

# CHIPSEC

version 1.6.2



**Platform Security Assessment Framework**

**May 07, 2021**

# Contents

<b>CHIPSEC</b>	<b>1</b>
Installation	1
Windows Installation	1
Install CHIPSEC Dependencies	1
Building	2
Turn off kernel driver signature checks	2
Alternate Build Methods	3
DAL Windows Installation	4
Prerequisites	4
Building	4
Linux Installation	4
Prerequisites	5
Building	5
Creating a Live Linux image	5
MacOS Installation	6
Install CHIPSEC Dependencies	6
Building Chipsec	6
Run CHIPSEC	6
CHIPSEC Cleanup	6
Build Errors	7
Building a Bootable USB drive with UEFI Shell (x64)	7
Installing CHIPSEC	7
Run CHIPSEC in UEFI Shell	7
Creating the Kali Linux Live USB	8
Installing CHIPSEC	8
Run CHIPSEC	8
Using CHIPSEC	8
Running CHIPSEC	8
Running in Shell	9
Using as a Python Package	9
chipsec_main options	10
chipsec_util options	10
Interpreting results	10
Results	11
Automated Tests	11
Tools	14
Module & Command Development	14

Architecture Overview	14
Core components	14
Modules & Tools	15
Platform detection	15
Methods for Platform Detection	15
Uses PCI VID and DID to detect processor and PCH	15
Chip information located in <code>chipsec/chipset.py</code>	15
Platform Configuration Options	15
Configuration Files	16
Configuration Files	16
Configuraiton File Example	16
Commands	23
utilcmd package	23
acpi_cmd module	23
chipset_cmd module	23
cmos_cmd module	24
config_cmd module	24
cpu_cmd module	25
decode_cmd module	25
deltas_cmd module	26
desc_cmd module	26
ec_cmd module	27
igd_cmd module	27
interrupts_cmd module	28
io_cmd module	28
iommu_cmd module	29
mem_cmd module	30
mmcfg_base_cmd module	30
mmcfg_cmd module	31
mmio_cmd module	31
msgbus_cmd module	31
msr_cmd module	32
pci_cmd module	32
reg_cmd module	33
smbios_cmd module	34
smbus_cmd module	34
spd_cmd module	34
spi_cmd module	35
spidesc_cmd module	36

tpm_cmd module	36
ucode_cmd module	37
uefi_cmd module	37
vmem_cmd module	38
vmm_cmd module	39
HAL (Hardware Abstraction Layer)	40
hal package	40
acpi module	40
acpi_tables module	40
cmos module	44
cpu module	45
cpuid module	46
ec module	46
hal_base module	47
igd module	47
interrupts module	48
io module	48
iobar module	49
iommu module	49
mmio module	50
msgbus module	51
msr module	53
paging module	54
pci module	56
pcidb module	57
physmem module	57
smbios module	58
smbus module	59
spd module	59
spi module	60
spi_descriptor module	62
spi_jedec_ids module	62
spi_uefi module	62
tpm module	64
tpm12_commands module	64
tpm_eventlog module	65
ucode module	65
uefi module	66
uefi_common module	67

uefi_fv module	72
uefi_platform module	73
uefi_search module	75
virtmem module	75
vmm module	76
OS Helpers & Drivers	77
OS Helpers and Drivers	77
Mostly invoked by HAL modules	77
Helpers import from BaseHelper	77
Create a New Helper	77
Example	77
helper package	78
dal package	78
dalhelper module	78
efi package	80
efihelper module	80
file package	81
filehelper module	81
linux package	83
cpuid module	83
legacy_pci module	84
linuxhelper module	84
osx package	88
osxhelper module	88
rwe package	90
rwehelper module	90
win package	92
win32helper module	92
basehelper module	95
helpers module	97
oshelper module	97
Fuzzing	99
fuzzing package	99
primitives module	99
CHIPSEC_MAIN Program Flow	101
CHIPSEC_UTIL Program Flow	101
Auxiliary components	102
Executable build scripts	102
CHIPSEC Modules	102

Introduction	102
Modules	107
modules package	107
bdw package	107
byt package	107
common package	107
cpu package	107
cpu_info module	107
ia_untrusted module	107
spectre_v2 module	108
secureboot package	109
variables module	109
uefi package	110
access_ufispec module	110
s3bootscript module	110
bios_kbrd_buffer module	111
bios_smi module	111
bios_ts module	112
bios_wp module	112
debugenabled module	112
ia32cfg module	113
me_mfg_mode module	113
memconfig module	114
memlock module	115
remap module	115
rtclock module	115
sgx_check module	116
smm module	116
smm_code_chk module	116
smm_dma module	117
smrr module	117
spd_wd module	118
spi_access module	118
spi_desc module	118
spi_fdopss module	119
spi_lock module	119
hsw package	120
ivb package	120
snb package	120

tools package	120
cpu package	120
sinkhole module	120
secureboot package	120
te module	120
smm package	122
rogue_mmio_bar module	122
smm_ptr module	122
uefi package	124
reputation module	124
s3script_modify module	124
scan_blocked module	126
scan_image module	127
uefivar_fuzz module	127
vmm package	128
hv package	128
define module	128
hypercall module	129
hypercallfuzz module	129
synth_dev module	130
synth_kbd module	130
vmbus module	131
vmbusfuzz module	133
vbox package	133
vbox_crash_apicbase module	133
xen package	134
define module	134
hypercall module	134
hypercallfuzz module	134
xsa188 module	135
common module	135
cpuid_fuzz module	136
ept_finder module	137
hypercallfuzz module	137
iofuzz module	138
msr_fuzz module	138
pcie_fuzz module	138
pcie_overlap_fuzz module	139
venom module	139





# 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.

## 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.

### Windows Installation

CHIPSEC supports the following versions:

Windows 7, 8, 8.1, 10 x86 and AMD64

Windows Server 2008, 2012, 2016 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

[pywin32](#): for Windows API support

```
pip install pywin32
```

[setuptools](#)

```
pip install setuptools
```

[WConio](#): Optional. For colored console output

[Visual Studio and WDK](#): for building the driver

[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

### Windows 10 64-bit

In CMD shell:

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

**Windows 10 64-bit / Windows 8, 8.1 64-bit (with Secure Boot enabled) / Windows Server 2016 64-bit / Windows Server 2012 64-bit (with Secure Boot enabled):**

Method 1: - In CMD shell:

- `shutdown /r /t 0 /o`
- Navigate: Troubleshooting > Advanced Settings > Startup Settings > Reboot
- After reset choose F7 "Disable driver signature checks"

Method 2:

- Disable Secure Boot in the BIOS setup screen then disable driver signature checks as in Windows 8 with Secure Boot disabled

**Windows 7 64-bit / Windows Server 2008 64-bit / Windows 8 (with Secure Boot disabled) / Windows Server 2012 (with Secure Boot disabled):**

Method 1:

- **Boot in Test mode (allows self-signed certificates)**

- Start CMD.EXE as Administrator `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 at all

**Alternate Build Methods****Build CHIPSEC kernel driver with Visual Studio**

- Open the Visual Studio project file (drivers/win7/chipsec\_hlpr.vcxproj) using Visual Studio
- Select Platform and configuration (X86 or x64, Release)
- Go to Build -> Build Solution

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

```
<CHIPSEC_ROOT_DIR>\chipsec\helper\win\win7_amd64 (or i386)
```

**Build CHIPSEC kernel driver (Old Method)**

Install WDK 7600 - (e.g. [WDK 7600.16385.1](#)) Open WDK build environment command prompt:

```
"x86 Free Build Environment" or "x64 Free Build Environment" (for release build)
"x86 Checked Build Environment" or "x64 Checked Build Environment" (for debug build)
cd <CHIPSEC_ROOT_DIR>\drivers\win7 build -cZg
```

If build process completed without errors, the driver binary ("chipsec\_hlpr.sys") will be in:

```
<CHIPSEC_ROOT_DIR>\drivers\win7\sys\amd64
```

Sign the driver

- As Administrator, run in "x64 Free Build Environment" (or "x64 Checked Build Environment"):

```
makecert -r -n "CN=Chipsec" -ss ChipsecCertStore -sr LocalMachine
cd <CHIPSEC_ROOT_DIR>\drivers\win7\sign
```

- Run "sign64\_sys.bat" to sign "chipsec\_hlpr.sys" file
- If any error/warning is returned, create a new certificate store and modify "sign64\_sys.bat" accordingly

Copy the driver to CHIPSEC framework

- On Windows x64, copy signed "chipsec\_hlpr.sys" to <CHIPSEC\_ROOT\_DIR>chipsechelperwinwin7\_amd64 directory
- On Windows x86, copy "chipsec\_hlpr.sys" to <CHIPSEC\_ROOT\_DIR>chipsechelperwinwin7\_x86 directory

Build the compression tools

Method 1:

## CHIPSEC

- Navigate to the chipsec\_toolscompression directory
- run the build.cmd

Method 2:

- Download compression tools from <https://github.com/tianocore/edk2-BaseTools-win32/archive/master.zip>
- Unzip the archive into the chipsec\_tools/compression/bin directory

### Alternate Method to load CHIPSEC service/driver

To create and start CHIPSEC service

```
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
```

## 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
```

## Linux Installation

Tested on:

## CHIPSEC

- 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.

### 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) python python-devel gcc nasm redhat-rpm-c
onfig elfutils-libelf-devel git
```

or

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

or

```
pacman -S python2 python2-setuptools nasm linux-headers
```

Install setuptools package:

```
pip install setuptools
```

### Building

Clone CHIPSEC source

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

Build the Driver and Compression Tools

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

### Creating a Live Linux image

1. Download things you will need:

- Desired Linux image (e.g. Fedora LXDE 64bit)
- [liveusb-creator](#)

2. Use liveusb-creator to image a USB stick with the desired Linux image. Include as much persistent storage as possible.

### 3. Reboot to USB

## MacOS Installation

### Warning

MacOS support is currently in Beta release. There's no support for M1 chips

## Install CHIPSEC Dependencies

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

### Note

CHIPSEC has deprecated support for Python2 since June 2020

Install XCODE from the App Store (for best results use version 11 or newer)

Install PIP and setuptools packages. Please see instructions [here](#)

Turn the System Integrity Protection (SIP) off. See [Configuring SIP](#)

An alternative to disabling SIP and allowing untrusted/unsigned kexts to load can be enabled by running the following command:

```
# csrutil enable --without kext
```

## Building Chipsec

Clone CHIPSEC Git repository:

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

## Run CHIPSEC

Follow steps in section "Using as a Python package" of [Running CHIPSEC](#)

To build chipsec.kext on your own and load please follow the instructions in drivers/osx/README

## CHIPSEC Cleanup

When done using CHIPSEC, ensure the driver is unloaded and re-enable the System Integrity Protection:

```
# kextunload -b com.google.chipsec
# csrutil enable
```

## Build Errors

xcodebuild requires xcode error during CHIPSEC install:

```
# sudo xcode-select -s /Applications/Xcode.app/Contents/Developer
```

## 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\_uefi\_[x64|i586|IA32].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 follow:

```
- fs0:
- efi
- boot
- bootx64.efi
- StdLib
- lib
- python.27
- [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
```

Next follow steps in section “Basic Usage” of [Running CHIPSEC](#)

### Creating the Kali Linux Live USB

[Download](#) and [install](#) Kali Linux

### Installing CHIPSEC

#### Install the dependencies

```
apt-get install python python-devel gcc nasm linux-headers-[version]-all
```

#### Note

Install the linux headers for the currently running version of the Linux kernel. You can determine this with `uname -r`

```
pip install setuptools
```

#### Install latest CHIPSEC release from PyPI repository

```
pip install chipsec
```

#### Install CHIPSEC package from latest source code

Copy CHIPSEC to the USB drive (or install `git`)

```
git clone https://github.com/chipsec/chipsec
python setup.py install
```

### Run CHIPSEC

Follow steps in section “Using as a Python package” of [Running CHIPSEC](#)

## 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.

### 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.



## Using CHIPSEC

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.main()
>>> chipsec_main.main(['-m', 'common.bios_wp'])
```

```
>>> import chipsec_util
>>> chipsec_util.main()
>>> chipsec_util.main(['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)
-a, --module_args _MODULE_ARGV	additional module arguments
-v, --verbose	verbose mode
-vv, --vverbose	very verbose HAL debug mode
--hal	HAL mode
-d, --debug	debug mode
-l, --log LOG	output to log file

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
-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
--record _TO_FILE	run chipsec and clone helper results into JSON file
--replay _FROM_FILE	replay a chipsec run with JSON file
--helper _HELPER	specify OS Helper
-nb, --no_banner	chipsec won't display banner information
--skip_config	skip configuration and driver loading

## chipsec\_util options

usage: chipsec\_util.py [options] <command>

Options:

-h, --help	show this message and exit
-v, --verbose	verbose mode
--hal	HAL mode
-d, --debug	debug mode
-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
Command _CMD	Util command to run
Command _ARGS	All numeric values are in hex <width> is in {1 - byte, 2 - word, 4 - dword}

## 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

Currently, the SKIPPED return value is ambiguous. The proposed **new** definition of the return values is listed below:

### Generic results meanings

Result	Meaning
PASSED	A <b>mitigation</b> to a known vulnerability has been detected
FAILED	A known <b>vulnerability</b> has been detected
WARNING	We have detected something that could be a vulnerability but <b>manual analysis is required</b> to confirm (inconclusive)
SKIPPED NOT IMPLEMENTED	CHIPSEC currently has not implemented support for this test on this platform
SKIPPED 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
DEPRECATED	At least one module uses deprecated API
EXCEPTION	At least one module threw an unexpected exception

## 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. Remapping 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. The contents might have not been cleared by pre-boot firmware or overwritten with garbage. 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 SKIP PED_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	

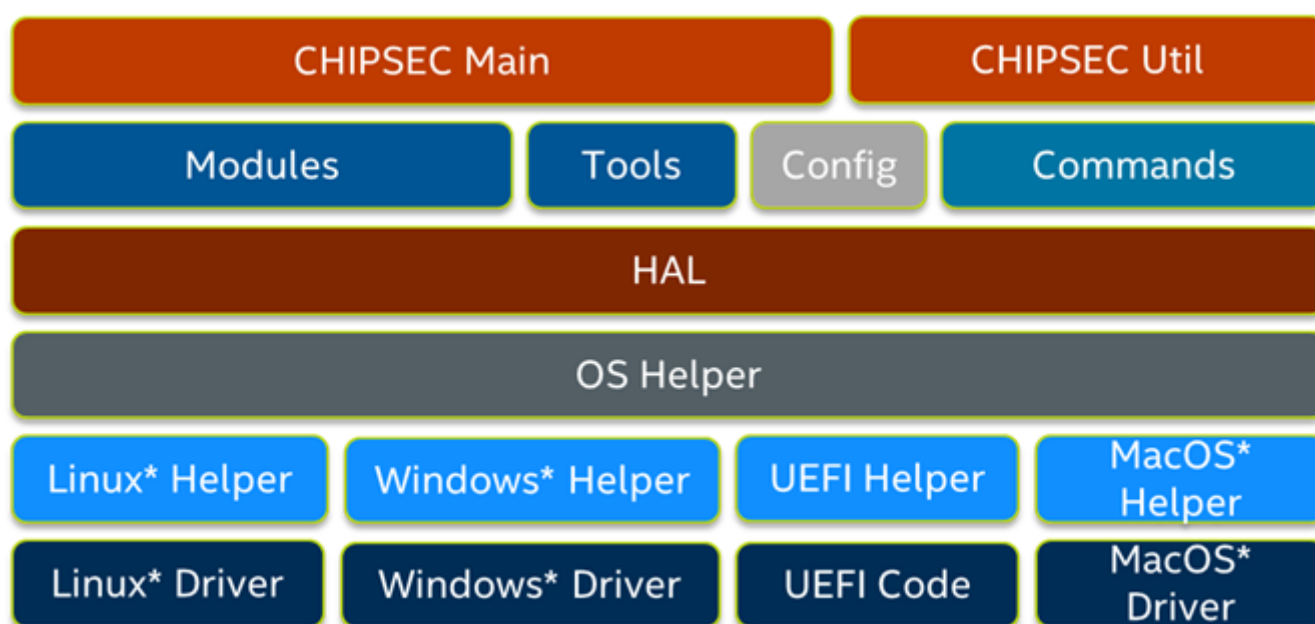
common.uefi.s3boot script	N/A	S3 Boot-Script and Dispatch entry-points do not appear to be protected	S3 Boot-Script is not in SMRAM but Dispatch entry-points appear to be protected. Recommend further testing	unfortunately, if the boot script is well protected (in SMRAM) we cannot find it at all and end 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.

