CHIPSEC

version 1.7.3

Platform Security Assesment Framework

October 27, 2021

Contents

CHIPSEC

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

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

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

Warning

Chipsec should only be used on test systems!

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

There are multiple reasons for that:

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

1. 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.

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

Contact

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

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

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

Mailing lists:

- CHIPSEC users: https://groups.google.com/forum/#!forum/chipsec-users
- CHIPSEC discussion list on 01.org: https://lists.01.org/hyperkitty/list/chipsec@lists.01.org/

Twitter:

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

Download CHIPSEC

GitHub repository

CHIPSEC source files are maintained in a GitHub repository:

GitHub Repo

Releases

You can find the latest release here:

Latest Release

Older releases can be found here

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

Python

Python downloads:

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

Installation

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

DAL Windows Installation

Prerequisites

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

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

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

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

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

Building

Clone CHIPSEC source

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

Linux Installation

Tested on:

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

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

Prerequisites

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

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

Install or update necessary dependencies before installing CHIPSEC:

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

or

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

or

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

Install setuptools package:

```
pip install setuptools
```

Building

Clone CHIPSEC source

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

Build the Driver and Compression Tools

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

Creating a Live Linux image

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

MacOS Installation

Warning

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

Install CHIPSEC Dependencies

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

Note

CHIPSEC has deprecated support for Python2 since June 2020

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

Install PIP and setuptools packages. Please see instructions here

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

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

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

Building Chipsec

Clone CHIPSEC Git repository:

Download CHIPSEC

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

Run CHIPSEC

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

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

CHIPSEC Cleanup

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

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

Build Errors

xcodebuild requires xcode error during CHIPSEC install:

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

Windows Installation

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

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

Install CHIPSEC Dependencies

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

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

To install requirements:

pip install -r windows_requirements.txt

which includes:

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

To compile the driver:

Visual Studio and WDK: for building the driver. For best results use the latest available (at least WDK 8 and VS 2012)

To clone the repo:

git: open source distributed version control system

Building

Clone CHIPSEC source

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

Build the Driver and Compression Tools

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

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

Turn off kernel driver signature checks

Windows 10 64-bit

In CMD shell:

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

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

Method 1:

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

Method 2:

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

Windows 10 (with Secure Boot disabled) / Windows 8 (with Secure Boot disabled) / Windows Server 2012 (with Secure Boot disabled):

Method 1:

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

Method 2:

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

Alternate Build Methods

Build CHIPSEC kernel driver with Visual Studio

Method 1:

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

Method 2:

- Open a VS developer command prompt
- `> cd <CHIPSEC_ROOT_DIR>\drivers\win7`
- **Build driver using msbuild command:**
  - **For 32 bit:**
    - `> msbuild`
  - **For 64 bit:**
    - `> msbuild /p:Platform=x64`

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

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

Build the compression tools

Method 1:

- Navigate to the chipsec_toolscompression directory
- run the build.cmd

Method 2:

- Download compression tools from https://github.com/tianocore/edk2-BaseTools-win32/archive/master.zip
- Unzip the archive into the chipsec_tools/compression/bin directory

Alternate Method to load CHIPSEC service/driver

To create and start CHIPSEC service

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

When finished running CHIPSEC stop/delete service:

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

Building a Bootable USB drive with UEFI Shell (x64)

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

Installing CHIPSEC

1. Extract the contents of `__install__/UEFI/chipsec_uefi_[x64|i586|IA32].zip` to the USB drive, as appropriate.
   - This will create a /efi/Tools directory with Python.efi and /efi/StdLib with subdirectories for dependencies.
2. Copy the contents of CHIPSEC to the USB drive.
   - The contents of your drive should look like follows:

```
-   fs0:
    -   efi
        -   boot
            -   bootx64.efi
        -   StdLib
            -   lib
                -   python.27
                    -   [lots of python files and directories]
        -   Tools
            -   Python.efi
    -   chipsec
        -   chipsec
            -   …
        -   chipsec_main.py
        -   chipsec_util.py
        -   …
```

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

Run CHIPSEC in UEFI Shell

```
fs0:
```

```
cd chipsec
```

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

Creating the Kali Linux Live USB

Download and install Kali Linux

Installing CHIPSEC

Install the dependencies

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

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

```
pip install setuptools
```

Install latest CHIPSEC release from PyPI repository

```
pip install chipsec
```

Install CHIPSEC package from latest source code

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

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

Run CHIPSEC

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

Using CHIPSEC

CHIPSEC should be launched as Administrator/root

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

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

Interpreting results

Note

DRAFT (work in progress)

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

Results

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

Generic results meanings

Result	Meaning
PASSED	A **mitigation** to a known vulnerability has been detected
FAILED	A known **vulnerability** has been detected
WARNING	We have detected something that could be a vulnerability but **manual analysis is required** to confirm (inconclusive)
SKIPPED NOT IMPLEMENTED	CHIPSEC currently has not implemented support for this test on this platform
SKIPPED NOT APPLICABLE	The issue checked by this module is not applicable to this platform. This result can be ignored
INFORMATION	This module does not check for a vulnerability. It just prints information about the system
ERROR	Something went wrong in the execution of CHIPSEC
DEPRECATED	At least one module uses deprecated API
EXCEPTION	At least one module threw an unexpected exception

Automated Tests

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

Modules results meanings

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

Remap	Memory Remap is configured correctly and locked	Memory Remap is not properly configured/locked. Remaping attack may be possible.	N/A	
smm_dma	TSEG is properly configured. SMRAM is protected from DMA attacks.	TSEG is properly configured, but the configuration is not locked or TSEG is not properly configured. Portions of SMRAM may be vulnerable to DMA attacks	TSEG is properly configured but can't determine if it covers entire SMRAM	
common.bios_kbrd_buffer	"Keyboard buffer is filled with common fill pattern" or "Keyboard buffer looks empty. Pre-boot passwords don't seem to be exposed	FAILED	Keyboard buffer is not empty. The test cannot determine conclusively if it contains pre-boot passwords.n The contents might have not been cleared by pre-boot firmware or overwritten with garbage.n Visually inspect the contents of keyboard buffer for pre-boot passwords (BIOS, HDD, full-disk encryption).	Also printing a message if size of buffer is revealed. "Was your password %d characters long?"
common.bios_smi	All required SMI sources seem to be enabled and locked	Not all required SMI sources are enabled and locked	Not all required SMI sources are enabled and locked, but SPI flash writes are still restricted to SMM	
common.bios_ts	BIOS Interface is locked (including Top Swap Mode)	BIOS Interface is not locked (including Top Swap Mode)	N/A	
common.bios_wp	BIOS is write protected	BIOS should enable all available SMM based write protection mechanisms or configure SPI protected ranges to protect the entire BIOS region. BIOS is NOT protected completely	N/A	
common.ia32cfg	IA32_FEATURE_CONTROL MSR is locked on all logical CPUs	IA32_FEATURE_CONTROL MSR is not locked on all logical CPUs	N/A	

