CHIPSEC

version 1.2.2

Platform Security Assesment Framework

November 06, 2015

Contents

CHIPSEC

Welcome to the CHIPSEC documentation!

Questions about CHIPSEC can be directed to chipsec@intel.com

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.

Description

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

CHIPSEC can run from Windows, Linux, and UEFI Shell.

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

Windows Installation

Supports the following client versions:

- Windows 8/8.1 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

Steps for installation:

1. Install Python

> ***Note***
>
> Tested on 2.7.x and Python 2.6.x (E.g. Python 2.7.6)

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

   - (REQUIRED) pywin32: for Windows API support
   - (OPTIONAL) WConio: if you need colored console output
   - (OPTIONAL) py2exe: if you need to build chipsec executables

> ***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)

     1. Start CMD.EXE as Adminstrator
     2. BcdEdit /set TESTSIGNING ON
     3. Reboot

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

     1. BcdEdit /set noIntegrityChecks ON
     2. BcdEdit /set loadoptions DISABLE_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)

1. sc create chipsec binpath=<PATH_TO_CHIPSEC_SYS> type= kernel DisplayName="Chipsec driver"
2. sc start chipsec

Then to stop/delete service:

1. sc stop chipsec
2. sc delete chipsec

Linux Installation

Tested on:

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

Installing CHIPSEC

1. You can use CHIPSEC on a desired Linux distribution or create a live Linux image on a USB flash drive and boot to it
   - For example, you can use liveusb-creator to create live Fedora image on a USB drive
2. Update and install necessary packages

```
#> yum install kernel kernel-devel-$(uname -r) python python-devel gcc nasm
```

or

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

Note

You can install the kernel headers for the currently installed version. That is why the above commands install `kernel-devel-$(uname -r)` or `linux-headers-$(uname -r)`.

3. Clone CHIPSEC Git repository
   - `git clone https://github.com/chipsec/chipsec`
4. Build Linux driver for CHIPSEC
   - `cd source/drivers/linux`

  - `make`

5. Load CHIPSEC driver in running system

    - (Optional) `chmod 755 run.sh`
    - `sudo ./run.sh` or `sudo make install`

   or

    - `cd source/scripts`
    - `chmod 755 compile_linux_driver.sh`
    - `sudo ./compile_linux_driver.sh`

6. Run CHIPSEC

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

7. Remove CHIPSEC driver after using

   `sudo make uninstall`

UEFI Shell Installation

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:

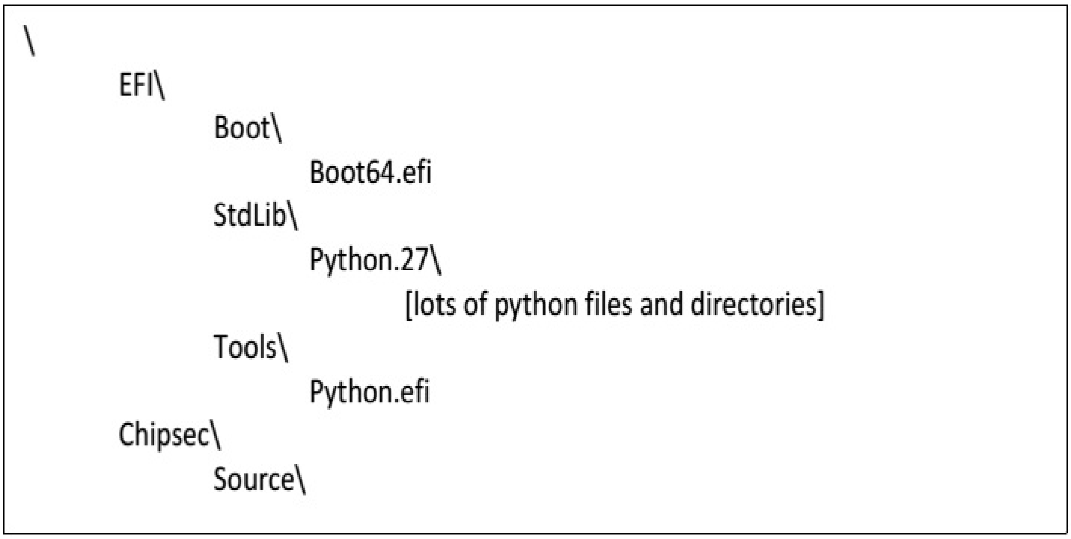
1. Download UDK from Tianocore (Tested with UDK2010.SR1)
2. Follow instructions in DuetPkg/ReadMe.txt to create a bootable USB thumb drive with UEFI Shell (DUET)

