

# CHIPSEC

version 1.3.7



**Platform Security Assessment Framework**

March 04, 2019



# Contents

<b>CHIPSEC</b>	<b>1</b>
<b>Installation</b>	<b>1</b>
Windows Installation	1
DAL Windows Installation	3
Linux Installation	4
MacOS Installation	5
UEFI Shell Installation	6
<b>Using CHIPSEC</b>	<b>7</b>
Command Line Usage	8
Using CHIPSEC as a Python Package	9
Writing Your Own Modules	9
<b>CHIPSEC Components and Structure</b>	<b>10</b>
Core components	11
Platform Configuration	12
OS/Environment Helpers	15
HW Abstraction Layer (HAL)	17
Utility command-line scripts	28
Auxiliary components	37
Executable build scripts	38
<b>CHIPSEC Modules</b>	<b>38</b>
Introduction	38
Modules Description	38



# CHIPSEC

Welcome to the CHIPSEC documentation!

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. When running CHIPSEC on client PC systems, Windows may be preferred. However, sometimes it may be preferable to assess platform security without interfering with the normal operating system. In these instances, CHIPSEC may be run from a bootable USB thumb drive - either a Live Linux image or a UEFI shell.

## Windows Installation

CHIPSEC supports the following versions:

- Windows 7, 8, 8.1, 10 x86 and 64-bit
- Windows Server 2008, 2012, 2016 x86 and 64-bit

NOTE: CHIPSEC has removed support for the [RWEverything](#) driver due to PCI configuration space access issues.

Please follow the steps below to install CHIPSEC framework on Windows:

1. Install [Python 2.7](#)
2. Install [pywin32](#) and `setuptools` packages:

```
pip install setuptools
```



```
pip install pypiwin32
```

To get colored console output, you may optionally want to install [WConio](#).

### 3. Clone CHIPSEC source

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

### 4. Build CHIPSEC kernel driver. Please follow the instructions in \drivers\win7\readme. Skip this step if you already have chipsec\_hlpr.sys driver binary for your version of Windows

If you already have a version of the driver you can copy the CHIPSEC driver (chipsec\_hlpr.sys) to the proper directory \chipsec\helper\win\win7\_<arch> where <arch> is "x86" or "amd64" (default path is \chipsec\helper\win\win7\_amd64)

### 5. Install CHIPSEC framework

Manually install CHIPSEC as a package:

```
python setup.py install
```

#### Note

When installing CHIPSEC on Windows, the driver isn't built automatically (as when installing on Linux). You'll need to build Windows driver and copy it to proper directory (see steps 3 and 4) prior to installing CHIPSEC

### 6. Turn off kernel driver signature checks

*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):*

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

Alternatively, disable Secure Boot in the BIOS setup screen then disable driver signature checks as 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):*

Boot in Test mode (allows self-signed certificates)

- Start CMD.EXE as Administrator
- BcdEdit /set TESTSIGNING ON
- Reboot

If that doesn't work, run these additional commands:

```
BcdEdit /set noIntegrityChecks ON
```

```
BcdEdit /set loadoptions DISABLE_INTEGRITY_CHECKS
```

Alternatively, press F8 when booting Windows and choose "No driver signatures enforcement" option to turn off driver signature checks

*Windows 10 64-bit:*

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

### 7. Run CHIPSEC

- Launch CMD.EXE as Administrator



- You can use commands below to run CHIPSEC. CHIPSEC will automatically load the driver and unload it when done.

```
python chipsec_main.py
```

```
python chipsec_util.py
```

- If CHIPSEC is used as a standalone tool, run above commands from where CHIPSEC is.

## Note

You can manually register and start CHIPSEC service/driver. CHIPSEC will attempt to connect to already running chipsec service.

To create and start chipsec service (in CMD.EXE)

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

```
sc start chipsec
```

Then to stop and delete chipsec service:

```
sc stop chipsec
```

```
sc delete chipsec
```

## DAL Windows Installation

### 1. Install Intel System Studio

Should include Python

### 2. Install pywin32 and setuptools packages:

```
pip install setuptools
```

```
pip install pypiwin32
```

### 3. Clone CHIPSEC source:

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

### 4. Install CHIPSEC support:

Manually install CHIPSEC as a package:

```
python setup.py install
```

### 5. Connect and halt platform:

Connect and set up debugger to system.

### 6. Import and run CHIPSEC main

#### 1. Launch a python Command-line.

#### 2. Import the chipsec\_main module

```
>>> import chipsec_main
```

#### 3. Run standard CHIPSEC modules:

```
>>> chipsec_main.main()
```

Example command-lines:

Run a specific module:

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



Generate a log file:

```
>>> chipsec_main.main(['-l', 'test.log'])
```

Note: the test.log file can be found in C:\intel\DAL

## 7. Import and run CHIPSEC util

1. Launch a python Command-line.

2. Import the IPC CLI module

```
>>> import ipccli
```

3. Import the chipsec\_util module

```
>>> import chipsec_util
```

4. Run CHIPSEC util and list available commands:

```
>>> chipsec_util.main()
```

Example command-lines:

Read SPI info...

```
>>> chipsec_util.main(['spi', 'info'])
```

Read MSR...

```
>>> chipsec_util.main(['msr', '0x1f2'])
```

# Linux Installation

Tested on:

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

## Installing necessary packages

You will need to install or update the following dependencies before installing CHIPSEC:

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

or

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

or

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

You can use CHIPSEC on a desired Linux distribution or create a live Linux image on a USB flash drive and boot to it. For example, you can use [liveusb-creator](#) to create live Fedora image on a USB drive

## Installing Manually

Clone chipsec Git repository and install it as a package:



## MacOS Installation

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

To use CHIPSEC *in place* without installing it:

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

## MacOS Installation

**WARNING: MacOS support is currently in Beta release**

### Install CHIPSEC Dependencies

Install XCODE from the App Store

Install Python 2.7, 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
```

### Installing CHIPSEC

Clone CHIPSEC Git repository:

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

To install and run CHIPSEC as a package:

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

To use CHIPSEC *in place* without installing it:

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

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
```

## UEFI Shell Installation

### Installing CHIPSEC for UEFI Shell

1. Extract contents of `__install__/UEFI/chipsec_uefi_<arch>.zip` to the EFI drive which can be either USB flash drive (e.g. DUET USB drive) or HDD/SSD hard drive (e.g. EFI System Partition). `<arch>` should correspond to your UEFI shell and can be `x64`, `ia32` or `i586`. This will create `/efi/Tools` directory with `Python.efi` and `/efi/StdLib` with subdirectories
2. Copy contents of CHIPSEC anywhere on the EFI drive (for example, to `chipsec` directory in root). The contents of your drive should look like follows:

```
\
  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
    ...
```