common.rtclock	Protected locations in RTC memory are locked	N/A	Protected locations in RTC memory are accessible (BIOS may not be using them)	
common.smm	Compatible SMRAM is locked down	Compatible SMRAM is not properly locked. Expected (D_LCK = 1, D_OPEN = 0)	N/A	Should return SKIPPED_NOT_APPLICABLE when compatible SMRAM is not enabled.
common.smrr	SMRR protection against cache attack is properly configured	SMRR protection against cache attack is not configured properly	N/A	
common.spi_access	SPI Flash Region Access Permissions in flash descriptor look ok	SPI Flash Region Access Permissions are not programmed securely in flash descriptor	Software has write access to GBe region in SPI flash" and "Certain SPI flash regions are writeable by software	we have observed production systems reacting badly when GBe was overwritten
common.spi_desc	SPI flash permissions prevent SW from writing to flash descriptor	SPI flash permissions allow SW to write flash descriptor	N/A	we can probably remove this now that we have spi_access
common.spi_fdopss	SPI Flash Descriptor Security Override is disabled	SPI Flash Descriptor Security Override is enabled	N/A	
common.spi_lock	SPI Flash Controller configuration is locked	SPI Flash Controller configuration is not locked	N/A	
common.cpu.spectre_v2	CPU and OS support hardware mitigations (enhanced IBRS and STIBP)	CPU mitigation (IBRS) is missing	CPU supports mitigation (IBRS) but doesn't support enhanced IBRS" or "CPU supports mitigation (enhanced IBRS) but OS is not using it" or "CPU supports mitigation (enhanced IBRS) but STIBP is not supported/enabled	
common.secureboot.variables	All Secure Boot UEFI variables are protected	Not all Secure Boot UEFI variables are protected' (failure when secure boot is enabled)	Not all Secure Boot UEFI variables are protected' (warning when secure boot is disabled)	
common.uefi.access_uefispec	All checked EFI variables are protected according to spec	Some EFI variables were not protected according to spec	Extra/Missing attributes	

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

Tools

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

Running CHIPSEC

CHIPSEC should be launched as Administrator/root.

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

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

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

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

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

```
# python chipsec_util.py
```

Running in Shell

Basic usage

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

For help, run

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

Using as a Python Package

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

To install and run CHIPSEC as a package:

```
# python setup.py install
```

```
# sudo chipsec_main
```

From the Python shell:

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

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

To use CHIPSEC *in place* without installing it:

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

```
# sudo python chipsec_main.py
```

chipsec_main options

```
usage: chipsec_main.py [options]

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

Advanced Options:
  -p, --platform _PLATFORM              explicitly specify platform code
  --pch _PCH                            explicitly specify PCH code
  -n, --no_driver                       chipsec won't need kernel mode functions so don't load chipsec driver
  -i, --ignore_platform                 run chipsec even if the platform is not recognized
  -j, --json _JSON_OUT                  specify filename for JSON output
  -x, --xml _XML_OUT                    specify filename for xml output (JUnit style)
  -k, --markdown                        specify filename for markdown output
  -t, --moduletype USER_MODULE_TAGS     run tests of a specific type (tag)
  --list_tags                           list all the available options for -t,--moduletype
  -I, --include IMPORT_PATHS            specify additional path to load modules from
  --failfast                            fail on any exception and exit (don't mask exceptions)
  --no_time                             don't log timestamps
  --deltas _DELTAS_FILE                 specifies a JSON log file to compute result deltas from
  --record _TO_FILE                     run chipsec and clone helper results into JSON file
  --replay _FROM_FILE                   replay a chipsec run with JSON file
  --helper _HELPER                      specify OS Helper
  -nb, --no_banner                      chipsec won't display banner information
  --skip_config                         skip configuration and driver loading
```