Installing CHIPSEC on bootable thumb drive with UEFI shell

1. Extract contents of __install__/UEFI/chipsec_uefi_x64.zip 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. The contents of your thumb drive should look like follows:



```
\
        EFI\
                Boot\
                        Boot64.efi
                StdLib\
                        Python.27\
                                [lots of python files and directories]
                Tools\
                        Python.efi
        Chipsec\
                Source\
```

Note

The USB drive should already include a UEFI Shell binary in /efi/boot. On 64-bit platforms this should be named `bootx64.efi`.

3. Reboot to the USB drive (this will load UEFI shell)
4. Run CHIPSEC in UEFI shell

    1. `fs0:`
    2. `cd source/tool`
    3. `python chipsec_main.py` or `python chipsec_util.py`

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

    - You'll also need to check out `StdLib` and `StdLibPrivateInternalFiles` packages from SVN
    - Alternatively download latest EADK (EDK II Application Development Kit). EADK includes `AppPkg/StdLib/StdLibPrivateInternalFiles`. Unfortunately, EADK Alpha 2 doesn't have Python 2.7.2 port so you'll need to check it out SVN.

2. Add functionality to Python port for UEFI

    - Python 2.7.2 port for UEFI is in `<UDK>\AppPkg\Applications\Python`
    - All chipsec related functions are in `<UDK>\AppPkg\Applications\Python\Efi\edk2module.c (#ifdef CHIPSEC)`
    - Asm functions are in `<UDK>\AppPkg\Applications\Python\Efi\cpu.asm`
    - e.g. <UDK> is C:UDK2010.SR1
    - Add cpu.asm under the Efi section in PythonCore.inf

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

Using CHIPSEC

CHIPSEC should be launched as Administrator/root.

- In command shell, run

```
# python chipsec_main.py
```

- For help, run

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

- **Command Line Usage**

```
# chipsec_main.py [options]
```

Options

-m --module	specify module to run (example: -m common.bios_wp)
-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 | QRK | BDW | IVT | AVN | HSW | SKL | HSX]
-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 --include	specify additional path to load modules from
--failfast	fail on any exception and exit (don't mask exceptions)

--no_time	don't log timestamps

Exit Code

CHIPSEC returns an integer exit code:

- Exit code is 0: all modules ran successfully and passed
- Exit code is not 0: each bit means the following:
    - Bit 0: SKIPPED at least one module was skipped
    - Bit 1: WARNING at least one module had a warning
    - Bit 2: DEPRECATED at least one module uses deprecated API
    - Bit 3: FAIL at least one module failed
    - Bit 4: ERROR at least one module wasn't able to run
    - Bit 5: EXCEPTION at least one module thrown an unexpected exceptions

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

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

- `# 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 help
```

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
```

