

CHIPSEC

Platform Security Assessment Framework

5/21/2014

Contents

1	Description	3
2	Installation	4
2.1	Windows Installation	4
2.2	UEFI Shell	6
2.2.1	Building bootable USB thumb drive with UEFI Shell.....	6
2.2.2	Installing CHIPSEC on bootable thumb drive with UEFI shell	6
2.2.3	Extending CHIPSEC functionality for UEFI.....	7
2.3	Linux Installation	8
2.3.1	Creating a Live Linux image with CHIPSEC:	8
3	Usage.....	10
3.1	Using CHIPSEC as a Python Package	10
3.2	Compiling CHIPSEC Executables on Windows.....	10
4	CHIPSEC Components and Structure	12
4.1	Core components.....	12
4.2	HW Abstraction Layer (HAL)	12
4.3	OS/Environment Helpers	13
4.4	Platform Configuration	13
4.5	CHIPSEC utility command-line scripts	13
4.6	CHIPSEC modules (security tests, tools)	13
4.7	Auxiliary components	14
4.8	Executable build scripts	14
5	CHIPSEC Extension Modules and API.....	15
5.1	Existing Modules	15
5.2	Writing Your Own Modules (security modules).....	15
5.3	Using CHIPSEC in a Python Shell	16
6	CHIPSEC Utilities.....	18
6.1	Accessing and Parsing the Contents of SPI Flash	18
6.1.1	Accessing SPI Flash.....	18
6.1.2	Parsing the SPI Flash	19
6.2	Accessing Memory Mapped PCI Configuration Space	19
6.3	Accessing PCI Configuration Space	19

6.4	UEFI Variable Access	20
6.5	Access to I/O Port Space	20
6.6	Access to MSRs	21
6.7	Access to Physical Memory.....	21
6.8	Access Operating System IDT and GDT.....	21

1 Description

CHIPSEC is a framework for analyzing security of PC platforms including hardware, system firmware including BIOS/UEFI and the configuration of platform components. It allows creating security test suite, security assessment tools for various low level components and interfaces as well as forensic capabilities for firmware.

CHIPSEC can run on any of these environments:

1. Windows (client and server)
2. Linux
3. UEFI Shell

2 Installation

CHIPSEC supports Windows, Linux, and UEFI shell. Circumstances surrounding the target platform may change which of these environments is most appropriate. When running CHIPSEC as part of a corporate IT management infrastructure, Windows may be preferred. However, sometimes it may be preferable to assess the platform security without interfering with the normal operating system. In these instances, CHIPSEC may be run from a bootable USB thumb drive - either a Live Linux image or a UEFI shell.

2.1 Windows Installation

Supports the following client versions:

- Windows 8 x86 and AMD64
- Windows 7 x86 and AMD64
- Windows XP (support discontinued)

Supports the following server versions:

- Windows Server 2012 x86 and AMD64
- Windows Server 2008 x86 and AMD64

1. Install Python (<http://www.python.org/download/>)

- Tested on 2.7.x and Python 2.6.x

- E.g. Python 2.7.6 (<http://www.python.org/download/releases/2.7.6/>)

2. Install additional packages for installed Python release (in any order)

- (REQUIRED) pywin32: for Windows API support (<http://sourceforge.net/projects/pywin32/>)

- (OPTIONAL) WConio : if you need colored console output
(<http://newcenturycomputers.net/projects/wconio.html>)

- (OPTIONAL) py2exe : if you need to build chipsec executables (<http://www.py2exe.org/>)

Note: packages have to match Python platform (e.g. AMD64 package on Python AMD64)

3. Turn off kernel driver signature checks

Windows 8 64-bit (with Secure Boot enabled) / Windows Server 2012 64-bit (with Secure Boot enabled):

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

OR

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

Windows 7 64-bit (AMD64) / Windows Server 2008 64-bit (AMD64) / Windows 8 (with Secure Boot disabled) / Windows Server 2012 (with Secure Boot disabled):

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

* Start CMD.EXE as Administrator

* `BcdEdit /set TESTSIGNING ON`

* Reboot

If doesn't work, run these additional commands:

* `BcdEdit /set noIntegrityChecks ON`

* `BcdEdit /set loadoptions DDISABLE_INTEGRITY_CHECKS`

OR

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

4. Notes on loading chipsec kernel driver:

- On Windows 7, launch CMD.EXE as Administrator
- CHIPSEC will attempt to automatically register and start its service (load driver) or call existing if it's already started.