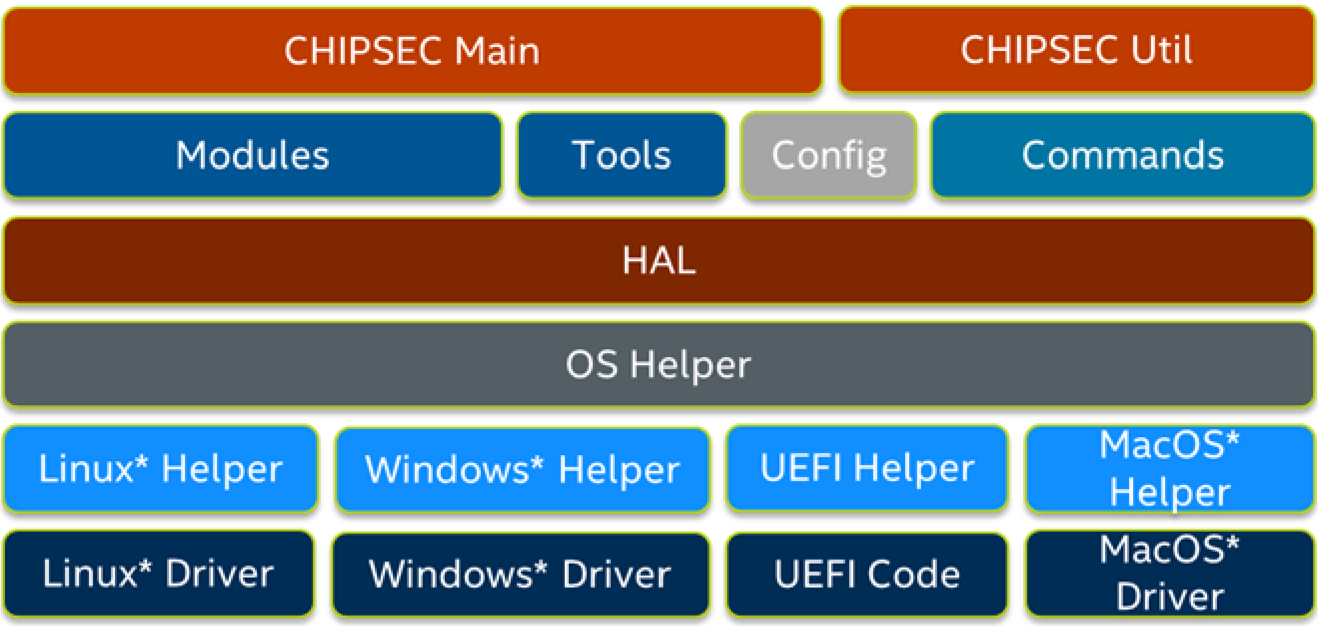
chipsec_util options

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

Options:
  -h, --help                  show this message and exit
  -v, --verbose               verbose mode
  --hal                       HAL mode
  -d, --debug                 debug mode
  -l, --log  LOG              output to log file
  -p, --platform _PLATFORM    explicitly specify platform code
  --pch _PCH                  explicitly specify PCH code
  -n, --no_driver             chipsec won't need kernel mode functions so don't load chipsec driver
  -i, --ignore_platform       run chipsec even if the platform is not recognized
  Command _CMD                Util command to run
  Command _ARGS               All numeric values are in hex <width> is in {1 - byte, 2 - word, 4 - dword}
```

Module & Command Development

Architecture Overview



CHIPSEC Architecture

Core components

`chipsec_main.py`	main application logic and automation functions
`chipsec_util.py`	utility functions (access to various hardware resources)
`chipsec/chipset.py`	chipset detection
`chipsec/command.py`	base class for util commands

`chipsec/defines.py`	common defines
`chipsec/file.py`	reading from/writing to files
`chipsec/logger.py`	logging functions
`chipsec/module.py`	generic functions to import and load modules
`chipsec/module_common.py`	base class for modules
`chipsec/result_deltas.py`	supports checking result deltas between test runs
`chipsec/testcase.py`	support for XML and JSON log file output
`chipsec/helper/helpers.py`	registry of supported OS helpers
`chipsec/helper/oshelper.py`	OS helper: wrapper around platform specific code that invokes kernel driver

Commands

Implement functionality of chipsec_util

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

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

> **Note**
>
> All numeric values in the instructions are in hex.

utilcmd package

acpi_cmd module

Command-line utility providing access to ACPI tables

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

Examples:

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

chipset_cmd module

usage as a standalone utility:

```
>>> chipsec_util platform
```

cmos_cmd module

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

Examples:

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

config_cmd module

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

Examples:

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

cpu_cmd module

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

Examples:

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

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.

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

For a list of fw types run:

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

Examples:

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

deltas_cmd module

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

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

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

desc_cmd module

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

IDT command:

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

Examples:

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

GDT command:

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

Examples:

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

LDT command:

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

Examples:

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

ec_cmd module

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

Examples:

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

igd_cmd module

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

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

Examples:

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

interrupts_cmd module

SMI command:

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

Examples:

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

NMI command:

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

io_cmd module

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

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

Examples:

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

iommu_cmd module

Command-line utility providing access to IOMMU engines

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

Examples:

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

lock_check_cmd module

mem_cmd module

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

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

Examples:

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

mmcfg_base_cmd module

The mmcfg_base command displays PCIe MMCFG Base/Size.

Usage:

```
>>> chipsec_util mmcfg_base
```

Examples:

```
>>> chipsec_util mmcfg_base
```

mmcfg_cmd module

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

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

Examples:

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

mmio_cmd module

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

Examples:

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

msgbus_cmd module

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

Examples:

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

msr_cmd module

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

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

Examples:

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

pci_cmd module

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

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

Examples:

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

reg_cmd module

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

Examples:

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

smbios_cmd module

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

Examples:

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

smbus_cmd module

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

Examples:

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

spd_cmd module

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

Examples:

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

spi_cmd module

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

Warning

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

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

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

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

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

Examples:

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

spidesc_cmd module

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

Examples:

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

tpm_cmd module

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

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

Examples:

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

ucode_cmd module

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

Examples:

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

uefi_cmd module

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

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

Examples:

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

vmem_cmd module

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

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

Examples:

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

vmm_cmd module

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

Examples:

```
>>> chipsec_util vmm hypercall 32 0 0 0 0 0
>>> chipsec_util vmm pt 0x524B01E
```

```
>>> chipsec_util vmm virtio
>>> chipsec_util vmm virtio 0:6.0
```

HAL (Hardware Abstraction Layer)

Useful abstractions for common tasks such as accessing the SPI

hal package

acpi module

HAL component providing access to and decoding of ACPI tables

acpi_tables module

HAL component decoding various ACPI tables

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 )
```

cpu module

CPU related functionality

cpuid module

CPUID information

usage:

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

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 )
```

hal_base module

Base for HAL Components

igd module

Working with Intel processor Integrated Graphics Device (IGD)

usage:

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

interrupts module

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

usage:

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

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 )
```

iobar module

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

usage:

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

iommu module

Access to IOMMU engines

locks module

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 )
```

msgbus module

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

References:

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

usage:

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

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()
```

paging module

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

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 )
```

pcidb module

PCI Vendor & Device ID data.

> ### Note
>
> THIS FILE WAS GENERATED
>
> Auto generated from:
>
> https://github.com/pciutils/pciids

physmem module

Access to physical memory

usage:

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

smbios module

HAL component providing access to and decoding of SMBIOS structures

smbus module

Access to SMBus Controller

spd module

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

References:

http://www.jedec.org/sites/default/files/docs/4_01_02R19.pdf
http://www.jedec.org/sites/default/files/docs/4_01_02_10R17.pdf
http://www.jedec.org/sites/default/files/docs/4_01_02_11R24.pdf
http://www.jedec.org/sites/default/files/docs/4_01_02_12R23A.pdf
https://www.simmtester.com/News/PublicationArticle/184 https://www.simmtester.com/News/PublicationArticle/153
https://www.simmtester.com/News/PublicationArticle/101 http://en.wikipedia.org/wiki/Serial_presence_detect

spi module

Access to SPI Flash parts

usage:

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

Note

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

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

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

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

spi_descriptor module

SPI Flash Descriptor binary parsing functionality

usage:

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

spi_jedec_ids module

JEDED ID : Manufacturers and Device IDs

spi_uefi module

UEFI firmware image parsing and manipulation functionality

usage:

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

tpm module

Trusted Platform Module (TPM) HAL component

https://trustedcomputinggroup.org

tpm12_commands module

Definition for TPMv1.2 commands to use with TPM HAL

TCG PC Client TPM Specification TCG TPM v1.2 Specification

tpm_eventlog module

Trusted Platform Module Event Log

Based on the following specifications:

TCG EFI Platform Specification For TPM Family 1.1 or 1.2

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

TCG EFI Protocol Specification, Family "2.0"

TCG PC Client Platform Firmware Profile Specification

ucode module

Microcode update specific functionality (for each CPU thread)

usage:

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

uefi module

Main UEFI component using platform specific and common UEFI functionality

uefi_common module

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

uefi_fv module

UEFI Firmware Volume Parsing/Modification Functionality

uefi_platform module

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

uefi_search module

UEFI image search auxilliary functionality

usage:

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

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 )
```