### Note

The EFI drive should already include a UEFI Shell binary in `/efi/boot`. On 64-bit platforms the shell will likely be named `bootx64.efi`

3. Run your UEFI shell

- If UEFI shell is on the USB removable drive, you'll need to boot off of the USB drive (rebooting will load UEFI shell).
- If your UEFI firmware allows booting from any file, choose to boot from your UEFI shell binary from the UEFI firmware setup options
- Some systems have embedded UEFI shell which can be booted from setup options

4. Run CHIPSEC in UEFI shell

```
1. fs0:
```

```
2. python chipsec_main.py or python chipsec_util.py
```





## **(OPTIONAL) Extending CHIPSEC functionality for UEFI**

Skip this section if you don't plan on extending native UEFI functionality for CHIPSEC.

Native functions accessing HW resources are built directly into Python UEFI port in built-in `edk2` module. If you want to add more native functionality to Python UEFI port for `chipsec`, you'll need to re-build Python for UEFI:

1. Check out [AppPkg with Python 2.7.2](#) port for UEFI from SVN
  - You'll also need to check out `StdLib` and `StdLibPrivateInternalFiles` packages from SVN
  - Alternatively download latest EADK ([EDK II Application Development Kit](#)). EADK includes `AppPkg/StdLib/StdLibPrivateInternalFiles`. Unfortunately, EADK Alpha 2 doesn't have Python 2.7.2 port so you'll need to check it out SVN.
2. Add functionality to Python port for UEFI
  - Python 2.7.2 port for UEFI is in `<UDK>\AppPkg\Applications\Python`
  - All `chipsec` related functions are in `<UDK>\AppPkg\Applications\Python\Efi\edk2module.c` (`#ifdef CHIPSEC`)
  - `Asm` functions are in `<UDK>\AppPkg\Applications\Python\Efi\cpu.asm` e.g. `<UDK>` is `C:\UDK2010.SR1`
  - Add `cpu.asm` under the `Efi` section in `PythonCore.inf`
3. Build `<UDK>/AppPkg` with Python
  - Read instructions in `<UDK>\AppPkg\ReadMe.txt` and `<UDK>\AppPkg\Applications\Python\PythonReadMe.txt`
  - Binaries of `AppPkg` and `Python` will be in `<UDK>\Build\AppPkg\DEBUG_MYTOOLS\X64\`
4. Create directories and copy Python files on DUET USB drive
  - Read instructions in `<UDK>\AppPkg\Applications\Python\PythonReadMe.txt`

## **(OPTIONAL) Building bootable USB thumb drive with UEFI Shell**

You can build 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
 

UEFI Shell <<https://github.com/tianocore/edk2/raw/master/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

## **Using CHIPSEC**

CHIPSEC should be launched as Administrator/root.

- In command shell, run
 

```
# python chipsec_main.py
```



```
# python chipsec_util.py
```

• For help, run

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

```
# python chipsec_util.py help
```

## Command Line Usage

```
usage: chipsec_main.py [options]
```

Options:

```
-h,
--help                show this message and exit
-m _MODULE,
--module _MODULE      specify module to run (example: -m common.bios_wp)
-a [_MODULE_ARGV [_MODULE_ARGV ...]],
--module_args [_MODULE_ARGV [_MODULE_ARGV ...]]
                        additional module arguments
-v,
--verbose              verbose mode
-d,
--debug                debug mode
-l LOG,
--log LOG              output to log file
```

Advanced Options:

```
-p {CFL,SNB,IVB,KBL,JKT,BYT,QRK,BDW,IVT,AVN,DNV,CHT,HSW,APL,SKL,HSX,BDX},
--platform {CFL,SNB,IVB,KBL,JKT,BYT,QRK,BDW,IVT,AVN,DNV,CHT,HSW,APL,SKL,HSX,BDX}
                        explicitly specify platform code
--pch {PCH_3XX,PCH_C620,PCH_1XX,PCH_2XX,PCH_C61X,PCH_C60X}
                        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_OUT,
--json _JSON_OUT       specify filename for JSON output
-x _XML_OUT,
--xml _XML_OUT          specify filename for xml output (JUnit style)
-t USER_MODULE_TAGS,
--moduletype USER_MODULE_TAGS
                        run tests of a specific type (tag)
--list_tags            list all the available options for -t,--moduletype
-I IMPORT_PATHS,
--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
```

Exit Code

```
-----
```

CHIPSEC returns an integer exit code:

- Exit code is 0: all modules ran successfully and passed
- Exit code is not 0: each bit means the following:
  - Bit 0: NOT IMPLEMENTED at least one module was not implemented for the platform
  - Bit 1: WARNING at least one module had a warning
  - Bit 2: DEPRECATED at least one module uses deprecated API
  - Bit 3: FAIL at least one module failed
  - Bit 4: ERROR at least one module wasn't able to run



- Bit 5: EXCEPTION at least one module threw an unexpected exception
- Bit 6: INFORMATION at least one module contained information
- Bit 7: NOT APPLICABLE at least one module was not applicable for the platform

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 if you want CHIPSEC to use native OS API rather than own kernel module. This option can also be used if loading kernel module is not needed to use desired functionality.

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`

## Using CHIPSEC as a Python Package

Install CHIPSEC manually or from PyPI as described in the Installation section.

You can then use CHIPSEC from your Python project or 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'])
```

## 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
```

When a module calls `get_control` or `set_control`, CHIPSEC will look up the control in the platform XML file, look up the corresponding register/field, and call `chipsec.chipset.read_register_field` or `chipsec.chipset.write_register_field`. This allows modules to be written for abstract *controls* that could be in different registers on different platforms.

