CHIPSEC

version 2.0.10



Platform Security Assesment Framework

November 06, 2026

Contents

CHIPSEC 2.0.10 **1**
Contact 1
Download CHIPSEC 2
GitHub repository 2
Releases 2
Python 2
Installation 2
Linux Installation 3
Creating a Live Linux image 3
Installing Kali Linux 3
Prerequisites 3
Installing CHIPSEC 4
Building CHIPSEC 4
Run CHIPSEC 4
DAL Windows Installation 4
Prerequisites 4
Building 5
Windows Installation 5
Install CHIPSEC Dependencies 5
Building 6
Turn off kernel driver signature checks 6
Alternate Build Methods 7
Windows PCI Filter Driver 8
Install PCI Filter Driver 9
Filter Driver Access PCI Config Space Test 11
Building a Bootable USB drive with UEFI Shell (x64) 12
Installing CHIPSEC 12
Run CHIPSEC in UEFI Shell 12
Building UEFI Python 3.6.8 (optional) 13
Using CHIPSEC 13
Interpreting results 13
Results 14
Automated Tests 14
Tools 17
Running CHIPSEC 17
Running in Shell 18
Using as a Python Package 18
chipsec_main options 19
chipsec_util options 19

Windows Exception Errors 19
STATUS_PRIVILEGED_INSTRUCTION (0xC0000096) 19
Windows driver: StartService error 21
Additional details 22
Module & Command Development 22
Architecture Overview 22
Core components 22
Commands 23
chipsec.utilcmd package 23
chipsec.utilcmd.acpi_cmd module 23
chipsec.utilcmd.chipset_cmd module 23
chipsec.utilcmd.cmos_cmd module 24
chipsec.utilcmd.config_cmd module 24
chipsec.utilcmd.cpu_cmd module 24
chipsec.utilcmd.decode_cmd module 24
chipsec.utilcmd.deltas_cmd module 25
chipsec.utilcmd.desc_cmd module 25
chipsec.utilcmd.ec_cmd module 25
chipsec.utilcmd.hals_cmd module 26
chipsec.utilcmd.hob_cmd module 26
chipsec.utilcmd.igd_cmd module 26
chipsec.utilcmd.interrupts_cmd module 27
chipsec.utilcmd.io_cmd module 27
chipsec.utilcmd.iommu_cmd module 27
chipsec.utilcmd.lock_check_cmd module 27
chipsec.utilcmd.mem_cmd module 28
chipsec.utilcmd.mm_msgbus_cmd module 28
chipsec.utilcmd.mmcfg_base_cmd module 28
chipsec.utilcmd.mmcfg_cmd module 29
chipsec.utilcmd.mmio_cmd module 29
chipsec.utilcmd.module_id_cmd module 29
chipsec.utilcmd.msgbus_cmd module 30
chipsec.utilcmd.msr_cmd module 30
chipsec.utilcmd.pci_cmd module 30
chipsec.utilcmd.reg_cmd module 31
chipsec.utilcmd.smbios_cmd module 31
chipsec.utilcmd.smbus_cmd module 31
chipsec.utilcmd.spd_cmd module 31
chipsec.utilcmd.spi_cmd module 32
chipsec.utilcmd.spidesc_cmd module 32
chipsec.utilcmd.tpm_cmd module 32

chipsec.utilcmd.txt_cmd module 33
chipsec.utilcmd.ucode_cmd module 33
chipsec.utilcmd.uefi_cmd module 33
chipsec.utilcmd.vmem_cmd module 34
chipsec.utilcmd.vmm_cmd module 34
HAL (Hardware Abstraction Layer) 34
chipsec.hal package 34
chipsec.hal.amd package 34
chipsec.hal.amd.cpu module 34
chipsec.hal.amd.psp module 35
chipsec.hal.common package 35
chipsec.hal.common.acpi module 35
chipsec.hal.common.cmos module 35
chipsec.hal.common.cpuid module 35
chipsec.hal.common.ec module 35
chipsec.hal.common.hob module 35
chipsec.hal.common.interrupts module 36
chipsec.hal.common.io module 36
chipsec.hal.common.iobar module 36
chipsec.hal.common.iommu module 36
chipsec.hal.common.locks module 36
chipsec.hal.common.memrange module 36
chipsec.hal.common.mmio module 37
chipsec.hal.common.msr module 37
chipsec.hal.common.pci module 37
chipsec.hal.common.physmem module 38
chipsec.hal.common.smbios module 38
chipsec.hal.common.smbus module 38
chipsec.hal.common.spd module 38
chipsec.hal.common.tpm module 38
chipsec.hal.common.uefi module 38
chipsec.hal.common.virtmem module 38
chipsec.hal.common.vmm module 39
chipsec.hal.intel package 39
chipsec.hal.intel.cpu module 39
chipsec.hal.intel.igd module 39
chipsec.hal.intel.mm_msgbus module 39
chipsec.hal.intel.mmcfg module 39
chipsec.hal.intel.msgbus module 40
chipsec.hal.intel.spi module 40
chipsec.hal.intel.spi_descriptor module 40

chipsec.hal.intel.ucode module 40
chipsec.hal.hal_base module 41
chipsec.hal.hals module 41
Fuzzing 41
chipsec.fuzzing package 41
chipsec.fuzzing.primitives module 41
CHIPSEC_MAIN Program Flow 41
CHIPSEC_UTIL Program Flow 41
Auxiliary components 42
Executable build scripts 42
Configuration Files 42
List of Cfg components 43
Developer Guide 47
Module layout and lifecycle 47
Core APIs 47
Development practices 48
Debugging and testing 49
OS Helpers and Drivers 49
Implementing a new backend 50
Helper components 50
chipsec.helper package 50
chipsec.helper.dal package 50
chipsec.helper.dal.dalhelper module 50
chipsec.helper.efi package 50
chipsec.helper.efi.efihelper module 50
chipsec.helper.linux package 50
chipsec.helper.linux.linuxhelper module 50
chipsec.helper.linuxnative package 51
chipsec.helper.linuxnative.cpuid module 51
chipsec.helper.linuxnative.legacy_pci module 51
chipsec.helper.linuxnative.linuxnativehelper module 51
chipsec.helper.record package 51
chipsec.helper.record.recordhelper module 51
chipsec.helper.replay package 51
chipsec.helper.replay.replayhelper module 51
chipsec.helper.windows package 51
chipsec.helper.windows.windowshelper module 51
chipsec.helper.basehelper module 51
chipsec.helper.nonehelper module 51
chipsec.helper.oshelper module 51
Platform Detection 52

Platform Configuration Options 52
Sample Module Code 52
Util Command 53
CHIPSEC Modules 53
Modules 58
chipsec.modules package 58
chipsec.modules.bdw package 58
chipsec.modules.byt package 58
chipsec.modules.common package 58
chipsec.modules.common.cpu package 58
chipsec.modules.common.cpu.cpu_info module 58
chipsec.modules.common.cpu.ia_untrusted module 58
chipsec.modules.common.cpu.spectre_v2 module 59
chipsec.modules.common.secureboot package 60
chipsec.modules.common.secureboot.variables module 60
chipsec.modules.common.uefi package 61
chipsec.modules.common.uefi.access_uefispec module 61
chipsec.modules.common.uefi.s3bootscript module 61
chipsec.modules.common.bios_kbrd_buffer module 62
chipsec.modules.common.bios_smi module 62
chipsec.modules.common.bios_ts module 63
chipsec.modules.common.bios_wp module 63
chipsec.modules.common.cet module 64
chipsec.modules.common.debugenabled module 64
chipsec.modules.common.firmware_info module 65
chipsec.modules.common.ia32cfg module 65
chipsec.modules.common.me_mfg_mode module 65
chipsec.modules.common.memconfig module 66
chipsec.modules.common.memlock module 67
chipsec.modules.common.remap module 67
chipsec.modules.common.rom_armor module 68
chipsec.modules.common.rtclock module 68
chipsec.modules.common.sgx_check module 68
chipsec.modules.common.sgx_check_sidekick module 70
chipsec.modules.common.smm module 70
chipsec.modules.common.smm_addr module 70
chipsec.modules.common.smm_close module 70
chipsec.modules.common.smm_code_chk module 71
chipsec.modules.common.smm_dma module 71
chipsec.modules.common.smm_lock module 72
chipsec.modules.common.smrr module 72

chipsec.modules.common.spd_wd module 73
chipsec.modules.common.spi_access module 73
chipsec.modules.common.spi_desc module 74
chipsec.modules.common.spi_fdopss module 74
chipsec.modules.common.spi_lock module 74
chipsec.modules.tools package 75
chipsec.modules.tools.cpu package 75
chipsec.modules.tools.cpu.sinkhole module 75
chipsec.modules.tools.secureboot package 76
chipsec.modules.tools.secureboot.te module 76
chipsec.modules.tools.smm package 76
chipsec.modules.tools.smm.rogue_mmio_bar module 76
chipsec.modules.tools.smm.smm_ptr module 77
chipsec.modules.tools.uefi package 78
chipsec.modules.tools.uefi.reputation module 78
chipsec.modules.tools.uefi.s3script_modify module 78
chipsec.modules.tools.uefi.scan_blocked module 79
chipsec.modules.tools.uefi.scan_image module 80
chipsec.modules.tools.uefi.uefivar_fuzz module 81
chipsec.modules.tools.vmm package 82
chipsec.modules.tools.vmm.hv package 82
chipsec.modules.tools.vmm.hv.define module 82
chipsec.modules.tools.vmm.hv.hypercall module 82
chipsec.modules.tools.vmm.hv.hypercallfuzz module 82
chipsec.modules.tools.vmm.hv.synth_dev module 83
chipsec.modules.tools.vmm.hv.synth_kbd module 83
chipsec.modules.tools.vmm.hv.vmbus module 83
chipsec.modules.tools.vmm.hv.vmbusfuzz module 83
chipsec.modules.tools.vmm.vbox package 83
chipsec.modules.tools.vmm.vbox.vbox_crash_apicbase module 83
chipsec.modules.tools.vmm.xen package 83
chipsec.modules.tools.vmm.xen.define module 83
chipsec.modules.tools.vmm.xen.hypercall module 84
chipsec.modules.tools.vmm.xen.hypercallfuzz module 84
chipsec.modules.tools.vmm.xen.xsa188 module 84
chipsec.modules.tools.vmm.cpuid_fuzz module 84
chipsec.modules.tools.vmm.ept_finder module 85
chipsec.modules.tools.vmm.hypercallfuzz module 86
chipsec.modules.tools.vmm.iofuzz module 86
chipsec.modules.tools.vmm.msr_fuzz module 87
chipsec.modules.tools.vmm.pcie_fuzz module 88

chipsec.modules.tools.vmm.pcie_overlap_fuzz module 89
chipsec.modules.tools.vmm.venom module 89
chipsec.modules.tools.wsmt module 90
Contribution and Style Guides 90
Python Version 90
Python Coding Style Guide 91
f-Strings 93
Type Hints 94
Underscores in Numeric Literals 95
Walrus Operator (:=) 96
Deprecate distutils module support 96
Sphinx Version 97
Generating Documentation 97
References 97

CHIPSEC 2.0.10

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*, and *UEFI shell*.

Warning

Chipsec should only be used on test systems!

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

There are multiple reasons for that:

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

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

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

Contact

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

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

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

Mailing list:

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

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

Twitter:

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

Discord:

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

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

Download CHIPSEC

GitHub repository

CHIPSEC source files are maintained in a GitHub repository:



Releases

You can find the latest release here:



Archived 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, DAL, and UEFI shell. Circumstances surrounding the target platform may change which of these environments is most appropriate.

Linux Installation

Tested on:

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

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

Creating a Live Linux image

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

Installing Kali Linux

Download and install Kali Linux

Prerequisites

Python 3.8 or higher (https://www.python.org/downloads/)

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

Install or update necessary dependencies before installing CHIPSEC:

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

or

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

or

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

To install requirements:

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

Installing CHIPSEC

Get latest CHIPSEC release from PyPI repository

```
pip install chipsec
```

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

Get CHIPSEC package from latest source code

Download zip from CHIPSEC repo

Download CHIPSEC

or

Clone CHIPSEC source

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

Building CHIPSEC

Build the Driver and Compression Tools

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

Run CHIPSEC

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

DAL Windows Installation

Prerequisites

Python 3.8 or higher (https://www.python.org/downloads/)

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

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

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

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

Building

Clone CHIPSEC source

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

Windows Installation

CHIPSEC supports the following versions:
Windows 8, 8.1, 10, 11 - x86 and AMD64
Windows Server 2012, 2016, 2019, 2022, 2025 - 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 3.8 or higher (https://www.python.org/downloads/)

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

To install requirements:

pip install -r windows_requirements.txt

which includes:

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

To compile the driver:

Visual Studio and WDK: for building the driver.