vmm module

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

Fuzzing

fuzzing package

primitives module

CHIPSEC_MAIN Program Flow

1. Select OS Helpers and Drivers
    - Load Driver (optional)
2. Detect Platform
3. Load Configuration Files

4. Load Modules
5. Run Loaded Modules
6. Report Results
7. Cleanup

CHIPSEC_UTIL Program Flow

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

Auxiliary components

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

Executable build scripts

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

Configuration Files

Provide a human readable abstraction for registers in the system

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

Broken into common and platform specific configuration files

Used to define controls, registers and bit fields

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

Correct platform configuration files loaded based off of platform detection

Configuration File Example

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

List of Cfg components

hsw

Path: chipsec\cfg\8086\hsw.xml

XML configuration file for Haswell based platforms

sfdp

Path: chipsec\cfg\8086\sfdp.xml

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

pch_4xxlp

Path: chipsec\cfg\8086\pch_4xxlp.xml

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

glk

Path: chipsec\cfg\8086\glk.xml

XML configuration for GLK

Document ID: 336561-001

icx

Path: chipsec\cfg\8086\icx.xml

XML configuration file for Icelake/Lewisburg Server

ivt

Path: chipsec\cfg\8086\ivt.xml

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

pch_2xx

Path: chipsec\cfg\8086\pch_2xx.xml

XML configuration file for 200 series PCH based platforms

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

cht

Path: chipsec\cfg\8086\cht.xml

XML configuration for Cherry Trail and Braswell SoCs

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

skx

Path: chipsec\cfg\8086\skx.xml

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

pch_5xxh

Path: chipsec\cfg\8086\pch_5xxh.xml

XML configuration file for 5XXH series pch

jkt

Path: chipsec\cfg\8086\jkt.xml

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

tpm12

Path: chipsec\cfg\8086\tpm12.xml

Contact information: chipsec@intel.com

apl

Path: chipsec\cfg\8086\apl.xml

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

pch_495

Path: chipsec\cfg\8086\pch_495.xml

XML configuration file for the 495 series PCH

snb

Path: chipsec\cfg\8086\snb.xml

XML configuration for Sandy Bridge based platforms

pch_3xx

Path: chipsec\cfg\8086\pch_3xx.xml

XML configuration file for the 300 series PCH https://www.intel.com/content/www/us/en/products/docs/chipsets/300-series-chipset-pch-datasheet-vol-2.html 337348-001

pch_3xxop

Path: chipsec\cfg\8086\pch_3xxop.xml

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

pch_5xxlp

Path: chipsec\cfg\8086\pch_5xxlp.xml

XML configuration file for 5XXLP series pch

bdx

Path: chipsec\cfg\8086\bdx.xml

XML configuration file for Broadwell Server based platforms Intel (c) Xeon Processor E5 v4 Product Family datasheet Vol. 2 Intel (c) Xeon Processor E7 v4 Product Family datasheet Vol. 2 Intel (c) C600 Series Chipset and

Intel (c) X79 Express Chipset datasheet Intel (c) C600 Series Chipset and Intel (c) X79 Express Chipset Specification Update Intel (c) C610 Series Chipset and Intel (c) X99 Chipset Platform Controller Hub (PCH) datasheet

cml

Path: chipsec\cfg\8086\cml.xml

XML configuration file for Comet Lake

skl

Path: chipsec\cfg\8086\skl.xml

XML configuration file for Skylake based platforms

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

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

tglu

Path: chipsec\cfg\8086\tglu.xml

Contact information: chipsec@intel.com

hsx

Path: chipsec\cfg\8086\hsx.xml

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

qrk

Path: chipsec\cfg\8086\qrk.xml

XML configuration for Quark based platforms

pch_4xx

Path: chipsec\cfg\8086\pch_4xx.xml

XML configuration file for 4XX pch

common

Path: chipsec\cfg\8086\common.xml

Common (default) XML platform configuration file

rkl

Path: chipsec\cfg\8086\rkl.xml

Contact information: chipsec@intel.com

pch_3xxlp

Path: chipsec\cfg\8086\pch_3xxlp.xml

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

whl

Path: chipsec\cfg\8086\whl.xml

XML configuration file for Whiskey Lake

n Generation Intel(R) Processor Family for U-Processor Platforms
tps://www.intel.com/content/www/us/en/processors/core/core-technical-resources.html
tps://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/300-series-chipset-on-package-pch-datasheet-v

bdw

Path: chipsec\cfg\8086\bdw.xml

XML configuration for Broadwell based platforms

byt

Path: chipsec\cfg\8086\byt.xml

XML configuration for Bay Trail based platforms

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

avn

Path: chipsec\cfg\8086\avn.xml

XML configuration for Avoton based platforms

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

pch_c620

Path: chipsec\cfg\8086\pch_c620.xml

XML configuration file for

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

dnv

Path: chipsec\cfg\8086\dnv.xml

XML configuration file for Denverton

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

icl

Path: chipsec\cfg\8086\icl.xml

XML configuration file for Ice Lake

pch_c60x

Path: chipsec\cfg\8086\pch_c60x.xml

XML configuration file for C600 series PCH

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

kbl

Path: chipsec\cfg\8086\kbl.xml

XML configuration file for Kaby Lake based platforms

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

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

iommu

Path: chipsec\cfg\8086\iommu.xml

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

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

ivb

Path: chipsec\cfg\8086\ivb.xml

XML configuration for IvyBridge based platforms

pch_c61x

Path: chipsec\cfg\8086\pch_c61x.xml

XML configuration file for C610 series PCH

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

pch_1xx

Path: chipsec\cfg\8086\pch_1xx.xml

XML configuration file for 100 series PCH based platforms

Contact information: chipsec@intel.com

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

cfl

Path: chipsec\cfg\8086\cfl.xml

XML configuration file for Coffee Lake

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

pch_4xxh

Path: chipsec\cfg\8086\pch_4xxh.xml

XML configuration file 4xxH PCH 620855

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 knowning which register provides it. (This is especially important for test reuse across platform generations.)

