
- `<vector>` : Code or name of a hypercall to be fuzzed (use info)
- `<iterations>` : Number of fuzzing iterations

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing,10 -l log.txt
>>> chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing-all,50 -l log.txt
>>> chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing-all-randomly,10,0x10000000 -l log.txt
```

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class HypercallFuzz
Bases: BaseModule

get_int (arg, base=10, defvalue=10000)

run (module_argv)

usage ()

chipsec.modules.tools.vmm.xen.xsa188 module

This module triggers host crash on vulnerable Xen 4.4

Reference:

- **Proof-of-concept module for Xen XSA-188**

- CVE-2016-7154: “use after free in FIFO event channel code”
- Discovered by Mikhail Gorobets

Usage:

```
chipsec_main.py -m tools.vmm.xen.xsa188
```

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.xen.xsa188
```

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class xsa188
Bases: BaseModule

run (module_argv)

Module contents

Submodules

chipsec.modules.tools.vmm.common module

Common functionality for VMM related modules/tools

class BaseModuleDebug
Bases: BaseModule

dbg (message: str)

err (message: str) → None

fatal (message: str) → None

hex (title: str, data: str, w=16) → None

info_bitwise (reg: int, desc: Dict[int, str]) → None

```
    msg (message: str) → None

class BaseModuleHwAccess
    Bases: BaseModuleSupport

    cpuid_info (eax: int, ecx: int, desc: str) → Tuple[int, int, int, int]

    rdmsr (msr: int) → Tuple[int, int]

    wrmsr (msr: int, value: int) → None

class BaseModuleSupport
    Bases: BaseModuleDebug

    add_initial_data (vector: int, buffer: str, status: str) → None

    dump_initial_data (filename: str) → None

    get_initial_data (statuses: List[str], vector: int, size: int, padding='\x00') → List[str]

    stats_event (name: str) → None

    stats_print (title: str) → None

    stats_reset () → None

get_int_arg (arg: str) → int

getrandbits (k) → x. Generates an int with k random bits.

hv_hciv (rep_start: int, rep_count: int, call_code: int, fast: int = 0) → int

overwrite (buffer: bytes, string: bytes, position: int) → bytes

rand_dd (n: int, rndbytes: int = 1, rndbits: int = 1) → List[int]

class session_logger (log: bool, details: str)
    Bases: object

    closefile () → None

    flush () → None

    write (message: str) → None

uuid (id: bytes) → str

weighted_choice (choices: List[Tuple[int, float]]) → int
```

chipsec.modules.tools.vmm.cpuid_fuzz module

Simple CPUID VMM emulation fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.cpuid_fuzz [-a random]
```

- `random` : Fuzz in random order (default is sequential)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.cpuid_fuzz
>>> chipsec_main.py -i -m tools.vmm.cpuid_fuzz -l log.txt
>>> chipsec_main.py -i -m tools.vmm.cpuid_fuzz -a random
```

Additional options set within the module:

- `_NO_EAX_TO_FUZZ` : No of EAX values to fuzz within each step
- `_EAX_FUZZ_STEP` : Step to fuzz range of EAX values
- `_NO_ITERATIONS_TO_FUZZ` : Number of iterations if *random* chosen
- `_FUZZ_ECX_RANDOM` : Fuzz ECX with random values?
- `_MAX_ECX` : Max ECX value
- `_EXCLUDE_CPUID` : Exclude the following EAX values from fuzzing
- `_FLUSH_LOG_EACH_ITER` : Flush log file after each iteration
- `_LOG_OUT_RESULTS` : Log output results

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class cpuid_fuzz
Bases: BaseModule

fuzz_CPUID (eax_start, random_order=False)

run (module_argv)

chipsec.modules.tools.vmm.ept_finder module

Extended Page Table (EPT) Finder

Usage:

```
chipsec_main -m tools.vmm.ept_finder [-a dump,<file_name>|file,<file_name>,<revision_id>]
```

- `dump` : Dump contents
- `file` : Load contents from file
- `<file_name>` : File name to read from or dump to
- `<revision_id>` : Revision ID (hex)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.ept_finder
>>> chipsec_main.py -i -m tools.vmm.ept_finder -a dump,my_file.bin
>>> chipsec_main.py -i -m tools.vmm.ept_finder -a file,my_file.bin,0x0
```

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class c_extended_page_tables_from_file (cs, read_from_file, par)
Bases: c_extended_page_tables

readmem (name, addr, size=4096)

class ept_finder
Bases: BaseModule

dump_dram (filename, pa, end_pa, buffer_size=1048576)

find_ept_pt (pt_addr_list, mincount, level)

find_vmcs_by_ept (ept_list, revision_id)

get_memory_ranges ()

read_physical_mem (addr, size=4096)

read_physical_mem_dword (addr)

run (module_argv)

chipsec.modules.tools.vmm.hypercallfuzz module

Pretty simple VMM hypercall fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.hypercallfuzz [-a <mode>,<vector_reg>,<maxval>,<iterations>]
```