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

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

To clone the repo:

git: open source distributed version control system

Building

Clone CHIPSEC source

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

Build the Driver and Compression Tools

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

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

Turn off kernel driver signature checks

Enable boot menu

In CMD shell:

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

With Secure Boot enabled:

Method 1:

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

Method 2:

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

With Secure Boot disabled:

7 - Download CHIPSEC

Method 1:

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

Method 2:

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

Alternate Build Methods

Build CHIPSEC kernel driver with Visual Studio

Method 1:

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

Method 2:

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

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

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

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

Build the compression tools

Method:

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

Alternate Method to load CHIPSEC service/driver

To create and start CHIPSEC service

```
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
```

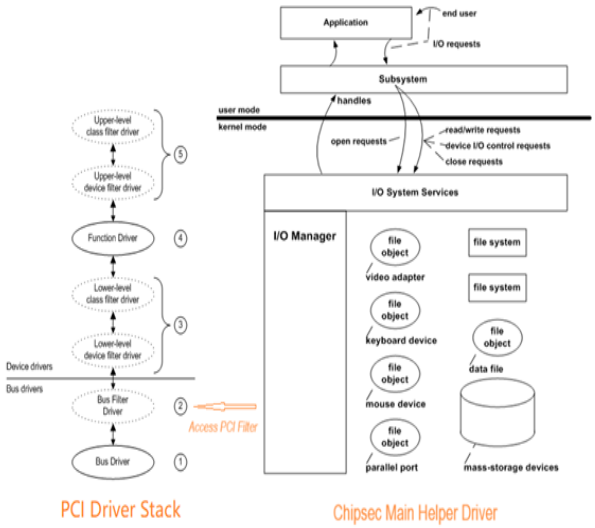
Windows PCI Filter Driver

Filter driver background

Since July 31, 2020 Microsoft has released Windows 2020-KB4568831 (OS Build 19041.423) Preview. Microsoft recommends to not access the PCI configuration space using the legacy API, as it might result in the Windows BSOD (Blue Screen of Death). The BSOD trigger condition is “Windows version >= (OS Build 19041.423) && Secure Devices (SDEV) ACPI table && VBS enabled”. Therefore, CHIPSEC now includes a PCI filter driver which supplements the original CHIPSEC Windows Driver to access the PCI configuration space. A system requires the PCI Filter Driver if the conditions above are met.

Windows devices that receive the 2020-KB4568831 (OS Build 19041.423) Preview or later updates restrict how processes can access peripheral component interconnect (PCI) device configuration space if a Secure Devices (SDEV) ACPI table is present and Virtualization-based Security (VBS) is running. Processes that have to access PCI device configuration space must use officially supported mechanisms.The SDEV table defines secure hardware devices in ACPI. VBS is enabled on a system if security features that use virtualization are enabled. Some examples of these features are Hypervisor Code Integrity or Windows Defender Credential Guard. The new restrictions are designed to prevent malicious processes from modifying the configuration space of secure devices. Device drivers or other system processes must not try to manipulate the configuration space of any PCI devices, except by using the Microsoft-provided bus interfaces or IRP. If a process tries to access PCI configuration space in an unsupported manner (such as by parsing MCFG table and mapping configuration space to virtual memory), Windows denies access to the process and generates a Stop error. For more detail please refer below link: https://learn.microsoft.com/en-us/troubleshoot/windows-client/performance/stop-error-lenovo-thinkpad-kb4568831-uefi

Filter Driver and Main Helper Driver Architecture

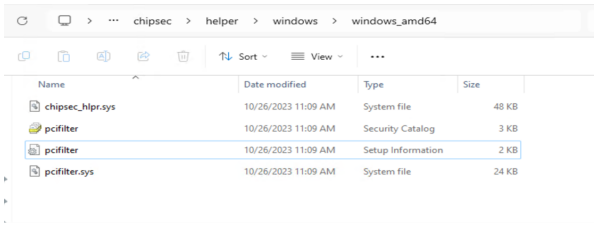


CHIPSEC Main & Filter Driver Architecture

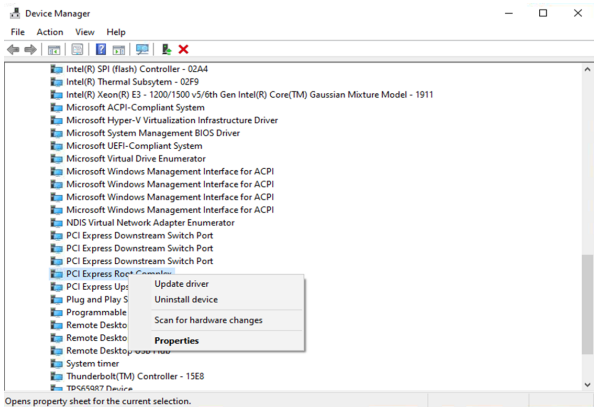
9 - Download CHIPSEC

Install PCI Filter Driver

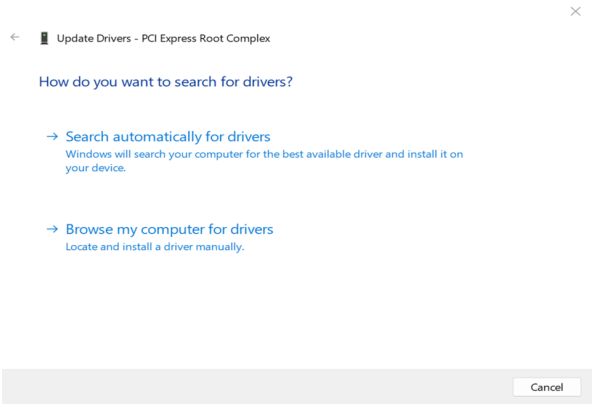
1. Locate the Filter Driver Files: chipsec/helper/windows/windows_amd64/



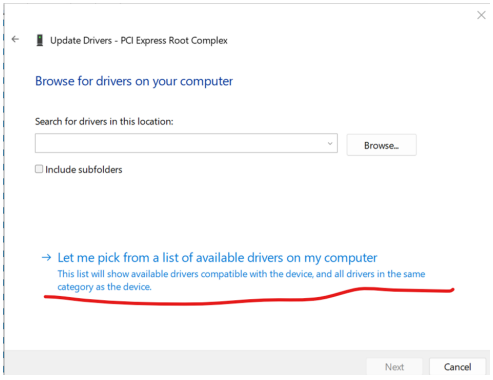
2. Update The PCI Device Driver From Device Manager



3. Browse The PCI Filter Driver

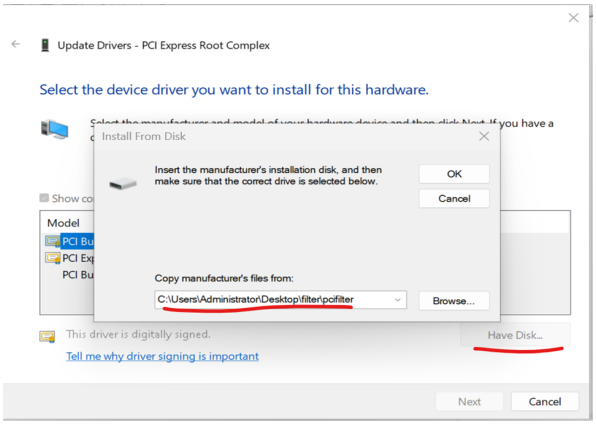


4. Manually Select The PCI Bus Filter Driver

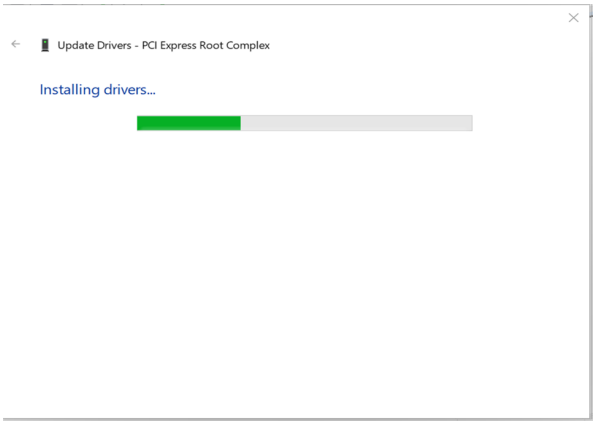


10 - Download CHIPSEC

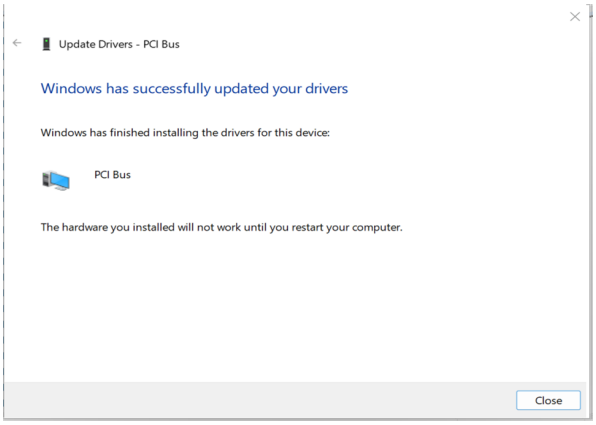
5. Install The Filter Driver From Disk



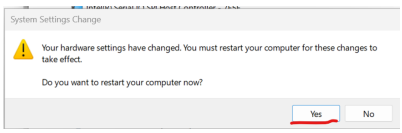
6. Installing The Filter Driver



7. Finish The Filter Driver Installing

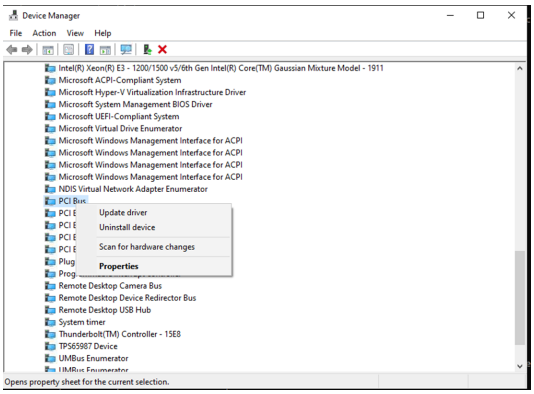


8. Restart Computer

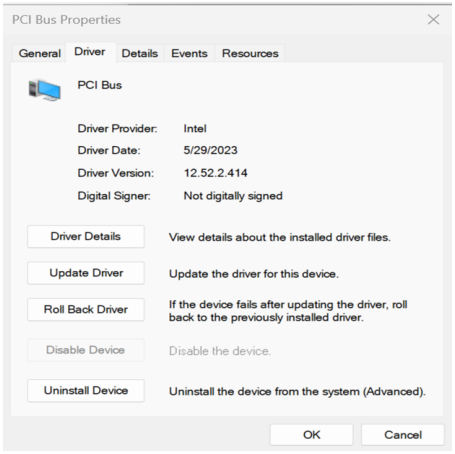


9. Check The Installed Device Driver From Device Manager

11 - Download CHIPSEC

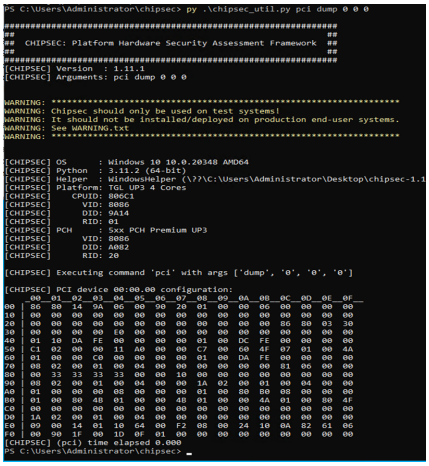


10. Check The Driver Device Info



Filter Driver Access PCI Config Space Test

Dump PCI Config Test




```
py chipsec_util.py pci dump 0 0 0
```

PCI Enumeration Test

12 - Download CHIPSEC

```
py chipsec_util.py pci enumerate
```

Building a Bootable USB drive with UEFI Shell (x64)

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

Installing CHIPSEC

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

   The contents of your drive should look like follows:

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

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

Run CHIPSEC in UEFI Shell

```
fs0:
```

`cd chipsec`

`python.efi chipsec_main.py` or `python.efi chipsec_util.py`

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

Building UEFI Python 3.6.8 (optional)

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

Using CHIPSEC

CHIPSEC should be launched as Administrator/root

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

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

Interpreting results

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

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

Results

Generic results meanings

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

Automated Tests

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

Modules results meanings

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

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

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

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

Tools

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

Running CHIPSEC

CHIPSEC should be launched as Administrator/root.