## 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

### Modules & Tools

Implementation of tests or other functionality for chipsec\_main

### Platform detection

#### Methods for Platform Detection

#### Uses PCI VID and DID to detect processor and PCH

Processor 0:0.0

PCH 0:31.0

#### Chip information located in chipsec/chipset.py

Currently requires VID of 0x8086

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

## 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>

## Configuration Files

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>
```

## apl

Path: chipsec\cfg\8086\apl.xml

XML configuration for Apollo Lake based SoCs document id 334818/334819

## avn

Path: chipsec\cfg\8086\avn.xml

XML configuration for Avoton based platforms

- Intel(R) Atom(TM) Processor C2000 Product Family for Microserver, September 2014  
<http://www.intel.com/content/www/us/en/processors/atom/atom-c2000-microserver-datasheet.html>

## bdw

Path: chipsec\cfg\8086\bdw.xml

XML configuration for Broadwell based platforms



#### *bdx*

Path: chipsec\cfg\8086\bdx.xml

XML configuration file for Broadwell Server based platforms Intel (c) Xeon Processor E5 v4 Product Family datasheet Vol. 2 Intel (c) Xeon Processor E7 v4 Product Family datasheet Vol. 2 Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset datasheet Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset Specification Update Intel (c) C610 Series Chipset and Intel (c) X99 Chipset Platform Controller Hub (PCH) datasheet

#### *byt*

Path: chipsec\cfg\8086\byt.xml

XML configuration for Bay Trail based platforms

- Intel(R) Atom(TM) Processor E3800 Product Family Datasheet, May 2016, Revision 4.0  
<http://www.intel.com/content/www/us/en/embedded/products/bay-trail/atom-e3800-family-datasheet.html>

#### *cfl*

Path: chipsec\cfg\8086\cfl.xml

XML configuration file for Coffee Lake

- 8th Generation Intel(R) Processor Family for S-Processor Platforms  
<https://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html>

#### *cht*

Path: chipsec\cfg\8086\cht.xml

XML configuration for Cherry Trail and Braswell SoCs

- Intel(R) Atom(TM) Processor Z8000 series datasheet  
<http://www.intel.com/content/www/us/en/processors/atom/atom-z8000-datasheet-vol-2.html>
- N-series Intel(R) Pentium(R) and Celeron(R) Processors Datasheet  
<http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/pentium-celeron-n-series-datasheet-vol-2.pdf>

#### *cml*

Path: chipsec\cfg\8086\cml.xml

XML configuration file for Comet Lake

#### *common*

Path: chipsec\cfg\8086\common.xml

Common (default) XML platform configuration file

#### *dnv*

Path: chipsec\cfg\8086\dnv.xml

XML configuration file for Denverton

- Intel Atom(R) Processor C3000 Product Family  
<https://www.intel.com/content/www/us/en/processors/atom/atom-technical-resources.html> 337018-002

#### *glk*

Path: chipsec\cfg\8086\glk.xml

#### **XML configuration for GLK**

Document ID: 336561-001

#### *hsw*

Path: chipsec\cfg\8086\hsw.xml

XML configuration file for Haswell based platforms

#### *hsx*

Path: chipsec\cfg\8086\hsx.xml

XML configuration file for Haswell Server based platforms Intel (c) Xeon Processor E5-1600/2400/2600/4600 v3 Product Family datasheet Vol. 2 Intel (c) Xeon Processor E7-8800/4800 v3 Product Family datasheet Vol. 2 Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset datasheet Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset Specification Update Intel (c) C610 Series Chipset and Intel (c) X99 Chipset Platform Controller Hub (PCH) datasheet

#### *icl*

Path: chipsec\cfg\8086\icl.xml

XML configuration file for Ice Lake

#### *iommu*

Path: chipsec\cfg\8086\iommu.xml

XML configuration file for Intel Virtualization Technology for Directed I/O (VT-d)

- Section 10 of Intel Virtualization Technology for Directed I/O  
<http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf>

#### *ivb*

Path: chipsec\cfg\8086\ivb.xml

XML configuration for IvyBridge based platforms

#### *ivt*

Path: chipsec\cfg\8086\ivt.xml

XML configuration file for Ivytown (Ivy Bridge-E) based platforms

#### *jkt*

Path: chipsec\cfg\8086\jkt.xml

XML configuration file for Jaketown (Sandy Bridge-E) based platforms

#### *kbl*

Path: chipsec\cfg\8086\kbl.xml

XML configuration file for Kaby Lake based platforms

<http://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html>

- 7th Generation Intel(R) Processor Families for U/Y-Platforms
- 7th Generation Intel(R) Processor Families I/O for U/Y-Platforms

#### *pch\_1xx*

Path: chipsec\cfg\8086\pch\_1xx.xml

XML configuration file for 100 series PCH based platforms

- Intel(R) 100 Series Chipset Family Platform Controller Hub (PCH)  
<http://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html>

#### *pch\_2xx*

Path: chipsec\cfg\8086\pch\_2xx.xml

XML configuration file for 200 series PCH based platforms

- Intel(R) 200 Series Chipset Family Platform Controller Hub (PCH)  
<http://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html>

#### *pch\_3xx*

Path: chipsec\cfg\8086\pch\_3xx.xml

XML configuration file for the 300 series PCH

<https://www.intel.com/content/www/us/en/products/docs/chipsets/300-series-chipset-pch-datasheet-vol-2.html>  
337348-001

#### *pch\_3xxlp*

Path: chipsec\cfg\8086\pch\_3xxlp.xml

XML configuration file for the 300 series LP (U/Y) PCH <https://www.intel.com/content/www/us/en/products/docs/processors/core/7th-and-8th-gen-core-family-mobile-u-y-processor-lines-i-o-datasheet-vol-2.html> 334659-005

#### *pch\_3xxop*

Path: chipsec\cfg\8086\pch\_3xxop.xml

XML configuration file for the 300 series On Package PCH <https://www.intel.com/content/www/us/en/products/docs/chipsets/300-series-chipset-on-package-pch-datasheet-vol-2.html> 337868-002

#### *pch\_495*

Path: chipsec\cfg\8086\pch\_495.xml

XML configuration file for the 495 series PCH

#### *pch\_4xx*

Path: chipsec\cfg\8086\pch\_4xx.xml

XML configuration file for 4XX pch

#### *pch\_4xxh*

Path: chipsec\cfg\8086\pch\_4xxh.xml

XML configuration file 4xxH PCH 620855

#### *pch\_4xxlp*

Path: chipsec\cfg\8086\pch\_4xxlp.xml

XML configuration file for the 400 series LP (U/H) PCH

#### *pch\_5xxh*

Path: chipsec\cfg\8086\pch\_5xxh.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2021, Intel Corporation

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; Version 2.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Contact information: [chipsec@intel.com](mailto:chipsec@intel.com)

#### *pch\_5xxlp*

Path: chipsec\cfg\8086\pch\_5xxlp.xml

XML configuration file for 5XX series pch

#### *pch\_c60x*

Path: chipsec\cfg\8086\pch\_c60x.xml

##### **XML configuration file for C600 series PCH**

Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset datasheet Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset Specification Update  
<https://ark.intel.com/products/series/98463/Intel-C600-Series-Chipsets>

#### *pch\_c61x*

Path: chipsec\cfg\8086\pch\_c61x.xml

##### **XML configuration file for C610 series PCH**

Intel (c) C610 Series Chipset and Intel (c) X99 Chipset Platform Controller Hub (PCH) datasheet  
<https://ark.intel.com/products/series/98915/Intel-C610-Series-Chipsets>

#### *pch\_c620*

Path: chipsec\cfg\8086\pch\_c620.xml

XML configuration file for

- Intel(R) C620 Series Chipset Family Platform Controller Hub  
<https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/c620-series-chipset-datasheet.pdf>

#### *qrk*

Path: chipsec\cfg\8086\qrk.xml

XML configuration for Quark based platforms

#### *rkl*

Path: chipsec\cfg\8086\rkl.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2021, Intel Corporation

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; Version 2.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Contact information: [chipsec@intel.com](mailto:chipsec@intel.com)

#### *sfdp*

Path: chipsec\cfg\8086\sfdp.xml

XML configuration for Serial Flash Discoverable Parameter feature document:  
<https://www.jedec.org/system/files/docs/JESD216D-01.pdf>

#### *skl*

Path: chipsec\cfg\8086\skl.xml

XML configuration file for Skylake based platforms

<http://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html>

- 6th Generation Intel(R) Processor Datasheet for U/Y-Platforms
- 6th Generation Intel(R) Processor I/O Datasheet for U/Y-Platforms
- 6th Generation Intel(R) Processor Datasheet for S-Platforms
- 6th Generation Intel(R) Processor Datasheet for H-Platforms
- Intel(R) 100 Series Chipset Family Platform Controller Hub (PCH)

#### *skx*

Path: chipsec\cfg\8086\skx.xml

XML configuration file for Skylake/Purely Server Intel (c) Xeon Processor Scalable Family datasheet Vol. 2

#### *snb*

Path: chipsec\cfg\8086\snb.xml

XML configuration for Sandy Bridge based platforms

#### *tglu*

Path: chipsec\cfg\8086\tglu.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2021, Intel Corporation

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; Version 2.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Contact information: [chipsec@intel.com](mailto:chipsec@intel.com)

#### *whl*

Path: chipsec\cfg\8086\whl.xml

XML configuration file for Whiskey Lake

6th Generation Intel(R) Processor Family for U-Processor Platforms

<https://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html>

<https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/300-series-chipset-on-package-pch-datasheet-v>

## 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.

#### utilcmd package

#### acpi\_cmd module

Command-line utility providing access to ACPI tables

```
class ACPICommand (argv, cs=None)
```

Bases: `chipsec.command.BaseCommand`

```
>>> 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
```

`acpi_list ()`

`acpi_table ()`

`requires_driver ()`

`run ()`

#### chipset\_cmd module

usage as a standalone utility:

```
>>> chipsec_util platform
```

```
class PlatformCommand (argv, cs=None)
```

Bases: `chipsec.command.BaseCommand`

`chipsec_util platform`

`requires_driver ()`

`run ()`

#### *cmos\_cmd module*

`class CMOSCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

```
>>> 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
```

`cmos_dump ()`

`cmos_readh ()`

`cmos_readl ()`

`cmos_writeh ()`

`cmos_writel ()`

`requires_driver ()`

`run ()`

#### *config\_cmd module*

`class CONFIGCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

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

Examples:

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

`bus_details (regi)`

`control_details (regi)`

`io_details (regi)`

`lock_details (regi)`

`mem_details (regi)`

`mmio_details (regi)`

`pci_details (regi)`



```
register_details (regi)

requires_driver ()

run ()

show ()
```

#### *cpu\_cmd module*

```
class CPUCCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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
```

```
cpu_cpuid ()
```

```
cpu_cr ()
```

```
cpu_info ()
```

```
cpu_pt ()
```

```
cpu_topology ()
```

```
requires_driver ()
```

```
run ()
```

#### *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.

Examples:

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

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

```
class DecodeCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

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

For a list of fw types run:

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

Examples:

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

**decode\_rom ()**

**decode\_types ()**

**requires\_driver ()**

**run ()**

### *deltas\_cmd module*

**class DeltasCommand (argv, cs=None)**

Bases: **chipsec.command.BaseCommand**

```
>>> 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

**requires\_driver ()**

**run ()**

### *desc\_cmd module*

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

**class GDTCommand (argv, cs=None)**

Bases: **chipsec.command.BaseCommand**

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

Examples:

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

```
>>> chipsec_util gdt
```

**requires\_driver ()**

**run ()**

**class IDTCommand (argv, cs=None)**

Bases: **chipsec.command.BaseCommand**

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

Examples:

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

```
>>> chipsec_util idt
```

**requires\_driver ()**

**run ()**

**class LDTCommand (argv, cs=None)**

**Bases: chipsec.command.BaseCommand**

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

**Examples:**

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

**requires\_driver ()**

**run ()**

#### *ec\_cmd module*

**class ECCCommand (argv, cs=None)**

**Bases: chipsec.command.BaseCommand**

```
>>> 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
```

**command ()**

**dump ()**

**index ()**

**read ()**

**requires\_driver ()**

**run ()**

**write ()**

#### *igd\_cmd module*

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

**class IgdCommand (argv, cs=None)**

**Bases: chipsec.command.BaseCommand**