- `mode` : *Hypercall fuzzing mode*
  - `exhaustive` : Fuzz all arguments exhaustively in range `[0:<maxval>]` (default)

        - `random` : Send random values in all registers in range `[0:<maxval>]`
    - `vector_reg` : Hypercall vector register
    - `maxval` : Maximum value of each register
    - `iterations` : Number of iterations in random mode

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.hypercallfuzz
>>> chipsec_main.py -i -m tools.vmm.hypercallfuzz -a random,22,0xFFFF,1000
```

Additional options set within the module:

- `DEFAULT_VECTOR_MAXVAL` : Default maximum value
- `DEFAULT_MAXVAL_EXHAUSTIVE` : Default maximum value for exhaustive testing
- `DEFAULT_MAXVAL_RANDOM` : Default maximum value for random testing
- `DEFAULT_RANDOM_ITERATIONS` : Default iterations for random testing
- `_FLUSH_LOG_EACH_ITER` : Set to flush log after each iteration
- `_LOG_ALL_GPRS` : Display log of each iteration values

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class hypercallfuzz
Bases: BaseModule

fuzz_generic_hypercalls ()

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

chipsec.modules.tools.vmm.iofuzz module

Simple port I/O VMM emulation fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.iofuzz [-a <mode>,<count>,<iterations>]
```

- `<mode>` **:** *SMI handlers testing mode*
  - `exhaustive` : Fuzz all I/O ports exhaustively (default)
  - `random` : Fuzz randomly chosen I/O ports
- `<count>` : Number of times to write to each port (default = 1000)
- `<iterations>` : Number of I/O ports to fuzz (default = 1000000 in random mode)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.iofuzz
>>> chipsec_main.py -i -m tools.vmm.iofuzz -a random,9000,4000000
```

Additional options set within the module:

- `MAX_PORTS` : Maximum ports
- `MAX_PORT_VALUE` : Maximum port value to use
- `DEFAULT_PORT_WRITE_COUNT` : Default port write count if not specified with switches
- `DEFAULT_RANDOM_ITERATIONS` : Default port write iterations if not specified with switches
- `_FLUSH_LOG_EACH_ITER` : Flush log after each iteration
- `_FUZZ_SPECIAL_VALUES` : Specify to use 1-2-4 byte values
- `_EXCLUDE_PORTS` : Ports to exclude (list)

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class iofuzz
Bases: BaseModule

fuzz_ports (iterations, write_count, random_order=False)

run (module_argv)

chipsec.modules.tools.vmm.msr_fuzz module

Simple CPU Module Specific Register (MSR) VMM emulation fuzzer

Usage:

```
chipsec_main -m tools.vmm.msr_fuzz [-a random]
```

- `-a` : random = use random values (default = sequential numbering)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.msr_fuzz
>>> chipsec_main.py -i -m tools.vmm.msr_fuzz -a random
```

Additional options set within the module:

- `_NO_ITERATIONS_TO_FUZZ` : Number of iterations to fuzz randomly
- `_READ_MSR` : Specify to read MSR when fuzzing it
- `_FLUSH_LOG_EACH_MSR` : Flush log file before each MSR
- `_FUZZ_VALUE_0_all1s` : Try all 0 & all 1 values to be written to each MSR
- `_FUZZ_VALUE_5A` : Try 0x5A values to be written to each MSR
- `_FUZZ_VALUE_RND` : Try random values to be written to each MSR
- `_EXCLUDE_MSR` : MSR values to exclude (list)

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class msr_fuzz
Bases: BaseModule

fuzz_MSRs (msr_addr_start, random_order=False)

run (module_argv)