- (OPTIONAL) You can manually register and start the service/driver. Follow below instructions before running CHIPSEC, then run it with "--exists" command-line option. CHIPSEC will not attempt to start the driver but will call already running driver.

* To start the service (in cmd.exe)

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

```
sc start chipsec
```

* Then to stop/delete service:

```
sc stop chipsec
```

```
sc delete chipsec
```

2.2 UEFI Shell

A zipped image of a USB stick that has been prepared according to the below instructions is available in `__install__`/UEFI/duet.img.zip. A USB stick, imaged from this file, can be used after CHIPSEC is copied to it.

2.2.1 Building bootable USB thumb drive with UEFI Shell

If you don't have bootable USB thumb drive with UEFI Shell yet, you need to build it:

1. Download UDK from Tianocore

<http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=UDK2010> (Tested with UDK2010.SR1)

2. Follow instructions in `DuetPkg/ReadMe.txt` to create a bootable USB thumb drive with UEFI Shell (DUET)

2.2.2 Installing CHIPSEC on bootable thumb drive with UEFI shell

1. Extract contents of `__install__`/UEFI/chipsec_uefi.7z to the DUET USB drive

- This will create `/efi/Tools` directory with `Python.efi` and `/efi/StdLib` with subdirectories

2. Copy contents of CHIPSEC (source/tool) to the DUET USB drive

3. Reboot to the USB drive (this will load UEFI shell)

4. Run CHIPSEC in UEFI shell

```
fs0:  
  
cd source/tool  
  
python chipsec_main.py (or python chipsec_util.py)
```

2.2.3 Extending CHIPSEC functionality for UEFI

You don't need to read this section if you don't plan on extending native UEFI functionality for CHIPSEC. Native functions accessing HW resources are built directly into Python UEFI port in built-in edk2 module.

If you want to add more native functionality to Python UEFI port for chipsec, you'll need to re-build Python for UEFI:

1. Check out AppPkg with Python 2.7.2 port for UEFI from SVN

<http://edk2.svn.sourceforge.net/svnroot/edk2/trunk/edk2>

- You'll also need to check out StdLib and StdLibPrivateInternalFiles packages from SVN

- Alternatively download latest EADK (EDK II Application Development Kit) from

http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDKII_EADK

EADK includes AppPkg/StdLib/StdLibPrivateInternalFiles. Unfortunately, EADK Alpha 2 doesn't have Python 2.7.2 port so you'll need to check it out SVN

2. Add functionality to Python port for UEFI

- Python 2.7.2 port for UEFI is in <UDK>\AppPkg\Applications\Python

- All chipsec related functions are in <UDK>\AppPkg\Applications\Python\Efi\edk2module.c "#ifdef CHIPSEC"

Asm functions are in <UDK>\AppPkg\Applications\Python\Efi\cpu.asm

- e.g. <UDK> is C:\UDK2010.SR1

- Add cpu.asm under the Efi section in PythonCore.inf

3. Build <UDK>/AppPkg with Python

- Read instructions in <UDK>\AppPkg\ReadMe.txt and <UDK>\AppPkg\Applications\Python\PythonReadMe.txt
 - Binaries of AppPkg and Python will be in <UDK>\Build\AppPkg\DEBUG_MYTOOLS\X64\
4. Create directories and copy Python files on DUET USB drive
- Do not use Python binaries from python_uefi.7z, copy newly generated
 - Read instructions in <UDK>\AppPkg\Applications\Python\PythonReadMe.txt

2.3 Linux Installation

Tested on:

- Linux 3.2.6 x32 (Mint/Ubuntu)
- Linux 2.6.32 x32 (Ubuntu)
- Fedora 20 LXDE 64bit

2.3.1 Creating a Live Linux image with CHIPSEC:

1. Download things you will need
 - a. Download chipsec
 - b. liveusb-creator: <https://fedorahosted.org/liveusb-creator/>
 - c. desired Linux image (e.g. 64bit Fedora 20 LXDE)
2. Use liveusb-creator to image a USB stick with the desired linux image. Include as much persistent storage as possible.
3. Reboot to USB
4. Update and install necessary packages

```
#> yum install kernel kernel-devel python python-devel gcc
```
5. Copy chipsec to the USB stick

Installing CHIPSEC:

6. Build Linux driver for CHIPSEC

```
cd source/drivers/linux  
  
make
```

7. Load CHIPSEC driver in running system

```
cd source/drivers/linux  
  
(Optional) chmod 755 run.sh  
  
sudo ./run.sh or sudo make install
```

8. Run CHIPSEC

```
cd source/tool  
  
sudo python chipsec_main.py (or sudo python chipsec_util.py)
```

9. Remove CHIPSEC driver after using

```
sudo make uninstall
```

3 Usage

CHIPSEC should be launched as Administrator/root.

- In command shell, run chipsec_main.py

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

USAGE: chipsec_main.py [options]

OPTIONS:

-m --module	specify module to run (example: -m common.bios)
-a --module_args	additional module arguments, format is 'arg0,arg1..'
-v --verbose	verbose mode
-l --log	output to log file

ADVANCED OPTIONS:

-p --platform	explicitly specify platform code. Should be among the supported platforms:
	[SNB IVB JKT BYT IVT HSW]
-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
-e --exists	chipsec service has already been manually installed and started (driver loaded).
-x --xml	specify filename for xml output (JUnit style).
-t --moduletype	run tests of a specific type (tag).
--list_tags	list all the available options for -t,--moduletype
-I --import	specify additional path to load modules from.

Use "--no-driver" command-line option if the module you are executing does not require loading kernel mode driver

Chipsec won't load/unload the driver and won't try to access existing driver

Use "--exists" command-line option if you manually installed and start chipsec driver (see "install_readme" file).

Otherwise chipsec will automatically attempt to create and start its service (load driver)

or open existing service if it's already started

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

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

3.1 Using CHIPSEC as a Python Package

The directory should contain the file `setup.py`. Install CHIPSEC into your system's site-packages directory:

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

3.2 Compiling CHIPSEC Executables on Windows

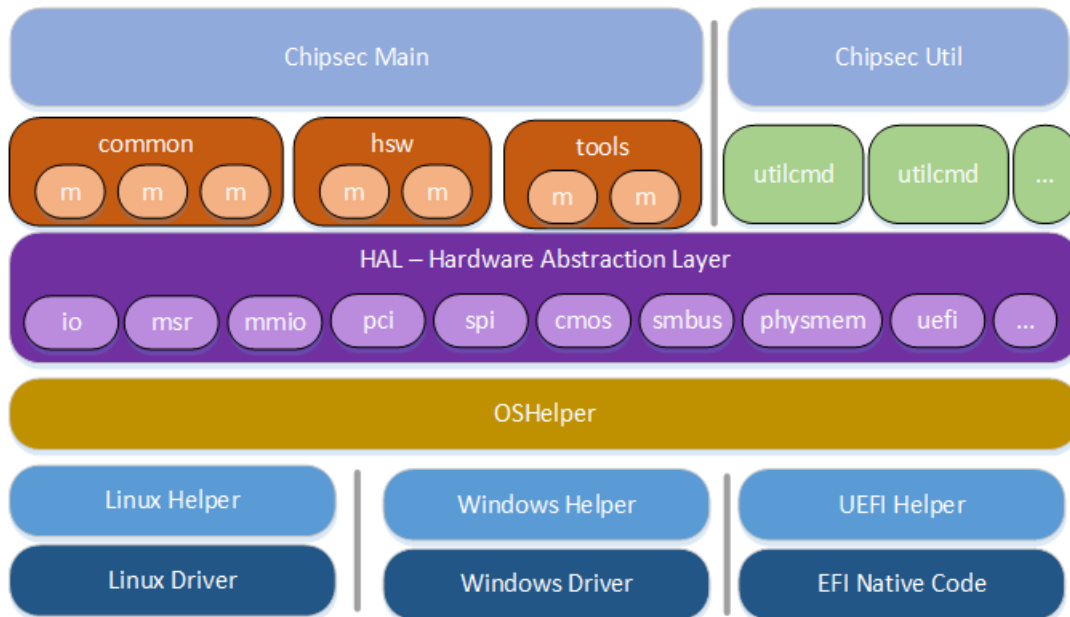
Directories "bin/<platform>" should already contain compiled CHIPSEC binaries:

```
"chipsec_main.exe", "chipsec_util.exe"
- To run all security tests run "chipsec_main.exe" from "bin"
directory:
  # chipsec_main.exe
- To access hardware resources run "chipsec_util.exe" from "bin"
directory:
  # chipsec_util.exe
```