```
>>> 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
```

`read_dma ()`

`requires_driver ()`

`run ()`

`write_dma ()`

### *interrupts cmd module*

`class NMICCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

`requires_driver ()`

`run ()`

`class SMICCommand (argv, cs=None)`

Bases: `chipsec.command.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 0x79efdfdf ed32d533-99e6-4209-9cc02d72cdd998a7 0x79dfaaaa payload.bin
```

`requires_driver ()`

`run ()`

`smi_count ()`

`smi_send ()`

`smi_smmc ()`

### *io cmd module*

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

`class PortIOCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

```
>>> 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
```

`io_list()`

`io_read()`

`io_write()`

`requires_driver()`

`run()`

### *iommu\_cmd module*

Command-line utility providing access to IOMMU engines

`class IOMMUCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

```
>>> 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
```

`iommu_config()`

`iommu_disable()`

`iommu_enable()`

`iommu_engine(cmd)`

`iommu_list()`

`iommu_pt()`

`iommu_status()`

`requires_driver()`

`run()`

**mem cmd module**

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

```
class MemCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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_
```

```
dump_region_to_path (path, pa_start, pa_end)
```

```
mem_allocate ()
```

```
mem_pagedump ()
```

```
mem_read ()
```

```
mem_readval ()
```

```
mem_search ()
```

```
mem_write ()
```

```
mem_writeval ()
```

```
requires_driver ()
```

```
run ()
```

**mmcfg\_base cmd module**

The mmcfg\_base command displays PCIe MMCFG Base/Size.

```
class MMCfgBaseCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> chipsec_util mmcfg_base
```

Examples:

```
>>> chipsec_util mmcfg_base
```

```
requires_driver ()
```

```
run ()
```

#### *mmcfg\_cmd module*

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

```
class MMCfgCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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
```

```
requires_driver ()
```

```
run ()
```

#### *mmio\_cmd module*

```
class MMIOCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

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

Examples:

```
>>> chipsec_util mmio list
>>> chipsec_util mmio dump MCHBAR
>>> chipsec_util mmio read SPIBAR 0x74 0x4
>>> chipsec_util mmio write SPIBAR 0x74 0x4 0xFFFF0000
```

```
dump_bar ()
```

```
listBars ()
```

```
read_bar ()
```

```
requires_driver ()
```

```
run ()
```

```
write_bar ()
```

#### *msgbus\_cmd module*

```
class MsgBusCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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
```

**msgbus\_message ()**

**msgbus\_mm\_read ()**

**msgbus\_mm\_write ()**

**msgbus\_read ()**

**msgbus\_write ()**

**requires\_driver ()**

**run ()**

### *msr\_cmd module*

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

**class MSRCommand (argv, cs=None)**

Bases: **chipsec.command.BaseCommand**

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

### Examples:

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

**requires\_driver ()**

**run ()**

### *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.

**class PCICCommand (argv, cs=None)**

Bases: **chipsec.command.BaseCommand**



```
>>> 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
```

`pci_cmd()`

`pci_dump()`

`pci_enumerate()`

`pci_read()`

`pci_write()`

`pci_xrom()`

`requires_driver()`

`run()`

### *reg\_cmd module*

`class RegisterCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

```
>>> 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
```

`reg_get_control()`

`reg_read()`

```
reg_read_field ()  
reg_set_control ()  
reg_write ()  
reg_write_field ()  
requires_driver ()  
run ()
```

#### *smbios\_cmd module*

```
class smbios_cmd (argv, cs=None)  
Bases: chipsec.command.BaseCommand
```

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

Examples:

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

```
requires_driver ()  
run ()  
smbios_ep ()  
smbios_get ()
```

#### *smbus\_cmd module*

```
class SMBusCommand (argv, cs=None)  
Bases: chipsec.command.BaseCommand
```

```
>>> 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
```

```
requires_driver ()  
run ()  
smbus_read ()  
smbus_write ()
```

#### *spd\_cmd module*

```
class SPDCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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
```

```
requires_driver ()
```

```
run ()
```

```
spd_detect ()
```

```
spd_dump ()
```

```
spd_read ()
```

```
spd_write ()
```

#### *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.

```
class SPICCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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
```

```
requires_driver ()

run ()

spi_disable_wp ()

spi_dump ()

spi_erase ()

spi_info ()

spi_jedec ()

spi_read ()

spi_sfdp ()

spi_write ()
```

#### spidesc\_cmd module

```
class SPIDescCommand (argv, cs=None)
Bases: chipsec.command.BaseCommand
```

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

Examples:

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

```
requires_driver ()

run ()
```

#### tpm\_cmd module

```
class TPMCommand (argv, cs=None)
Bases: chipsec.command.BaseCommand
```

```
>>> 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 pccrread 0 17
>>> chipsec_util tpm command continueselftest 0
```

```
no_driver_cmd = ['parse_log']
```

```

requires_driver ()

run ()

tpm_command ()

tpm_parse ()

tpm_state ()

```

#### ucode\_cmd module

```
class UCodeCommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```
>>> 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

```

```

requires_driver ()

run ()

ucode_decode ()

ucode_id ()

ucode_load ()

```

#### uefi\_cmd module

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

```
class UEFICommand (argv, cs=None)
```

```
Bases: chipsec.command.BaseCommand
```

```

>>> 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 --fwtype <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-EEEEEEEEEEEEEE freeform lzma uefi.raw mydriver.efi
>>> chipsec_util uefi replace AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEEEE bios.bin new_bios.bin mydriver.efi
```

`assemble ()`

`decode ()`

`insert_after ()`

`insert_before ()`

`keys ()`

`nvram ()`

`nvram_auth ()`

`remove ()`

`replace ()`

`requires_driver ()`

`run ()`

`s3bootscript ()`

`tables ()`

`var_delete ()`

`var_find ()`

`var_list ()`

`var_read ()`

`var_write ()`

#### *vmem\_cmd module*

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

`class VMemCommand (argv, cs=None)`

Bases: `chipsec.command.BaseCommand`

```
>>> 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
```

**requires\_driver ()****run ()****vmem\_allocate ()****vmem\_getphys ()****vmem\_read ()****vmem\_readval ()****vmem\_search ()****vmem\_write ()****vmem\_writeval ()****vmm\_cmd module****class VMMCommand (argv, cs=None)****Bases: chipsec.command.BaseCommand**

```
>>> 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
```

**requires\_driver ()****run ()****vmm\_hypercall ()****vmm\_pt ()****vmm\_virtio ()**

## HAL (Hardware Abstraction Layer)

Useful abstractions for common tasks such as accessing the SPI

### hal package

#### acpi module

HAL component providing access to and decoding of ACPI tables

```
class ACPI (cs)
    Bases: chipsec.hal.hal_base.HALBase

    dump_ACPI_table (name, isfile=False)

    find_RSDP ()

    get_ACPI_table (name, isfile=False)

    get_ACPI_table_list ()

    get_DSDT_from_FADT ()

    get_SDT (search_rsdp=True)

    get_parse_ACPI_table (name, isfile=False)

    get_table_list_from_SDT (sdt, is_xsdt)

    is_ACPI_table_present (name)

    print_ACPI_table_list ()

    read_RSDP (rsdp_pa)

class ACPI_TABLE_HEADER (Signature, Length, Revision, Checksum, OEMID, OEMTableID, OEMRevision,
    CreatorID, CreatorRevision)
    Bases: chipsec.hal.acpi.ACPI_TABLE_HEADER

exception AcpiRuntimeError
    Bases: RuntimeError

class RSDP (table_content)
    Bases: object

    is_RSDP_valid ()
```

#### acpi\_tables module

HAL component decoding various ACPI tables

```
class ACPI_TABLE
    Bases: object
```



```

parse (table_content)

class ACPI_TABLE_APIC_GICC_CPU (Type, Length, Reserved, CPUIntNumber, ACPIProcUID, Flags,
ParkingProtocolVersion, PerformanceInterruptGSIV, ParkedAddress, PhysicalAddress, GICV, GICH,
VGICMaintenanceInterrupt, GICRBaseAddress, MPIDR)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GICC_CPU

class ACPI_TABLE_APIC_GIC_DISTRIBUTOR (Type, Length, Reserved, GICID, PhysicalBaseAddress,
SystemVectorBase, Reserved2)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GIC_DISTRIBUTOR

class ACPI_TABLE_APIC_GIC_MSI (Type, Length, Reserved, GICMSIFrameID, PhysicalBaseAddress, Flags,
SPICount, SPIBase)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GIC_MSI

class ACPI_TABLE_APIC_GIC_REDISTRIBUTOR (Type, Length, Reserved, DiscoverRangeBaseAdd,
DiscoverRangeLength)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GIC_REDISTRIBUTOR

class ACPI_TABLE_APIC_INTERRUPT_SOURCE_OVERRIDE (Type, Length, Bus, Source, GlobalSysIntBase, Flags)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_INTERRUPT_SOURCE_OVERRIDE

class ACPI_TABLE_APIC_IOAPIC (Type, Length, IOAPICID, Reserved, IOAPICAddr, GlobalSysIntBase)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_IOAPIC

class ACPI_TABLE_APIC_IOSAPIC (Type, Length, IOAPICID, Reserved, GlobalSysIntBase, IOSAPICAddress)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_IOSAPIC

class ACPI_TABLE_APIC_LAPIC_ADDRESS_OVERRIDE (Type, Length, Reserved, LocalAPICAddress)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_LAPIC_ADDRESS_OVERRIDE

class ACPI_TABLE_APIC_LAPIC_NMI (Type, Length, ACPIProcessorID, Flags, LocalAPICLINT)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_LAPIC_NMI

class ACPI_TABLE_APIC_Lx2APIC_NMI (Type, Length, Flags, ACPIProcUID, Localx2APICLINT, Reserved)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_Lx2APIC_NMI

class ACPI_TABLE_APIC_NMI_SOURCE (Type, Length, Flags, GlobalSysIntBase)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_NMI_SOURCE

class ACPI_TABLE_APIC_PLATFORM_INTERRUPT_SOURCES (Type, Length, Flags, InterruptType, ProcID,
ProcEID, IOSAPICVector, GlobalSystemInterrupt, PlatIntSourceFlags)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PLATFORM_INTERRUPT_SOURCES

class ACPI_TABLE_APIC_PROCESSOR_LAPIC (Type, Length, ACPIProcID, APICID, Flags)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PROCESSOR_LAPIC

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

class ACPI_TABLE_APIC_PROCESSOR_Lx2APIC (Type, Length, Reserved, x2APICID, Flags, ACPIProcUID)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PROCESSOR_Lx2APIC

class ACPI_TABLE_DMAR_ANDD (Type, Length, Reserved, ACPIDevNum, ACPIObjectName)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_ANDD

class ACPI_TABLE_DMAR_ATSR (Type, Length, Flags, Reserved, SegmentNumber, DeviceScope)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_ATSR

```

```

class ACPI_TABLE_DMAR_DRHD (Type, Length, Flags, Reserved, SegmentNumber, RegisterBaseAddr,
DeviceScope)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_DRHD

class ACPI_TABLE_DMAR_DeviceScope (Type, Length, Reserved, EnumerationID, StartBusNum, Path)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_DeviceScope

class ACPI_TABLE_DMAR_RHSA (Type, Length, Reserved, RegisterBaseAddr, ProximityDomain)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_RHSA

class ACPI_TABLE_DMAR_RMRR (Type, Length, Reserved, SegmentNumber, RMRBaseAddr, RMRLimitAddr,
DeviceScope)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_RMRR

class APIC
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    get_structure_APIC (value, DataStructure)

    parse (table_content)

class BERT (cs)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

    parseErrorBlock (table_content)

    parseGenErrorEntries (table_content)

    parseSectionType (table_content)

    parseTime (table_content)

class BGRT
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

class DMAR
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

class EINJ
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

    parseAddress (table_content)

    parseInjection (table_content)

    parseInjectionActionTable (table_contents, numInjections)

class ERST
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

```

```
    parseActionTable (table_content, instrCountEntry)

    parseAddress (table_content)

    parseInstructionEntry (table_content)

class FADT
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    get_DSDT_address_to_use ()

    parse (table_content)

class GAS (table_content)
    Bases: object

    get_info ()

class HEST
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    machineBankParser (table_content)

    parse (table_content)

    parseAMCES (table_content)

    parseAMCS (table_content, type)

    parseAddress (table_content)

    parseErrEntry (table_content)

    parseGHESS (table_content, type)

    parseNMIStructure (table_content)

    parseNotify (table_content)

    parsePCIE (table_content, type)

class MSCT
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

    parseProx (table_content, val)

    parseProxDomInfoStruct (table_contents, num)

class NFIT (header)
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    flushHintAddrStruct (tableLen, table_content)

    interleave (tableLen, table_content)

    nvdimmblockDataWindowsRegionStruct (tableLen, table_content)
```

```
nvdimmmControlRegionStructMark (tableLen, table_content)

parse (table_content)

parseMAP (tableLen, table_content)

parseSPA (tableLen, table_content)

parseStructures (table_content)

platCapStruct (tableLen, table_content)

smbiosManagementInfo (tableLen, table_content)

class RASF
Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

class RSDT
Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

class SPMI
Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)

    parseAddress (table_content)

    parseNonUID (table_content)

    parseUID (table_content)

class UEFI_TABLE
Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    get_commbuf_info ()

    parse (table_content)

class XSDT
Bases: chipsec.hal.acpi_tables.ACPI_TABLE

    parse (table_content)
```

#### 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: chipsec.hal.hal_base.HALBase  
  
  dump ()  
  
  dump_high ()  
  
  dump_low ()  
  
  read_cmos_high (offset)  
  
  read_cmos_low (offset)  
  
  write_cmos_high (offset, value)  
  
  write_cmos_low (offset, value)  
  
exception CmosAccessError  
  Bases: RuntimeError  
  
exception CmosRuntimeError  
  Bases: RuntimeError
```

#### cpu module

CPU related functionality

```
class CPU (cs)  
  Bases: chipsec.hal.hal_base.HALBase  
  
  check_SMRR_supported ()  
  
  check_vmm ()  
  
  cpuid (eax, ecx)  
  
  dump_page_tables (cr3, pt_fname=None)  
  
  dump_page_tables_all ()  
  
  get_SMRAM ()  
  
  get_SMRR ()  
  
  get_SMRR_SMRAM ()  
  
  get_TSEG ()  
  
  get_cpu_topology ()  
  
  get_number_logical_processor_per_core ()  
  
  get_number_logical_processor_per_package ()
```

```
get_number_physical_processor_per_package ()

get_number_sockets_from_APIC_table ()

get_number_threads_from_APIC_table ()

is_HT_active ()

read_cr (cpu_thread_id, cr_number)

write_cr (cpu_thread_id, cr_number, value)

exception CPURuntimeError
  Bases: RuntimeError
```