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

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

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

2. Get the current status of the control:

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

3. React based on the status of the control:

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

4. Return:

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

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

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

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

- Modules common to all platform which CHIPSEC supports can be located in `chipsec/modules/common` directory

If a new platform needs to be added:

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

> ### Seealso
>
> Creating CHIPSEC modules and commands

OS Helpers and Drivers

Provide common interfaces to interact with system drivers/commands

Mostly invoked by HAL modules

Directly invoking helpers from modules should be minimized

Helpers import from BaseHelper

Override applicable functions – default is to generate exception

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

Create a New Helper

Helper needs to be added to the import list either within helpers.py or custom_helpers.py

Example

The new helper should be added to either `chipsec/helper/helpers.py` or `chipsec/helper/custom_helpers.py`

A new helper folder should be created under `chipsec/helper/new_helper`

`chipsec/helper/new_helper/__init__.py` within the new folder needs to add the helper to avail_helpers list

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

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

`chipsec/helper/new_helper/new_helper.py` should import from Helper Base Class

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

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

Helper components

helper package

dal package

dalhelper module

Intel DFx Abstraction Layer (DAL) helper

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

efi package

efihelper module

On UEFI use the efi package functions

file package

filehelper module

Use results from a json file

linux package

cpuid module

legacy_pci module

linuxhelper module

Linux helper

osx package

osxhelper module

OSX helper

rwe package

rwehelper module

win package

win32helper module

basehelper module

helpers module

oshelper module

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

Methods for Platform Detection

Uses PCI VID and DID to detect processor and PCH

Processor 0:0.0

PCH 0:31.0

Chip information located in `chipsec/chipset.py`

Currently requires VID of 0x8086

DID is used as the lookup key

If there are matching DID, will fall back to cpuid check for CPU

Platform Configuration Options

Select a specific platform using the `–p` flag

Specify PCH using the `--pch` flag

Ignore the platform specific registers using the `-i` flag

CHIPSEC Modules

A CHIPSEC module is just a python class that inherits from BaseModule and implements `is_supported` and `run`. Modules are stored under the chipsec installation directory in a subdirectory "modules". The "modules" directory contains one subdirectory for each chipset that chipsec supports. There is also a directory for common modules that should apply to every platform.

`chipsec/modules/`	modules including tests or tools (that's where most of the chipsec functionality is)
`chipsec/modules/common/`	modules common to all platforms
`chipsec/modules/<platform>/`	modules specific to <platform>
`chipsec/modules/tools/`	security tools based on CHIPSEC framework (fuzzers, etc.)

Internally the chipsec application uses the concept of a module name, which is a string of the form: `common.bios_wp`. This means module `common.bios_wp` is a python script called `bios_wp.py` that is stored at `<ROOT_DIR>\chipsec\modules\common\`.

Modules can be mapped to one or more security vulnerabilities being checked. More information also found in the documentation for any individual module.

Known vulnerabilities can be mapped to CHIPSEC modules as follows:

Attack Surface/Vector: Firmware protections in ROM

Vulnerability Description	CHIPSEC Module	Example
SMI event configuration is not locked	common.bios_smi	
SPI flash descriptor is not protected	common.spi_desc	
SPI controller security override is enabled	common.spi_fdopss	
SPI flash controller is not locked	common.spi_lock	
Device-specific SPI flash protection is not used	chipsec_util spi write (manual analysis)	
SMM BIOS write protection is not correctly used	common.bios_wp	
Flash protected ranges do not protect bios region	common.bios_wp	
BIOS interface is not locked	common.bios_ts	

Attack Surface/Vector: Runtime protection of SMRAM

Vulnerability Description	CHIPSEC Module	Example
Compatability SMRAM is not locked	common.smm	
SMM cache attack	common.smrr	
Memory remapping vulnerability in SMM protection	remap	
DMA protections of SMRAM are not in use	smm_dma	
Graphics aperture redirection of SMRAM	chipsec_util memconfig remap	
Memory sinkhole vulnerability	tools.cpu.sinkhole	

Attack Surface/Vector: Secure boot - Incorrect protection of secure boot configuration

Vulnerability Description	CHIPSEC Module	Example
Root certificate	common.bios_wp, common.secureboot.variables	
Key exchange keys and whitelist/blacklist	common.secureboot.variables	
Controls in setup variable (CSM enable/disable, image verification policies, secure boot enable/disable, clear/restore keys)	chipsec_util uefi var-find Setup	
TE header confusion	tools.secureboot.te	
UEFI NVRAM is not write protected	common.bios_wp	
Insecure handling of secure boot disable	chipsec_util uefi var-list	

Attack Surface/Vector: Persistent firmware configuration

Vulnerability Description	CHIPSEC Module	Example
Secure boot configuration is stored in unprotected variable	common.secureboot.variables, chipsec_util uefi var-list	
Variable permissions are not set according to specification	common.uefi.access_uefispec	
Sensitive data (like passwords) are stored in uefi variables	chipsec_util uefi var-list (manual analysis)	
Firmware doesn't sanitize pointers/addresses stored in variables	chipsec_util uefi var-list (manual analysis)	
Firmware hangs on invalid variable content	chipsec_util uefi var-write, chipsec_util uefi var-delete (manual analysis)	
Hardware configuration stored in unprotected variables	chipsec_util uefi var-list (manual analysis)	
Re-creating variables with less restrictive permissions	chipsec_util uefi var-write (manual analysis)	
Variable NVRAM overflow	chipsec_util uefi var-write (manual analysis)	
Critical configuration is stored in unprotected CMOS	chipsec_util cmos, common.rtclock	

Attack Surface/Vector: Platform hardware configuration

Vulnerability Description	CHIPSEC Module	Example
Boot block top-swap mode is not locked	common.bios_ts	
Architectural features not locked	common.ia32cfg	
Memory mamp is not locked	memconfig	
IOMMU usage	chipsec_util iommu	
Memory remapping is not locked	remap	

Attack Surface/Vector: Runtime firmware (eg. SMI handlers)

Vulnerability Description	CHIPSEC Module	Example
SMI handlers use pointers/addresses from OS without validation	tools.smm.smm_ptr	
Legacy SMI handlers call legacy BIOS outside SMRAM		
INT15 in legacy SMI handlers		

UEFI SMI handlers call UEFI services outside SMRAM		
Malicious CommBuffer pointer and contents		
Race condition during SMI handler		
Authenticated variables SMI handler is not implemented	chipsec_util uefi var-write	
SmmRuntime vulnerability	tools.uefi.blacklist	

Attack Surface/Vector: Boot time firmware

Vulnerability Description	CHIPSEC Module	Example
Software vulnerabilities when parsing, decompressing, and loading data from ROM		
Software vulnerabilities in implementation of digital signature verification		
Pointers stored in UEFI variables and used during boot	chipsec_util uefi var-write	
Loading unsigned PCI option ROMs	chipsec_util pci xrom	
Boot hangs due to error condition (eg. ASSERT)		

Attack Surface/Vector: Power state transitions (eg. resume from sleep)

Vulnerability Description	CHIPSEC Module	Example
Insufficient protection of S3 boot script table	common.uefi.s3bootscript, tools.uefi.s3script_modify	
Dispatch opcodes in S3 boot script call functions in unprotected memory	common.uefi.s3bootscript, tools.uefi.s3script_modify	
S3 boot script interpreter stored in unprotected memory		
Pointer to S3 boot script table in unprotected UEFI variable	common.uefi.s3bootscript, tools.uefi.s3script_modify	
Critical setting not recorded in S3 boot script table	chipsec_util uefi s3bootscript (manual analysis)	
OS waking vector in ACPI tables can be modified	chipsec_util acpi dump (manual analysis)	
Using pointers on S3 resume stored in unprotected UEFI variables	chipsec_util uefi var-write	