chipsec.modules.tools.vmm.pcie_fuzz module

Simple PCIe device Memory-Mapped I/O (MMIO) and I/O ranges VMM emulation fuzzer

Usage:

```
chipsec_main -m tools.vmm.pcie_fuzz [-a <bus> <dev> <fun>]
```

- `<bus>` : Bus # to fuzz (in hex)
- `<dev>` : Device # to fuzz (in hex)
- `<fun>` : Function # to fuzz (in hex)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.pcie_fuzz
>>> chipsec_main.py -i -m tools.vmm.pcie_fuzz -l log.txt
>>> chipsec_main.py -i -m tools.vmm.pcie_fuzz -a 0 1f 0
```

Additional options set within the module:

- `IO_FUZZ` : Set to fuzz IO BARs
- `CALC_BAR_SIZE` : Set to calculate BAR sizes
- `TIMEOUT` : Timeout between memory writes (seconds)
- `ACTIVE_RANGE` : Set to fuzz MMIO BAR in Active range
- `BIT_FLIP` : Set to fuzz using bit flips
- `_EXCLUDE_BAR` : BARs to exclude (list)

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class pcie_fuzz
Bases: BaseModule

find_active_range (bar, size)

fuzz_io_bar (bar, size=256)

fuzz_mmio_bar (bar, is64bit, size=4096)

fuzz_mmio_bar_in_active_range (bar, is64bit, list)

fuzz_mmio_bar_in_active_range_bit_flip (bar, is64bit, list)

fuzz_mmio_bar_in_active_range_random (bar, is64bit, list)

fuzz_mmio_bar_random (bar, is64bit, size=4096)

fuzz_offset (bar, reg_off, reg_value, is64bit)

fuzz_pcie_device (b, d, f)

fuzz_unaligned (bar, reg_off, is64bit)

run (module_argv)

chipsec.modules.tools.vmm.pcie_overlap_fuzz module

PCIe device Memory-Mapped I/O (MMIO) ranges VMM emulation fuzzer which first overlaps MMIO BARs of all available PCIe devices then fuzzes them by writing garbage if corresponding option is enabled

Usage:

```
chipsec_main.py -i -m tools.vmm.pcie_overlap_fuzz
```

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.pcie_overlap_fuzz -l log.txt
```

Additional options set within the module:

- `OVERLAP_MODE` : Set overlap direction
- `FUZZ_OVERLAP` : Set for fuzz overlaps
- `FUZZ_RANDOM` : Set to fuzz in random mode
- `_EXCLUDE_MMIO_BAR1` : List 1 of MMIO bars to exclude
- `_EXCLUDE_MMIO_BAR2` : List 2 of MMIO bars to exclude

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class pcie_overlap_fuzz
Bases: BaseModule

fuzz_mmio_bar (bar, is64bit, size=4096)

fuzz_mmio_bar_random (bar, is64bit, size=4096)

fuzz_offset (bar, reg_off, reg_value, is64bit)

fuzz_overlap_pcie_device (pcie_devices)

fuzz_unaligned (bar, reg_off, is64bit)

overlap_mmio_range (bus1, dev1, fun1, is64bit1, off1, bus2, dev2, fun2, is64bit2, off2, direction)

run (module_argv)

chipsec.modules.tools.vmm.venom module

QEMU VENOM vulnerability DoS PoC test

Reference:

- Module is based on PoC by Marcus Meissner
- VENOM: QEMU vulnerability (CVE-2015-3456)

Usage:

```
chipsec_main.py -i -m tools.vmm.venom
```

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.venom
```

Additional options set within the module:

- `ITER_COUNT` : Iteration count
- `FDC_PORT_DATA_FIFO` : FDC DATA FIFO port
- `FDC_CMD_WRVAL` : FDC Command write value
- `FD_CMD` : FD Command

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

class venom
Bases: BaseModule

run (module_argv)

venom_impl ()

Module contents

Submodules

chipsec.modules.tools.generate_test_id module

Generate a test ID using hashlib from the test's file name (no file extension). Hash is truncated to 28 bits.

Usage:

```
chipsec_main -m common.tools.generate_test_id -a <test name>
```

Examples:

```
>>> chipsec_main.py -m common.tools.generate_test_id -a remap
>>> chipsec_main.py -m common.tools.generate_test_id -a s3bootscript
>>> chipsec_main.py -m common.tools.generate_test_id -a bios_ts
```

class generate_test_id
Bases: BaseModule

generate_id (test_name: str) → int

is_supported () → bool
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv: List[str]) → int

usage ()

chipsec.modules.tools.wsmt module

The Windows SMM Security Mitigation Table (WSMT) is an ACPI table defined by Microsoft that allows system firmware to confirm to the operating system that certain security best practices have been implemented in System Management Mode (SMM) software.

Reference:

- See <https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-uefi-wsmt> for more details.

Usage:

```
chipsec_main -m common.wsmt
```

Examples:

```
>>> chipsec_main.py -m common.wsmt
```

Note

- Analysis is only necessary if Windows is the primary OS

class wsmt
Bases: BaseModule

check_wsmt ()

is_supported ()
This method should be overwritten by the module returning True or False depending whether or not this module is supported in the currently running platform. To access the currently running platform use

run (module_argv)

Module contents

Module contents

Contribution and Style Guides

Python Version

All Python code, and PEP support, must to be limited to the features supported by **Python 3.6.8**.

This is earliest version of Python utilized by CHIPSEC, the version of the EFI Shell Python.

Python Coding Style Guide

CHIPSEC mostly follows the PEP8 with some exceptions. This attempts to highlight those as well as clarify others.

Consistency and readability are the goal but not at the expense of readability or functionality.

If in doubt, follow the existing code style and formatting.

1. PEP8

   PEP 8 is a set of recommended code style guidelines (conventions) for Python.

   PEP 8

2. Linting tools

   CHIPSEC includes a Flake8 configuration file

   CHIPSEC flake8 config

3. Zen of Python

   Great philosophy around Python building principles.

   PEP 20

4. Headers and Comments

   Use single line comments, a single hash/number sign/octothorpe '#'.

   Should contain a space immediately after the '#'.

```
# Good header comment
```

5. Single vs Double Quotes

   Single quotes are encouraged but can vary with use case.

   Avoid using backslashes '\' in strings.

```
'This is a preferred "string".'
"Also an acceptable 'string'."

"Avoid making this \"string\"."
```

6. Imports

   Import order:

   1. Python standard library
   2. Third-party imports
   3. CHIPSEC and local application imports

   Avoid using `import *` or `from import *`. This could pollute the namespace.

```
# Good
import sys
from chipsec.module_common import BaseModule, ModuleResult

# Bad - using '*' and importing sys after local imports
import *
from chipsec.module_common import *
import sys
```

   Avoid using `from __future__ imports`. These may not work on older or all interpreter versions required in all supported environments.

7. Line Length

   Maximum line length should be 120 characters.

   If at or near this limit, consider rewriting (eg simplifying) the line instead of breaking it into multiple lines.

   Long lines can be an indication that too many things are happening at once and/or difficult to read.

8. Class Names

   `HAL` and `utilcmd` **classes** should use **UpperCamelCase** (**PascalCase**) Words and acronyms are capitalized with no spaces or underscores.

   Test **module** class names MUST match the module name which are typically **snake_case**

9. Constants

   Constants should use **CAPITALIZATION_WITH_UNDERSCORES**

10. Variable Names

   Variable names should use **snake_case**

   Lower-case text with underscores between words.

11. Local Variable Names (private)

Prefixed with an underscore, **_private_variable**

Not a hard rule but will help minimize any variable name collisions with upstream namespace.

12 Dunder (double underscore)

- Avoid using `__dunders__` when naming variables. Should be used for functions that overwrite or add to classes and only as needed.

  Dunders utilize double (two) underscore characters before and after the name.

13 Code Indents

- CHIPSEC uses 4 space 'tabbed' indents.

  No mixing spaces and tabs.

  - 1 indent = 4 spaces
  - No tabs

  Recommend updating any IDE used to use 4 space indents by default to help avoid mixing tabs with spaces in the code.

14 Operator Precedence, Comparisons, and Parentheses

- If in doubt, wrap evaluated operators into logical sections if using multiple operators or improves readability.

  While not needed in most cases, it can improve readability and limit the possibility of 'left-to-right chaining' issues.

```
# Preferred
if (test1 == True) or (test2 in data_list):
    return True

# Avoid.  Legal but behavior may not be immediately evident.
if True is False == False:
    return False
```

15 Whitespace

- No whitespace inside parentheses, brackets, or braces.

  No whitespace before a comma, colon, or semicolons.

  Use whitespace after a comma, colon, or semicolon.

  Use whitespace around operators: +, -, *, **, /, //, %, =, ==, <, >, <=, >=, <>, !=, is, in, is not, not in, <<, >>, &, |, ^

  No trailing whitespace.

16 Non-ASCII Characters

- If including any non-ASCII characters anywhere in a python file, include the python encoding comment at the beginning of the file.

```
# -*- coding: utf-8 -*-
```

  No non-ASCII class, function, or variable names.

17 Docstrings

- Use three double-quotes for all docstrings.