#### cpuid module

CPUID information

usage:

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

```
class CpuID (cs)
  Bases: chipsec.hal.hal_base.HALBase

  cpuid (eax, ecx)
```

```
exception CpuIDRuntimeError
  Bases: RuntimeError
```

#### 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: chipsec.hal.hal_base.HALBase

  read_data ()

  read_idx (offset)

  read_memory (offset)

  read_memory_extended (word_offset)
```

```
read_range (start_offset, size)

write_command (command)

write_data (data)

write_idx (offset, value)

write_memory (offset, data)

write_memory_extended (word_offset, data)

write_range (start_offset, buffer)
```

#### *hal\_base module*

Base for HAL Components

```
class HALBase (cs)
    Bases: object
```

#### *igd module*

Working with Intel processor Integrated Graphics Device (IGD)

**usage:**

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

```
class IGD (cs)
    Bases: chipsec.hal.hal_base.HALBase

    dump_GGTT_PTEs (num)

    get_GGTT_PTE_from_PA (pa)

    get_GGTT_PTE_from_PA_gen8 (pa)

    get_GGTT_PTE_from_PA_legacy (pa)

    get_GGTT_base ()

    get_GMADR ()

    get_GTTMMADR ()

    get_PA_from_PTE (pte)

    get_PA_from_PTE_gen8 (pte)

    get_PA_from_PTE_legacy (pte)

    get_PTE_size ()

    gfx_aperture_dma_read_write (address, size=4, value=None, pte_num=0)
```

```

is_device_enabled ()

is_enabled ()

is_legacy_gen ()

read_GGTT_PTE (pte_num)

write_GGTT_PTE (pte_num, pte)

write_GGTT_PTE_from_PA (pte_num, pa)

exception IGDRuntimeError
Bases: RuntimeError

```

#### interrupts module

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

usage:

```

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

```

```

class Interrupts (cs)
Bases: chipsec.hal.hal_base.HALBase

find_ACPI_SMI_Buffer ()

find_smmc (start, end)

send_ACPI_SMI (thread_id, smi_num, buf_addr, invoc_reg, guid, data)

send_NMI ()

send_SMI_APMC (SMI_code_port_value, SMI_data_port_value)

send_SW_SMI (thread_id, SMI_code_port_value, SMI_data_port_value, _rax, _rbx, _rcx, _rdx, _rsi, _rdi)

send_smmc_SMI (smmc, guid, payload, payload_loc, CommandPort=0, DataPort=0)

```

#### 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, range_size, size=1)

```



```
read_IO (range_base, range_size, size=1)

read_port_byte (io_port)

read_port_dword (io_port)

read_port_word (io_port)

write_port_byte (io_port, value)

write_port_dword (io_port, value)

write_port_word (io_port, value)

exception PortIORuntimeError
Bases: RuntimeError
```

#### *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: chipsec.hal.hal_base.HALBase

dump_IO_BAR (bar_name, size=1)

get_IO_BAR_base_address (bar_name)

is_IO_BAR_defined (bar_name)

is_IO_BAR_enabled (bar_name)

list_IO_BARs ()

read_IO_BAR (bar_name, size=1)

read_IO_BAR_reg (bar_name, offset, size)

write_IO_BAR_reg (bar_name, offset, size, value)

exception IOBARNotFoundError
Bases: RuntimeError

exception IOBARRuntimeError
Bases: RuntimeError
```

#### *iommu module*

Access to IOMMU engines

```

class IOMMU (cs)
    Bases: chipsec.hal.hal_base.HALBase

    dump_IOMMU_configuration (iommu_engine)

    dump_IOMMU_page_tables (iommu_engine)

    dump_IOMMU_status (iommu_engine)

    get_IOMMU_Base_Address (iommu_engine)

    is_IOMMU_Engine_Enabled (iommu_engine)

    is_IOMMU_Translation_Enabled (iommu_engine)

    set_IOMMU_Translation (iommu_engine, te)

exception IOMMUError
    Bases: RuntimeError

```

#### 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 MMIO (cs)
    Bases: chipsec.hal.hal_base.HALBase

    dump_MMIO (bar_base, size)

    dump_MMIO_BAR (bar_name)

    get_MMCFG_base_address ()

    get_MMIO_BAR_base_address (bar_name)

    is_MMIO_BAR_defined (bar_name)

    is_MMIO_BAR_enabled (bar_name)

```

```
is_MMIO_BAR_programmed (bar_name)

list_MMIO_BARS ()

read_MMIO (bar_base, size)

read_MMIO_BAR (bar_name)

read_MMIO_BAR_reg (bar_name, offset, size=4)

read_MMIO_reg (bar_base, offset, size=4, bar_size=None)

read_MMIO_reg_byte (bar_base, offset)

read_MMIO_reg_dword (bar_base, offset)

read_MMIO_reg_word (bar_base, offset)

read_mmcfg_reg (bus, dev, fun, off, size)

write_MMIO_BAR_reg (bar_name, offset, value, size=4)

write_MMIO_reg (bar_base, offset, value, size=4, bar_size=None)

write_MMIO_reg_byte (bar_base, offset, value)

write_MMIO_reg_dword (bar_base, offset, value)

write_MMIO_reg_word (bar_base, offset, value)

write_mmcfg_reg (bus, dev, fun, off, size, value)
```

#### *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: chipsec.hal.hal_base.HALBase
```

```
mm_msgbus_reg_read (port, register)
mm_msgbus_reg_write (port, register, data)
```

```
msgbus_read_message (port, register, opcode)

msgbus_reg_read (port, register)

msgbus_reg_write (port, register, data)

msgbus_send_message (port, register, opcode, data=None)

msgbus_write_message (port, register, opcode, data)
```

```
exception MsgBusRuntimeError
Bases: RuntimeError
```

#### 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, num_entries=None)

GDT_all (num_entries=None)

IDT (cpu_thread_id, num_entries=None)

IDT_all (num_entries=None)

dump_Descriptor_Table (cpu_thread_id, code, num_entries=None)

get_Desc_Table_Register (cpu_thread_id, code)

get_GDTR (cpu_thread_id)

get_IDTR (cpu_thread_id)

get_LDTR (cpu_thread_id)

get_cpu_core_count ()

get_cpu_thread_count ()

read_msr (cpu_thread_id, msr_addr)

write_msr (cpu_thread_id, msr_addr, eax, edx)

exception MsrRuntimeError
Bases: RuntimeError
```

## paging module

*exception* InvalidMemoryAddress

Bases: RuntimeError

*class* c\_4level\_page\_tables (cs)

Bases: chipsec.hal.paging.c\_paging

get\_attr (entry)

get\_virt\_addr (pml4e\_index, pdpte\_index=0, pde\_index=0, pte\_index=0)

is\_bigpage (entry)

is\_present (entry)

print\_entry (lvl, pa, va=0, perm=)

read\_entry\_by\_virt\_addr (virt)

read\_page\_tables (ptr)

read\_pd (addr, pml4e\_index, pdpte\_index)

read\_pdpt (addr, pml4e\_index)

read\_pml4 (addr)

read\_pt (addr, pml4e\_index, pdpte\_index, pde\_index)

*class* c\_extended\_page\_tables (cs)

Bases: chipsec.hal.paging.c\_4level\_page\_tables

get\_attr (entry)

is\_bigpage (entry)

is\_present (entry)

map\_bigpage\_1G (virt, i)

read\_pt\_and\_show\_status (path, name, ptr)

*class* c\_ia32e\_page\_tables (cs)

Bases: chipsec.hal.paging.c\_4level\_page\_tables

get\_attr (entry)

is\_bigpage (entry)

is\_present (entry)

*class* c\_pae\_page\_tables (cs)

Bases: chipsec.hal.paging.c\_ia32e\_page\_tables

read\_page\_tables (ptr)

read\_pdpt (addr, pml4e\_index=None)

```
read_pml4(addr)

class c_paging(cs)
    Bases: chipsec.hal.paging.c_paging_with_2nd_level_translation,
chipsec.hal.paging.c_translation

    check_misconfig(addr_list)

    get_canonical(va)

    get_field(entry, desc)

    load_configuration(path)

    print_info(name)

    read_entries(info, addr, size=8)

    read_page_tables(entry)

    read_pt_and_show_status(path, name, ptr)

    save_configuration(path)

    set_field(value, desc)

class c_paging_memory_access(cs)
    Bases: object

    readmem(name, addr, size=4096)

class c_paging_with_2nd_level_translation(cs)
    Bases: chipsec.hal.paging.c_paging_memory_access

    readmem(name, addr, size=4096)

class c_reverse_translation(translation)
    Bases: object

    get_reverse_translation(addr)

class c_translation
    Bases: object

    add_page(virt, phys, size, attr)

    del_page(addr)

    expand_pages(exp_size)

    get_address_space()

    get_mem_range(noattr=False)

    get_pages_by_physaddr(addr)

    get_translation(addr)

    is_translation_exist(addr, mask, size)
```

```

class c_vtd_page_tables (cs)
    Bases: chipsec.hal.paging.c_extended_page_tables

    print_context_entry (source_id, cee)

    read_ce (addr, ree_index)

    read_page_tables (ptr)

    read_pt_and_show_status (path, name, ptr)

    read_re (addr)

    read_vtd_context (path, ptr)

```

#### 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_deviceBars( 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: chipsec.hal.pci.EFI_XROM_HEADER

```

```

class PCI_XROM_HEADER (Signature, ArchSpecific, PCIROffset)
    Bases: chipsec.hal.pci.PCI_XROM_HEADER

```

```

class Pci (cs)
    Bases: object

    calc_bar_size (bus, dev, fun, off, reg)

    dump_pci_config (bus, device, function)

    enumerate_devices ()

    enumerate_xroms (try_init=False, xrom_dump=False, xrom_addr=None)

    find_XROM (bus, dev, fun, try_init=False, xrom_dump=False, xrom_addr=None)

    get_DIDVID (bus, dev, fun)

    get_deviceBars (bus, dev, fun, bCalcSize=False)

    is_enabled (bus, dev, fun)

    parse_XROM (xrom, xrom_dump=False)

```



```
print_pci_config_all ()

read_byte (bus, device, function, address)

read_dword (bus, device, function, address)

read_word (bus, device, function, address)

write_byte (bus, device, function, address, byte_value)

write_dword (bus, device, function, address, dword_value)

write_word (bus, device, function, address, word_value)

exception PciDeviceNotFoundError
    Bases: RuntimeError

exception PciRuntimeError
    Bases: RuntimeError

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

class XROM_HEADER (Signature, InitSize, InitEP, Reserved, PCIROffset)
    Bases: chipsec.hal.pci.XROM_HEADER

get_device_name_by_didvid (vid, did)

get_vendor_name_by_vid (vid)

print_pci_XROMs (_xroms)

print_pci_devices (_devices)
```

#### *pcidb module*

PCI Vendor & Device ID data.

#### **Note**

THIS FILE WAS GENERATED

Auto generated from:

<https://github.com/pciutils/pciids>

#### *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: chipsec.hal.hal_base.HALBase

    alloc_physical_mem (length, max_phys_address=18446744073709551615)

    free_physical_mem (pa)

    map_io_space (pa, length, cache_type)

    read_physical_mem (phys_address, length)

    read_physical_mem_byte (phys_address)

    read_physical_mem_dword (phys_address)

    read_physical_mem_word (phys_address)

    set_mem_bit (addr, bit)

    va2pa (va)

    write_physical_mem (phys_address, length, buf)

    write_physical_mem_byte (phys_address, byte_value)

    write_physical_mem_dword (phys_address, dword_value)

    write_physical_mem_word (phys_address, word_value)

exception MemoryAccessError
    Bases: RuntimeError

exception MemoryRuntimeError
    Bases: RuntimeError

```

#### smbios module

```

class SMBIOS (cs)
    Bases: chipsec.hal.hal_base.HALBase

    find_smbios_table ()

    get_decoded_structs (struct_type=None, force_32bit=False)

    get_header (raw_data)

    get_raw_structs (struct_type=None, force_32bit=False)
        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)

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: chipsec.hal.smbios.SMBIOS_2_x_ENTRY_POINT

class SMBIOS_3_x_ENTRY_POINT (Anchor, EntryCs, EntryLen, MajorVer, MinorVer, Docrev, EntryRev, Reserved,
MaxSize, TableAddr)
Bases: chipsec.hal.smbios.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: chipsec.hal.smbios.SMBIOS_BIOS_INFO_2_0_ENTRY

class SMBIOS_STRUCT_HEADER (Type, Length, Handle)
Bases: chipsec.hal.smbios.SMBIOS_STRUCT_HEADER

class SMBIOS_SYSTEM_INFO_2_0 (type, length, handle, manufacturer_str, product_str, version_str, serial_str,
strings)
Bases: chipsec.hal.smbios.SMBIOS_SYSTEM_INFO_2_0_ENTRY
```

#### *smbus module*

Access to SMBus Controller

```
class SMBus (cs)
Bases: chipsec.hal.hal_base.HALBase

display_SMBus_info ()

enable_SMBus_host_controller ()

get_SMBus_Base_Address ()

get_SMBus_HCFG ()

is_SMBus_enabled ()

is_SMBus_host_controller_enabled ()

is_SMBus_supported ()

read_byte (target_address, offset)

read_range (target_address, start_offset, size)

reset_SMBus_controller ()

write_byte (target_address, offset, value)

write_range (target_address, start_offset, buffer)
```

#### *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_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_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_11R24.pdf)

[http://www.jedec.org/sites/default/files/docs/4\\_01\\_02\\_12R23A.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](http://en.wikipedia.org/wiki/Serial_presence_detect)

```
class SPD (smbus)
```

```
    Bases: object
```

```
    decode (device=160)
```

```
    detect ()
```

```
    dump_spd_rom (device=160)
```

```
    getDRAMDeviceType (device=160)
```

```
    getModuleType (device=160)
```

```
    isECC (device=160)
```

```
    isSPDPresent (device=160)
```

```
    read_byte (offset, device=160)
```

```
    read_range (start_offset, size, device=160)
```

```
    write_byte (offset, value, device=160)
```

```
    write_range (start_offset, buffer, device=160)
```

```
class SPD_DDR (SPDBytes, TotalBytes, DeviceType, RowAddressCount)
```

```
    Bases: chipsec.hal.spd.SPD_DDR
```

```
class SPD_DDR2 (SPDBytes, TotalBytes, DeviceType, RowAddressCount)
```

```
    Bases: chipsec.hal.spd.SPD_DDR2
```

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

```
    Bases: chipsec.hal.spd.SPD_DDR3
```