Attack Surface/Vector: Firmware update

Vulnerability Description	CHIPSEC Module	Example
Software vulnerabilities when parsing firmware updates		
Unauthenticated firmware updates		
Runtime firmware update that can be interrupted		
Signature not checked on capsule update executable		

Attack Surface/Vector: Network interfaces

Vulnerability Description	CHIPSEC Module	Example
Software vulnerabilities when handling messages over network interfaces		
Booting unauthenticated firmware over unprotected network interfaces		

Attack Surface/Vector: Misc

Vulnerability Description	CHIPSEC Module	Example
BIOS keyboard buffer is not cleared during boot	common.bios_kbrd_buffer	
DMA attack from devices during firmware execution		

Modules

modules package

bdw package

byt package

common package

cpu package

cpu_info module

Displays CPU information

ia_untrusted module

IA Untrusted checks

spectre_v2 module

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

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

1. Indirect Branch Restricted Speculation (IBRS) and Indirect Branch Predictor Barrier (IBPB): CPUID.(EAX=7H,ECX=0):EDX[26] == 1
2. Single Thread Indirect Branch Predictors (STIBP): CPUID.(EAX=7H,ECX=0):EDX[27] == 1 IA32_SPEC_CTRL[STIBP] == 1
3. Enhanced IBRS: CPUID.(EAX=7H,ECX=0):EDX[29] == 1 IA32_ARCH_CAPABILITIES[IBRS_ALL] == 1 IA32_SPEC_CTRL[IBRS] == 1
4. @TODO: Mitigation for Rogue Data Cache Load (RDCL): CPUID.(EAX=7H,ECX=0):EDX[29] == 1 IA32_ARCH_CAPABILITIES[RDCL_NO] == 1

In addition to checking if CPU supports and OS enables all mitigations, we need to check that relevant MSR bits are set consistently on all logical processors (CPU threads).

The module returns the following results:

FAILED:

IBRS/IBPB is not supported

WARNING:

IBRS/IBPB is supported

Enhanced IBRS is not supported

WARNING:

IBRS/IBPB is supported

Enhanced IBRS is supported

Enhanced IBRS is not enabled by the OS

WARNING:

IBRS/IBPB is supported

STIBP is not supported or not enabled by the OS

PASSED:

IBRS/IBPB is supported

Enhanced IBRS is supported

Enhanced IBRS is enabled by the OS

STIBP is supported

Notes:

- The module returns WARNING when CPU doesn't support enhanced IBRS Even though OS/software may use basic IBRS by setting IA32_SPEC_CTRL[IBRS] when necessary, we have no way to verify this
- The module returns WARNING when CPU supports enhanced IBRS but OS doesn't set IA32_SPEC_CTRL[IBRS] Under enhanced IBRS, OS can set IA32_SPEC_CTRL[IBRS] once to take advantage of IBRS protection
- The module returns WARNING when CPU doesn't support STIBP or OS doesn't enable it Per Speculative Execution Side Channel Mitigations: "enabling IBRS prevents software operating on one logical processor from controlling the predicted targets of indirect branches executed on another logical processor. For that reason, it is not necessary to enable STIBP when IBRS is enabled"
- OS/software may implement "retpoline" mitigation for Spectre variant 2 instead of using CPU hardware IBRS/IBPB

@TODO: we should verify CPUID.07H:EDX on all logical CPUs as well because it may differ if ucode update wasn't loaded on all CPU cores

Hardware registers used:

- CPUID.(EAX=7H,ECX=0):EDX[26] - enumerates support for IBRS and IBPB
- CPUID.(EAX=7H,ECX=0):EDX[27] - enumerates support for STIBP
- CPUID.(EAX=7H,ECX=0):EDX[29] - enumerates support for the IA32_ARCH_CAPABILITIES MSR
- IA32_ARCH_CAPABILITIES[IBRS_ALL] - enumerates support for enhanced IBRS
- IA32_ARCH_CAPABILITIES[RCDL_NO] - enumerates support RCDL mitigation
- IA32_SPEC_CTRL[IBRS] - enable control for enhanced IBRS by the software/OS
- IA32_SPEC_CTRL[STIBP] - enable control for STIBP by the software/OS

References:

- Reading privileged memory with a side-channel by Jann Horn, Google Project Zero: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- Spectre: https://spectreattack.com/spectre.pdf
- Meltdown: https://meltdownattack.com/meltdown.pdf

- Speculative Execution Side Channel Mitigations: https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

- Retpoline: a software construct for preventing branch-target-injection: https://support.google.com/faqs/answer/7625886

secureboot package

variables module

UEFI 2.4 spec Section 28

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

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

uefi package

access_uefispec module

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

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

usage:

`chipsec_main -m common.uefi.access_uefispec [-a modify]`

- `-a modify`: Attempt to modify each variable in addition to checking attributes

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -m common.uefi.access_uefispec
>>> chipsec_main.py -m common.uefi.access_uefispec -a modify
```

NOTE: Requires an OS with UEFI Runtime API support.

s3bootscript module

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

References:

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

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

Attacks on UEFI Security by Rafal Wojtczuk and Corey Kallenberg.

Attacking UEFI Boot Script by Rafal Wojtczuk and Corey Kallenberg.

Exploiting UEFI boot script table vulnerability by Dmytro Oleksiuk.

Usage:

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

- `-a <script_address>`: Specify the bootscript address

Where:

- [ ]: optional line

Examples:

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

> **Note**
>
> Requires an OS with UEFI Runtime API support.

bios_kbrd_buffer module

DEFCON 16: Bypassing Pre-boot Authentication Passwords by Instrumenting the BIOS Keyboard Buffer by Jonathan Brossard

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

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

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)

bios_ts module

Checks for BIOS Interface Lock including Top Swap Mode

BIOS Boot Hijacking and VMware Vulnerabilities Digging by Bing Sun

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.

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 diabled and locked.

Hardware registers used: IA32_DEBUG_INTERFACE[DEBUGENABLE] IA32_DEBUG_INTERFACE[DEBUGELOCK] IA32_DEBUG_INTERFACE[DEBUGEOCCURED] P2SB_DCI.DCI_CONTROL_REG[HDCIEN]

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

me_mfg_mode module

This module checks that ME Manufacturing mode is not enabled

References:

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

PCI_DEVS.H

```
#define PCH_DEV_SLOT_CSE        0x16
#define  PCH_DEVFN_CSE          _PCH_DEVFN(CSE, 0)
#define  PCH_DEV_CSE            _PCH_DEV(CSE, 0)
```

https://github.com/coreboot/coreboot/blob/master/src/soc/intel/apollolake/cse.c