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

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

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

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

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

  `# python chipsec_util.py`

Running in Shell

Basic usage

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

For help, run

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

Using as a Python Package

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

To install and run CHIPSEC as a package:

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

From the Python shell:

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

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

To use CHIPSEC *in place* without installing it:

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

chipsec_main options

```
usage: chipsec_main.py [options]

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

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

chipsec_util options

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

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

Windows Exception Errors

Windows exception errors have many different causes and at least as many possible solutions. The below ‘fixes’ may need to be used in combination to fix any of these issues.

STATUS_PRIVILEGED_INSTRUCTION (0xC0000096)

This type of error often occurs on newer Windows OSes, often caused by the Windows *Device Guard* feature. THIS IS A KNOWN ISSUE WITH WINDOWS AND NOT A CHIPSEC BUG.

For CHIPSEC to work, the Windows *Device Guard* feature needs to be disabled.

Example errors:

```
ERROR: HW Access Violation: DeviceIoControl returned STATUS_PRIVILEGED_INSTRUCTION (0xC0000096)

chipsec.helper.oshelper.HWAccessViolationError: HW Access Violation: DeviceIoControl returned STATUS_PRIVILEGED_INSTRUCTION (0xC0000096)
```

Device Guard and Credential Guard hardware readiness tool

Device Guard and Credential Guard hardware readiness tool

Steps to disable Device Guard:

1. Download the Microsoft *Device Guard and Credential Guard hardware readiness tool*
2. Extract zip contents
3. Run this command (via PowerShell): `DG_Readiness.ps1 -Disable`
4. Reboot system and press **<F3>** when prompted (twice)
5. May take several reboots before finishing

Note

Microsoft seems to no longer use “Device Guard”, instead now uses WDAC (Windows Defender Application Control).

If the readiness tool doesn’t work…

Workaround:

1. Launch *Local Group Policy Editor*
    a. Run -> `gpedit.msc`
2. Change *Device Guard* setting to *Disabled*
    a. Computer Configuration > Administrative Templates > System > Device Guard
    b. Switch *Device Guard* from “Not Configured” to “Disabled”
3. Save changes and exit *Group Policy Editor*
    4. Reboot

Note

Windows Home version may NOT have *Group Policy Editor* loaded by default.

Windows Home workaround:

There is no guarantee that these steps will work. If they don’t, it is recommended to use *Windows Pro* or *Windows Enterprise* versions that include the *Group Policy Editor*.

1. From a Windows PowerShell *Run as Administrator*, run these commands sequentially:

```
FOR %F IN ("%SystemRoot%\servicing\Packages\Microsoft-Windows-GroupPolicy-ClientTools-Package~*.mum") DO (
DISM /Online /NoRestart /Add-Package:"%F"
)

FOR %F IN ("%SystemRoot%\servicing\Packages\Microsoft-Windows-GroupPolicy-ClientExtensions-Package~*.mum") DO (
DISM /Online /NoRestart /Add-Package:"%F"
)
```

2. After these commands complete, reboot the system. The *Group Policy Editor* should now be installed and ready to run the above steps to disable *Device Guard*.

> **Note**
>
> Occasionally Windows does not include the needed *Administrative template* for *Device Guard* to be visible and the template will have to be updated. In this case, download the below update and run it on the system. Reboot the system and verify that *Device Guard* is now visible.
>
> Updated Windows 10 Administrative Template: https://www.microsoft.com/en-us/download/100591
>
> It is possible that the template update does NOT have the needed permissions to properly update the template. A manual update may be needed. Don't forget to reboot after.

3. Manually copy the **.admx** and **.adml** files from the update folder **C:\Program Files (x86)\Microsoft Group Policy\Windows 10 November 2019 Update (1909)\PolicyDefinitions** to **C:\Windows\PolicyDefinitions**

If this last step does not work, you will have to use a different OS.

ERROR: Exception occurred during …

This error, like the previous issue, appears to be related to a Windows Server OS and its security protections. Often happening when trying to read an MSR, MMIO, or device. Windows Server OS appears to be blocking read access to them. MSR and MMIO reads will simply fail with this error message. Device (PCI) reads appear to mask the device and return all 0xF's, causing an error parsing them.

One or more of the above mentioned solutions *may* fix this.

Example MSR/MMIO error:

```
ERROR: Exception occurred during chipsec.modules.common.smm_dma.run(): ''TSEGMB''
```

Example PCI error:

```
ERROR: Exception occurred during chipsec.modules.common.bios_smi.run(): 'integer out of range for 'H' format code'
```

Windows driver: StartService error

A Windows security feature called HVCI can cause issues running CHIPSEC. HVCI should be disabled. Follow the steps described above to disable HVCI.

Follow these steps to disable HVCI:

How to disable HVCI

Full details on HVCI

Example error messages:

```
ERROR: service 'chipsec' didn't start: The parameter is incorrect. (87)
[-] Traceback (most recent call last):
  File "\chipsec\helper\win\win32helper.py", line 434, in start
    win32serviceutil.StartService( SERVICE_NAME )
  File "\lib\site-packages\win32\lib\win32serviceutil.py", line 417, in StartService
    win32service.StartService(hs, args)
pywintypes.error: (87, 'StartService', 'The parameter is incorrect.')
```

And…

```
chipsec.helper.oshelper.OsHelperError: service 'chipsec' didn't start: The parameter is incorrect. (87)
```

Additional details

Windows Device Guard

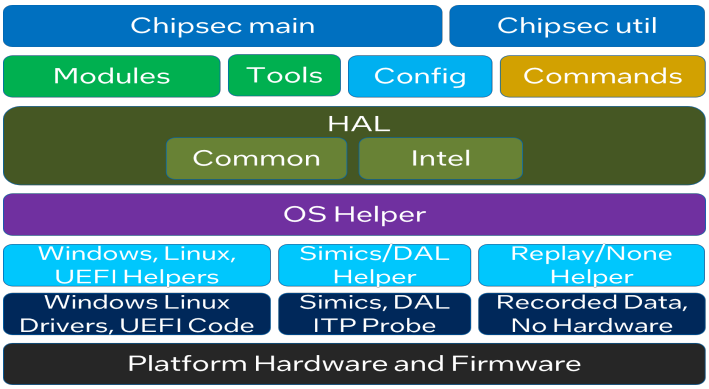
Windows Credential Guard

WDAC – Windows Defender Application Control

Hypervisor-Protected Code Integrity (HVCI)

Module & Command Development

Architecture Overview



CHIPSEC Architecture

Core components

`chipsec_main.py`	main application logic and module execution
`chipsec_util.py`	command-line access to CHIPSEC utility commands
`chipsec/chipset.py`	chipset detection and CHIPSEC initialization
`chipsec/command.py`	base class for utility commands
`chipsec/library/defines.py`	common constants and definitions
`chipsec/library/file.py`	file and path utilities
`chipsec/library/logger.py`	logging functions
`chipsec/module.py`	module discovery and loading
`chipsec/module_common.py`	base class for security modules
`chipsec/library/result_deltas.py`	checking result deltas between test runs

`chipsec/testcase.py`	result reporting and XML/JSON output
`chipsec/helper/basehelper.py`	interface implemented by OS helpers
`chipsec/helper/oshelper.py`	discovers and selects an OS helper backend
`chipsec/hal/`	hardware abstraction layer implementations

Commands

Implement functionality of chipsec_util.

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

Warning

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

Note

All numeric values in the instructions are in hex.

chipsec.utilcmd package

chipsec.utilcmd.acpi_cmd module

Command-line utility providing access to ACPI tables

```
>>> chipsec_util acpi list
>>> chipsec_util acpi table <name>|<file_path>
>>> chipsec_util acpi mmio-regions
```

Examples:

```
>>> chipsec_util acpi list
>>> chipsec_util acpi table XSDT
>>> chipsec_util acpi table acpi_table.bin
>>> chipsec_util acpi mmio-regions
>>> chipsec_util acpi mmio-regions --space systemmemory
```

chipsec.utilcmd.chipset_cmd module

usage as a standalone utility:

```
>>> chipsec_util platform
```

chipsec.utilcmd.cmos_cmd module

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

Examples:

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

chipsec.utilcmd.config_cmd module

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

Examples:

```
>>> chipsec_util config show ALL
>>> chipsec_util config show [CONFIG_PCI|MEMORY_RANGES|MM_MSGBUS|MSGBUS|IO|MSR]
>>> chipsec_util config show SCOPE
```

chipsec.utilcmd.cpu_cmd module

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

Examples:

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

chipsec.utilcmd.decode_cmd module

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

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

Usage:

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

For a list of fw types run:

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

Examples:

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

Note

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

chipsec.utilcmd.deltas_cmd module

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

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

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

chipsec.utilcmd.desc_cmd module

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

IDT command:

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

Examples:

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

GDT command:

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

Examples:

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

LDT command:

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

Examples:

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

chipsec.utilcmd.ec_cmd module

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

Examples:

```
>>> chipsec_util ec dump
>>> chipsec_util ec command 0x001
```

```
>>> chipsec_util ec read    0x2F
>>> chipsec_util ec write   0x2F 0x00
>>> chipsec_util ec index
```

chipsec.utilcmd.hals_cmd module

Requires the Driver. Lists all available HALs (Hardware Abstraction Layers). >>> chipsec_util hals list >>> chipsec_util hals listloadable >>> chipsec_util hals funcs <hal_name>

Examples:

```
>>> chipsec_util hals list
>>> chipsec_util hals listloadable
>>> chipsec_util hals funcs Pci
```

chipsec.utilcmd.hob_cmd module

The hob command provides access to the PI Hand-Off Block (HOB) list.

The HOB list is only resident in memory while running in the EFI environment.

Both list and dump accept optional filters: a HOB type (EFI_HOB_TYPE value) and a full or partial GUID matched against the GUID fields of each HOB.

The read subcommand reports every HOB matching an XML-declared definition, decoded into its named fields.

```
>>> chipsec_util hob list [<type>] [<guid>]
>>> chipsec_util hob dump [<type>] [<guid>]
>>> chipsec_util hob read <register>
```

HOB types (EFI_HOB_TYPE):

> 0x0001 HANDOFF (PHIT) 0x0008 FV2 0x0002 MEMORY_ALLOCATION 0x0009 LOAD_PEIM_UNUSED 0x0003 RESOURCE_DESCRIPTOR 0x000A UEFI_CAPSULE 0x0004 GUID_EXTENSION 0x000B FV3 0x0005 FV 0xFFFE UNUSED 0x0006 CPU 0xFFFF END_OF_HOB_LIST 0x0007 MEMORY_POOL

Examples:

```
>>> chipsec_util hob list
>>> chipsec_util hob list 0x4
>>> chipsec_util hob list 0x4 EA296D92-0B69-423C-8C28-33B4E0A91268
>>> chipsec_util hob list 0 EA296D92
>>> chipsec_util hob dump
>>> chipsec_util hob dump 0x5
>>> chipsec_util hob read 8086.HOB.PEI_PCD_DATABASE
>>> chipsec_util hob read PEI_PCD_DATABASE
```

chipsec.utilcmd.igd_cmd module

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

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

Examples:

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

chipsec.utilcmd.interrupts_cmd module

SMI command:

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

Examples:

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

NMI command:

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

chipsec.utilcmd.io_cmd module

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

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

Examples:

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

chipsec.utilcmd.iommu_cmd module

Command-line utility providing access to IOMMU engines

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

Examples:

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

chipsec.utilcmd.lock_check_cmd module

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

Examples:

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

KEY:

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

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

chipsec.utilcmd.mem_cmd module

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

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

Examples:

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

chipsec.utilcmd.mm_msgbus_cmd module

```
>>> chipsec_util mm_msgbus read     <port> <register>
>>> chipsec_util mm_msgbus write    <port> <register> <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
```

Examples:

```
>>> chipsec_util mm_msgbus read  0x3 0x2E
>>> chipsec_util msgbus mm_write 0x3 0x27 0xE0000001
```

chipsec.utilcmd.mmcfg_base_cmd module

The mmcfg_base command displays PCIe MMCFG Base/Size.

Usage:

```
>>> chipsec_util mmcfg_base
```

Examples:

```
>>> chipsec_util mmcfg_base
```

chipsec.utilcmd.mmcfg_cmd module

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

```
>>> chipsec_util mmcfg base
>>> chipsec_util mmcfg read <bus> <device> <function> <offset> <width>
>>> chipsec_util mmcfg write <bus> <device> <function> <offset> <width> <value>
>>> chipsec_util mmcfg ec
```

Examples:

```
>>> chipsec_util mmcfg base
>>> chipsec_util mmcfg read 0 0 0 0x200 4
>>> chipsec_util mmcfg write 0 0 0 0x200 1 0x1A
>>> chipsec_util mmcfg ec
```

chipsec.utilcmd.mmio_cmd module