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

```
    Bases: chipsec.hal.spd.SPD_DDR4
```

```
SPD_REVISION (revision)
```

```
dram_device_type_name (dram_type)
```

```
module_type_name (module_type)
```

#### *spi module*

Access to SPI Flash parts

**usage:**

```
>>> read_spi( spi_flg, length )
>>> write_spi( spi_flg, buf )
```

```
>>> erase_spi_block( spi_flg )
>>> 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: chipsec.hal.hal_base.HALBase

  check_hardware_sequencing ()

  disable_BIOS_write_protection ()

  display_BIOS_region ()

  display_BIOS_write_protection ()

  display_SPI_Flash_Descriptor ()

  display_SPI_Flash_Regions ()

  display_SPI_Protected_Ranges ()

  display_SPI_Ranges_Access_Permissions ()

  display_SPI_map ()

  display_SPI_opcode_info ()

  erase_spi_block (spi_flg)

  get_SPI_JEDEC_ID ()

  get_SPI_JEDEC_ID_decoded ()

  get_SPI_MMIO_base ()

  get_SPI_Protected_Range (pr_num)

  get_SPI_SFDP ()

  get_SPI_region (spi_region_id)

  get_SPI_regions (all_regions=True)

  ptmesg (offset)

  read_spi (spi_flg, data_byte_count)
```

```
read_spi_to_file (spi_flg, data_byte_count, filename)

spi_reg_read (reg, size=4)

spi_reg_write (reg, value, size=4)

write_spi (spi_flg, buf)

write_spi_from_file (spi_flg, filename)

exception SpiAccessError
  Bases: RuntimeError

exception SpiRuntimeError
  Bases: RuntimeError

get_SPI_region (flreg)
```

#### *spi\_descriptor module*

SPI Flash Descriptor binary parsing functionality

**usage:**

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

```
get_SPI_master (flmstr)
```

```
get_spi_flash_descriptor (rom)
```

```
get_spi_regions (fd)
```

```
parse_spi_flash_descriptor (cs, rom)
```

#### *spi\_jedec\_ids module*

JEDED ID : Manufacturers and Device IDs

```
class JEDEC_ID
  Bases: object
```

```
DEVICE = {12722199: 'MX25L6408', 12722200: 'MX25L12805', 15679511: 'W25Q64FV (SPI)', 15679512:
'W25Q128 (SPI)', 15679513: 'W25Q256', 15687703: 'W25Q64FV (QPI)', 15687704: 'W25Q128 (QPI)', 15691798:
'W25Q32JV'}
```

```
MANUFACTURER = {194: 'Macronix', 239: 'Winbond'}
```

#### *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, parent, modn)
```

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

```
    Bases: json.encoder.JSONEncoder
```

```
default (obj)
```

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 (_uefi, fv_img, fwtype)
```

```
build_efi_model (_uefi, data, fwtype)
```

```
build_efi_modules_tree (_uefi, fwtype, data, Size, offset, polarity)
```

```
build_efi_tree (_uefi, data, fwtype)
```

```
compress_image (_uefi, image, compression_type)
```

```
decode_uefi_region (_uefi, pth, fname, fwtype, filetype=[])
```

```
decompress_section_data (_uefi, section_dir_path, sec_fs_name, compressed_data, compression_type,
remove_files=False)
```

```
dump_efi_module (mod, parent, modn, path)
```

```
modify_uefi_region (data, command, guid, uefi_file="")
```

```
parse_uefi_region_from_file (_uefi, filename, fwtype, outpath=None, filetype=[])
```

```
save_efi_tree (_uefi, modules, parent=None, save_modules=True, path=None, save_log=True, lvl=0)
```

```
save_efi_tree_filetype (modules, parent=None, path=None, lvl=0, filetype=[], save=False)
```

```
search_efi_tree (modules, search_callback, match_module_types=0, findall=True)
```

```
update_efi_tree (modules, parent_guid=None)
```

### *tpm module*

Trusted Platform Module (TPM) HAL component

<https://trustedcomputinggroup.org>

**class** **TPM** (**cs**)

Bases: **chipsec.hal.hal\_base.HALBase**

**command** (**commandName**, **locality**, **command\_argv**)

Send command to the TPM and receive data

**dump\_access** (**locality**)

View the contents of the register used to gain ownership of the TPM

**dump\_didvid** (**locality**)

TPM's Vendor and Device ID

**dump\_intcap** (**locality**)

Provides information of which interrupts that particular TPM supports

**dump\_intenable** (**locality**)

View the contents of the register used to enable specific interrupts

**dump\_rid** (**locality**)

TPM's Revision ID

**dump\_status** (**locality**)

View general status details

**class** **TPM\_RESPONSE\_HEADER** (**ResponseTag**, **DataSize**, **ReturnCode**)

Bases: **chipsec.hal.tpm.TPM\_RESPONSE\_HEADER**

**exception** **TpmRuntimeError**

Bases: **RuntimeError**

### *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**)

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**)

**getcap** (**command\_argv**)

Returns current information regarding the TPM CapArea - Capabilities Area SubCapSize - Size of SubCapabilities SubCap - Subcapabilities

**nvread** (**command\_argv**)

Read a value from the NV store Index, Offset, Size

**pcrread** (**command\_argv**)

The TPM\_PCRRead operation provides non-cryptographic reporting of the contents of a named PCR



### **startup** (command\_argv)

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

### *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

[https://trustedcomputinggroup.org/wp-content/uploads/TCG\\_EFI\\_Platform\\_1\\_22\\_Final\\_-v15.pdf](https://trustedcomputinggroup.org/wp-content/uploads/TCG_EFI_Platform_1_22_Final_-v15.pdf)

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

[https://trustedcomputinggroup.org/wp-content/uploads/TCG\\_PCClientImplementation\\_1-21\\_1\\_00.pdf](https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientImplementation_1-21_1_00.pdf)

TCG EFI Protocol Specification, Family "2.0"

<https://trustedcomputinggroup.org/wp-content/uploads/EFI-Protocol-Specification-rev13-160330final.pdf>

TCG PC Client Platform Firmware Profile Specification [https://trustedcomputinggroup.org/wp-content/uploads/PC-ClientSpecific\\_Platform\\_Profile\\_for\\_TPM\\_2p0\\_Systems\\_v51.pdf](https://trustedcomputinggroup.org/wp-content/uploads/PC-ClientSpecific_Platform_Profile_for_TPM_2p0_Systems_v51.pdf)

**class** EFIFirmwareBlob (\*args)

Bases: `chipsec.hal.tpm_eventlog.TcgPcrEvent`

**class** PcrLogParser (log)

Bases: `object`

Iterator over the events of a log.

**next** ()

**class** SCRTMVersion (\*args)

Bases: `chipsec.hal.tpm_eventlog.TcgPcrEvent`

**class** TcgPcrEvent (pcr\_index, event\_type, digest, event\_size, event)

Bases: `object`

An Event (TPM 1.2 format) as recorded in the SML.

**classmethod** parse (log)

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)

Simple wrapper around PcrLogParser.

### *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 ()

  load_ucode_update (cpu_thread_id, ucode_buf)

  ucode_update_id (cpu_thread_id)

  update_ucode (cpu_thread_id, ucode_file)

  update_ucode_all_cpus (ucode_file)

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

  dump_ucode_update_header (pdb_ucode_buffer)

  read_ucode_file (ucode_filename)
```

#### uefi module

Main UEFI component using platform specific and common UEFI functionality

```
class UEFI (cs)
  Bases: chipsec.hal.hal_base.HALBase

  compress_EFI_binary (uncompressed_name, compressed_name, compression_type)

  decompress_EFI_binary (compressed_name, uncompressed_name, compression_type)

  delete_EFI_variable (name, guid)

  dump_EFI_tables ()

  dump_EFI_variables_from_SPI ()

  find_EFI_BootServices_Table ()

  find_EFI_Configuration_Table ()

  find_EFI_DXEservices_Table ()

  find_EFI_RuntimeServices_Table ()

  find_EFI_System_Table ()

  find_EFI_Table (table_sig)

  find_EFI_variable_store (rom_buffer)

  find_s3_bootscript ()
```

```

get_EFI_variable (name, guid, filename=None)

get_s3_bootscript (log_script=False)

list_EFI_variables ()

parse_EFI_variables (iname, rom, authvars, _fw_type=None)

read_EFI_variables (efi_var_store, authvars)

read_EFI_variables_from_SPI (BIOS_region_base, BIOS_region_size)

read_EFI_variables_from_file (filename)

set_EFI_variable (name, guid, var, datasize=None, attrs=None)

set_EFI_variable_from_file (name, guid, filename, datasize=None, attrs=None)

set_FWType (efi_nvram_format)

decode_EFI_variables (efi_vars, nvram_pth)

get_attr_string (attr)

get_auth_attr_string (attr)

identify_EFI_NVRAM (buffer)

parse_script (script, log_script=False)

print_efi_variable (offset, efi_var_buf, EFI_var_header, efi_var_name, efi_var_data, efi_var_guid,
efi_var_attributes)

print_sorted_EFI_variables (variables)

```

#### 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: `chipsec.hal.uefi_common.EFI_BOOT_SERVICES_TABLE`

```

class EFI_CONFIGURATION_TABLE

```

Bases: `object`

```

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

```

Bases: `chipsec.hal.uefi_common.EFI_DXE_SERVICES_TABLE`

`EFI_ERROR_STR` (`error`)

Translates an EFI\_STATUS value into its corresponding textual representation.

`EFI_GUID_STR` (`guid`)

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

Bases: `chipsec.hal.uefi_common.EFI_RUNTIME_SERVICES_TABLE`

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

Bases: `chipsec.hal.uefi_common.EFI_SYSTEM_TABLE`

`EFI_SYSTEM_TABLE_REVISION` (`revision`)

`class EFI_TABLE_HEADER` (`Signature`, `Revision`, `HeaderSize`, `CRC32`, `Reserved`)

Bases: `chipsec.hal.uefi_common.EFI_TABLE_HEADER`

`class EFI_VENDOR_TABLE` (`VendorGuid`, `VendorTable`)

Bases: `chipsec.hal.uefi_common.EFI_VENDOR_TABLE`

`VendorGuid` ()

Alias for field number 0

`IS_EFI_VARIABLE_AUTHENTICATED` (`attr`)

`IS_VARIABLE_ATTRIBUTE` (`_c`, `_Mask`)

`class S3BOOTSCRIPT_ENTRY` (`script_type`, `index`, `offset_in_script`, `length`, `data=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: `chipsec.hal.uefi_common.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: chipsec.hal.uefi_common.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, size)

bit_set (value, mask, polarity=False)

get_3b_size (s)

get_nvar_name (nvram, name_offset, isAscii)

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

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

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

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

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

class op_terminate (opcode, size)
    Bases: object

class op_unknown (opcode, size)
    Bases: object

parse_auth_var (db, decode_dir)

parse_efivar_file (fname, var=None, var_type=1)

parse_esal_var (db, decode_dir)

parse_external (data)

parse_pkcs7 (data)
```

```
parse_rsa2048 (data)
parse_rsa2048_sha1 (data)
parse_rsa2048_sha256 (data)
parse_sb_db (db, decode_dir)
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)
```

#### uefi\_fv module

```
DecodeSection (SecType, SecBody, SecHeaderSize)

class EFI_FILE (Offset, Guid, Type, Attributes, State, Checksum, Size, Image, HeaderSize, UD, CalcSum)
    Bases: chipsec.hal.uefi_fv.EFI_MODULE

class EFI_FV (Offset, Guid, Size, Attributes, HeaderSize, Checksum, ExtHeaderOffset, Image, CalcSum)
    Bases: chipsec.hal.uefi_fv.EFI_MODULE

class EFI_MODULE (Offset, Guid, HeaderSize, Attributes, Image)
    Bases: object

    calc_hashes (off=0)

    name ()

class EFI_SECTION (Offset, Name, Type, Image, HeaderSize, Size)
    Bases: chipsec.hal.uefi_fv.EFI_MODULE

    name ()

FvChecksum16 (buffer)

FvChecksum8 (buffer)

FvSum16 (buffer)

FvSum8 (buffer)
```



```

GetFvHeader (buffer, off=0)

NextFwFile (FvImage, FvLength, fof, polarity)

NextFwFileSection (sections, ssize, sof, polarity)

NextFwVolume (buffer, off=0)

ValidateFwVolumeHeader (ZeroVector, FsGuid, FvLength, HeaderLength, ExtHeaderOffset, Reserved, size)

align_image (image, size=8, fill='x00')

assemble_uefi_file (guid, image)

assemble_uefi_raw (image)

assemble_uefi_section (image, uncompressed_size, compression_type)

get_guid_bin (guid)

```

#### 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: chipsec.hal.uefi_platform.EFI_HDR_NVAR1

class EFI_HDR_VSS (StartId, State, Reserved, Attributes, NameSize, DataSize, guid)
    Bases: chipsec.hal.uefi_platform.EFI_HDR_VSS

class EFI_HDR_VSS_APPLE (StartId, State, Reserved, Attributes, NameSize, DataSize, guid, unknown)
    Bases: chipsec.hal.uefi_platform.EFI_HDR_VSS_APPLE

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

EFIvar_EVSA (nvram_buf)

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, _Mask)

ParsePFS (data)

class PfsFile (data, concat=False)
    Bases: object

    parse ()

class PfsFileSection (data)
    Bases: object

    parse ()

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: chipsec.hal.uefi_platform.UEFI_VARIABLE_HEADER