```
fwsts1 = dump_status(1, PCI_ME_HFSTS1);
# Minimal decoding is done here in order to call out most important
# pieces. Manufacturing mode needs to be locked down prior to shipping
# the product so it's called out explicitly.
printk(BIOS_DEBUG, "ME: Manufacturing Mode      : %s", (fwsts1 & (1 << 0x4)) ? "YES" : "NO");
```

PCH.H

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

ME.H

```
struct me_hfs {
        u32 working_state: 4;
        u32 mfg_mode: 1;
        u32 fpt_bad: 1;
        u32 operation_state: 3;
        u32 fw_init_complete: 1;
        u32 ft_bup_ld_flr: 1;
        u32 update_in_progress: 1;
        u32 error_code: 4;
        u32 operation_mode: 4;
        u32 reserved: 4;
        u32 boot_options_present: 1;
        u32 ack_data: 3;
        u32 bios_msg_ack: 4;
} __packed;
```

ME_STATUS.C

```
printk(BIOS_DEBUG, "ME: Manufacturing Mode      : %s", hfs->mfg_mode ? "YES" : "NO");
```

This module checks the following:

HFS.MFG_MODE BDF: 0:22:0 offset 0x40 - Bit [4]

The module returns the following results:

FAILED : HFS.MFG_MODE is set

PASSED : HFS.MFG_MODE is not set.

Hardware registers used:

HFS

memconfig module

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

memlock module

This module checks if memory configuration is locked to protect SMM

Reference: https://github.com/coreboot/coreboot/blob/master/src/cpu/intel/model_206ax/finalize.c https://github.com/coreboot/coreboot/blob/master/src/soc/intel/broadwell/include/soc/msr.h

This module checks the following: - MSR_LT_LOCK_MEMORY MSR (0x2E7) - Bit [0]

The module returns the following results: FAILED : MSR_LT_LOCK_MEMORY[0] is not set PASSED : MSR_LT_LOCK_MEMORY[0] is set.

Hardware registers used: MSR_LT_LOCK_MEMORY

Usage:

`chipsec_main -m common.memlock`

Example:

```
>>> chipsec_main.py -m common.memlock
```

> ***Note***
>
> This module will not run on Atom based platforms.

remap module

Check Memory Remapping Configuration

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

Usage:

`chipsec_main -m common.remap`

Example:

```
>>> chipsec_main.py -m common.remap
```

> **Note**
>
> This module will only run on Core platforms.

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.

sgx_check module

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

smm module

Compatible SMM memory (SMRAM) Protection check module This CHIPSEC module simply reads SMRAMC and checks that D_LCK is set.

Reference: In 2006, Security Issues Related to Pentium System Management Mode outlined a configuration issue where compatibility SMRAM was not locked on some platforms. This means that ring 0 software was able to modify System Management Mode (SMM) code and data that should have been protected.

In Compatability SMRAM (CSEG), access to memory is defined by the SMRAMC register. When SMRAMC[D_LCK] is not set by the BIOS, SMRAM can be accessed even when the CPU is not in SMM. Such attacks were also described in Using CPU SMM to Circumvent OS Security Functions and Using SMM for Other Purposes.

usage:

`chipsec_main -m common.smm`

Examples:

```
>>> chipsec_main.py -m common.smm
```

This module will only run on client (core) platforms that have PCI0.0.0_SMRAMC defined.

smm_code_chk module

SMM_Code_Chk_En is a bit found in the MSR_SMM_FEATURE_CONTROL register. Once set to ‘1’, any CPU that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE. As such, enabling and locking this bit is an important step in mitigating SMM call-out vulnerabilities. This CHIPSEC module simply reads the register and checks that SMM_Code_Chk_En is set and locked.

smm_dma module

Just like SMRAM needs to be protected from software executing on the CPU, it also needs to be protected from devices that have direct access to DRAM (DMA). Protection from DMA is configured through proper programming of SMRAM memory range. If BIOS does not correctly configure and lock the configuration, then malware could reprogram configuration and open SMRAM area to DMA access, allowing manipulation of memory that should have been protected.

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

This module examines the configuration and locking of SMRAM range configuration protecting from DMA attacks. If it fails, then DMA protection may not be securely configured to protect SMRAM.

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-cachable 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.

spd_wd module

This module checks that SPD Write Disable bit in SMBus controller has been set

References:

Intel 8 Series/C220 Series Chipset Family Platform Controller Hub datasheet Intel 300 Series Chipset Families Platform Controller Hub datasheet

This module checks the following:

SMBUS_HCFG.SPD_WD

The module returns the following results:

PASSED : SMBUS_HCFG.SPD_WD is set

FAILED : SMBUS_HCFG.SPD_WD is not set and SPDs were detected

INFORMATION: SMBUS_HCFG.SPD_WD is not set, but no SPDs were detected

Hardware registers used:

SMBUS_HCFG

Usage:

```
chipsec_main -m common.spd_wd
```

Examples:

```
>>> chipsec_main.py -m common.spd_wd
```

> ***Note***
>
> **This module will only run if:**

- SMBUS device is enabled
- SMBUS_HCFG.SPD_WD is defined for the platform

spi_access module

Checks SPI Flash Region Access Permissions programmed in the Flash Descriptor

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.

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.

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.

wsmt module

The Windows SMM Security Mitigation Table (WSMT) is an ACPI table defined by Microsoft that allows system firmware to confirm to the operating system that certain security best practices have been implemented in System Management Mode (SMM) software. See <https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-uefi-wsmt> for more details.

hsw package

ivb package

snb package

tools package

cpu package

sinkhole module

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

NOTE: The system may hang when running this test. In that case, the mitigation to this issue is likely working but we may not be handling the exception generated.

References:

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

secureboot package

te module

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

Usage:

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

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

Examples:

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

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

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

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

Restore bootloaders listed in te.cfg file:

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

smm package

rogue_mmio_bar module

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

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

Usage:

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

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

Example:

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

Note

Look for ‘changes found’ messages for items that should be further investigated.

Warning

When running this test, system may freeze, reboot, etc. This is not unexpected behavior and not generally considered a failure.

smm_ptr module

CanSecWest 2015 A New Class of Vulnerability in SMI Handlers of BIOS/UEFI Firmware

A tool to test SMI handlers for pointer validation vulnerabilities

Usage: `chipsec_main -m tools.smm.smm_ptr -l log.txt \`
`[-a <mode>,<config_file>|<smic_start:smic_end>,<size>,<address>]`

- `mode`: SMI fuzzing mode
    - `config` = use SMI configuration file <config_file>

- `fuzz` = fuzz all SMI handlers with code in the range <smic_start:smic_end>
- `fuzzmore` = fuzz mode + pass 2nd-order pointers within buffer to SMI handlers

- `size`: size of the memory buffer (in Hex)
- `address`: physical address of memory buffer to pass in GP regs to SMI handlers (in Hex)
  - `smram` = option passes address of SMRAM base (system may hang in this mode!)

In `config` mode, SMI configuration file should have the following format