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

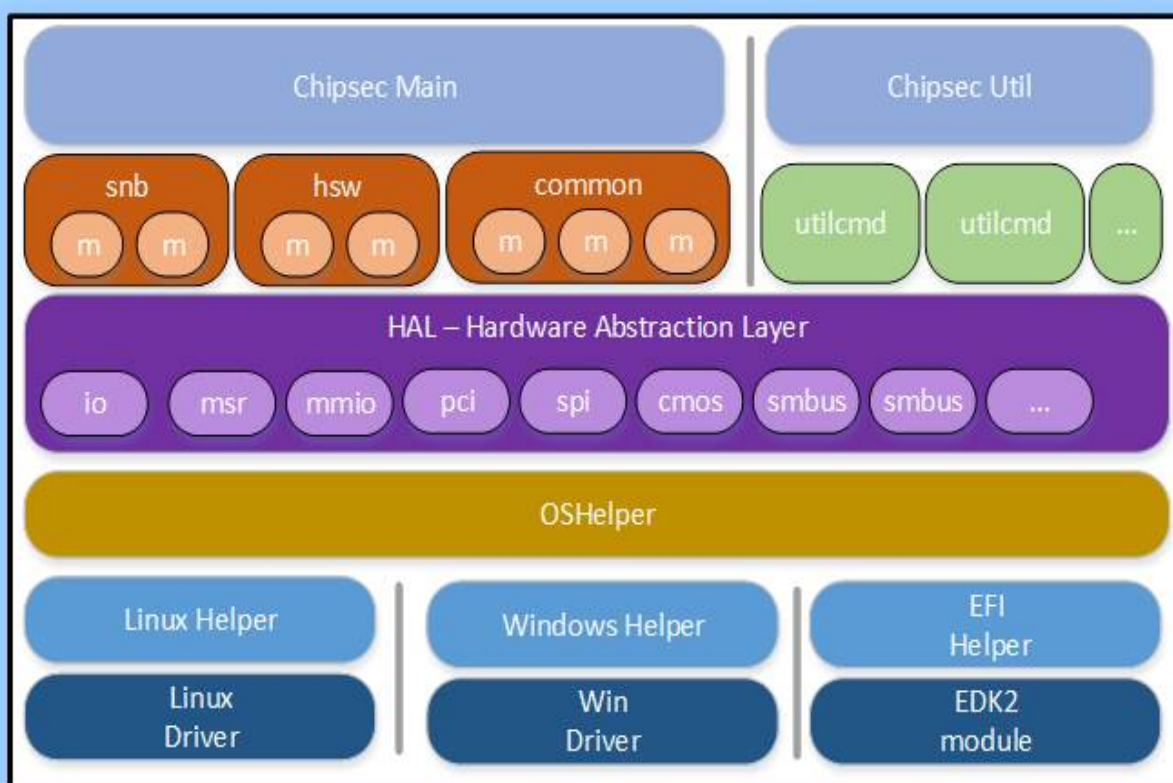
Copy your module into the `chipsec/modules/` directory structure

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

If a new platform needs to be added:

- Modify `chipsec/chipset.py` to include the Device ID for the platform you are adding
- Review the platform datasheet and include appropriate information in an XML configuration file for the platform. Place this file in `chipsec/cfg`. 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).

## CHIPSEC Components and Structure



## Core components

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



## Platform Configuration

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

### *chipsec.cfg.apl.xml*

XML configuration for Apollo Lake based SoCs

### *chipsec.cfg.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>

### *chipsec.cfg.bdw.xml*

XML configuration for Broadwell based platforms

### *chipsec.cfg.bdx.xml*

XML configuration file for Broadwell Server based platforms

### *chipsec.cfg.byx.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>

### *chipsec.cfg.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>

### *chipsec.cfg.chipsec\_cfg.xsd*



XML Schema for platform configuration XML files

### ***chipsec.cfg.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>

### ***chipsec.cfg.common.xml***

Common (default) XML platform configuration file

### ***chipsec.cfg.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>

### ***chipsec.cfg.hsw.xml***

XML configuration file for Haswell based platforms

### ***chipsec.cfg.hsx.xml***

XML configuration file for Haswell Server based platforms

### ***chipsec.cfg.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>

### ***chipsec.cfg.ivt.xml***

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



### ***chipsec.cfg.jkt.xml***

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

### ***chipsec.cfg.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

### ***chipsec.cfg.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>

### ***chipsec.cfg.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>

### ***chipsec.cfg.pch\_3xx.xml***

XML configuration file for the 300 series PCH

### ***chipsec.cfg.pch\_c60x.xml***

XML configuration file for C600 series PCH

- Intel(R) C600 Series Chipset Family Platform Controller Hub (PCH)  
<https://ark.intel.com/products/series/98463/Intel-C600-Series-Chipsets>

### ***chipsec.cfg.pch\_c61x.xml***

XML configuration file for C610 series PCH

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





## *chipsec.cfg.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>

## *chipsec.cfg.qrk.xml*

XML configuration for Quark based platforms

## *chipsec.cfg.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)

## *chipsec.cfg.template.xml*

Template for XML configuration file

# OS/Environment Helpers

## *chipsec.helper.dal.dalhelper module*

Intel DFX Abstraction Layer (DAL) helper

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

*exception* `DALHelperError`

Bases: `exceptions.RuntimeError`

## *chipsec.helper.efi.efihelper module*

On UEFI use the efi package functions

*exception* `EfiHelperError`

Bases: `exceptions.RuntimeError`



## *chipsec.helper.linux.helper module*

Linux helper

```
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.
```

## *chipsec.helper.osx.helper module*

OSX helper

## *chipsec.helper.rwe.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/>

```
class EFI_HDR_WIN
    Bases: chipsec.helper.rwe.rwehelper.EFI_HDR_WIN

class PCI_BDF
    Bases: ctypes.Structure
```

## *chipsec.helper.win.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/>

```
class EFI_HDR_WIN
    Bases: chipsec.helper.win.win32helper.EFI_HDR_WIN

class PCI_BDF
    Bases: ctypes.Structure
```

## *chipsec.helper.helpers module*



## *chipsec.helper.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`

*exception* `OsHelperError` (msg, errorcode)  
Bases: `exceptions.RuntimeError`

*exception* `UnimplementedAPIError` (api\_name)  
Bases: `chipsec.helper.oshelper.OsHelperError`

*exception* `UnimplementedNativeAPIError` (api\_name)  
Bases: `chipsec.helper.oshelper.UnimplementedAPIError`

## HW Abstraction Layer (HAL)

Components responsible for access to hardware (Hardware Abstraction Layer)

## *chipsec.hal.acpi module*

HAL component providing access to and decoding of ACPI tables

*class* `ACPI_TABLE_HEADER`  
Bases: `chipsec.hal.acpi.ACPI_TABLE_HEADER`

*exception* `AcpiRuntimeError`  
Bases: `exceptions.RuntimeError`

## *chipsec.hal.acpi\_tables module*

HAL component decoding various ACPI tables

*class* `ACPI_TABLE_APIC_GICC_CPU`  
Bases: `chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GICC_CPU`