```
"""String description docstring."""
```

18 Semicolons

- Do not use semicolons.

19 Try Except

- Avoid using nested try-except.

  The routine you are calling, may already be using one.

20 Avoid for-else and while-else loops

- The loop behavior for these can be counterintuitive.

  If they have to be used, make sure to properly document the expected behavior / work-flow.

f-Strings

PEP versions supported by CHIPSEC

PEP / bpo	Title	Summary	Python Version	Supported
PEP 498	Literal String Interpolation	Adds a new string formatting mechanism: Literal String Interpolation, f-strings	3.6	Yes
bpo 36817	Add = to f-strings for easier debugging	f-strings support = for self-documenting expressions	3.8	No
PEP 701	Syntactic formalization of f-strings	Lift some restrictions from PEP 498 and formalize grammar for f-strings	3.12	No

Type Hints

For more information on Python Type Hints:

PEP 483 - The Theory of Type Hints

This table lists which Type Hint PEPs are in scope for CHIPSEC.

PEP versions supported by CHIPSEC

PEP	Title	Summary	Python Version	Supported
PEP 3107	Function Annotations	Syntax for adding arbitrary metadata annotations to Python functions	3.0	Yes
PEP 362	Function Signature Object	Contains all necessary information about a function and its parameters	3.3	Yes

PEP 484	Type Hints	Standard syntax for type annotations	3.5	Yes
PEP 526	Syntax for Variable Annotations	Adds syntax for annotating the types of variables	3.6	Yes
PEP 544	Protocols: Structural subtyping (static duck typing)	Specify type metadata for static type checkers and other third-party tools	3.8	No
PEP 585	Type Hinting Generics In Standard Collections	Enable support for the generics syntax in all standard collections currently available in the typing module	3.9	No
PEP 586	Literal Types	Literal types indicate that some expression has literally a specific value(s).	3.8	No
PEP 589	TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys	Support dictionary object with a specific set of string keys, each with a value of a specific type	3.8	No
PEP 593	Flexible function and variable annotations	Adds an Annotated type to the typing module to decorate existing types with context-specific metadata.	3.9	No
PEP 604	Allow writing union types as X | Y	Overload the | operator on types to allow writing Union[X, Y] as X | Y	3.10	No
PEP 612	Parameter Specification Variables	Proposes typing.ParamSpec and typing.Concatenate to support forwarding parameter types of one callable over to another callable	3.10	No
PEP 613	Explicit Type Aliases	Formalizes a way to explicitly declare an assignment as a type alias	3.10	No
PEP 646	Variadic Generics	Introduce TypeVarTuple, enabling parameterisation with an arbitrary number of types	3.11	No
PEP 647	User-Defined Type Guards	Specifies a way for programs to influence conditional type narrowing employed by a type checker based on runtime checks	3.11	No
PEP 655	Marking individual TypedDict items as required or potentially-missing	Two new notations: Required[], which can be used on individual items of a TypedDict to mark them as required, and NotRequired[]	3.11	No
PEP 673	Self Type	Methods that return an instance of their class	3.10	No

PEP 675	Arbitrary Literal String Type	Introduces supertype of literal string types: LiteralString	3.11	No
PEP 681	Data Class Transforms	Provides a way for third-party libraries to indicate that certain decorator functions, classes, and metaclasses provide behaviors similar to dataclasses	3.11	No
PEP 692	Using TypedDict for more precise kwargs typing	A new syntax for specifying kwargs type as a TypedDict without breaking current behavior	3.12	No
PEP 695	Type Parameter Syntax	A syntax for specifying type parameters within a generic class, function, or type alias. And introduces a new statement for declaring type aliases.	3.12	No
PEP 698	Override Decorator for Static Typing	Adds @override decorator to allow type checkers to prevent a class of bugs that occur when a base class changes methods that are inherited by derived classes.	3.12	No

Underscores in Numeric Literals

Underscores in Numeric Literals are supported, even encouraged, but not required. For consistency, follow the grouping examples presented in the PEP abstract.

PEP versions supported by CHIPSEC

PEP	Title	Summary	Python Version	Supported
PEP 515	Underscores in Numeric Literals	Extends Python's syntax so that underscores can be used as visual separators for grouping purposes in numerical literals	3.6	Yes

Walrus Operator (:=)

At this time, Assignment Expressions (Walrus operator) are not supported.

PEP versions supported by CHIPSEC

PEP	Title	Summary	Python Version	Supported
PEP 572	Assignment Expressions	Adds a way to assign to variables within an expression	3.8	No

Deprecate distutils module support

Python 3.12 will deprecate and remove the distutils module. In order for CHIPSEC to support this and furture versions of Python, setuptools should be used instead of distutils.

The setuptools module has been updated to fully replace distutils but requires an up-to-date version.

- Minimum setuptools version: 62.0.0 (requires Python >= 3.7)
- Recommended setuptools version: latest

Note: If you get any *setuptools.command.build* errors, verify that you have (at least) the minimum setuptools version.

PEP versions supported by CHIPSEC

PEP / bpo	Title	Summary	Python Version	Supported
PEP 632	Deprecate distutils module	Mark the distutils module as deprecated (3.10) and then remove it (3.12)	3.12	Yes