UEFI_VARIABLE_STORE_HEADER_SIZE = 28
EFI_VARIABLE_HEADER_AUTH = "<HBB128sIIHH8s"
EFI_VARIABLE_HEADER_AUTH_SIZE = struct.calcsize(EFI_VARIABLE_HEADER_AUTH)
EFI_VARIABLE_HEADER = "<HBBIIHH8s"
EFI_VARIABLE_HEADER_SIZE = struct.calcsize(EFI_VARIABLE_HEADER)

class VARIABLE_STORE_HEADER_VSS (Signature, Size, Format, State, Reserved, Reserved1)
    Bases: chipsec.hal.uefi_platform.VARIABLE_STORE_HEADER_VSS

class VARIABLE_STORE_HEADER_VSS2 (Signature, Size, Format, State, Reserved, Reserved1)
    Bases: chipsec.hal.uefi_platform.VARIABLE_STORE_HEADER_VSS2

create_s3bootscript_entry_buffer (script_type, op, index=None)

decode_s3bs_opcode (s3bootscript_type, script_data)

decode_s3bs_opcode_def (data)

decode_s3bs_opcode_edkcompat (data)

encode_s3bootscript_entry (entry)

encode_s3bs_opcode (s3bootscript_type, op)

encode_s3bs_opcode_def (op)

encode_s3bs_opcode_edkcompat (op)

getEFIvariables_NVAR (nvram_buf)

getEFIvariables_NVAR_simple (nvram_buf)

```

```
getEFIvariables_UEFI (nvram_buf)
getEFIvariables_UEFI_AUTH (nvram_buf)
getEFIvariables_VSS (nvram_buf)
getEFIvariables_VSS2 (nvram_buf)
getEFIvariables_VSS2_AUTH (nvram_buf)
getEFIvariables_VSS_APPLE (nvram_buf)
getEFIvariables_VSS_AUTH (nvram_buf)
getNVstore_EFI (nvram_buf)
getNVstore_EFI_AUTH (nvram_buf)
getNVstore_EVSA (nvram_buf)
getNVstore_NVAR (nvram_buf)
getNVstore_NVAR_simple (nvram_buf)
getNVstore_VSS (nvram_buf)
getNVstore_VSS2 (nvram_buf)
getNVstore_VSS2_AUTH (nvram_buf)
getNVstore_VSS_APPLE (nvram_buf)
getNVstore_VSS_AUTH (nvram_buf)
id_s3bootscript_type (script, log_script=False)
isCorrectVSStype (nvram_buf, vss_type)
parse_s3bootscript_entry (s3bootscript_type, script, off, log_script=False)
```

#### *uefi\_search module*

UEFI image search auxilliary functionality

**usage:**

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

```
check_match_criteria (efi, criteria, _log)
```

```
check_rules (efi, rules, entry_name, _log, bLog=True)
```

#### *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 )
```

*exception* **MemoryAccessError**

Bases: **RuntimeError**

*exception* **MemoryRuntimeError**

Bases: **RuntimeError**

*class* **VirtMemory** (**cs**)

Bases: **chipsec.hal.hal\_base.HALBase**

**alloc\_virtual\_mem** (**length**, **max\_phys\_address=18446744073709551615**)

**free\_virtual\_mem** (**virt\_address**)

**read\_virtual\_mem** (**virt\_address**, **length**)

**read\_virtual\_mem\_byte** (**virt\_address**)

**read\_virtual\_mem\_dword** (**virt\_address**)

**read\_virtual\_mem\_word** (**virt\_address**)

**va2pa** (**va**)

**write\_virtual\_mem** (**virt\_address**, **length**, **buf**)

**write\_virtual\_mem\_byte** (**virt\_address**, **byte\_value**)

**write\_virtual\_mem\_dword** (**virt\_address**, **dword\_value**)

**write\_virtual\_mem\_word** (**virt\_address**, **word\_value**)

#### **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**, **pt\_fname=None**)

**hypercall** (**rax**, **rbx**, **rcx**, **rdx**, **rdi**, **rsi**, **r8=0**, **r9=0**, **r10=0**, **r11=0**, **xmm\_buffer=0**)

**hypercall64\_extended\_fast** (**hypervisor\_input\_value**, **parameter\_block**)

**hypercall64\_fast** (**hypervisor\_input\_value**, **param0=0**, **param1=0**)

**hypercall64\_five\_args** (**vector**, **arg1=0**, **arg2=0**, **arg3=0**, **arg4=0**, **arg5=0**)

**hypercall64\_memory\_based** (**hypervisor\_input\_value**, **parameters**, **size=0**)

**init** ()

```
exception VM/RuntimeError
Bases: RuntimeError

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

dump_device ()

get_virtio_devices (devices)
```

### OS Helpers & Drivers

Provides a translation layer to convert a common interface to OS specific driver calls

#### 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 to the import list either within helpers.py or custom\_helpers.py

#### Example

The new helper should be added to either chipsec/helper/helpers.py or chipsec/helper/custom\_helpers.py

A new helper folder should be created under chipsec/helper/new\_helper

chipsec/helper/new\_helper/\_\_init\_\_.py within the new folder needs to add the helper to avail\_helpers list

```
import platform
from chipsec.helper.oshelper import avail_helpers

if "linux" == platform.system().lower():
    __all__ = [ "linuxhelper" ]
    avail_helpers.append("linuxhelper")
else:
    __all__ = [ ]
```

chipsec/helper/new\_helper/new\_helper.py 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 package*

*dal package*

*dalhelper module*

Intel DFX Abstraction Layer (DAL) helper

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

```
class DALHelper
    Bases: chipsec.helper.basehelper.Helper

    EFI_supported ()

    cpuid (eax, ecx)

    create (start_driver)

    dal_version ()

    delete (start_driver)

    delete_EFI_variable (name, guid)

    find_thread ()

    get_ACPI_SDT ()

    get_ACPI_table (table_name)

    get_EFI_variable (name, guid, attrs=None)

    get_affinity ()

    get_descriptor_table (cpu_thread_id, desc_table_code)

    get_threads_count ()

    get_tool_info (tool_type)

    list_EFI_variables ()

    load_ucode_update (core_id, ucode_update_buf)

    map_io_space (physical_address, length, cache_type)
```

```
msgbus_send_message (mcr, mcrx, mdr=None)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

native_delete_EFI_variable (name, guid)

native_get_ACPI_table (table_name)

native_get_EFI_variable (name, guid, attrs=None)

native_list_EFI_variables ()

native_set_EFI_variable (name, guid, data, datasize, attrs=None)

pci_addr (bus, device, function, offset)

read_cr (cpu_thread_id, cr_number)

read_io_port (io_port, size)

read_mmio_reg (phys_address, size)

read_msr (thread, msr_addr)

read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

read_physical_mem (phys_address, length, bitwise=False)

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

set_EFI_variable (name, guid, data, datasize, attrs=None)

set_affinity (value)

start (start_driver, driver_exhists=False)

stop (start_driver)

target_machine ()

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (thread, msr_addr, eax, edx)

write_pci_reg (bus, device, function, address, dword_value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, buf)

write_physical_mem (phys_address, length, buf, bitwise=False)
```

*exception* DALHelperError

Bases: RuntimeError

get\_helper ()

*efi package*

*efihelper module*

On UEFI use the efi package functions

*class* EfiHelper

Bases: chipsec.helper.basehelper.Helper

EFI\_supported ()

alloc\_phys\_mem (length, max\_pa)

cpuid (eax, ecx)

create (start\_driver)

delete (start\_driver)

delete\_EFI\_variable (name, guid)

get\_ACPI\_SDT ()

get\_EFI\_variable (name, guidstr)

get\_EFI\_variable\_full (name, guidstr)

get\_descriptor\_table (cpu\_thread\_id, desc\_table\_code)

get\_threads\_count ()

get\_tool\_info (tool\_type)

getcwd ()

list\_EFI\_variables ()

load\_ucode\_update (cpu\_thread\_id, ucode\_update\_buf)

map\_io\_space (physical\_address, length, cache\_type)

msgbus\_send\_message (mcr, mcrx, mdr=None)

msgbus\_send\_read\_message (mcr, mcrx)

msgbus\_send\_write\_message (mcr, mcrx, mdr)

pa2va (pa)

read\_cr (cpu\_thread\_id, cr\_number)



```
read_io_port (io_port, size)

read_mmio_reg (phys_address, size)

read_msr (cpu_thread_id, msr_addr)

read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

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

set_EFI_variable (name, guidstr, data, datasize=None, attrs=7)

set_affinity (value)

start (start_driver, driver_exists=False)

stop (start_driver)

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (cpu_thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, address, value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, buf)

exception EfiHelperError
    Bases: RuntimeError

get_helper ()
```

*file package*

*filehelper module*

Use results from a json file

```
class FileCmds (filename)
    Bases: object

    AddElement (cmd, args, ret)

    Load ()

    Save ()
```

```
getElement (cmd, args)

class FileHelper
  Bases: chipsec.helper.basehelper.Helper

  EFI_supported ()

  alloc_phys_mem (length, max_phys_address)

  compress_file (FileName, OutputFileName, CompressionType)

  cpuid (eax, ecx)

  create (start_driver)

  decompress_file (CompressedFileName, OutputFileName, CompressionType)

  delete (start_driver)

  delete_EFI_variable (name, guid)

  free_phys_mem (physical_address)

  get_ACPI_SDT ()

  get_ACPI_table (table_name)

  get_EFI_variable (name, guid)

  get_affinity ()

  get_descriptor_table (cpu_thread_id, desc_table_code)

  get_threads_count ()

  getcwd ()

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

  list_EFI_variables ()

  load_ucode_update (cpu_thread_id, ucode_update_buf)

  map_io_space (physical_address, length, cache_type)

  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, size)

  read_mmio_reg (phys_address, size)

  read_msr (cpu_thread_id, msr_addr)
```

```
read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

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

set_EFI_variable (name, guid, data, datasize=None, attrs=None)

set_affinity (value)

start (start_driver, from_file=None)

stop (start_driver)

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (cpu_thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, address, value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, buf)

get_helper ()
```

*linux package*

*cpuid module*

```
class CPUID
    Bases: object

class CPUID_struct
    Bases: ctypes.Structure

    eax
        Structure/Union member

    ebx
        Structure/Union member

    ecx
        Structure/Union member

    edx
        Structure/Union member
```

*legacy\_pci module*

```

class LEGACY_PCI
    Bases: object

    read_pci_config (bus, dev, func, offset)

    write_pci_config (bus, dev, func, offset, value)

class PORTS
    Bases: object

    inl (port)

    outl (value, port)

```

*linuxhelper module*

Linux helper

```

class LinuxHelper
    Bases: chipsec.helper.basehelper.Helper

    DEVICE_NAME = '/dev/chipsec'

    DEV_MEM = '/dev/mem'

    DEV_PORT = '/dev/port'

    DKMS_DIR = '/var/lib/dkms/'

    EFIVARS_get_EFI_variable (name, guid)

    EFIVARS_get_efivar_from_sys (filename)

    EFIVARS_list_EFI_variables ()

    EFIVARS_set_EFI_variable (name, guid, value, attrs=None)

    EFI_supported ()

    MODULE_NAME = 'chipsec'

    SUPPORT_KERNEL26_GET_PAGE_IS_RAM = False

    SUPPORT_KERNEL26_GET_PHYS_MEM_ACCESS_PROT = False

    VARS_get_EFI_variable (name, guid)

    VARS_get_efivar_from_sys (filename)

    VARS_list_EFI_variables ()

    VARS_set_EFI_variable (name, guid, value)

    alloc_phys_mem (num_bytes, max_addr)

```

```

close ()

compress_file (FileName, OutputFileName, CompressionType)

compute_ioctlbase (itype='C')

cpuid (eax, ecx)

create (start_driver)

decompress_file (CompressedFileName, OutputFileName, CompressionType)

decompression_oder_type1 = [1, 2]

decompression_oder_type2 = [1, 2, 3, 4]

delete (start_driver)

delete_EFI_variable (name, guid)

devmem_available ()
    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 ()
    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 ()
    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 (physmem)

get_ACPI_SDT ()

get_ACPI_table (table_name)

get_EFI_variable (name, guid, attrs=None)

get_affinity ()

get_descriptor_table (cpu_thread_id, desc_table_code)

get_dkms_module_location ()

get_page_is_ram ()

get_phys_mem_access_prot ()

get_threads_count ()

get_tool_info (tool_type)

getcwd ()

```

```

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

init (start_driver)

ioctl (nr, args, *mutate_flag)

kern_get_EFI_variable (name, guid)

kern_get_EFI_variable_full (name, guid)

kern_list_EFI_variables ()

kern_set_EFI_variable (name, guid, value, attr=7)

list_EFI_variables ()

load_chipsec_module ()

load_ucode_update (cpu_thread_id, ucode_update_buf)

map_io_space (base, size, cache_type)

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

msgbus_send_message (mcr, mcrx, mdr=None)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

native_cpuid (eax, ecx)

native_delete_EFI_variable (name, guid)

native_get_ACPI_table ()

native_get_EFI_variable (name, guid, attrs=None)

native_list_EFI_variables ()

native_map_io_space (base, size, cache_type)
    Map to memory a specific region.

native_read_io_port (io_port, size)

native_read_mmio_reg (bar_base, bar_size, offset, size)

native_read_msr (thread_id, msr_addr)

native_read_pci_reg (bus, device, function, offset, size, domain=0)

native_read_phys_mem (phys_address_hi, phys_address_lo, length)

native_set_EFI_variable (name, guid, data, datasize, attrs=None)

native_write_io_port (io_port, newval, size)

```

```
native_write_mmio_reg (bar_base, bar_size, offset, size, value)

native_write_msr (thread_id, msr_addr, eax, edx)

native_write_pci_reg (bus, device, function, offset, value, size=4, domain=0)

native_write_phys_mem (phys_address_hi, phys_address_lo, length, newval)

read_cr (cpu_thread_id, cr_number)

read_io_port (io_port, size)

read_mmio_reg (phys_address, size)

read_msr (thread_id, msr_addr)

read_pci_reg (bus, device, function, offset, size=4)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

retpoline_enabled ()

rotate_list (list, n)

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

set_EFI_variable (name, guid, data, datasize, attrs=None)

set_affinity (thread_id)

start (start_driver, driver_exists=False)

stop (start_driver)

unknown_decompress (CompressedFileName, OutputFileName)

unknown_efi_decompress (CompressedFileName, OutputFileName)

use_efivars ()

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, offset, value, size=4)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, newval)

class MemoryMapping (fileno, length, flags, prot, offset)
    Bases: mmap.mmap
    Memory mapping based on Python's mmap.
```

This subclass keeps tracks of the start and end of the mapping.

```
get_helper ()
```

```
osx package
```

```
osxhelper module
```