*class* `ACPI_TABLE_APIC_GIC_DISTRIBUTOR`  
Bases: `chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GIC_DISTRIBUTOR`

*class* `ACPI_TABLE_APIC_GIC_MSI`  
Bases: `chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GIC_MSI`

*class* `ACPI_TABLE_APIC_GIC_REDISTRIBUTOR`  
Bases: `chipsec.hal.acpi_tables.ACPI_TABLE_APIC_GIC_REDISTRIBUTOR`

*class* `ACPI_TABLE_APIC_INTERRUPT_SOURCE_OVERRIDE`  
Bases: `chipsec.hal.acpi_tables.ACPI_TABLE_APIC_INTERRUPT_SOURCE_OVERRIDE`

*class* `ACPI_TABLE_APIC_IOAPIC`  
Bases: `chipsec.hal.acpi_tables.ACPI_TABLE_APIC_IOAPIC`



```
class ACPI_TABLE_APIC_IOSAPIC
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_IOSAPIC

class ACPI_TABLE_APIC_LAPIC_ADDRESS_OVERRIDE
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_LAPIC_ADDRESS_OVERRIDE

class ACPI_TABLE_APIC_LAPIC_NMI
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_LAPIC_NMI

class ACPI_TABLE_APIC_Lx2APIC_NMI
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_Lx2APIC_NMI

class ACPI_TABLE_APIC_NMI_SOURCE
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_NMI_SOURCE

class ACPI_TABLE_APIC_PLATFORM_INTERRUPT_SOURCES
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PLATFORM_INTERRUPT_SOURCES

class ACPI_TABLE_APIC_PROCESSOR_LAPIC
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PROCESSOR_LAPIC

class ACPI_TABLE_APIC_PROCESSOR_LSAPIC
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PROCESSOR_LSAPIC

class ACPI_TABLE_APIC_PROCESSOR_Lx2APIC
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_APIC_PROCESSOR_Lx2APIC

class ACPI_TABLE_DMAR_ANDD
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_ANDD

class ACPI_TABLE_DMAR_ATSR
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_ATSR

class ACPI_TABLE_DMAR_DRHD
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_DRHD

class ACPI_TABLE_DMAR_DeviceScope
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_DeviceScope

class ACPI_TABLE_DMAR_RHSA
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_RHSA

class ACPI_TABLE_DMAR_RMRR
    Bases: chipsec.hal.acpi_tables.ACPI_TABLE_DMAR_RMRR
```

## chipsec.hal.cmos module

CMOS memory specific functions (dump, read/write)

usage:

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



## HW Abstraction Layer (HAL)

*exception CmosAccessError*  
Bases: `exceptions.RuntimeError`

*exception CmosRuntimeError*  
Bases: `exceptions.RuntimeError`

### *chipsec.hal.cpu module*

CPU related functionality

*exception CPURuntimeError*  
Bases: `exceptions.RuntimeError`

### *chipsec.hal.cpuid module*

CPUID information

**usage:**

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

*exception CpuIDRuntimeError*  
Bases: `exceptions.RuntimeError`

### *chipsec.hal.ec module*

Access to Embedded Controller (EC)

**Usage:**

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

### *chipsec.hal.hal\_base module*

Base for HAL Components

### *chipsec.hal.igd module*

Working with Intel processor Integrated Graphics Device (IGD)

**usage:**

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



**exception** `IGDRuntimeError`  
Bases: `exceptions.RuntimeError`

## *chipsec.hal.interrupts module*

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

**usage:**

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

## *chipsec.hal.io module*

Access to Port I/O

**usage:**

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

**exception** `PortIORuntimeError`  
Bases: `exceptions.RuntimeError`

## *chipsec.hal.io\_bar 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 )
```

**exception** `IOBARNotFoundError`  
Bases: `exceptions.RuntimeError`

**exception** `IOBARRuntimeError`  
Bases: `exceptions.RuntimeError`

## *chipsec.hal.iommu module*

Access to IOMMU engines

**exception** `IOMMUErrror`  
Bases: `exceptions.RuntimeError`



## chipsec.hal.mmio module

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

usage:

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

Access MMIO by BAR name:

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

Access Memory Mapped Config Space:

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

DEPRECATED: Access MMIO by BAR id:

```
>>> read_MMIOBAR_reg( cs, mmio.MMIO_BAR_MCHBAR, 0x0 )
>>> write_MMIOBAR_reg( cs, mmio.MMIO_BAR_MCHBAR, 0xFFFFFFFF )
>>> get_MMIO_base_address( cs, mmio.MMIO_BAR_MCHBAR )
```

## chipsec.hal.msgbus module

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

References:

- 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>  
 (sections 3.6 and 13.4.6 - 13.4.8)
- 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 )
```

exception **MsgBusRuntimeError**

Bases: **exceptions.RuntimeError**

## chipsec.hal.msr module

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

usage:



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

**exception MsrRuntimeError**

Bases: `exceptions.RuntimeError`

## *chipsec.hal.paging module*

**exception InvalidMemoryAddress**

Bases: `exceptions.RuntimeError`

## *chipsec.hal.pci module*

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

**usage:**

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

**class EFI\_XROM\_HEADER**

Bases: `chipsec.hal.pci.EFI_XROM_HEADER`

**class PCI\_XROM\_HEADER**

Bases: `chipsec.hal.pci.PCI_XROM_HEADER`

**exception PciDeviceNotFoundError**

Bases: `exceptions.RuntimeError`

**exception PciRuntimeError**

Bases: `exceptions.RuntimeError`

**class XROM\_HEADER**

Bases: `chipsec.hal.pci.XROM_HEADER`

## *chipsec.hal.pcidb module*

PCI Vendor & Device ID data.