```
SMI_code=<SMI code> or *
SMI_data=<SMI data> or *
RAX=<value of RAX> or * or PTR or VAL
RBX=<value of RBX> or * or PTR or VAL
RCX=<value of RCX> or * or PTR or VAL
RDX=<value of RDX> or * or PTR or VAL
RSI=<value of RSI> or * or PTR or VAL
RDI=<value of RDI> or * or PTR or VAL
[PTR_OFFSET=<offset to pointer in the buffer>]
[SIG=<signature>]
[SIG_OFFSET=<offset to signature in the buffer>]
[Name=<SMI name>]
[Desc=<SMI description>]
```

Where

- `[]`: optional line
- `*`: Don't Care (the module will replace * with 0x0)
- PTR: Physical address SMI handler will write to (the module will replace PTR with physical address provided as a command-line argument)
- VAL: Value SMI handler will write to PTR address (the module will replace VAL with hardcoded _FILL_VALUE_xx)

uefi package

reputation module

This module checks current contents of UEFI firmware ROM or specified firmware image for bad EFI binaries as per the VirusTotal API. These can be EFI firmware volumes, EFI executable binaries (PEI modules, DXE drivers..) or EFI sections. The module can find EFI binaries by their UI names, EFI GUIDs, MD5/SHA-1/SHA-256 hashes or contents matching specified regular expressions.

Important! This module can only detect bad or vulnerable EFI modules based on the file's reputation on VT.

Usage:

chipsec_main.py -i -m tools.uefi.reputation -a <vt_api_key>[,<vt_threshold>,<fw_image>]

vt_api_key : *API key to VirusTotal. Can be obtained by visting https://www.virustotal.com/gui/join-us.*

This argument must be specified.

vt_threshold : *The minimal number of different AV vendors on VT which must claim an EFI module is malicious*

before failing the test. Defaults to 10.

fw_image : *Full file path to UEFI firmware image*

If not specified, the module will dump firmware image directly from ROM

s3script_modify module

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

Usage:

Replacing existing opcode:

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

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

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

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

Adding new opcode:

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

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

Examples:

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

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

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

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

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

By default this test will allocate memory and write write `0xB007B007` that location.

After executing this, if the system is vulnerable to boot script modification, you should find the given value in the allocated memory location.

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

The option will look for a dispatch opcode in the script and modify the opcode to point to a different entry point. The new entry point will contain a HLT instruction.

After executing this, if the system is vulnerable to boot script modification, the system should hang on resume from S3.

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

The option will look for a dispatch opcode in the script and will modify memory at the entry point for that opcode. The modified instructions will contain a HLT instruction.

After executing this, if the system is vulnerable to dispatch opcode entry point modification, the system should hang on resume from S3.

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

The option will add a new opcode which writes to PCI config, MMIO or I/O registers with specified values.

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

The option will add a new DISPATCH opcode to the script with entry point to either existing or newly allocated memory.

scan_blocked module

This module checks current contents of UEFI firmware ROM or specified firmware image for blocked EFI binaries which can be EFI firmware volumes, EFI executable binaries (PEI modules, DXE drivers..) or EFI sections. The module can find EFI binaries by their UI names, EFI GUIDs, MD5/SHA-1/SHA-256 hashes or contents matching specified regular expressions.

Important! This module can only detect what it knows about from its config file. If a bad or vulnerable binary is not detected then its 'signature' needs to be added to the config.

Usage:

```
chipsec_main.py -i -m tools.uefi.scan_blocked [-a <fw_image>,<blockedlist>]
```

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

Examples:

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

Dumps UEFI firmware image from flash memory device, decodes it and checks for blocked EFI modules defined in the default config `blockedlist.json`

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

Decodes `uefi.rom` binary with UEFI firmware image and checks for blocked EFI modules defined in `blockedlist.json` config

Note: `-i` and `--no_driver` arguments can be used in this case because the test does not depend on the platform and no kernel driver is required when firmware image is specified

scan_image module

The module can generate a list of EFI executables from (U)EFI firmware file or extracted from flash ROM, and then later check firmware image in flash ROM or file against this list of expected executables

Usage:

```
chipsec_main -m tools.uefi.scan_image [-a generate|check,<json>,<fw_image>]
```

- `generate` **Generates a list of EFI executable binaries from the UEFI**
  firmware image (default)
- `check` **Decodes UEFI firmware image and checks all EFI executable**
  binaries against a specified list
- `json` **JSON file with configuration of allowed list EFI**
  executables (default = `efilist.json`)
- `fw_image` **Full file path to UEFI firmware image. If not specified,**
  the module will dump firmware image directly from ROM

Examples:

```
>>> chipsec_main -m tools.uefi.scan_image
```

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

```
>>> chipsec_main -i -n -m tools.uefi.scan_image -a generate,efilist.json,uefi.rom
```

Creates a list of EFI executable binaries in `efilist.json` from `uefi.rom` firmware binary

```
>>> chipsec_main -i -n -m tools.uefi.scan_image -a check,efilist.json,uefi.rom
```

Decodes `uefi.rom` UEFI firmware image binary and checks all EFI executables in it against a list defined in `efilist.json`

Note: `-i` and `-n` arguments can be used when specifying firmware file because the module doesn't depend on the platform and doesn't need kernel driver

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
```

vmm package

hv package

define module

Hyper-V specific defines

hypercall module

Hyper-V specific hypercall functionality

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`

synth_dev module

Hyper-V VMBus synthetic device generic fuzzer

Usage:

Print channel offers:

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

Fuzzing device with specified relid:

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

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

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`

vmbus module

Hyper-V VMBus functionality

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`

vbox package

vbox_crash_apicbase module

PoC test for Host OS Crash when writing to IA32_APIC_BASE MSR (Oracle VirtualBox CVE-2015-0377) http://www.oracle.com/technetwork/topics/security/cpujan2015-1972971.html

Usage:

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

xen package

define module

Xen specific defines

hypercall module

Xen specific hypercall functionality

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
```

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
```

common module

Common functionality for VMM related modules/tools

cpuid_fuzz module

Simple CPUID VMM emulation fuzzer

Usage:

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

ept_finder module

Usage:

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

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

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
```

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
```

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
```

pcie_overlap_fuzz module

PCIe device Memory-Mapped I/O (MMIO) ranges VMM emulation fuzzer which first overlaps MMIO BARs of all available PCIe devices then fuzzes them by writing garbage if corresponding option is enabled

Usage:

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

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
```