If directory "bin" doesn't exist, then you can compile CHIPSEC executables:

- Install "py2exe" package from <http://www.py2exe.org>
- From root chipsec directory run "build_exe_<platform>.py" as follows:
 # python build_exe_<platform>.py py2exe
- chipsec_main.exe, chipsec_util.exe executables and required libraries will be created in
 "bin/<platform>" directory

4 CHIPSEC Components and Structure



4.1 Core components

<code>chipsec_main.py</code>	- main application logic and
<code>automation functions</code>	
<code>chipsec_util.py</code>	- utility functions (access to
<code>various hardware resources)</code>	
<code>chipsec/chipset.py</code>	- chipset detection
<code>chipsec/logger.py</code>	- logging functions
<code>chipsec/file.py</code>	- reading from/writing to files
<code>chipsec/module_common.py</code>	- common include file for modules
<code>chipsec/helper/oshelper.py</code>	- OS helper: wrapper around platform
<code>specific code that invokes kernel driver</code>	
<code>chipsec/helper/xmlout.py</code>	- support for JUnit compatible XML
<code>output (-x command-line option)</code>	

4.2 HW Abstraction Layer (HAL)

<code>chipsec/hal/</code>	- components responsible for access
<code>to hardware (Hardware Abstraction Layer):</code>	
<code>chipsec/hal/pci.py</code>	- Access to PCIe config space
<code>chipsec/hal/pcidb.py</code>	- Database of PCIe vendor and device
<code>IDs</code>	
<code>chipsec/hal/physmem.py</code>	- Access to physical memory
<code>chipsec/hal/msr.py</code>	- Access to CPU resources (for each
<code>CPU thread): Model Specific Registers (MSR), IDT/GDT</code>	
<code>chipsec/hal/mmio.py</code>	- Access to MMIO (Memory Mapped IO)
<code>BARs and Memory-Mapped PCI Configuration Space (MMCFG)</code>	
<code>chipsec/hal/spi.py</code>	- Access to SPI Flash parts

chipsec/hal/ucode.py	- Microcode update specific
functionality (for each CPU thread)	
chipsec/hal/io.py	- Access to Port I/O Space
chipsec/hal/smbus.py	- Access to SMBus Controller in the
PCH	
chipsec/hal/uefi.py	- Main UEFI component using platform
specific and common UEFI functionality	
chipsec/hal/uefi_common.py	- Common UEFI functionality (EFI
variables, db/dbx decode, etc.)	
chipsec/hal/uefi_platform.py	- Platform specific UEFI
functionality (parsing platform specific EFI NVRAM, capsules, etc.)	
chipsec/hal/interrupts.py	- CPU Interrupts specific functions
(SMI, NMI)	
chipsec/hal/cmos.py	- CMOS memory specific functions
(dump, read/write)	
chipsec/hal/cpuid.py	- CPUID information
chipsec/hal/spi_descriptor.py	- SPI Flash Descriptor binary
parsing functionality	