## Note

THIS FILE WAS GENERATED

Auto generated from:

<http://www.pcidatabase.com/vendors.php?sort=id>

<http://www.pcidatabase.com/reports.php?type=csv>

## chipsec.hal.physmem module

Access to physical memory

usage:

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

exception **MemoryAccessError**

Bases: **exceptions.RuntimeError**

exception **MemoryRuntimeError**

Bases: **exceptions.RuntimeError**

## chipsec.hal.smbus module

Access to SMBus Controller

## chipsec.hal.spd module

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

References:

[http://www.jedec.org/sites/default/files/docs/4\\_01\\_02R19.pdf](http://www.jedec.org/sites/default/files/docs/4_01_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)

<http://www.simmtester.com/page/news/showpubnews.asp?num=184>

<http://www.simmtester.com/page/news/showpubnews.asp?num=153>

<http://www.simmtester.com/page/news/showpubnews.asp?num=101>

[http://en.wikipedia.org/wiki/Serial\\_presence\\_detect](http://en.wikipedia.org/wiki/Serial_presence_detect)

class **SPD\_DDR**

Bases: **chipsec.hal.spd.SPD\_DDR**

class **SPD\_DDR2**

Bases: **chipsec.hal.spd.SPD\_DDR2**

class **SPD\_DDR3**

Bases: **chipsec.hal.spd.SPD\_DDR3**



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

## chipsec.hal.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)

```
exception SpiAccessError
    Bases: exceptions.RuntimeError
```

```
exception SpiRuntimeError
    Bases: exceptions.RuntimeError
```

## chipsec.hal.spi\_descriptor module

SPI Flash Descriptor binary parsing functionality

usage:

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

## chipsec.hal.spi\_jedec\_ids module

JEDED ID : Manufacturers and Device IDs

## chipsec.hal.spi\_uefi module

UEFI firmware image parsing and manipulation functionality

usage:

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



## chipsec.hal.tpm module

Trusted Platform Module (TPM) HAL component

<https://trustedcomputinggroup.org>

**class** `TPM_RESPONSE_HEADER`

Bases: `chipsec.hal.tpm.TPM_RESPONSE_HEADER`

**exception** `TpmRuntimeError`

Bases: `exceptions.RuntimeError`

## chipsec.hal.tpm12\_commands module

Definition for TPMv1.2 commands to use with TPM HAL

TCG PC Client TPM Specification TCG TPM v1.2 Specification

**continueselftest** (`command_argv`)

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.

**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

## chipsec.hal.tpm\_eventlog module

Trusted Platform Module Event Log

Based on the following specifications:

TCG EFI Platform Specification For TPM Family 1.1 or 1.2

[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.

```
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.

## *chipsec.hal.unicode module*

Microcode update specific functionality (for each CPU thread)

**usage:**

```
>>> unicode_update_id( 0 )
>>> load_unicode_update( 0, unicode_buf )
>>> update_unicode_all_cpus( 'unicode.pdb' )
>>> dump_unicode_update_header( 'unicode.pdb' )
```

```
class UnicodeUpdateHeader
```

Bases: `chipsec.hal.unicode.UnicodeUpdateHeader`

## *chipsec.hal.uefi module*

Main UEFI component using platform specific and common UEFI functionality

## *chipsec.hal.uefi\_common module*

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

```
class EFI_BOOT_SERVICES_TABLE
```

Bases: `chipsec.hal.uefi_common.EFI_BOOT_SERVICES_TABLE`

```
class EFI_DXE_SERVICES_TABLE
```

Bases: `chipsec.hal.uefi_common.EFI_DXE_SERVICES_TABLE`

```
class EFI_RUNTIME_SERVICES_TABLE
```

Bases: `chipsec.hal.uefi_common.EFI_RUNTIME_SERVICES_TABLE`



```
class EFI_SYSTEM_TABLE
    Bases: chipsec.hal.uefi_common.EFI_SYSTEM_TABLE

class EFI_TABLE_HEADER
    Bases: chipsec.hal.uefi_common.EFI_TABLE_HEADER

class EFI_VENDOR_TABLE
    Bases: chipsec.hal.uefi_common.EFI_VENDOR_TABLE

class VARIABLE_STORE_HEADER
    Bases: chipsec.hal.uefi_common.VARIABLE_STORE_HEADER
```

## *chipsec.hal.uefi\_platform module*

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

```
class EFI_HDR_NVAR1
    Bases: chipsec.hal.uefi_platform.EFI_HDR_NVAR1

class EFI_HDR_VSS
    Bases: chipsec.hal.uefi_platform.EFI_HDR_VSS

class EFI_HDR_VSS_APPLE
    Bases: chipsec.hal.uefi_platform.EFI_HDR_VSS_APPLE

class EFI_HDR_VSS_AUTH
    Bases: chipsec.hal.uefi_platform.EFI_HDR_VSS_AUTH

class UEFI_VARIABLE_HEADER
    Bases: chipsec.hal.uefi_platform.UEFI_VARIABLE_HEADER

UEFI_VARIABLE_STORE_HEADER_SIZE = 28
EFI_VARIABLE_HEADER_AUTH = "<HBB128sIIIHH8s" EFI_VARIABLE_HEADER_AUTH_SIZE =
    struct.calcsize(EFI_VARIABLE_HEADER_AUTH)
EFI_VARIABLE_HEADER = "<HBBIIIIHH8s" EFI_VARIABLE_HEADER_SIZE =
    struct.calcsize(EFI_VARIABLE_HEADER)

class VARIABLE_STORE_HEADER_VSS
    Bases: chipsec.hal.uefi_platform.VARIABLE_STORE_HEADER_VSS
```

## *chipsec.hal.uefi\_search module*

UEFI image search auxilliary functionality

**usage:**

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

## *chipsec.hal.virtmem module*

Access to virtual memory

**usage:**



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

exception **MemoryAccessError**

Bases: **exceptions.RuntimeError**

exception **MemoryRuntimeError**

Bases: **exceptions.RuntimeError**

## chipsec.hal.vmm module

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

exception **VMMRuntimeError**

Bases: **exceptions.RuntimeError**

## Utility command-line scripts

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

### Warning

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

### Note

All numeric values in the instructions are in hex.

## chipsec.utilcmd.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
```



## *chipsec.utilcmd.chipset\_cmd module*

usage as a standalone utility:

```
>>> chipsec_util platform
```

```
class PlatformCommand (argv, cs=None)
  Bases: chipsec.command.BaseCommand
  chipsec_util platform
```

## *chipsec.utilcmd.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 rl 0x0
>>> chipsec_util cmos wh 0x0 0xCC
```

## *chipsec.utilcmd.cpu\_cmd module*

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

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

Examples:

```
>>> chipsec_util cpu info
>>> chipsec_util cpu cr 0 0
>>> chipsec_util cpu cr 0 4 0x0
>>> chipsec_util cpu cpuid 40000000
>>> chipsec_util cpu pt
```

## *chipsec.utilcmd.decode\_cmd module*

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

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
```

## *chipsec.utilcmd.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

## *chipsec.utilcmd.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 idt|gdt|ldt [cpu_id]
```

Examples:

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

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

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

```
>>> chipsec_util idt|gdt|ldt [cpu_id]
```

Examples:

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

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

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