OSX helper

```
class OSXHelper
    Bases: chipsec.helper.basehelper.Helper

    DEVICE_NAME = '/dev/chipsec'

    DRIVER_NAME = 'chipsec.kext'

    EFI_supported ()

    alloc_phys_mem (num_bytes, max_addr)

    close ()

    compress_file (FileName, OutputFileName, CompressionType)

    cpuid (eax, ecx)

    create (start_driver)

    decompress_file (CompressedFileName, OutputFileName, CompressionType)

    decompression_order_type1 = [1, 2]

    decompression_order_type2 = [1, 2, 3, 4]

    delete (start_driver)

    delete_EFI_variable (name, guid)

    get_EFI_variable (name, guid, attrs=None)

    get_affinity ()

    get_descriptor_table (cpu_thread_id, desc_table_code)

    get_threads_count ()

    get_tool_info (tool_type)

    getcwd ()

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

    init (start_driver)
```



```
ioctl (ioctl_name, args)

list_EFI_variables ()

load_driver ()

load_ucode_update (cpu_thread_id, ucode_update_buf)

map_io_space (base, size, cache_type)

mem_read_block (addr, sz)

mem_write_block (addr, sz, newval)

msgbus_send_message (mcr, mcrx, mdr=None)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

native_delete_EFI_variable (name, guid)

native_get_EFI_variable (name, guid, attrs=None)

native_list_EFI_variables ()

native_set_EFI_variable (name, guid, data, datasize, attrs=None)

read_cr (cpu_thread_id, cr_number)

read_io_port (io_port, size)

read_mmio_reg (phys_address, size)

read_msr (thread_id, msr_addr)

read_pci_reg (bus, device, function, offset, size=4)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (addr_hi, addr_lo, size)

retpoline_enabled ()

rotate_list (list, n)

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

set_EFI_variable (name, guid, data, datasize, attrs=None)

set_affinity (thread_id)

start (start_driver, driver_exists=False)

stop (start_driver)

unknown_decompress (CompressedFileName, OutputFileName)

unknown_efi_decompress (CompressedFileName, OutputFileName)
```

```
write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, offset, value, size=4)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (addr_hi, addr_lo, size, value)

get_helper ()
```

#### *rwe package*

#### *rwehelper module*

Management and communication with Windows kernel mode driver which provides access to hardware resources

### **Note**

On Windows you need to install pywin32 Python extension corresponding to your Python version:  
<http://sourceforge.net/projects/pywin32/>

```
CTL_CODE (DeviceType, Function, Method, Access)

class EFI_HDR_WIN (Size, DataOffset, DataSize, Attributes, guid)
    Bases: chipsec.helper.rwe.rwehelper.EFI_HDR_WIN

class PCI_BDF
    Bases: ctypes.Structure

    BUS
        Structure/Union member

    DEV
        Structure/Union member

    FUNC
        Structure/Union member

    OFF
        Structure/Union member

    cfg_address ()

class RweHelper
    Bases: chipsec.helper.basehelper.Helper

    EFI_supported ()
```

```
alloc_phys_mem (length, max_pa)

check_driver_handle ()

cpuid (eax, ecx)

create (start_driver)

decompress_file (CompressedFileName, OutputFileName, CompressionType)

delete (start_driver)

delete_EFI_variable (name, guid)

free_phys_mem (physical_address)

get_ACPI_SDT ()

get_ACPI_table (table_name)

get_EFI_variable (name, guid, attrs=None)

get_EFI_variable_full (name, guid, attrs=None)

get_affinity ()

get_compression_tool_path (compression_type)

get_descriptor_table (cpu_thread_id, desc_table_code)

get_driver_handle ()

get_threads_count ()

get_tool_info (tool_type)

get_tool_path (tool_type)

getcwd ()

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

list_EFI_variables (infcls=2)

load_ucode_update (cpu_thread_id, ucode_update_buf)

map_io_space (physical_address, length, cache_type)

msgbus_send_message (mcr, mcrx, mdr=None)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

native_get_ACPI_table (table_name)

native_read_phys_mem (phys_address_hi, phys_address_lo, length)

native_write_phys_mem (phys_address_hi, phys_address_lo, length, buf)
```

```
read_cr (cpu_thread_id, cr_number)

read_io_port (io_port, size)

read_mmio_reg (phys_address, size)

read_msr (cpu_thread_id, msr_addr)

read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

retpoline_enabled ()

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

set_EFI_variable (name, guid, data, datasize, attrs)

set_affinity (value)

show_warning ()

start (start_driver, driver_exists=False)

stop (start_driver)

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (cpu_thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, address, value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, buf)

getEFIVariables_NtEnumerateSystemEnvironmentValuesEx2 (nvram_buf)

get_helper ()

guid_str (guid0, guid1, guid2, guid3)

packl_ctypes (lnum, bitlength)
```

*win package*

*win32helper module*

Management and communication with Windows kernel mode driver which provides access to hardware resources

**Note**

On Windows you need to install pywin32 Python extension corresponding to your Python version:  
<http://sourceforge.net/projects/pywin32/>

`CTL_CODE` (`DeviceType`, `Function`, `Method`, `Access`)

`class EFI_HDR_WIN` (`Size`, `DataOffset`, `DataSize`, `Attributes`, `guid`)  
 Bases: `chipsec.helper.win.win32helper.EFI_HDR_WIN`

`class PCI_BDF`  
 Bases: `_ctypes.Structure`

`BUS`  
 Structure/Union member

`DEV`  
 Structure/Union member

`FUNC`  
 Structure/Union member

`OFF`  
 Structure/Union member

`class Win32Helper`  
 Bases: `chipsec.helper.basehelper.Helper`

`EFI_supported()`

`alloc_phys_mem(length, max_pa)`

`check_driver_handle()`

`compress_file(FileName, OutputFileName, CompressionType)`

`cpuid(eax, ecx)`

`create(start_driver)`

`decompress_file(CompressedFileName, OutputFileName, CompressionType)`

`decompression_oder_type1 = [1, 2]`

`decompression_oder_type2 = [1, 2, 3, 4]`

`delete(start_driver)`

`delete_EFI_variable(name, guid)`

`free_phys_mem(physical_address)`

`get_ACPI_SDT()`

`get_ACPI_table(table_name)`

```
get_EFI_variable (name, guid, attrs=None)

get_EFI_variable_full (name, guid, attrs=None)

get_affinity ()

get_descriptor_table (cpu_thread_id, desc_table_code)

get_driver_handle ()

get_threads_count ()

getcwd ()

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

list_EFI_variables (infcls=2)

load_ucode_update (cpu_thread_id, ucode_update_buf)

map_io_space (physical_address, length, cache_type)

msgbus_send_message (mcr, mcrx, mdr=None)

msgbus_send_read_message (mcr, mcrx)

msgbus_send_write_message (mcr, mcrx, mdr)

native_get_ACPI_table (table_name)

native_read_phys_mem (phys_address_hi, phys_address_lo, length)

native_write_phys_mem (phys_address_hi, phys_address_lo, length, buf)

read_cr (cpu_thread_id, cr_number)

read_io_port (io_port, size)

read_mmio_reg (phys_address, size)

read_msr (cpu_thread_id, msr_addr)

read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

retpoline_enabled ()

rotate_list (list, n)

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

set_EFI_variable (name, guid, data, datasize, attrs)

set_affinity (value)

show_warning ()
```

```

start (start_driver, driver_exists=False)

stop (start_driver)

unknown_decompress (CompressedFileName, OutputFileName)

unknown_efi_decompress (CompressedFileName, OutputFileName)

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

write_mmio_reg (phys_address, size, value)

write_msr (cpu_thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, address, value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, buf)

getEFIvariables_NtEnumerateSystemEnvironmentValuesEx2 (nvram_buf)

get_helper ()

packl_ctypes (lnum, bitlength)

```

#### basehelper module

```

class Helper
    Bases: object

    EFI_supported ()

    alloc_phys_mem (length, max_phys_address)

    compress_file (FileName, OutputFileName, CompressionType)

    cpuid (eax, ecx)

    create (start_driver)

    decompress_file (CompressedFileName, OutputFileName, CompressionType)

    delete (start_driver)

    delete_EFI_variable (name, guid)

    free_phys_mem (physical_address)

    get_ACPI_SDT ()

    get_ACPI_table (table_name)

```

```

get_EFI_variable (name, guid)

get_affinity ()

get_descriptor_table (cpu_thread_id, desc_table_code)

get_info ()

get_threads_count ()

getcwd ()

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

list_EFI_variables ()

load_ucode_update (cpu_thread_id, ucode_update_buf)

map_io_space (physical_address, length, cache_type)

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, size)

read_mmio_reg (phys_address, size)

read_msr (cpu_thread_id, msr_addr)

read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports

read_phys_mem (phys_address_hi, phys_address_lo, length)

retpoline_enabled ()

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

set_EFI_variable (name, guid, data, datasize=None, attrs=None)

set_affinity (value)

start (start_driver, from_file=None)

stop (start_driver)

use_native_api ()

va2pa (va)

write_cr (cpu_thread_id, cr_number, value)

write_io_port (io_port, value, size)

```



```
write_mmio_reg (phys_address, size, value)

write_msr (cpu_thread_id, msr_addr, eax, edx)

write_pci_reg (bus, device, function, address, value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_phys_mem (phys_address_hi, phys_address_lo, length, buf)
```

#### *helpers module*

#### *oshelper module*

Abstracts support for various OS/environments, wrapper around platform specific code that invokes kernel driver

```
exception HWAccessViolationError (msg, errorcode)
    Bases: chipsec.helper.oshelper.OsHelperError
```

```
class OsHelper
    Bases: object
```

```
EFI_supported ()

alloc_physical_mem (length, max_phys_address)

compress_file (FileName, OutputFileName, CompressionType)

cpuid (eax, ecx)

decompress_file (CompressedFileName, OutputFileName, CompressionType)

delete_EFI_variable (name, guid)

free_physical_mem (physical_address)

get_ACPI_SDT ()

get_ACPI_table (table_name)

get_EFI_variable (name, guid)

get_affinity ()

get_descriptor_table (cpu_thread_id, desc_table_code)

get_threads_count ()

getcwd ()

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

is_dal ()

is_efi ()
```

```
is_linux ()
is_macos ()
is_win8_or_greater ()
is_windows ()
list_EFI_variables ()
loadHelpers ()
load_ucode_update (cpu_thread_id, ucode_update_buf)
map_io_space (physical_address, length, cache_type)
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, size)
read_mmio_reg (bar_base, size, offset=0, bar_size=None)
read_msr (cpu_thread_id, msr_addr)
read_pci_reg (bus, device, function, address, size)
    Read PCI configuration registers via legacy CF8/CFC ports
read_physical_mem (phys_address, length)
retpoline_enabled ()
send_sw_smi (cpu_thread_id, SMI_code_data, _rax, _rbx, _rcx, _rdx, _rsi, _rdi)
set_EFI_variable (name, guid, data, datasize=None, attrs=None)
set_affinity (value)
start (start_driver, driver_exists=None, to_file=None, from_file=False)
stop (start_driver)
use_native_api ()
va2pa (va)
write_cr (cpu_thread_id, cr_number, value)
write_io_port (io_port, value, size)
write_mmio_reg (bar_base, size, value, offset=0, bar_size=None)
write_msr (cpu_thread_id, msr_addr, eax, edx)
```

```
write_pci_reg(bus, device, function, address, value, size)
    Write PCI configuration registers via legacy CF8/CFC ports

write_physical_mem(phys_address, length, buf)

exception OsHelperError(msg, errorcode)
    Bases: RuntimeError

exception UnimplementedAPIError(api_name)
    Bases: chipsec.helper.oshelper.OsHelperError

exception UnimplementedNativeAPIError(api_name)
    Bases: chipsec.helper.oshelper.UnimplementedAPIError

f_mod_zip(x)

get_tools_path()

helper()

map_modname_zip(x)
```

## Fuzzing

### fuzzing package

### 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: chipsec.fuzzing.primitives.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: chipsec.fuzzing.primitives.bit_field

class delim (value, fuzzable=True, name=None)
    Bases: chipsec.fuzzing.primitives.base_primitive

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

class group (name, values)
    Bases: chipsec.fuzzing.primitives.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.

isinteger (var)

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

class random_data (value, min_length, max_length, max_mutations=25, fuzzable=True, step=None, name=None)
    Bases: chipsec.fuzzing.primitives.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: chipsec.fuzzing.primitives.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: chipsec.fuzzing.primitives.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: chipsec.fuzzing.primitives.bit_field
```

## 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

## CHIPSEC Modules

### Introduction

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.)

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.

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	
SMI configuration is not locked (SMI race condition)	common.smi_lock	

### ***Attack Surface/Vector: Runtime protection of SMRAM***

<b>Vulnerability Description</b>	<b>CHIPSEC Module</b>	<b>Example</b>
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***

<b>Vulnerability Description</b>	<b>CHIPSEC Module</b>	<b>Example</b>
Root certificate	common.bios_wp, common.secureboot.variables	
Key exchange keys and whitelist/blacklist	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 mamp 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.blacklist	

### 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

*modules package*

*bdw package*

*byt package*

*common package*

*cpu package*

*cpu\_info module*

Displays CPU information

`class cpu_info`

Bases: `chipsec.module_common.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)`

*ia\_untrusted module*

IA Untrusted checks

`class ia_untrusted`

Bases: `chipsec.module_common.BaseModule`

`check_untrusted ()`

`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)`

**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: chipsec.module_common.BaseModule
```

```
    check_spectre_mitigations ()
```

```
    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)
```

*secureboot package*

*variables module*

UEFI 2.4 spec Section 28

Verify that all Secure Boot key UEFI variables are authenticated (BS+RT+AT) and protected from unauthorized modification.

Use '-a modify' option for the module to also try to write/corrupt the variables.

```
class variables
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    can_modify (name, guid, data, attrs)
```

```
    check_secureboot_variable_attributes (do_modify)
```

### `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)`

## *uefi package*

### *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.

#### `class access_uefispec`

Bases: `chipsec.module_common.BaseModule`

#### `can_modify (name, guid, data)`

#### `check_vars (do_modify)`

#### `diff_var (data1, data2)`

### `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)`

## *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>]
```

Examples:

```
>>> chipsec_main.py -m common.uefi.s3bootscript
>>> chipsec_main.py -m common.uefi.s3bootscript -a 0x00000000BDE10000
```

#### `class s3bootscript`

Bases: `chipsec.module_common.BaseModule`

#### `check_dispatch_opcodes (bootscript_entries)`

```
check_s3_bootscript (bootscript_pa)
```

```
check_s3_bootscripts (bsaddress=None)
```

```
is_inside_SMRAM (pa)
```

```
is_inside_SPI (pa)
```

```
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)
```

#### *bios\_kbrd\_buffer module*

DEFCON 16: [Bypassing Pre-boot Authentication Passwords by Instrumenting the BIOS Keyboard Buffer](#) by Jonathan Brossard

Checks for BIOS/HDD password exposure through BIOS keyboard buffer.

Checks for exposure of pre-boot passwords (BIOS/HDD/pre-bot authentication SW) in the BIOS keyboard buffer.