4.3 OS/Environment Helpers

chipsec/helper/win/	- Windows helper
chipsec/helper/linux/	- Linux helper
chipsec/helper/efi/	- UEFI/EFI shell helper

4.4 Platform Configuration

chipsec/cfg/	- platform specific configuration
includes	
chipsec/cfg/common.py	- common configuration
chipsec/cfg/<platform>.py	- configuration for a specific
<platform>	

4.5 CHIPSEC utility command-line scripts

chipsec/utilcmd/	- command-line extensions for
chipsec_util.py	
chipsec/utilcmd/<command>_cmd.py	- implements "chipsec_util
<command>" command-line extension	

4.6 CHIPSEC modules (security tests, tools)

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_code>/
<platform_code> platform

- modules specific to

chipsec/modules/tools/
CHIPSEC framework (fuzzers, etc.)

- security tools based on

4.7 Auxiliary components

bist.cmd
various basic HW functionality to make sure
setup.py
CHIPSEC as a package

- built-in self test for
it's not broken
- setup script to install

4.8 Executable build scripts

<CHIPSEC_ROOT>/build/build_exe_*.py
Windows executables

- make files to build

5 CHIPSEC Extension Modules and API

In the most basic sense, a platform module is just a python script with a top-level function called `check_all()`. These modules are stored under the chipsec installation directory in a subdirectory "modules". The "modules" directory contains one subdirectory for each chipset that chipsec supports. 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\`.

5.1 Existing Modules

Each published module can be mapped to a publication that details the issue being checked. The mapping is provided below.

Chipsec Module	Publication
Modules/common/bios_ts.py	"BIOS Boot Hijacking and VMware Vulnerabilities Digging" - Sun Bing
Modules/common/bios_kbrd_buffer.py	DEFCON 16: "Bypassing Pre-boot Authentication Passwords by Instrumenting the BIOS Keyboard Buffer" – Jonathan Brossard
Modules/common/bios_wp.py	Black Hat USA 2013 "BIOS Security" – MITRE (Kovah, Butterworth, Kallenberg)
	NoSuchCon 2013 "BIOS Chronomancy: Fixing the Static Root of Trust for Measurement" – MITRE (Kovah, Butterworth, Kallenberg)
	Parsing of SPI descriptor access permissions is implemented in "ich_descriptors_tool" which is part of open source flashrom .
Modules/common/smm.py	CanSecWest 2006 "Security Issues Related to Pentium System Management Mode" – Dufлот
Modules/common/smrr.py	"Attacking SMM Memory via Intel CPU Cache Poisoning" – ITL (Rutkowska, Wojtczuk)
	"Getting into the SMRAM: SMM Reloaded" – Dufлот, Levillain, Morin, Grumelard
Modules/common/spi_lock.py	FLOCKDN is in flashrom and MITRE's Copernicus
Modules/common/secure boot/keys.py	UEFI 2.4 spec
Modules/common/secure boot/variables.py	UEFI 2.4 spec

5.2 Writing Your Own Modules (security modules)

Implement a function called `run()` in your module.

- Use other chipsec components for support
- See 'CHIPSEC Components/API' section

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

- Modules specific to certain chipset should be in `chipsec/modules/<chipset_code>` directory
- Modules common to all supported chipsets should be in `chipsec/modules/common` directory

If a new chipset needs to be added:

- Create directory for the new chipset in `chipsec/modules`
- Create empty `__init__.py` in new directory
- Modify `chipsec/chipset.py` to include detection for the chipset you are adding

5.3 Using CHIPSEC in a Python Shell

The `chipsec.app` component can also be run from a python interactive shell or used in other python scripts. The `chipsec.app` module contains application logic in the form of a set of python functions for this purpose:

`run_module('module_path')`
Immediately calls `module.check_all()` and returns. Does not affect internal loaded modules list.