```
>>> chipsec_util mmio list
>>> chipsec_util mmio dump <MMIO_BAR_name> [offset] [length] [--instance/-i B:D.F]
>>> chipsec_util mmio dump-abs <MMIO_base_address> [offset] [length]
>>> chipsec_util mmio read <MMIO_BAR_name> <offset> <width> [--instance/-i B:D.F]
>>> chipsec_util mmio read-abs <MMIO_base_address> <offset> <width>
>>> chipsec_util mmio write <MMIO_BAR_name> <offset> <width> <value> [--instance/-i B:D.F]
>>> chipsec_util mmio write-abs <MMIO_base_address> <offset> <width> <value>
```

For BARs with multiple PCI device instances, use –instance/-i B:D.F to select which instance to target. Run ‘mmio list’ to see available instances. When –instance/-i is omitted, the first discovered instance is used.

Examples:

```
>>> chipsec_util mmio list
>>> chipsec_util mmio dump 8086.HOSTCTL.MCHBAR
>>> chipsec_util mmio dump 8086.HOSTCTL.MCHBAR --instance 00:00.0
>>> chipsec_util mmio dump-abs 0xFE010000 0x70 0x10
>>> chipsec_util mmio read SPIBAR 0x74 0x4
>>> chipsec_util mmio read 8086.HOSTCTL.MCHBAR 0x0 0x4 -i 00:00.0
>>> chipsec_util mmio read-abs 0xFE010000 0x74 0x04
>>> chipsec_util mmio write SPIBAR 0x74 0x4 0xFFFF0000
>>> chipsec_util mmio write SPIBAR 0x74 0x4 0xFFFF0000 -i 00:00.0
>>> chipsec_util mmio write-abs 0xFE010000 0x74 0x04 0xFFFF0000
```

chipsec.utilcmd.module_id_cmd module

Generate a module ID using hashlib from the module’s file name (no file extension). Hash is truncated to 28 bits. For module names use full path as seen in example below.

Usage:

`chipsec_util id name <module name>` `chipsec_util id hash <module hash>`

Examples:

```
>>> chipsec_util.py id name chipsec.modules.common.remap
>>> chipsec_util.py id hash 0x67eb58d
```

chipsec.utilcmd.msgbus_cmd module

```
>>> chipsec_util msgbus read     <port> <register>
>>> chipsec_util msgbus 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 message  0x3 0x2E 0x10
>>> chipsec_util msgbus message  0x3 0x2E 0x11 0x0
```

chipsec.utilcmd.msr_cmd module

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

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

Examples:

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

chipsec.utilcmd.pci_cmd module

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

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

Examples:

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

chipsec.utilcmd.reg_cmd module

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

Examples:

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

chipsec.utilcmd.smbios_cmd module

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

Examples:

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

chipsec.utilcmd.smbus_cmd module

SMBus chipsec utility

Usage:

```
>>> chipsec_util smbus read <device_addr> <offset> [size] [--OnSemi] [flags]
>>> chipsec_util smbus read_block <device_addr> <offset> [flags]
>>> chipsec_util smbus write <device_addr> <offset> <byte_val> [size] [--OnSemi] [flags]
>>> chipsec_util smbus process_call <device_addr> <offset> <byte_val> [flags]
>>> chipsec_util smbus scan [<start> [<stop>]] [flags]
>>> chipsec_util smbus dump <device_addr> [<start> [<stop>]] [flags]
```

flags = –addr8b, –mmio, –i2c

Examples:

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

chipsec.utilcmd.spd_cmd module

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

Examples:

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

```
>>> chipsec_util spd read 0xA0 0x0
>>> chipsec_util spd write 0xA0 0x0 0xAA
```

chipsec.utilcmd.spi_cmd module

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

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

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

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

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

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

Examples:

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

chipsec.utilcmd.spidesc_cmd module

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

Examples:

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

chipsec.utilcmd.tpm_cmd module

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

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

Examples:

```
>>> chipsec_util tpm parse_log binary_bios_measurements
>>> chipsec_util tpm state 0
```

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

chipsec.utilcmd.txt_cmd module

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

Usage:

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

chipsec.utilcmd.ucode_cmd module

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

Examples:

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

chipsec.utilcmd.uefi_cmd module

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

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

Examples:

```
>>> chipsec_util uefi types
>>> chipsec_util uefi var-list
>>> chipsec_util uefi var-list-spi
>>> 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 dbx-check x64 dbx_info_msft_latest.json
>>> chipsec_util uefi dbx-check aarch64 C:\dbx\dbx_info_msft_latest.json
>>> 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
```

chipsec.utilcmd.vmem_cmd module

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

```
>>> chipsec_util vmem <op> <virtual_address> <length> [value|buffer_file]
>>>
>>> <virtual_address> : 64-bit virtual 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 <virtual_address>
>>> <buffer_file>      : file with the contents to be written to memory at <virtual_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
```

chipsec.utilcmd.vmm_cmd module

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

Examples:

```
>>> chipsec_util vmm hypercall 32 0 0 0 0 0
>>> chipsec_util vmm pt 0x524B01E
>>> chipsec_util vmm virtio
>>> chipsec_util vmm is_virtio 0 6 0
```

HAL (Hardware Abstraction Layer)

Useful abstractions for common tasks such as accessing the SPI.

chipsec.hal package

chipsec.hal.amd package

chipsec.hal.amd.cpu module

CPU related functionality

chipsec.hal.amd.psp module

chipsec.hal.common package

chipsec.hal.common.acpi module

HAL component providing access to and decoding of ACPI tables

chipsec.hal.common.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 )
```

chipsec.hal.common.cpuid module

CPUID information

usage:

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

chipsec.hal.common.ec module

Access to Embedded Controller (EC)

Usage:

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

chipsec.hal.common.hob module

HAL component providing access to the PI Hand-Off Block (HOB) list

The HOB list is published in the EFI Configuration Table under the gEfiHobListGuid (EFI_HOB_LIST_GUID) vendor table entry and is only available while the HOB list is still resident in memory (i.e. when running in the EFI environment).

chipsec.hal.common.interrupts module

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

usage:

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

chipsec.hal.common.io module

Access to Port I/O

usage:

```
>>> read(0x61, 1)
>>> read_port_byte(0x61)
>>> read_port_word(0x61)
>>> read_port_dword(0x61)
>>> read_range(0x71, 0x4, 1)
>>> dump_range(0x71, 0x4, 1)
>>> write(0x71, 0, 1)
>>> write_port_byte(0x61, 0xAA)
>>> write_port_word(0x61, 0xAAAA)
>>> write_port_dword(0x61, 0xAAAAAAAA)
```

chipsec.hal.common.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 )
```

chipsec.hal.common.iommu module

Access to IOMMU engines - uses DMAR table for dynamic VT-d engine discovery

chipsec.hal.common.locks module

chipsec.hal.common.memrange 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 )
```

chipsec.hal.common.mmio module

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

usage:

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

Access MMIO by BAR name:

```
>>> read_MMIO_BAR_reg('MCHBAR', 0x0, 4)
>>> write_MMIO_BAR_reg('MCHBAR', 0x0, 0xFFFFFFFF, 4)
>>> get_MMIO_BAR_base_address('8086.HOSTCTL.MCHBAR')
>>> is_MMIO_BAR_enabled('8086.HOSTCTL.MCHBAR')
>>> is_MMIO_BAR_programmed('8086.HOSTCTL.MCHBAR')
>>> dump_MMIO_BAR('8086.HOSTCTL.MCHBAR')
>>> list_MMIO_BARs()
```

chipsec.hal.common.msr module

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

usage:

```
>>> read( 0x8B )
>>> write( 0x79, 0x12345678 )
>>> get_cpu_thread_count()
>>> get_IDTR( 0 )
>>> get_GDTR( 0 )
>>> dump_Descriptor_Table( 0, DESCRIPTOR_TABLE_CODE_IDTR )
>>> IDT( 0 )
>>> GDT( 0 )
>>> IDT_all()
>>> GDT_all()
```

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

chipsec.hal.common.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 )
```

chipsec.hal.common.smbios module

HAL component providing access to and decoding of SMBIOS structures

chipsec.hal.common.smbus module

Access to SMBus Controller

chipsec.hal.common.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

chipsec.hal.common.tpm module

Trusted Platform Module (TPM) HAL component

https://trustedcomputinggroup.org

chipsec.hal.common.uefi module

Main UEFI component using platform specific and common UEFI functionality

chipsec.hal.common.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 )
```

chipsec.hal.common.vmm module

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

chipsec.hal.intel package

chipsec.hal.intel.cpu module

CPU related functionality

chipsec.hal.intel.igd module

Working with Intel processor Integrated Graphics Device (IGD)

usage:

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

chipsec.hal.intel.mm_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:

```
>>> get_sbreg_base_address( )
>>> read( port, register )
>>> write( port, register, data )
```

chipsec.hal.intel.mmcfg module

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

usage:

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)
```

chipsec.hal.intel.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 )
```

chipsec.hal.intel.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)

chipsec.hal.intel.spi_descriptor module

SPI Flash Descriptor binary parsing functionality

usage:

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

chipsec.hal.intel.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' )
```

chipsec.hal.hal_base module

Base for HAL Components

chipsec.hal.hals module

Fuzzing

chipsec.fuzzing package

chipsec.fuzzing.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

CHIPSEC configuration maps platform and device identities to register, field, control and BAR definitions. Platform files are stored under `chipsec/cfg/<vendor-id>/`; reusable definitions are grouped into XML fragments under IP directories. Platform detection selects the platform files, whose device entries refer to those fragments with `config`.

For example, `chipsec/cfg/8086/kbl.xml` selects a configuration fragment for the SPI device:

```
<pci>
   <device name="SPI" bus="0x00" dev="0x1F" fun="5"
              config="SPI.spi0.xml" />
</pci>
```

The referenced `chipsec/cfg/8086/SPI/spi0.xml` contains the register and control definitions:

```
<configuration>
   <registers>
      <register name="BC" type="pcicfg" offset="0xDC" size="4"
                   desc="BIOS Control">
         <field name="BLE" bit="1" size="1" desc="BIOS Lock Enable" />
      </register>
   </registers>
   <controls>
      <control name="BiosLockEnable" register="BC" field="BLE"
                  desc="BIOS Lock Enable" />
   </controls>
</configuration>
```

The platform XML is parsed for device and other platform configuration sections; register, control and lock definitions belong in referenced configuration fragments. Placing those sections directly in a platform XML does not load them in the current configuration pipeline. Follow a nearby platform and its referenced fragments when adding definitions.

List of Cfg components

arl

Path: chipsec\cfg\8086\arl.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

pch_3xxlp_orig.

Path: chipsec\cfg\8086\pch_3xxlp_orig.xmls

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

kbl_orig.

Path: chipsec\cfg\8086\kbl_orig.xmls

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

pmc_i440fx

Path: chipsec\cfg\8086\pmc_i440fx.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

q35

Path: chipsec\cfg\8086\q35.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

tglu

Path: chipsec\cfg\8086\tglu.xml

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

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

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

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

Contact information: chipsec@intel.com

kbl

Path: chipsec\cfg\8086\kbl.xml

XML configuration file for Ice Lake

lnl

Path: chipsec\cfg\8086\lnl.xml

XML configuration file for Lunar Lake

pch_7xxOP

Path: chipsec\cfg\8086\pch_7xxOP.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

pch_6xxp

Path: chipsec\cfg\8086\pch_6xxp.xml

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

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

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

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

Contact information: chipsec@intel.com

pch_7xx

Path: chipsec\cfg\8086\pch_7xx.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

adl

Path: chipsec\cfg\8086\adl.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2023, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

pch_8xx

Path: chipsec\cfg\8086\pch_8xx.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

mtl

Path: chipsec\cfg\8086\mtl.xml

XML configuration file for Ice Lake

rpl

Path: chipsec\cfg\8086\rpl.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

arlh

Path: chipsec\cfg\8086\arlh.xml

CHIPSEC: Platform Security Assessment Framework Copyright (c) 2026, Intel Corporation

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

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

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

Contact information: chipsec@intel.com

pch_5xxlp

Path: chipsec\cfg\8086\pch_5xxlp.xml

XML configuration file for 5XX series pch

Developer Guide

CHIPSEC modules are Python classes that assess or exercise a platform through the CHIPSEC framework. Module code runs with privileged hardware access on supported systems. Develop and test on a dedicated test system, and understand the effect of every hardware read or write before running a module.

Module layout and lifecycle

Place modules under `chipsec/modules/`. Use `common/` for checks intended to work across supported platforms, or a platform-code directory for a platform-specific check. The module name is its path relative to `chipsec/modules` without the `.py` suffix; for example, `chipsec/modules/common/bios_wp.py` is run as `common.bios_wp`. The module class name must match the Python file's base name.

Subclass `BaseModule` from `chipsec.module_common`. CHIPSEC calls `is_supported()` before `run(module_argv)`. Return `False` from `is_supported` when required hardware or configuration is unavailable; the module is then reported as not applicable. `run` receives the arguments provided with `-a` and returns the module result/return code. See Sample Module Code for a starting point.

Core APIs

`BaseModule` provides these commonly used objects:

- `self.cs` **is the active chipset object. Use its configuration interfaces**
  to resolve platform-specific registers and controls instead of duplicating platform detection in a module.
- `self.cs.register` **accesses configured registers. Use**
  `get_list_by_name(name)` to get an `ObjList` of defined and enabled registers, then call `objlist.read()` to read their values.
- `self.cs.control` **accesses abstract controls. A control maps a logical**

security setting to the register and field used by a platform. `get_list_by_name(name)` returns an `ObjList` of defined and enabled controls; call `objlist.read()` to read their values.

- `self.cs.device` **provides configured device information. For common**

  hardware operations, use the corresponding HAL exposed under `self.cs.hals` (for example, `self.cs.hals.spi` or `self.cs.hals.uefi`). See Architecture Overview and the generated CHIPSEC HAL API documentation for the available HALs.

- `self.logger` **provides** `start_test`**,** `log`**,** `log_verbose`**,**

  `log_warning`, `log_passed`, `log_failed` and `log_important` for consistent output.

- `self.result` **tracks return-code status bits. Use**

  `self.result.setStatusBit(...)` to add a meaningful status such as `LOCKS`, `CONFIGURATION` or `VERIFY`. Follow nearby modules for the result style used in that part of the codebase; CHIPSEC retains the legacy `ModuleResult` interface for compatibility.

Platform controls are declared in configuration XML, not hard-coded into a portable module. For example:

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

A module can then query the abstract control without knowing which register implements it on each platform:

```
lock = self.cs.control.get_list_by_name('BiosLockEnable')
lock.read_and_verbose_print()
if lock.is_all_value(1):
        self.logger.log_passed('BIOS Lock is set.')
else:
        self.logger.log_failed('BIOS Lock is not set.')
```

See Configuration Files for configuration structure and examples.

Development practices

- **Prefer read-only checks. If a check must modify hardware state, make the**

  change explicit, preserve the original value, and restore it on every exit path. Document the risk and require an appropriate test platform.

- **Use abstract controls and existing HALs so modules can be reused across**

  platform generations. Put platform-specific register mappings in configuration XML.

- **Treat missing definitions, disabled devices, failed reads and all-ones**

  reads as unknown or unsupported input, not as evidence that a protection passed or failed. Report a warning or not-applicable result as appropriate.

- **Validate assumptions and ranges before using hardware-derived values. Keep**

  parsing and result handling explicit, and avoid swallowing exceptions that indicate a programming error.

- **Keep checks focused, log the observed state and expected state, and add**

  unit tests for parsing, decision logic and error paths. Follow the Python version and style rules in Python Version.

- **Add module tags and metadata that accurately describe the module. Use**

  existing modules in the same directory as examples of repository conventions.

Debugging and testing

Start with a focused run from the CHIPSEC repository root:

```
python chipsec_main.py -m common.example
```

Increase diagnostic output with `-v` or `-vv` (verbose, HAL and debug logging), and save it with `-l example.log`. Use `--failfast` when you need exceptions to stop the run rather than be masked by normal module handling. `--no-driver` is only suitable for operations that do not need kernel-driver access.

When diagnosing a module:

- **If it is not found, check the module path/name, class name, and any** additional module path passed with `-I`.
- **If it is not applicable, inspect** `is_supported` **and verify that the** platform configuration defines and enables each required device, register and control.
- **If a result is unexpected, use verbose output to inspect the values read,** then confirm the register mapping and expected semantics against the platform documentation.
- **If the module raises an exception, reproduce it with a single-module run** and `--failfast`; add a focused test for the failing input or decision path.

Run a focused unit test from the repository root, for example:

```
python -m pytest tests/modules/test_example.py -q
```

Hardware validation is separate from unit testing and must be performed only on an appropriate test system. See Running CHIPSEC for command line options and the contribution guide for contribution requirements.

Seealso

- Architecture Overview
- Configuration Files
- Sample Module Code
- Creating CHIPSEC modules and commands

OS Helpers and Drivers

OS helpers provide low-level, operating-system-specific access to drivers and system interfaces such as PCI configuration space, MSRs, physical memory and UEFI variables. HAL implementations use the active helper to perform these operations. Module authors should normally use the HAL API rather than invoke the helper directly.

OS helper backends live in `chipsec/helper/<name>/<name>helper.py`. During initialization, `OsHelper` discovers these packages and selects the backend for the current environment. Each backend exports a `get_helper()` factory that returns its helper instance.

Implementing a new backend

Subclass `Helper` from `chipsec.helper.basehelper` and implement its abstract operations. The interface includes driver lifecycle, PCI/MMIO, physical memory, I/O ports, CPU registers, CPUID, and UEFI/ACPI services. Implement only the behavior the target environment supports, and follow an existing backend such as `chipsec/helper/linux/linuxhelper.py` for its module structure and `get_helper()` factory. A helper backend is a platform-integration change; ordinary security checks should be implemented as modules and use the existing HALs.

Helper components

chipsec.helper package

chipsec.helper.dal package

chipsec.helper.dal.dalhelper module

Intel DFx Abstraction Layer (DAL) helper

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

chipsec.helper.efi package

chipsec.helper.efi.efihelper module

On UEFI use the efi package functions

chipsec.helper.linux package

chipsec.helper.linux.linuxhelper module

Linux helper

chipsec.helper.linuxnative package

chipsec.helper.linuxnative.cpuid module

chipsec.helper.linuxnative.legacy_pci module

chipsec.helper.linuxnative.linuxnativehelper module

Native Linux helper

chipsec.helper.record package

chipsec.helper.record.recordhelper module

chipsec.helper.replay package

chipsec.helper.replay.replayhelper module

chipsec.helper.windows package

chipsec.helper.windows.windowshelper module

chipsec.helper.basehelper module

chipsec.helper.nonehelper module

chipsec.helper.oshelper module

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

Platform Detection

CHIPSEC identifies the processor platform and, where applicable, the PCH using the SKU metadata and PCI device IDs in its configuration tree. PCI enumeration provides device identities; CPU configuration can also use CPUID detection values to distinguish platforms that share device IDs. The platform-detection and configuration-loading flow is implemented by `chipsec/chipset.py` and `chipsec/config.py`.

Platform Configuration Options

Use `-p` or `--platform` to request a platform code explicitly. Use `--pch` to request a PCH code. These options select configuration data; they do not change the hardware being detected. The `-i` / `--ignore_platform` option is deprecated and should not be used for normal operation.

Warning

Using `-p`, `--pch` or `-i` is not recommended. These options can cause many devices and registers to be detected incorrectly or not detected at all. Use them only if you understand their effects. Their practical behavior changed between CHIPSEC 1.x.x and 2.x.x.

Sample Module Code

This template shows the minimum structure for a module named `common.example`. Save it as `chipsec/modules/common/example.py` and replace `ExampleControl` with a control defined for the platforms the module supports. The class name must match the file's base name.

```
from typing import List

from chipsec.library.returncode import ModuleResult
from chipsec.module_common import BaseModule


class example(BaseModule):
    def __init__(self):
        super().__init__()

    def is_supported(self) -> bool:
        self.control = self.cs.control.get_list_by_name('ExampleControl')
        return bool(self.control)

    def run(self, module_argv: List[str]) -> int:
        self.logger.start_test('Example control check')
        self.control.read_and_verbose_print()

        if self.control.is_all_value(1):
            self.logger.log_passed('Example control is set.')
            result = ModuleResult.PASSED
        else:
            self.logger.log_failed('Example control is not set.')
            result = ModuleResult.FAILED

        return self.result.getReturnCode(result)
```

Run the module with `python chipsec_main.py -m common.example`. Modules that accept arguments receive them as a list in `module_argv`; invoke them with `-a`. For module APIs, result handling, safety practices and debugging, see Developer Guide.

Util Command

```
# chipsec_util.py commands live in chipsec/utilcmd/
# Example file name: <command_display_name>_cmd.py

from argparse import ArgumentParser

from chipsec.command import BaseCommand, toLoad

class CommandClass(BaseCommand):
    """
        >>> chipsec_util command_display_name action
    """
    def requirements(self) -> toLoad:
        return toLoad.All

    def parse_arguments(self):
        parser = ArgumentParser(prog='chipsec_util command_display_name', usage=CommandClass.__doc__)
        subparsers = parser.add_subparsers()
        parser_entrypoint = subparsers.add_parser('action')
        parser_entrypoint.set_defaults(func=self.action)
        parser.parse_args(self.argv, namespace=self)

    def action(self):
        return

    def run(self):
        self.func()

commands = {'command_display_name': CommandClass}
```

CHIPSEC Modules

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

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

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

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

Known vulnerabilities can be mapped to CHIPSEC modules as follows:

Attack Surface/Vector: Firmware protections in ROM

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

Attack Surface/Vector: Runtime protection of SMRAM

Vulnerability Description	CHIPSEC Module	Example
Compatibility SMRAM is not locked	common.smm	
SMM cache attack	common.smrr	
Memory remapping vulnerability in SMM protection	remap	
DMA protections of SMRAM are not in use	smm_dma	
Graphics aperture redirection of SMRAM	Manual analysis (no dedicated current utility command)	
Memory sinkhole vulnerability	tools.cpu.sinkhole	

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

Vulnerability Description	CHIPSEC Module	Example
Root certificate	common.bios_wp, common.secureboot.variables	
Key exchange keys	common.secureboot.variables	
Controls in setup variable (CSM enable/disable, image verification policies, secure boot enable/disable, clear/restore keys)	chipsec_util uefi var-find Setup	

TE header confusion	tools.secureboot.te	
UEFI NVRAM is not write protected	common.bios_wp	
Insecure handling of secure boot disable	chipsec_util uefi var-list	

Attack Surface/Vector: Persistent firmware configuration

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

Attack Surface/Vector: Platform hardware configuration

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

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

Vulnerability Description	CHIPSEC Module	Example
SMI handlers use pointers/addresses from OS without validation	tools.smm.smm_ptr	
Legacy SMI handlers call legacy BIOS outside SMRAM		
INT15 in legacy SMI handlers		
UEFI SMI handlers call UEFI services outside SMRAM		
Malicious CommBuffer pointer and contents		
Race condition during SMI handler		
Authenticated variables SMI handler is not implemented	chipsec_util uefi var-write	
SmmRuntime vulnerability	tools.uefi.scan_blocked	

Attack Surface/Vector: Boot time firmware

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

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

Vulnerability Description	CHIPSEC Module	Example
Insufficient protection of S3 boot script table	common.uefi.s3bootscript, tools.uefi.s3script_modify	
Dispatch opcodes in S3 boot script call functions in unprotected memory	common.uefi.s3bootscript, tools.uefi.s3script_modify	

S3 boot script interpreter stored in unprotected memory		
Pointer to S3 boot script table in unprotected UEFI variable	common.uefi.s3bootscript, tools.uefi.s3script_modify	
Critical setting not recorded in S3 boot script table	chipsec_util uefi s3bootscript (manual analysis)	
OS waking vector in ACPI tables can be modified	chipsec_util acpi dump (manual analysis)	
Using pointers on S3 resume stored in unprotected UEFI variables	chipsec_util uefi var-write	

Attack Surface/Vector: Firmware update

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

Attack Surface/Vector: Network interfaces

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

Attack Surface/Vector: Misc

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

Modules

chipsec.modules package

chipsec.modules.bdw package

chipsec.modules.byt package

chipsec.modules.common package

chipsec.modules.common.cpu package

chipsec.modules.common.cpu.cpu_info module

Displays CPU information

Reference:

- **Intel 64 and IA-32 Architectures Software Developer Manual (SDM)**
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Usage:

```
chipsec_main -m common.cpu.cpu_info
```

Examples:

```
>>> chipsec_main.py -m common.cpu.cpu_info
```

Registers used:

- IA32_BIOS_SIGN_ID.MICROCODE

chipsec.modules.common.cpu.ia_untrusted module

IA Untrusted checks

Usage:

```
chipsec_main -m common.cpu.ia_untrusted
```

Examples:

```
>>> chipsec_main.py -m common.cpu.ia_untrusted
```

Registers used:

- MSR_BIOS_DONE.IA_UNTRUSTED

- MSR_BIOS_DONE.SoC_BIOS_DONE

chipsec.modules.common.cpu.spectre_v2 module

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

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

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

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

The module returns the following results:

FAILED:

IBRS/IBPB is not supported

WARNING:

IBRS/IBPB is supported

Enhanced IBRS is not supported

WARNING:

IBRS/IBPB is supported

Enhanced IBRS is supported

Enhanced IBRS is not enabled by the OS

WARNING:

IBRS/IBPB is supported

STIBP is not supported or not enabled by the OS

PASSED:

IBRS/IBPB is supported

Enhanced IBRS is supported

Enhanced IBRS is enabled by the OS

STIBP is supported

Notes:

- The module returns WARNING when CPU doesn't support enhanced IBRS Even though OS/software may use basic IBRS by setting IA32_SPEC_CTRL[IBRS] when necessary, we have no way to verify this

- The module returns WARNING when CPU supports enhanced IBRS but OS doesn't set IA32_SPEC_CTRL[IBRS] Under enhanced IBRS, OS can set IA32_SPEC_CTRL[IBRS] once to take advantage of IBRS protection
- The module returns WARNING when CPU doesn't support STIBP or OS doesn't enable it Per Speculative Execution Side Channel Mitigations: "enabling IBRS prevents software operating on one logical processor from controlling the predicted targets of indirect branches executed on another logical processor. For that reason, it is not necessary to enable STIBP when IBRS is enabled"
- OS/software may implement "retpoline" mitigation for Spectre variant 2 instead of using CPU hardware IBRS/IBPB

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

Hardware registers used:

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

References:

- Reading privileged memory with a side-channel by Jann Horn, Google Project Zero: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- Spectre: https://spectreattack.com/spectre.pdf
- Meltdown: https://meltdownattack.com/meltdown.pdf
- Speculative Execution Side Channel Mitigations: https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf
- Retpoline: a software construct for preventing branch-target-injection: https://support.google.com/faqs/answer/7625886

chipsec.modules.common.secureboot package

chipsec.modules.common.secureboot.variables module

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

Reference:

- UEFI 2.4 spec Section 28

Usage:

`chipsec_main -m common.secureboot.variables [-a modify]` - `-a` : modify = will try to write/corrupt the variables

Where:

- [ ]: optional line

Examples:

```
>>> chipsec_main.py -m common.secureboot.variables
>>> chipsec_main.py -m common.secureboot.variables -a modify
```

Note

- Module is not supported in all environments.

chipsec.modules.common.uefi package

chipsec.modules.common.uefi.access_uefispec module

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

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

usage:

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

- [-a modify[,attribute]]: Attempt to modify each variable in addition to checking attributes; optionally change the attribute as well

Where:

- [ ]: optional line

Examples:

```
>>> chipsec_main.py -m common.uefi.access_uefispec
>>> chipsec_main.py -m common.uefi.access_uefispec -a modify
>>> chipsec_main.py -m common.uefi.access_uefispec -a modify,7
```

NOTE: Requires an OS with UEFI Runtime API support.

chipsec.modules.common.uefi.s3bootscript module

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

References:

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

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

Attacks on UEFI Security by Rafal Wojtczuk and Corey Kallenberg.

Attacking UEFI Boot Script by Rafal Wojtczuk and Corey Kallenberg.

Exploiting UEFI boot script table vulnerability by Dmytro Oleksiuk.

Usage:

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

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

Where:

- `[]`: optional line

Examples:

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

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

chipsec.modules.common.bios_kbrd_buffer module

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

Reference:

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

Usage:

```
chipsec_main -m common.bios_kbrd_buffer
```

Examples:

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

chipsec.modules.common.bios_smi module

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

References:

- Setup for Failure: Defeating SecureBoot by Corey Kallenberg, Xeno Kovah, John Butterworth, Sam Cornwell
- Summary of Attacks Against BIOS and Secure Boot

Usage:

```
chipsec_main -m common.bios_smi
```

Examples:

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

Registers used:

- SmmBiosWriteProtection (Control)
- TCOSMILock (Control)
- SMILock (Control)
- BiosWriteEnable (Control)

chipsec.modules.common.bios_ts module

Checks for BIOS Interface Lock including Top Swap Mode

References:

- BIOS Boot Hijacking and VMware Vulnerabilities Digging by Bing Sun

Usage:

```
chipsec_main -m common.bios_ts
```

Examples:

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

Registers used:

- BiosInterfaceLockDown (control)
- TopSwapStatus (control)
- TopSwap (control)

chipsec.modules.common.bios_wp module

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

This module checks both mechanisms. In order to pass this test using SPI controller configuration, the SPI Protected Range registers (PR0-4) will need to cover the entire BIOS region. Often, if this configuration is used at all, it is used only to protect part of the BIOS region (usually the boot block). If other important data (eg. NVRAM) is not protected, however, some vulnerabilities may be possible.

A Tale of One Software Bypass of Windows 8 Secure Boot In a system where certain BIOS data was not protected, malware may be able to write to the Platform Key stored on the flash, thereby disabling secure boot.

SMM based write protection is controlled from the BIOS Control Register. When the BIOS Write Protect Disable bit is set (sometimes called BIOSWE or BIOS Write Enable), then writes are allowed. When cleared, it can also be locked with the BIOS Lock Enable (BLE) bit. When locked, attempts to change the WPD bit will result in generation of an SMI. This way, the SMI handler can decide whether to perform the write.

As demonstrated in the Speed Racer issue, a race condition may exist between the outstanding write and processing of the SMI that is generated. For this reason, the EISS bit (sometimes called SMM_BWP or SMM BIOS Write Protection) must be set to ensure that only SMM can write to the SPI flash.

References:

- A Tale of One Software Bypass of Windows 8 Secure Boot
- Speed Racer

Usage:

```
chipsec_main -m common.bios_wp
```

Examples:

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

Registers used: (n = 0,1,2,3,4)

- BiosLockEnable (Control)
- BiosWriteEnable (Control)

- SmmBiosWriteProtection (Control)
- PRn.PRB
- PRn.RPE
- PRn.PRL
- PRn.WPE

Note

- Module will fail if SMM-based protection is not correctly configured and SPI protected ranges (PR registers) do not protect the entire BIOS region.

chipsec.modules.common.cet module

Reports CET Settings

Usage:

```
chipsec_main -m common.cet
```

Example:

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

Note

- cpuid(7, 0) must return 1 at bit 7 to run
- IA32_U_CET and IA32_S_CET must be defined for addition information.
- Module is INFORMATION only and does NOT return a Pass/Fail

chipsec.modules.common.debugenabled module

This module checks if the system has debug features turned on, specifically the Direct Connect Interface (DCI).

This module checks the following bits: 1. HDCIEN bit in the DCI Control Register 2. Debug enable bit in the IA32_DEBUG_INTERFACE MSR 3. Debug lock bit in the IA32_DEBUG_INTERFACE MSR 4. Debug occurred bit in the IA32_DEBUG_INTERFACE MSR

Usage:

```
chipsec_main -m common.debugenabled
```

Examples:

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

The module returns the following results:

- **FAILED** : Any one of the debug features is enabled or unlocked.
- **PASSED** : All debug feature are disabled and locked.

Registers used:

- IA32_DEBUG_INTERFACE[DEBUGENABLE]
- IA32_DEBUG_INTERFACE[DEBUGELOCK]
- IA32_DEBUG_INTERFACE[DEBUGEOCCURED]
- P2SB_DCI.DCI_CONTROL_REG[HDCIEN]

chipsec.modules.common.firmware_info module

Displays firmware (BIOS/UEFI) information gathered from SMBIOS and OS interfaces.

Usage:

```
chipsec_main -m common.firmware_info
```

Examples:

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

chipsec.modules.common.ia32cfg module

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

Reference:

- **Intel 64 and IA-32 Architectures Software Developer Manual (SDM)**
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Usage:

```
chipsec_main -m common.ia32cfg
```

Examples:

```
>>> chipsec_main.py -m common.ia32cfg
```

Registers used:

- IA32_FEATURE_CONTROL
- Ia32FeatureControlLock (control)

chipsec.modules.common.me_mfg_mode module

This module checks that ME Manufacturing mode is not enabled.

References:

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

PCI_DEVS.H

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

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

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

PCH.H

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

ME.H

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

ME_STATUS.C

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

This module checks the following:

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

Usage:

`chipsec_main -m common.me_mfg_mode`

Examples:

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

The module returns the following results:

FAILED : HFS.MFG_MODE is set

PASSED : HFS.MFG_MODE is not set.

Hardware registers used:

- HFS.MFG_MODE

chipsec.modules.common.memconfig module

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

Usage:

`chipsec_main -m common.memconfig`

Example:

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

Note

- This module will only run on Core (client) platforms.

chipsec.modules.common.memlock module

This module checks if memory configuration is locked to protect SMM

Reference:

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

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

The module returns the following results:

- **FAILED** : LT_LOCK_MEMORY[0] is not set
- **PASSED** : LT_LOCK_MEMORY[0] is set
- **ERROR** : Problem reading LT_LOCK_MEMORY values

Usage:

```
chipsec_main -m common.memlock
```

Example:

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

Registers used:

- LT_LOCK_MEMORY

Note

- This module will not run on Atom based platforms.

chipsec.modules.common.remap module

Check Memory Remapping Configuration

Reference:

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

Usage:

```
chipsec_main -m common.remap
```

Example:

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

Registers used:

- 8086.HOSTCTL.MCHBAR*.REMAPBASE

- 8086.HOSTCTL.MCHBAR*.REMAPLIMIT
- 8086.HOSTCTL.TOUUD
- 8086.HOSTCTL.TOLUD
- 8086.HOSTCTL.TSEGMB

Note

- This module will only run on Core platforms.

chipsec.modules.common.rom_armor module

This module verifies support for Rom Armor and SPI ROM protections.

Reference:

usage:

```
chipsec_main -m common.rom_armor
```

Examples:

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

chipsec.modules.common.rtclock module

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

Usage:

```
chipsec_main -m common.rtclock [-a modify]
```

- `-a modify`: Attempt to modify CMOS values

Examples:

```
>>> chipsec_main.py -m common.rtclock
>>> chipsec_main.py -m common.rtclock -a modify
```

Registers used:

- RC.LL
- RC.UL

Note

- This module will only run on Core platforms

chipsec.modules.common.sgx_check module

Check SGX related configuration

Reference:

- SGX BWG, CDI/IBP#: 565432

Usage:

```
chipsec_main -m common.sgx_check
```

Examples:

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

Registers used:

- IA32_FEATURE_CONTROL.SGX_GLOBAL_EN
- IA32_FEATURE_CONTROL.LOCK
- IA32_DEBUG_INTERFACE.ENABLE
- IA32_DEBUG_INTERFACE.LOCK
- MTRRCAP.PRMRR
- PRMRR_VALID_CONFIG
- PRMRR_PHYBASE.PRMRR_BASE_ADDRESS_FIELDS
- PRMRR_PHYBASE.PRMRR_MEMTYPE
- PRMRR_MASK.PRMRR_mask_bits
- PRMRR_MASK.PRMRR_VLD
- PRMRR_MASK.PRMRR_LOCK
- PRMRR_UNCORE_PHYBASE.PRMRR_BASE_ADDRESS_FIELDS
- PRMRR_UNCORE_MASK.PRMRR_mask_bits
- PRMRR_UNCORE_MASK.PRMRR_VLD
- PRMRR_UNCORE_MASK.PRMRR_LOCK
- BIOS_SE_SVN.PFAT_SE_SVN
- BIOS_SE_SVN.ANC_SE_SVN
- BIOS_SE_SVN.SCLEAN_SE_SVN
- BIOS_SE_SVN.SINIT_SE_SVN
- BIOS_SE_SVN_STATUS.LOCK
- SGX_DEBUG_MODE.SGX_DEBUG_MODE_STATUS_BIT

Note

- Will not run within the EFI Shell

chipsec.modules.common.sgx_check_sidekick module

chipsec.modules.common.smm module

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

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

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

usage:

```
chipsec_main -m common.smm
```

Examples:

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

This module will only run on client (core) platforms that have SMRAMC defined.

chipsec.modules.common.smm_addr module

CPU SMM Addr

This module checks to see that SMMMask has Tseg and Aseg programmed correctly. It also verifies that CPU access to SMM is blocked while not in SMM.

Usage:

```
chipsec_main -m common.smm_addr
```

Examples:

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

Registers used:

AMD: - SMMMASK.TMTypeDram - SMMMASK.AMTypeDram - SMMMASK.TValid - SMMMASK.AValid - SMM_BASE

chipsec.modules.common.smm_close module

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

Reference: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7014.html

usage:

```
chipsec_main -m common.smm_close
```

Examples:

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

This module will only run on AMD platforms with SMMMask defined.

chipsec.modules.common.smm_code_chk module

SMM_CODE_CHK_EN (SMM Call-Out) Protection check

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

Reference:

- **Intel 64 and IA-32 Architectures Software Developer Manual (SDM)**
  - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Usage:

```
chipsec_main -m common.smm_code_chk
```

Examples:

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

Registers used:

- SMM_FEATURE_CONTROL.LOCK
- SMM_FEATURE_CONTROL.SMM_CODE_CHK_EN

> **Note**
>
> - SMM_FEATURE_CONTROL may not be defined or readable on all platforms.

chipsec.modules.common.smm_dma module

SMM TSEG Range Configuration Checks

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

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

References:

- System Management Mode Design and Security Issues
- Summary of Attack against BIOS and Secure Boot

Usage:

```
chipsec_main -m smm_dma
```

Examples:

```
>>> chipsec_main.py -m smm_dma
```

Registers used:

- TSEGBaseLock (control)

- TSEGLimitLock (control)
- MSR_BIOS_DONE.IA_UNTRUSTED
- PCI0.0.0_TSEGMB.TSEGMB
- PCI0.0.0_BGSM.BGSM
- IA32_SMRR_PHYSBASE.PHYSBASE
- IA32_SMRR_PHYSMASK.PHYSMASK

Supported Platforms:

- Core (client)

chipsec.modules.common.smm_lock module

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

Reference:

usage:

```
chipsec_main -m common.smm_lock
```

Examples:

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

This module will only run on platforms that have HWCR defined.

chipsec.modules.common.smrr module

CPU SMM Cache Poisoning / System Management Range Registers check

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

Reference:

Researchers demonstrated a way to use CPU cache to effectively change values in SMRAM in Attacking SMM Memory via Intel CPU Cache Poisoning and Getting into the SMRAM: SMM Reloaded . If ring 0 software can make SMRAM cacheable and then populate cache lines at SMBASE with exploit code, then when an SMI is triggered, the CPU could execute the exploit code from cache. System Management Mode Range Registers (SMRRs) force non-cachable behavior and block access to SMRAM when the CPU is not in SMM. These registers need to be enabled/configured by the BIOS.

Usage:

```
chipsec_main -m common.smrr [-a modify]
```

- `-a modify`: Attempt to modify memory at SMRR base

Examples:

```
>>> chipsec_main.py -m common.smrr
>>> chipsec_main.py -m common.smrr -a modify
```

Registers used:

- IA32_SMRR_PHYSBASE.PHYSBASE
- IA32_SMRR_PHYSBASE.TYPE
- IA32_SMRR_PHYSMASK.PHYSMASK

- IA32_SMRR_PHYSMASK.VALID

chipsec.modules.common.spd_wd module

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

References:

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

This module checks the following:

HCFG.SPD_WD

The module returns the following results:

PASSED : HCFG.SPD_WD is set

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

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

Hardware registers used:

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
> - HCFG.SPD_WD is defined for the platform

chipsec.modules.common.spi_access module

SPI Flash Region Access Control

Checks SPI Flash Region Access Permissions programmed in the Flash Descriptor

Usage:

```
chipsec_main -m common.spi_access
```

Examples:

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

Registers used:

- 8086.SPI.HSFS.FDV
- 8086.SPI.FRAP.BRWA

Important

- Some platforms may use alternate means of protecting these regions. Consider this when assessing results.

chipsec.modules.common.spi_desc module

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

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

Usage:

```
chipsec_main -m common.spi_desc
```

Examples:

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

Registers used:

- FRAP.BRRA
- FRAP.BRWA

chipsec.modules.common.spi_fdopss module

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

Usage:

```
chipsec_main -m common.spi_fdopss
```

Examples:

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

Registers used:

- HSFS.FDOPSS

chipsec.modules.common.spi_lock module

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

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

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

Reference:

- flashrom

- Copernicus: Question Your Assumptions about BIOS Security

Usage:

```
chipsec_main -m common.spi_lock
```

Examples:

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

Registers used:

- FlashLockDown (control)
- SpiWriteStatusDis (control)

chipsec.modules.tools package

chipsec.modules.tools.cpu package

chipsec.modules.tools.cpu.sinkhole module

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

References:

- Memory Sinkhole presentation by Christopher Domas
- Memory Sinkhole whitepaper

Usage:

```
chipsec_main -m tools.cpu.sinkhole
```

Examples:

```
>>> chipsec_main.py -m tools.cpu.sinkhole
```

Registers used:

- IA32_APIC_BASE.APICBase
- IA32_SMRR_PHYSBASE.PHYSBASE
- IA32_SMRR_PHYSMASK

Note

- Supported OS: Windows or Linux

Warning

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

chipsec.modules.tools.secureboot package

chipsec.modules.tools.secureboot.te module

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

Usage:

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

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

Examples:

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

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

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

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

Restore bootloaders listed in te.cfg file:

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

chipsec.modules.tools.smm package

chipsec.modules.tools.smm.rogue_mmio_bar module

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

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

Usage:

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

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

Example:

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

Note

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

Warning

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

chipsec.modules.tools.smm.smm_ptr module

A tool to test SMI handlers for pointer validation vulnerabilities

Reference:

- **Presented in CanSecWest 2015:**
  - c7zero.info: A New Class of Vulnerability in SMI Handlers of BIOS/UEFI Firmware

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

- `mode`: SMI fuzzing mode
  - `config` = use SMI configuration file <config_file>
  - `fuzz` = fuzz all SMI handlers with code in the range <smic_start:smic_end>
  - `fuzzmore` = fuzz mode + pass 2nd-order pointers within buffer to SMI handlers
  - `scan` = fuzz mode + time measurement to identify SMIs that trigger long-running code paths
- `size`: size of the memory buffer (in Hex)
- `address`: physical address of memory buffer to pass in GP regs to SMI handlers (in Hex)
  - `smram` = option passes address of SMRAM base (system may hang in this mode!)

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

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

Where:

- `[]`: optional line

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

Examples:

```
>>> chipsec_main.py -m tools.smm.smm_ptr
>>> chipsec_main.py -m tools.smm.smm_ptr -a fuzzmore,0x0:0xFF -l smm.log
>>> chipsec_main.py -m tools.smm.smm_ptr -a scan,0x0:0xff
```

Warning

- This is a potentially destructive test

chipsec.modules.tools.uefi package

chipsec.modules.tools.uefi.reputation module

chipsec.modules.tools.uefi.s3script_modify module

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

Usage:

Replacing existing opcode:

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

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

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

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

Adding new opcode:

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

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

Examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

chipsec.modules.tools.uefi.scan_blocked module

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

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

Usage:

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

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

Examples:

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

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

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

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

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

chipsec.modules.tools.uefi.scan_image module

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

Usage:

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

- `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
- `norm` **Optional,** `generate` **with** `fw_image` **only. Adds** `sha256_norm` to each PE32 and TE entry: the SHA-256 of the module with its placement undone (relocations reversed; ImageBase, TimeDateStamp, CheckSum and section relocation pointers zeroed), prefixed with the normalization used (`uefi-pe-rebase0.v1:sha256:...` or `uefi-te-rebase0.v1:sha256:...`). Omitted where the module cannot be normalized exactly. `check` ignores the field

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 generate,efilist.json,uefi.rom,norm
```

Same, and also records each module's `sha256_norm`, which stays the same wherever the module is placed in flash

```
>>> 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

chipsec.modules.tools.uefi.uefivar_fuzz module

The module is fuzzing UEFI Variable interface.

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

Usage:

```
chipsec_main -m tools.uefi.uefivar_fuzz [-a <options>]
```

Options:

```
[-a <test>,<iterations>,<seed>,<test_case>]
```

- `test` : UEFI variable interface to fuzz (all, name, guid, attrib, data, size)
- `iterations` : Number of tests to perform (default = 1000)
- `seed` : RNG seed to use
- `test_case` : Test case # to skip to (combined with seed, can be used to skip to failing test)

All module arguments are optional

Examples::

```
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz -a all,100000
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz -a data,1000,123456789
>>> chipsec_main.py -m tools.uefi.uefivar_fuzz -a name,1,123456789,94
```

Note

- This module returns a WARNING by default to indicate that a manual review is needed.
- Writes may generate 'ERROR's, this can be expected behavior if the environment rejects them.

Warning

- This module modifies contents of non-volatile SPI flash memory (UEFI Variable NVRAM).
- This may render system UNBOOTABLE if firmware doesn't properly handle variable update/delete operations.

Important

- Evaluate the platform for expected behavior to determine PASS/FAIL.

- Behavior can include platform stability and retaining protections.

chipsec.modules.tools.vmm package

chipsec.modules.tools.vmm.hv package

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

Hyper-V specific defines

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

Hyper-V specific hypercall functionality

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

Hyper-V hypercall fuzzer

Usage:

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

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

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

chipsec.modules.tools.vmm.hv.synth_dev module

chipsec.modules.tools.vmm.hv.synth_kbd module

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

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

chipsec.modules.tools.vmm.vbox package

chipsec.modules.tools.vmm.vbox.vbox_crash_apicbase module

Oracle VirtualBox CVE-2015-0377 check

Reference:

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

Usage:

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

Examples:

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

Registers used:

- IA32_APIC_BASE

> **Warning**
>
> - Module can cause VMM/Host OS to crash; if so, this is a FAILURE

chipsec.modules.tools.vmm.xen package

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

Xen specific defines

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

Xen specific hypercall functionality

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

Xen hypercall fuzzer

Usage:

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

- `mode` : fuzzing mode
  - `help` : Prints this help
  - `info` : Hypervisor information
  - `fuzzing` : Fuzzing specified hypercall
  - `fuzzing-all` : Fuzzing all hypercalls
  - `fuzzing-all-randomly` : Fuzzing random hypercalls
- `<vector>` : Code or name of a hypercall to be fuzzed (use info)
- `<iterations>` : Number of fuzzing iterations

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing,10 -l log.txt
>>> chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing-all,50 -l log.txt
>>> chipsec_main.py -i -m tools.vmm.xen.hypercallfuzz -a fuzzing-all-randomly,10,0x10000000 -l log.txt
```

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.xen.xsa188 module

chipsec.modules.tools.vmm.cpuid_fuzz module

Simple CPUID VMM emulation fuzzer

Usage:

```
chipsec_main.py -i -m tools.vmm.cpuid_fuzz [-a random]
```

- `random` : Fuzz in random order (default is sequential)

Where:

- `[]`: optional line

Examples:

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

Additional options set within the module:

- `_NO_EAX_TO_FUZZ` : No of EAX values to fuzz within each step
- `_EAX_FUZZ_STEP` : Step to fuzz range of EAX values
- `_NO_ITERATIONS_TO_FUZZ` : Number of iterations if *random* chosen
- `_FUZZ_ECX_RANDOM` : Fuzz ECX with random values?
- `_MAX_ECX` : Max ECX value
- `_EXCLUDE_CPUID` : Exclude the following EAX values from fuzzing
- `_FLUSH_LOG_EACH_ITER` : Flush log file after each iteration
- `_LOG_OUT_RESULTS` : Log output results

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.ept_finder module

Extended Page Table (EPT) Finder

Usage:

```
chipsec_main -m tools.vmm.ept_finder [-a dump,<file_name>|file,<file_name>,<
revision_id>]
```

- `dump` : Dump contents
- `file` : Load contents from file
- `<file_name>` : File name to read from or dump to
- `<revision_id>` : Revision ID (hex)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.ept_finder
>>> chipsec_main.py -i -m tools.vmm.ept_finder -a dump,my_file.bin
>>> chipsec_main.py -i -m tools.vmm.ept_finder -a file,my_file.bin,0x0
```

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.hypercallfuzz module

chipsec.modules.tools.vmm.iofuzz module

Simple port I/O VMM emulation fuzzer

Usage:

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

- `<mode>` **:** *SMI handlers testing mode*
  - `exhaustive` : Fuzz all I/O ports exhaustively (default)
  - `random` : Fuzz randomly chosen I/O ports
- `<count>` : Number of times to write to each port (default = 1000)
- `<iterations>` : Number of I/O ports to fuzz (default = 1000000 in random mode)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.iofuzz
>>> chipsec_main.py -i -m tools.vmm.iofuzz -a random,9000,4000000
```

Additional options set within the module:

- `MAX_PORTS` : Maximum ports
- `MAX_PORT_VALUE` : Maximum port value to use
- `DEFAULT_PORT_WRITE_COUNT` : Default port write count if not specified with switches
- `DEFAULT_RANDOM_ITERATIONS` : Default port write iterations if not specified with switches
- `_FLUSH_LOG_EACH_ITER` : Flush log after each iteration
- `_FUZZ_SPECIAL_VALUES` : Specify to use 1-2-4 byte values
- `_EXCLUDE_PORTS` : Ports to exclude (list)