```
>>> chipsec_util idt|gdt|ldt [cpu_id]
```

Examples:

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

## *chipsec.utilcmd.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     <start_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
```

## chipsec.utilcmd.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
```

## chipsec.utilcmd.interrupts\_cmd module

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

Bases: **chipsec.command.BaseCommand**

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

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

Bases: **chipsec.command.BaseCommand**

```
>>> chipsec_util smi count
>>> chipsec_util smi <thread_id> <SMI_code> <SMI_data> [RAX] [RBX] [RCX] [RDX] [RSI] [RDI]
```

Examples:

```
>>> chipsec_util smi count
>>> chipsec_util smi 0x0 0xDE 0x0
>>> chipsec_util smi 0x0 0xDE 0x0 0xAAAAAAAAAAAAAAAA ..
```

## chipsec.utilcmd.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 <io_port> <width> [value]
```

Examples:

```
>>> chipsec_util io list
>>> chipsec_util io 0x61 1
>>> chipsec_util io 0x430 byte 0x0
```

## *chipsec.utilcmd.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
```

## *chipsec.utilcmd.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
>>> <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    00102030405060708090A0B0C0D0E0F
>>> chipsec_util mem allocate    0x1000
>>> chipsec_util mem pagedump    0xFED00000          0x100000
>>> chipsec_util mem search     0xF0000              0x10000  _SM_
```

## *chipsec.utilcmd.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
```

## chipsec.utilcmd.mmio\_cmd module

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

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

```
>>> chipsec_util mmio list
>>> chipsec_util mmio dump <MMIO_BAR_name>
>>> 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
```

## chipsec.utilcmd.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
```

## chipsec.utilcmd.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 0x8B 0x0 0x0 0
```

## chipsec.utilcmd.pci\_cmd module

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

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

Bases: **chipsec.command.BaseCommand**

```
>>> chipsec_util pci enumerate
>>> chipsec_util pci <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 0 0 0 0x00
>>> chipsec_util pci 0 0 0 0x88 byte 0x1A
>>> chipsec_util pci 0 0x1F 0 0xDC 1 0x1
>>> chipsec_util pci 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
```

## chipsec.utilcmd.reg\_cmd module

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

Bases: **chipsec.command.BaseCommand**

```
>>> chipsec_util reg read <reg_name>
>>> chipsec_util reg read <reg_name> <field_name>
```

Examples:

```
>>> chipsec_util reg read SMBUS_VID
>>> chipsec_util reg read HSFC FGO
```

## chipsec.utilcmd.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
```



## chipsec.utilcmd.spd\_cmd module

```
class SPDCCommand (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 read 0xA0 0x0
>>> chipsec_util spd write 0xA0 0x0 0xAA
```

## chipsec.utilcmd.spi\_cmd module

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

### Warning

Particular care must be taken when using the spi write and spi erase functions. These could make your system unbootable.

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

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

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

```
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 jedec
>>> chipsec_util spi jedec decode
```

## chipsec.utilcmd.spidesc\_cmd module

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

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

```
>>> chipsec_util spidesc [rom]
```

Examples:

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



## chipsec.utilcmd.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
```

## chipsec.utilcmd.unicode\_cmd module

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

Bases: `chipsec.command.BaseCommand`

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

Examples:

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

## chipsec.utilcmd.uefi\_cmd module

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

```
class UEFICCommand (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 <rom_file> [fwtype]
>>> 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 nvram uefi.rom vss_auth
>>> chipsec_util uefi keys db.bin
```



```
>>> chipsec_util uefi tables
>>> chipsec_util uefi s3bootscript
>>> chipsec_util uefi assemble AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE freeform lzma uefi.raw mydriver.efi
>>> chipsec_util uefi replace AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE bios.bin new_bios.bin mydriver.efi
```

## chipsec.utilcmd.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 pagedump 0xFED00000          0x100000
>>> chipsec_util vmem search 0xF0000               0x10000    _SM_
>>> chipsec_util vmem getphys 0xFED00000
```

## chipsec.utilcmd.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
```

## 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. Consult the documentation for an individual module for more information.

## Modules Description

### *chipsec.modules.common.cpu.spectre\_v2 module*

The module checks if system includes hardware mitigations for Speculative Execution Side Channel. Specifically, it verifies that the system supports CPU mitigations for Branch Target Injection vulnerability a.k.a. Spectre Variant 2 (CVE-2017-5715)

The module checks if the following hardware mitigations are supported by the CPU and enabled by the OS/software:

1. Indirect Branch Restricted Speculation (IBRS) and Indirect Branch Predictor Barrier (IBPB):  
CPUID.(EAX=7H,ECX=0):EDX[26] == 1
2. Single Thread Indirect Branch Predictors (STIBP): CPUID.(EAX=7H,ECX=0):EDX[27] == 1  
IA32\_SPEC\_CTRL[STIBP] == 1
3. Enhanced IBRS: CPUID.(EAX=7H,ECX=0):EDX[29] == 1 IA32\_ARCH\_CAPABILITIES[IBRS\_ALL] == 1  
IA32\_SPEC\_CTRL[IBRS] == 1
4. @TODO: Mitigation for Rogue Data Cache Load (RDCL): CPUID.(EAX=7H,ECX=0):EDX[29] == 1  
IA32\_ARCH\_CAPABILITIES[RDCL\_NO] == 1





## Executable build scripts

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:



## Executable build scripts

- 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>

### *chipsec.modules.common.secureboot.variables module*

UEFI 2.4 spec Section 28

Verify that all Secure Boot key/whitelist/blacklist 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.

### *chipsec.modules.common.uefi.access\_uefispec module*

Checks protection of UEFI variables defined in the UEFI spec to have certain permissions.

Returns failure if variable attributes are not as defined in table 11 "Global Variables" of the UEFI spec.

### *chipsec.modules.common.uefi.s3bootscript module*

Checks protections of the S3 resume boot-script implemented by the UEFI based firmware

References:

VU#976132 UEFI implementations do not properly secure the EFI S3 Resume Boot Path boot script

Technical Details of the S3 Resume Boot Script Vulnerability by Intel Security's Advanced Threat Research team.

Attacks on UEFI Security by Rafal Wojtczuk and Corey Kallenberg.

Attacking UEFI Boot Script by Rafal Wojtczuk and Corey Kallenberg.

Exploiting UEFI boot script table vulnerability by Dmytro Oleksiuk.

Usage:

```
>>> chipsec_main.py -m common.uefi.s3bootscript [-a <script_address>]
```

Examples:

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

### *chipsec.modules.common.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.



## Executable build scripts

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

### ***chipsec.modules.common.bios\_smi module***

The module checks that SMI events configuration is locked down - Global SMI Enable/SMI Lock - TCO SMI Enable/TCO Lock

References:

[Setup for Failure: Defeating SecureBoot](#) by Corey Kallenberg, Xeno Kovah, John Butterworth, Sam Cornwell

[Summary of Attacks Against BIOS and Secure Boot](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) (<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>)

### ***chipsec.modules.common.bios\_ts module***

Checks for BIOS Interface Lock including Top Swap Mode

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

### ***chipsec.modules.common.bios\_wp module***

The BIOS region in flash can be protected either using SMM-based protection or using configuration in the SPI controller. However, the SPI controller configuration is set once and locked, which would prevent writes later.

This module 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.

### ***chipsec.modules.common.ia32cfg module***

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

Reference: Intel Software Developer's Manual



## chipsec.modules.common.me\_mfg\_mode module

This module checks that ME Manufacturing mode is not enabled

References:

<https://blog.ptsecurity.com/2018/10/intel-me-manufacturing-mode-macbook.html>

[https://github.com/coreboot/coreboot/blob/master/src/soc/intel/\\*/include/soc/pci\\_devs.h](https://github.com/coreboot/coreboot/blob/master/src/soc/intel/*/include/soc/pci_devs.h)

Code:

```
#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>

Code:

```
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");
```

[https://github.com/coreboot/coreboot/blob/master/src/southbridge/intel\\*/pch.h](https://github.com/coreboot/coreboot/blob/master/src/southbridge/intel*/pch.h)

Code:

```
#define PCH_ME_DEV                PCI_DEV(0, 0x16, 0)
```

[https://github.com/coreboot/coreboot/blob/master/src/southbridge/intel\\*/me.h](https://github.com/coreboot/coreboot/blob/master/src/southbridge/intel*/me.h)

Code:

```
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;
```

[https://github.com/coreboot/coreboot/blob/master/src/southbridge/intel\\*/me\\_status.c](https://github.com/coreboot/coreboot/blob/master/src/southbridge/intel*/me_status.c)

Code:

```
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



### ***chipsec.modules.common.memlock module***

This module checks if memory configuration is locked to protect SMM

Reference: [https://github.com/coreboot/coreboot/blob/master/src/cpu/intel/model\\_206ax/finalize.c](https://github.com/coreboot/coreboot/blob/master/src/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

### ***chipsec.modules.common.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.

### ***chipsec.modules.common.sgx\_check module***

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

### ***chipsec.modules.common.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.

### ***chipsec.modules.common.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.

### ***chipsec.modules.common.spi\_access module***

Checks SPI Flash Region Access Permissions programmed in the Flash Descriptor



### ***chipsec.modules.common.spi\_desc module***

The SPI Flash Descriptor indicates read/write permissions for devices to access regions of the flash memory. This module simply reads the Flash Descriptor and checks that software cannot modify the Flash Descriptor itself. If software can write to the Flash Descriptor, then software could bypass any protection defined by it. While often used for debugging, this should not be the case on production systems.

This module checks that software cannot write to the flash descriptor.

### ***chipsec.modules.common.spi\_fdopss module***

Checks for SPI Controller Flash Descriptor Security Override Pin Strap (FDOPSS). On some systems, this may be routed to a jumper on the motherboard.

### ***chipsec.modules.common.spi\_lock module***

The configuration of the SPI controller, including protected ranges (PR0-PR4), is locked by HSFS[FLOCKDN] until reset. If not locked, the controller configuration may be bypassed by reprogramming these registers.

This vulnerability (not setting FLOCKDN) is also checked by other tools, including [flashrom](#) and Copernicus by MITRE (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.

### ***chipsec.modules.tools.cpu.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).

### ***chipsec.modules.tools.secureboot.te module***

Tool to test for 'TE Header' vulnerability in Secure Boot implementations as described in [All Your Boot Are Belong To Us](#)

**Usage:**

```
chipsec_main.py -m tools.secureboot.te [-a <mode>,<cfg_file>,<efi_file>]
```

- <mode>



- `generate_te` (default) convert PE EFI binary `<efi_file>` to TE binary
- `replace_bootloader` replace bootloader files listed in `<cfg_file>` on ESP with modified `<efi_file>`
- `restore_bootloader` restore original bootloader files from `.bak` files
- `<cfg_file>` path to config file listing paths to bootloader files to replace
- `<efi_file>` path to EFI binary to convert to TE binary. If no file path is provided, the tool will look for `Shell.efi`

Examples:

Convert `Shell.efi` PE/COFF EFI executable to TE executable:

```
chipsec_main.py -m tools.secureboot.te -a generate_te,Shell.efi
```

Replace bootloaders listed in `te.cfg` file with TE version of `Shell.efi` executable:

```
chipsec_main.py -m tools.secureboot.te -a replace_bootloader,te.cfg,Shell.efi
```

Restore bootloaders listed in `te.cfg` file:

```
chipsec_main.py -m tools.secureboot.te -a restore_bootloader,te.cfg
```

## *chipsec.modules.tools.smm.rogue\_mmio\_bar module*

Experimental module that may help checking SMM firmware for MMIO BAR hijacking vulnerabilities described in the following presentation:

[BARing the System: New vulnerabilities in Coreboot & UEFI based systems](#) by Intel Advanced Threat Research team at RECon Brussels 2017

**Usage:**

```
chipsec_main -m tools.smm.rogue_mmio_bar [-a <smi_start:smi_end>,<b:d.f>]
```

- `smi_start:smi_end`: range of SMI codes (written to IO port 0xB2)
- `b:d.f`: PCIe bus/device/function in b:d.f format (in hex)

**Example:**

```
>>> chipsec_main.py -m tools.smm.rogue_mmio_bar -a 0x00:0x80
>>> chipsec_main.py -m tools.smm.rogue_mmio_bar -a 0x00:0xFF,0:1C.0
```

## *chipsec.modules.tools.smm.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)



## Executable build scripts

- 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: **exceptions.RuntimeError**

## chipsec.modules.tools.uefi.blacklist module

This module checks current contents of UEFI firmware ROM or specified firmware image for black-listed 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.blacklist [-a <fw_image>,<blacklist>]
```

- fw\_image Full file path to UEFI firmware image. If not specified, the module will dump firmware image directly from ROM
- blacklist JSON file with configuration of black-listed EFI binaries (default = blacklist.json). Config file should be located in the same directory as this module

**Examples:**

```
>>> chipsec_main.py -m tools.uefi.blacklist
```

Dumps UEFI firmware image from flash memory device, decodes it and checks for black-listed EFI modules defined in the default config blacklist.json

```
>>> chipsec_main.py -i --no_driver -m tools.uefi.blacklist -a uefi.rom,blacklist.json
```

Decodes uefi.rom binary with UEFI firmware image and checks for black-listed EFI modules defined in blacklist.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

## *chipsec.modules.tools.uefi.s3script\_modify module*

This module will attempt to modify the S3 Boot Script on the platform. Doing this could cause the platform to malfunction. Use with care!