`load_module('module_path')`
Loads a module into the internal module list for batch processing

`unload_module('module_path')`
Unloads a module from the internal module list

`load_my_modules()`
Loads all modules from "modules\common" and (if the current chipset is recognized) `modules\<chipset_code>` into an internal list for batch processing.

`un_loaded_modules()`
Calls the `check_all()` function from every module in the internal loaded modules list

`clear_loaded_modules()`
Empties the internal loaded module list

```
run_all_checks()
```

Calls `load_my_modules()` followed by `run_loaded_modules()`. This function executes all existing security checks for a given chipset/platform. Calling this function in Python shell is equivalent to executing standalone `chipsec_main.py` or `chipsec_main.exe`.

Example:

```
>>> import chipsec_main
>>> chipsec_main._cs.init(True) # if chipsec driver is not running
>>> chipsec_main.load_module('chipsec/modules/common/bios_wp.py')
>>> chipsec_main.run_loaded_modules()
```

6 CHIPSEC Utilities

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

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

NOTE: All numeric values are in hex.

6.1 Accessing and Parsing the Contents of SPI Flash

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.

6.1.1 Accessing SPI Flash

The spi utility command provides access to read, write, and erase the SPI flash.

```
chipsec_util spi info|dump|read|write|erase [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 erase 0x7FFFF0
```

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 below.

6.1.2 Parsing the SPI Flash

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.

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

<fw_type> should be in [evsa | nvar | uefi | vss | vss_new]

Examples:

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

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

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 `nvar` directory does not appear and the list of `nvar` variables is empty, try again with another type.

6.2 Accessing Memory Mapped PCI Configuration Space

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

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

Examples:

```
chipsec_util mmcfg 0 0 0 0x88 4
```

```
chipsec_util mmcfg 0 0 0 0x88 byte 0x1A
```

```
chipsec_util mmcfg 0 0x1F 0 0xDC 1 0x1
```

```
chipsec_util mmcfg 0 0 0 0x98 dword 0x004E0040
```

6.3 Accessing PCI Configuration Space

The `pci` command can enumerate PCI devices and allow direct access to them by bus/device/function.

```
chipsec_util pci enumerate
```

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

Examples:

```
chipsec_util pci enumerate
chipsec_util pci 0 0 0 0x88 4
chipsec_util pci 0 0 0 0x88 byte 0x1A
chipsec_util pci 0 0x1F 0 0xDC 1 0x1
chipsec_util pci 0 0 0 0x98 dword 0x004E0040
```

6.4 UEFI Variable Access

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

```
chipsec_util uefi var-list
chipsec_util uefi var-read|var-write <name> <GUID> <efi_variable_file>
chipsec_util uefi nvram[-auth] <fw_type> [rom_file]
<fw_type> should be in [ evsa | nvar | uefi | vss | vss_new ]
chipsec_util uefi keys <keyvar_file>
<keyvar_file> should be one of the following EFI variables
[ SecureBoot | SetupMode | CustomMode | PK | KEK | db | dbx ]
```

Examples:

```
chipsec_util uefi var-list
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 nvram vss bios.rom
chipsec_util uefi nvram-auth vss
chipsec_util uefi decode uefi.bin fwtype
chipsec_util uefi keys db.bin
```

6.5 Access to I/O Port Space

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

```
chipsec_util io <io_port> <width> [value]
```

Examples:

```
chipsec_util io 0x61 1
```

```
chipsec_util io 0x430 byte 0x0
```

6.6 Access to MSRs

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

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

Examples:

```
chipsec_util msr 0x3A
```

```
chipsec_util msr 0x8B 0x0 0x0 0
```

6.7 Access to Physical Memory

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

```
chipsec_util mem <phys_addr_hi> <phys_addr_lo> <length> [value]
```

Examples:

```
chipsec_util mem 0x0 0x41E 0x20
```

```
chipsec_util mem 0x0 0xA0000 4 0x9090CCCC
```

```
chipsec_util mem 0x0 0xFED40000 0x4
```

6.8 Access Operating System IDT and GDT

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

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

Examples:

```
chipsec_util idt 0
```

```
chipsec_util gdt
```