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.msr_fuzz module

Simple CPU Module Specific Register (MSR) VMM emulation fuzzer

Usage:

```
chipsec_main -m tools.vmm.msr_fuzz [-a random]
```

- `-a` : random = use random values (default = sequential numbering)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.msr_fuzz
>>> chipsec_main.py -i -m tools.vmm.msr_fuzz -a random
```

Additional options set within the module:

- `_NO_ITERATIONS_TO_FUZZ` : Number of iterations to fuzz randomly
- `_READ_MSR` : Specify to read MSR when fuzzing it
- `_FLUSH_LOG_EACH_MSR` : Flush log file before each MSR
- `_FUZZ_VALUE_0_all1s` : Try all 0 & all 1 values to be written to each MSR
- `_FUZZ_VALUE_5A` : Try 0x5A values to be written to each MSR
- `_FUZZ_VALUE_RND` : Try random values to be written to each MSR
- `_EXCLUDE_MSR` : MSR values to exclude (list)

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.pcie_fuzz module

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

Usage:

```
chipsec_main -m tools.vmm.pcie_fuzz [-a <bus> <dev> <fun>]
```

- `<bus>` : Bus # to fuzz (in hex)
- `<dev>` : Device # to fuzz (in hex)
- `<fun>` : Function # to fuzz (in hex)

Where:

- `[]`: optional line

Examples:

```
>>> chipsec_main.py -i -m tools.vmm.pcie_fuzz
>>> chipsec_main.py -i -m tools.vmm.pcie_fuzz -l log.txt
>>> chipsec_main.py -i -m tools.vmm.pcie_fuzz -a 0 1f 0
```

Additional options set within the module:

- `IO_FUZZ` : Set to fuzz IO BARs
- `CALC_BAR_SIZE` : Set to calculate BAR sizes
- `TIMEOUT` : Timeout between memory writes (seconds)
- `ACTIVE_RANGE` : Set to fuzz MMIO BAR in Active range
- `BIT_FLIP` : Set to fuzz using bit flips
- `_EXCLUDE_BAR` : BARs to exclude (list)

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.pcie_overlap_fuzz module

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

Usage:

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

Examples:

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

Additional options set within the module:

- `OVERLAP_MODE` : Set overlap direction
- `FUZZ_OVERLAP` : Set for fuzz overlaps
- `FUZZ_RANDOM` : Set to fuzz in random mode
- `_EXCLUDE_MMIO_BAR1` : List 1 of MMIO bars to exclude
- `_EXCLUDE_MMIO_BAR2` : List 2 of MMIO bars to exclude

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.vmm.venom module

QEMU VENOM vulnerability DoS PoC test

Reference:

- Module is based on PoC by Marcus Meissner
- VENOM: QEMU vulnerability (CVE-2015-3456)

Usage:

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

Examples:

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

Additional options set within the module:

- `ITER_COUNT` : Iteration count
- `FDC_PORT_DATA_FIFO` : FDC DATA FIFO port
- `FDC_CMD_WRVAL` : FDC Command write value

- `FD_CMD` : FD Command

Note

- Returns a Warning by default
- System may be in an unknown state, further evaluation may be needed

Important

- This module is designed to run in a VM environment
- Behavior on physical HW is undefined

chipsec.modules.tools.wsmt module

The Windows SMM Security Mitigation Table (WSMT) is an ACPI table defined by Microsoft that allows system firmware to confirm to the operating system that certain security best practices have been implemented in System Management Mode (SMM) software.

Reference:

- See <https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-uefi-wsmt> for more details.

Usage:

```
chipsec_main -m common.wsmt
```

Examples:

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

Note

- Analysis is only necessary if Windows is the primary OS

Contribution and Style Guides

Python Version

All Python code, and PEP support, must to be limited to the features supported by **Python 3.6.8**.

This is earliest version of Python utilized by CHIPSEC, the version of the EFI Shell Python.

Python Coding Style Guide

CHIPSEC mostly follows the PEP8 with some exceptions. This attempts to highlight those as well as clarify others.

Consistency and readability are the goal but not at the expense of readability or functionality.

If in doubt, follow the existing code style and formatting.

1. PEP8

   PEP 8 is a set of recommended code style guidelines (conventions) for Python.

   PEP 8

2. Linting tools

   CHIPSEC includes a Flake8 configuration file

   CHIPSEC flake8 config

3. Zen of Python

   Great philosophy around Python building principles.

   PEP 20

4. Headers and Comments

   Use single line comments, a single hash/number sign/octothorpe '#'.

   Should contain a space immediately after the '#'.

```
# Good header comment
```

5. Single vs Double Quotes

   Single quotes are encouraged but can vary with use case.

   Avoid using backslashes '\' in strings.

```
'This is a preferred "string".'
"Also an acceptable 'string'."

"Avoid making this \"string\"."
```

6. Imports

   Import order:

   1. Python standard library
   2. Third-party imports
   3. CHIPSEC and local application imports

   Avoid using `import *` or `from import *`. This could pollute the namespace.

```
# Good
import sys
from chipsec.module_common import BaseModule
from chipsec.library.returncode import ModuleResult

# Bad - using '*' and importing sys after local imports
import *
from chipsec.module_common import *
import sys
```

   Avoid using `from __future__ imports`. These may not work on older or all interpreter versions required in all supported environments.

7. Line Length

   Maximum line length should be 120 characters.

   If at or near this limit, consider rewriting (eg simplifying) the line instead of breaking it into multiple lines.

   Long lines can be an indication that too many things are happening at once and/or difficult to read.

8. Class Names

   `HAL` and `utilcmd` **classes** should use **UpperCamelCase** (**PascalCase**) Words and acronyms are capitalized with no spaces or underscores.

   Test **module** class names MUST match the module name which are typically **snake_case**

9. Constants

   Constants should use **CAPITALIZATION_WITH_UNDERSCORES**

10 Variable Names

- Variable names should use **snake_case**

  Lower-case text with underscores between words.

11 Local Variable Names (private)

- Prefixed with an underscore, **_private_variable**

  Not a hard rule but will help minimize any variable name collisions with upstream namespace.

12 Dunder (double underscore)

- Avoid using `__dunders__` when naming variables. Should be used for functions that overwrite or add to classes and only as needed.

  Dunders utilize double (two) underscore characters before and after the name.

13 Code Indents

- CHIPSEC uses 4 space ‘tabbed’ indents.

  No mixing spaces and tabs.

  - 1 indent = 4 spaces
  - No tabs

  Recommend updating any IDE used to use 4 space indents by default to help avoid mixing tabs with spaces in the code.

14 Operator Precedence, Comparisons, and Parentheses

- If in doubt, wrap evaluated operators into logical sections if using multiple operators or improves readability.

  While not needed in most cases, it can improve readability and limit the possibility of ‘left-to-right chaining’ issues.

```
# Preferred
if (test1 == True) or (test2 in data_list):
    return True

# Avoid.  Legal but behavior may not be immediately evident.
if True is False == False:
    return False
```

15 Whitespace

- No whitespace inside parentheses, brackets, or braces.

No whitespace before a comma, colon, or semicolons.

Use whitespace after a comma, colon, or semicolon.

Use whitespace around operators: +, -, *, **, /, //, %, =, ==, <, >, <=, >=, <>, !=, is, in, is not, not in, <<, >>, &, |, ^

No trailing whitespace.

16 Non-ASCII Characters

- If including any non-ASCII characters anywhere in a python file, include the python encoding comment at the beginning of the file.

```
# -*- coding: utf-8 -*-
```

No non-ASCII class, function, or variable names.

17 Docstrings

- Use three double-quotes for all docstrings.

```
"""String description docstring."""
```

18 Semicolons

- Do not use semicolons.

19 Try Except

- Avoid using nested try-except.

  The routine you are calling, may already be using one.

20 Avoid for-else and while-else loops

- The loop behavior for these can be counterintuitive.

  If they have to be used, make sure to properly document the expected behavior / work-flow.

f-Strings

PEP versions supported by CHIPSEC

PEP / bpo	Title	Summary	Python Version	Supported
PEP 498	Literal String Interpolation	Adds a new string formatting mechanism: Literal String Interpolation, f-strings	3.6	Yes
bpo 36817	Add = to f-strings for easier debugging	f-strings support = for self-documenting expressions	3.8	No
PEP 701	Syntactic formalization of f-strings	Lift some restrictions from PEP 498 and formalize grammar for f-strings	3.12	No

Type Hints

For more information on Python Type Hints:

PEP 483 - The Theory of Type Hints

This table lists which Type Hint PEPs are in scope for CHIPSEC.

PEP versions supported by CHIPSEC

PEP	Title	Summary	Python Version	Supported
PEP 3107	Function Annotations	Syntax for adding arbitrary metadata annotations to Python functions	3.0	Yes
PEP 362	Function Signature Object	Contains all necessary information about a function and its parameters	3.3	Yes
PEP 484	Type Hints	Standard syntax for type annotations	3.5	Yes
PEP 526	Syntax for Variable Annotations	Adds syntax for annotating the types of variables	3.6	Yes
PEP 544	Protocols: Structural subtyping (static duck typing)	Specify type metadata for static type checkers and other third-party tools	3.8	No
PEP 585	Type Hinting Generics In Standard Collections	Enable support for the generics syntax in all standard collections currently available in the typing module	3.9	No
PEP 586	Literal Types	Literal types indicate that some expression has literally a specific value(s).	3.8	No
PEP 589	TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys	Support dictionary object with a specific set of string keys, each with a value of a specific type	3.8	No
PEP 593	Flexible function and variable annotations	Adds an Annotated type to the typing module to decorate existing types with context-specific metadata.	3.9	No
PEP 604	Allow writing union types as X | Y	Overload the | operator on types to allow writing Union[X, Y] as X | Y	3.10	No
PEP 612	Parameter Specification Variables	Proposes typing.ParamSpec and typing.Concatenate to support forwarding parameter types of one callable over to another callable	3.10	No

PEP 613	Explicit Type Aliases	Formalizes a way to explicitly declare an assignment as a type alias	3.10	No
PEP 646	Variadic Generics	Introduce TypeVarTuple, enabling parameterisation with an arbitrary number of types	3.11	No
PEP 647	User-Defined Type Guards	Specifies a way for programs to influence conditional type narrowing employed by a type checker based on runtime checks	3.11	No
PEP 655	Marking individual TypedDict items as required or potentially-missing	Two new notations: Required[], which can be used on individual items of a TypedDict to mark them as required, and NotRequired[]	3.11	No
PEP 673	Self Type	Methods that return an instance of their class	3.10	No
PEP 675	Arbitrary Literal String Type	Introduces supertype of literal string types: LiteralString	3.11	No
PEP 681	Data Class Transforms	Provides a way for third-party libraries to indicate that certain decorator functions, classes, and metaclasses provide behaviors similar to dataclasses	3.11	No
PEP 692	Using TypedDict for more precise kwargs typing	A new syntax for specifying kwargs type as a TypedDict without breaking current behavior	3.12	No
PEP 695	Type Parameter Syntax	A syntax for specifying type parameters within a generic class, function, or type alias. And introduces a new statement for declaring type aliases.	3.12	No
PEP 698	Override Decorator for Static Typing	Adds @override decorator to allow type checkers to prevent a class of bugs that occur when a base class changes methods that are inherited by derived classes.	3.12	No

Underscores in Numeric Literals

Underscores in Numeric Literals are supported, even encouraged, but not required. For consistency, follow the grouping examples presented in the PEP abstract.

PEP versions supported by CHIPSEC

PEP	Title	Summary	Python Version	Supported
PEP 515	Underscores in Numeric Literals	Extends Python's syntax so that underscores can be used as visual separators for grouping purposes in numerical literals	3.6	Yes

Walrus Operator (:=)

At this time, Assignment Expressions (Walrus operator) are not supported.

PEP versions supported by CHIPSEC

PEP	Title	Summary	Python Version	Supported
PEP 572	Assignment Expressions	Adds a way to assign to variables within an expression	3.8	No

Deprecate distutils module support

Python 3.12 will deprecate and remove the distutils module. In order for CHIPSEC to support this and furture versions of Python, setuptools should be used instead of distutils.

The setuptools module has been updated to fully replace distutils but requires an up-to-date version.

- Minimum setuptools version: 62.0.0 (requires Python >= 3.8)
- Recommended setuptools version: latest

Note: If you get any *setuptools.command.build* errors, verify that you have (at least) the minimum setuptools version.

PEP versions supported by CHIPSEC

PEP / bpo	Title	Summary	Python Version	Supported
PEP 632	Deprecate distutils module	Mark the distutils module as deprecated (3.10) and then remove it (3.12)	3.12	Yes

Sphinx Version

The versions of Sphinx that can be utilized to generate CHIPSEC's documentation are 6.X.X, 7.X.X and 8.X.X.

Generating Documentation

Use the script in the docs folder to automatically generate CHIPSEC's documentation using Sphinx. It generates PDF plus either HTML or JSON formats.

```
python3 create_manual.py [format]
```

`format` - html or json

```
python3 create_manual.py
```

```
python3 create_manual.py html
```

```
python3 create_manual.py json
```

References

- Sphinx Apidoc
- Sphinx Build
- Autodoc