### Usage:

Replacing existing opcode:

```
chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,<reg_opcode>,<address>,<value>
    <reg_opcode> = pci_wr|mmio_wr|io_wr|pci_rw|mmio_rw|io_rw

chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,mem[,<address>,<value>]

chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,dispatch``

chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,dispatch_ep``
```

Adding new opcode:

```
chipsec_main.py -m tools.uefi.s3script_modify -a add_op,<reg_opcode>,<address>,<value>,<width>
    <reg_opcode> = pci_wr|mmio_wr|io_wr

chipsec_main.py -m tools.uefi.s3script_modify -a add_op,dispatch[,<entrypoint>]
```

### Examples:

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,<reg_opcode>,<address>,<value>
>>> <reg_opcode> = pci_wr|mmio_wr|io_wr|pci_rw|mmio_rw|io_rw
```

The option will look for a script opcode that writes to PCI config, MMIO or I/O registers and modify the opcode to write the given value to the register with the given address.

After executing this, if the system is vulnerable to boot script modification, the hardware configuration will have changed according to given `<reg_opcode>`.

```
>>> chipsec_main.py -m tools.uefi.s3script_modify -a replace_op,mem
```

The option will look for a script opcode that writes to memory and modify the opcode to write the given value to the given address.

By default this test will allocate memory and write `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.

## *chipsec.modules.tools.uefi.uefivar\_fuzz module*

The module is fuzzing UEFI Variable interface.

The module is using UEFI SetVariable interface to write new UEFI variables to SPI flash NVRAM with randomized name/attributes/GUID/data/size.

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
```

## *chipsec.modules.tools.uefi.whitelist 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/whitelisted] executables

Usage:

```
chipsec_main -m tools.uefi.whitelist [-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 white-listed 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.whitelist
```



## Executable build scripts

Creates a list of EFI executable binaries in `efilist.json` from the firmware image extracted from ROM

```
>>> chipsec_main -i -n -m tools.uefi.whitelist -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.whitelist -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

## *chipsec.modules.tools.vmm.hv.define module*

Hyper-V specific defines

## *chipsec.modules.tools.vmm.hv.hypercall module*

Hyper-V specific hypercall functionality

**getrandbits** (k) → x. Generates a long int with k random bits.

## *chipsec.modules.tools.vmm.hv.hypercallfuzz module*

Hyper-V hypercall fuzzer

### Usage:

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

- mode fuzzing mode
  - = status-fuzzing finding parameters with hypercall success status
  - = params-info shows input parameters valid ranges
  - = params-fuzzing parameters fuzzing based on their valid ranges
  - = custom-fuzzing fuzzing of known hypercalls
- vector hypercall vector
- iterations number of hypercall iterations

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

**getrandbits** (k) → x. Generates a long int with k random bits.

## *chipsec.modules.tools.vmm.hv.synth\_dev module*

Hyper-V VMBus synthetic device generic fuzzer

### Usage:

Print channel offers:

```
chipsec_main.py -i -m tools.vmm.hv.synth_dev -a info
```



## Executable build scripts

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

### *chipsec.modules.tools.vmm.hv.synth\_kbd module*

Hyper-V VMBus synthetic keyboard fuzzer. Fuzzes inbound ring buffer in VMBus virtual keyboard device.

#### Usage:

```
chipsec_main.py -i -m tools.vmm.hv.synth_kbd -a fuzz -l log.txt
```

Note: the fuzzer is incompatible with native VMBus driver (`vmbus.sys`). To use it, remove `vmbus.sys`  
`getrandbits` (`k`) → `x`. Generates a long int with `k` random bits.

### *chipsec.modules.tools.vmm.hv.vmbus module*

Hyper-V VMBus functionality

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

### *chipsec.modules.tools.vmm.hv.vmbusfuzz module*

Hyper-V VMBus generic fuzzer

#### Usage:

```
chipsec_main.py -i -m tools.vmm.hv.vmbusfuzz -a fuzz,<parameters> -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`  
`getrandbits` (`k`) → `x`. Generates a long int with `k` random bits.

### *chipsec.modules.tools.vmm.vbox.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
```

### *chipsec.modules.tools.vmm.xen.define module*



Executable build scripts

Xen specific defines

## *chipsec.modules.tools.vmm.xen.hypercall module*

Xen specific hypercall functionality

**getrandbits** (k) → x. Generates a long int with k random bits.

## *chipsec.modules.tools.vmm.xen.hypercallfuzz module*

Xen hypercall fuzzer

### Usage:

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

**getrandbits** (k) → x. Generates a long int with k random bits.

## *chipsec.modules.tools.vmm.xen.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
```

## *chipsec.modules.tools.vmm.common module*

Common functionality for VMM related modules/tools

**getrandbits** (k) → x. Generates a long int with k random bits.



## ***chipsec.modules.tools.vmm.cpuid\_fuzz module***

Simple CPUID VMM emulation fuzzer

### **Usage:**

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

## ***chipsec.modules.tools.vmm.hypercallfuzz module***

Pretty simple VMM hypercall fuzzer

### **Usage:**

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

## ***chipsec.modules.tools.vmm.iofuzz module***

Simple port I/O VMM emulation fuzzer

### **Usage:**

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

## ***chipsec.modules.tools.vmm.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
```

## ***chipsec.modules.tools.vmm.pcie\_fuzz module***

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

### **Usage:**

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

## ***chipsec.modules.tools.vmm.pcie\_overlap\_fuzz module***



## Executable build scripts

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
```

## *chipsec.modules.tools.vmm.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
```

## *chipsec.modules.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]

## *chipsec.modules.memconfig module*

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

## *chipsec.modules.remap module*

Preventing & Detecting Xen Hypervisor Subversions by Joanna Rutkowska & Rafal Wojtczuk

Check Memory Remapping Configuration

## *chipsec.modules.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*



Executable build scripts

*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.