```
class bios_kbrd_buffer
```

```
Bases: chipsec.module_common.BaseModule
```

```
check_BIOS_keyboard_buffer ()
```

```
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)
```

#### *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* (<https://www.defcon.org/images/defcon-22/dc-22-presentations/Bulygin-Bazhaniul-Furtak-Loucaides/DEFCON-22-Bulygin-Bazhaniul-Furtak-Loucaides-Summary-of-attacks-against-BIOS-UPDATED.pdf>)

```
class bios_smi
```

```
Bases: chipsec.module_common.BaseModule
```

```
check_SMI_locks ()
```

```
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)
```

*bios\_ts module*

Checks for BIOS Interface Lock including Top Swap Mode

[BIOS Boot Hijacking and VMware Vulnerabilities Digging](#) by Bing Sun

```
class bios_ts
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    check_bios_iface_lock ()
```

```
    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)
```

*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 does check 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](#) described just such an attack. 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.

This module common.bios\_wp 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: chipsec.module_common.BaseModule
```

```
    check_BIOS_write_protection ()
```

```
    check_SPI_protected_ranges ()
```

```
    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)
```

*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

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.

Hardware registers used: IA32\_DEBUG\_INTERFACE[DEBUGENABLE]  
IA32\_DEBUG\_INTERFACE[DEBUGELOCK] IA32\_DEBUG\_INTERFACE[DEBUGEOCCURED]  
P2SB\_DCI.DCI\_CONTROL\_REG[HDCIEN]

```
class debugenabled
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    check_cpu_debug_enable ()
```

```
    check_dci ()
```

```
    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)
```

### ia32cfg module

Tests that IA-32/IA-64 architectural features are configured and locked, including IA32 Model Specific Registers (MSRs)

Reference: Intel Software Developer's Manual

```
class ia32cfg
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    check_ia32feature_control ()
```

```
    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)
```

### 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]

The module returns the following results:

FAILED : HFS.MFG\_MODE is set

PASSED : HFS.MFG\_MODE is not set.

Hardware registers used:

HFS

```
class me_mfg_mode
```

```
Bases: chipsec.module_common.BaseModule
```

```
check_me_mfg_mode ()
```

```
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)
```

## memconfig module

This module verifies memory map secure configuration, i.e. that memory map registers are correctly configured and locked down.

```
class memconfig
```

```
Bases: chipsec.module_common.BaseModule
```

```
check_memmap_locks ()
```

```
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)
```

### *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/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.

Hardware registers used: MSR\_LT\_LOCK\_MEMORY

```
class memlock
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    check_MSR_LT_LOCK_MEMORY ()
```

```
    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)
```

### *remap module*

[Preventing & Detecting Xen Hypervisor Subversions](#) by Joanna Rutkowska & Rafal Wojtczuk

Check Memory Remapping Configuration

```
class remap
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    check_remap_config ()
```

```
    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)
```

### *rtclock module*

Checks for RTC memory locks. Since we do not know what RTC memory will be used for on a specific platform, we return WARNING (rather than FAILED) if the memory is not locked.

```
class rtclock
```

```
    Bases: chipsec.module_common.BaseModule
```

```
    check_rtclock ()
```

### `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)`

#### *sgx\_check module*

Check SGX related configuration Reference: SGX BWG, CDI/IBP#: 565432

`class sgx_check`

Bases: `chipsec.module_common.BaseModule`

`check_sgx_config ()`

### `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)`

#### *smm module*

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 Compatibility 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](#).

This CHIPSEC module simply reads SMRAMC and checks that D\_LCK is set.

`class smm`

Bases: `chipsec.module_common.BaseModule`

`check_SMRAMC ()`

### `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)`

#### *smm\_code\_chk module*

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.

`class smm_code_chk`

Bases: `chipsec.module_common.BaseModule`

`check_SMM_Code_Chk_En ()`

**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)**

**smm\_dma module**

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.

DMA attacks were discussed in [Programmed I/O accesses: a threat to Virtual Machine Monitors?](#) and [System Management Mode Design and Security Issues](#). This is also discussed in *Summary of Attack against BIOS and Secure Boot* <https://www.defcon.org/images/defcon-22/dc-22-presentations/Bulygin-Bazhaniul-Furtak-Loucaides/DEFCON-22-Bulygin-Bazhaniul-Furtak-Loucaides-Summary-of-attacks-against-BIOS-UPDATED.pdf>

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.

**class smm\_dma**

Bases: **chipsec.module\_common.BaseModule**

**check\_tseg\_config ()**

**check\_tseg\_locks ()**

**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)**

**smrr module**

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-cacheable behavior and block access to SMRAM when the CPU is not in SMM. These registers need to be enabled/configured by the BIOS.

This module checks to see that SMRRs are enabled and configured.

**class smrr**

Bases: **chipsec.module\_common.BaseModule**

**check\_SMRR (do\_modify)**

**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)**

**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

**class spd\_wd**

Bases: `chipsec.module_common.BaseModule`

`check_spd_wd ()`

`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)`

**spi\_access module**

Checks SPI Flash Region Access Permissions programmed in the Flash Descriptor

**class spi\_access**

Bases: `chipsec.module_common.BaseModule`

`check_flash_access_permissions ()`

`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)`

**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.

**class spi\_desc**

Bases: `chipsec.module_common.BaseModule`

```
check_flash_access_permissions ()
```

```
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)
```

#### *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.

```
class spi_fdopss
```

```
Bases: chipsec.module_common.BaseModule
```

```
check_fd_security_override_strap ()
```

```
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)
```

#### *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 (ref: *Copernicus: Question Your Assumptions about BIOS Security* <http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about>).

This module checks that the SPI Flash Controller configuration is locked.

```
class spi_lock
```

```
Bases: chipsec.module_common.BaseModule
```

```
check_spi_lock ()
```

```
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)
```

#### *hsw package*

#### *ivb package*

#### *snb package*

#### *tools package*

#### *cpu package*

#### *sinkhole module*

This module checks if CPU is affected by 'The SMM memory sinkhole' vulnerability by Christopher Domas

NOTE: 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.

References:

The Memory Sinkhole by Christopher Domas: <https://www.blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation.pdf> (presentation) and <https://www.blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation-wp.pdf> (whitepaper).

```
class sinkhole
```

```
    Bases: chipsec.module_common.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)
```

#### *secureboot package*

#### *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: `chipsec.module_common.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` ()

**smm package****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
```

**DIFF (s, t, sz)**

```
class rogue_mmio_bar
```

```
Bases: chipsec.module_common.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)
```

**smm\_ptr module**

CanSecWest 2015 [A New Class of Vulnerability in SMI Handlers of BIOS/UEFI Firmware](#)

A tool to test SMI handlers for pointer validation vulnerabilities

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)

*exception* `BadSMIDetected`

Bases: `RuntimeError`

`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: `chipsec.module_common.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)`

**uefi package****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 visiting <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

**class reputation**

Bases: **chipsec.module\_common.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 ()**

**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 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: chipsec.module_common.BaseModule
```

```
    DISPATCH_ENTRYPOINT_INSTR = '\x90\x90\x00'
```

```
    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)**

### 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: `chipsec.module_common.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 ()****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: `chipsec.module_common.BaseModule`

**check\_list** (`json_pth`)

**generate\_efilist** (`json_pth`)

**genlist\_callback** (`efi_module`)

**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** ()

**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.

Note: 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.

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
```

**class uefivar\_fuzz**

Bases: **chipsec.module\_common.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 ()**

*vmm package*

*hv package*

*define module*

Hyper-V specific defines

**get\_hypercall\_name (code, defvalue=)**

**get\_hypercall\_status (code, defvalue=)**

**get\_msr\_name (code, defvalue=)**



`set_variables (varlist)`

#### *hypercall module*

Hyper-V specific hypercall functionality

```
class HyperVHypercall
    Bases: chipsec.modules.tools.vmm.common.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.
```

#### *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: `chipsec.module_common.BaseModule`

`run (module_argv)`

`usage ()`

`getrandbits (k)` → x. Generates an int with k random bits.

#### *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: `chipsec.modules.tools.vmm.hv.vmbus.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: `chipsec.module_common.BaseModule`

`run (module_argv)`

`usage ()`

#### *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: chipsec.modules.tools.vmm.hv.vmbus.RingBuffer

    ringbuffer_read ()
```

`getrandbits (k)` → x. Generates an int with k random bits.

```
class synth_kbd
    Bases: chipsec.module_common.BaseModule

    run (module_argv)

    usage ()
```

#### *vmbus module*

Hyper-V VMBus functionality

```
class HyperV
    Bases: chipsec.modules.tools.vmm.common.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: chipsec.modules.tools.vmm.common.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: chipsec.modules.tools.vmm.hv.vmbus.HyperV

    vmbus_clear ()
```

[illegible]

```
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.
```

#### vmbusfuzz module

Hyper-V VMBus generic fuzzer

**Usage:**

```
chipsec_main.py -i -m tools.vmm.hv.vmbusfuzz -a fuzz,<parameters> -l log.txt
```

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

Note: the fuzzer is incompatible with native VMBus driver (vmbus.sys). To use it, remove vmbus.sys

```
class VMBusFuzz
```

```
Bases: chipsec.modules.tools.vmm.hv.vmbus.VMBusDiscovery
```

```
hv_post_msg (message)
```

```
run (module_argv)
```

```
usage ()
```

```
vmbus_test1_run ()
```

getrandbits (k) → x. Generates an int with k random bits.

#### vbox package

#### vbox\_crash\_apicbase module

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
```

```
class vbox_crash_apicbase
```

```
Bases: chipsec.module_common.BaseModule
```

```
run (module_argv)
```

*xen package**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)
```

*hypercall module*

Xen specific hypercall functionality

```
class XenHypercall
    Bases: chipsec.modules.tools.vmm.common.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.
```

*hypercallfuzz module*

Xen hypercall fuzzer

**Usage:**

```
chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz \
-a <mode>[,<vector>,<iterations>] -l log.txt
```

- 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 sched_op,10 -l log.txt
chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a xen_version,50 -l log.txt
chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a set_timer_op,10,0x10000000 -l log.txt
```

**class HypercallFuzz**

Bases: `chipsec.module_common.BaseModule`

`get_int (arg, base=10, defvalue=10000)`

`run (module_argv)`

`usage ()`

`getrandbits (k)` → x. Generates an int with k random bits.

#### *xsa188 module*

Proof-of-concept module for Xen XSA-188 (<https://xenbits.xen.org/xsa/advisory-188.html>) CVE-2016-7154: “use after free in FIFO event channel code” Discovered by Mikhail Gorobets

This module triggers host crash on vulnerable Xen 4.4

**Usage:**

```
chipsec_main.py -m tools.vmm.xen.xsa188
```

**class xsa188**

Bases: `chipsec.module_common.BaseModule`

`run (module_argv)`

#### *common module*

Common functionality for VMM related modules/tools

**class BaseModuleDebug**

Bases: `chipsec.module_common.BaseModule`

`dbg (message)`

`err (message)`

`fatal (message)`

`hex (title, data, w=16)`

```

    info_bitwise (reg, desc)

    msg (message)

class BaseModuleHwAccess
    Bases: chipsec.modules.tools.vmm.common.BaseModuleSupport

    cpuid_info (eax, ecx, desc)

    rdmsr (msr)

    wrmsr (msr, value)

class BaseModuleSupport
    Bases: chipsec.modules.tools.vmm.common.BaseModuleDebug

    add_initial_data (vector, buffer, status)

    dump_initial_data (filename)

    get_initial_data (statuses, vector, size, padding='\x00')

    stats_event (name)

    stats_print (title)

    stats_reset ()

get_int_arg (arg)

getrandbits (k) → x. Generates an int with k random bits.

hv_hciv (rep_start, rep_count, call_code, fast=0)

overwrite (buffer, string, position)

rand_dd (n, rndbytes=1, rndbits=1)

class session_logger (log, details)
    Bases: object

    closefile ()

    write (message)

uuid (id)

weighted_choice (choices)

```

#### cpuid\_fuzz module

Simple CPUID VMM emulation fuzzer

#### Usage:

```
chipsec_main.py -i -m tools.vmm.cpuid_fuzz -l log.txt
```

```
class cpuid_fuzz
```



Bases: `chipsec.module_common.BaseModule`

`fuzz_CPUID (eax_start, random_order=False)`

`run (module_argv)`

#### *ept\_finder module*

##### Usage:

```
chipsec_main.py -i -m tools.vmm.ept_finder
```

`class c_extended_page_tables_from_file (cs, read_from_file, par)`

Bases: `chipsec.hal.paging.c_extended_page_tables`

`readmem (name, addr, size=4096)`

`class ept_finder`

Bases: `chipsec.module_common.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)`

`usage ()`

#### *hypercallfuzz module*

Pretty simple VMM hypercall fuzzer

##### Usage:

```
chipsec_main.py -i -m tools.vmm.hypercallfuzz \  
[-a <mode>,<vector_reg>,<maxval>,<iterations>] -l log.txt
```

- 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

`class hypercallfuzz`

Bases: `chipsec.module_common.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)`

`usage ()`

#### *iofuzz module*

Simple port I/O VMM emulation fuzzer

**Usage:**

```
chipsec_main.py -i -m tools.vmm.iofuzz [-a <mode>,<count>,<iterations>] -l log.txt
```

`class iofuzz`

Bases: `chipsec.module_common.BaseModule`

`fuzz_ports (iterations, write_count, random_order=False)`

`run (module_argv)`

#### *msr\_fuzz module*

Simple CPU Module Specific Register (MSR) VMM emulation fuzzer

**Usage:**

```
chipsec_main.py -i -m tools.vmm.msr_fuzz [-a random] -l log.txt
```

`class msr_fuzz`

Bases: `chipsec.module_common.BaseModule`

`fuzz_MSRS (msr_addr_start, random_order=False)`

`run (module_argv)`

#### *pcie\_fuzz module*

Simple PCIe device Memory-Mapped I/O (MMIO) and I/O ranges VMM emulation fuzzer

**Usage:**

```
chipsec_main.py -i -m tools.vmm.pcie_fuzz -l log.txt
```

`class pcie_fuzz`

Bases: `chipsec.module_common.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)

```

#### 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 -l log.txt
```

```
class pcie_overlap_fuzz
```

```
Bases: chipsec.module_common.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)

```

#### venom module

QEMU VENOM vulnerability DoS PoC test Module is based on PoC by Marcus Meissner (<https://marc.info/?l=oss-security&m=143155206320935&w=2>)

##### Usage:

```
chipsec_main.py -i -m tools.vmm.venom
```

```
class venom
```

```
Bases: chipsec.module_common.BaseModule
```

```

run (module_argv)

venom_impl ()

```

## 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 `chispec/cfg`):

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

2. Get the current status of the control:

```
ble = chipsec.chipset.get_control( self.cs, 'BiosLockEnable' )
```

3. React based on the status of the control:

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

4. Return:

```
if ble: return ModuleResult.PASSED
else: return ModuleResult.FAILED
```

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 `chipsesec_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).

## See also

[Creating CHIPSEC modules and commands](#)