then to run use this command:

```
# python -m chipsec_main
```

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 and 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 "modulescommon" 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()
```

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 the build 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

Using CHIPSEC with DAL/ITP2

- Set the "USEDAL" environment variable, then run from command line as usual. Operations will go through DAL to target platform

Note

All actions which do no specify a thread explicitly use Core 0 Thread 0, this cannot be reconfigured currently

Warning

Using chipsec over DAL at a command prompt is excruciatingly slow, as the DAL stack must be initialized and torn down for every command. Importing chipsec in a DAL CLI session will give much better performance

- Alternatively, launch the DAL CLI and enter `import chipsec_util` or `import chipsec_main` to load desired chipsec functions

Writing Your Own Modules (security modules)

See `chipsec/modules/module_template.py` for an example. Your module class should subclass BaseModule and implement at least the methods named `is_supported` and `run`. When chipsec_main runs, it will first run `is_supported` and if that returns true, then it will call `run`.

As of CHIPSEC version 1.2.0, CHIPSEC implements an abstract name for platform *controls*. Module authors are encouraged to create controls in the XML configuration files for important platform configuration information and then use `get_control` and `set_control` within modules. This abstraction allows modules to test for the abstract control without knowning which register provides it. (This is especially important for test reuse across platform generations.)

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

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

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

Get the current status of the control:

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

React based on the status of the control:

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

Return:

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

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

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

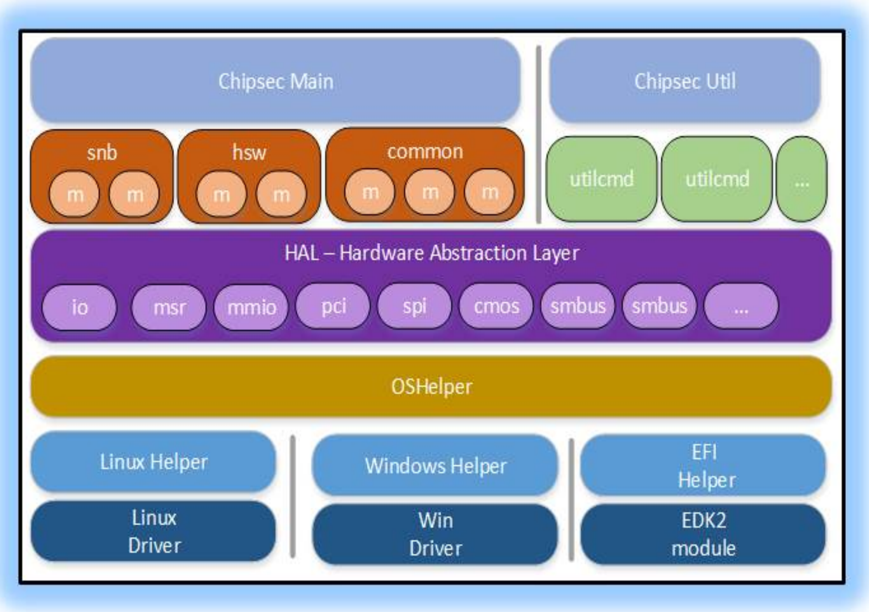
Copy your module into the chipsec/modules/ directory structure

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

If a new platform needs to be added:

- Create directory for the new platform in chipsec/modules
- Create empty __init__.py in the new directory
- Modify chipsec/chipset.py to include the Device ID for the platform you are adding
- Review the platform datasheet and include appropriate information in an XML configuration file for the platform. Place this file in chipsec/cfg. Registers that are correctly defined in `common.xml` will be inherited and do not need to be added. Use `common.xml` as an example. It is based on the 4th Generation Intel Core platform (Haswell).

CHIPSEC Components and Structure



Core components

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

Security modules (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>`	modules specific to <platform>
`chipsec/modules/tools/`	security tools based on CHIPSEC framework (fuzzers, etc.)

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

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

Each published module can be mapped to a publication that details the issue being checked (consult the documentation for an individual module for more information).

chipsec.modules.common.secureboot.variables module

UEFI 2.4 spec Section 28

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

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.secureboot.variables
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.secureboot.variables
[*] running loaded modules ..

[*] running module: chipsec.modules.common.secureboot.variables
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/secureboot/variables.py
[x][ =======================================================================
[x][ Module: Attributes of Secure Boot EFI Variables
[x][ =======================================================================
[*] Checking protections of UEFI variable 8be4df61-93ca-11d2-aa0d-00e098032b8c:SecureBoot
[*] Checking protections of UEFI variable 8be4df61-93ca-11d2-aa0d-00e098032b8c:SetupMode
[*] Checking protections of UEFI variable 8be4df61-93ca-11d2-aa0d-00e098032b8c:PK
[+] Variable 8be4df61-93ca-11d2-aa0d-00e098032b8c:PK is authenticated (TIME_BASED_AUTHENTICATED_WRITE_ACCESS)
[*] Checking protections of UEFI variable 8be4df61-93ca-11d2-aa0d-00e098032b8c:KEK
[+] Variable 8be4df61-93ca-11d2-aa0d-00e098032b8c:KEK is authenticated (TIME_BASED_AUTHENTICATED_WRITE_ACCESS)
[*] Checking protections of UEFI variable d719b2cb-3d3a-4596-a3bc-dad00e67656f:db
[+] Variable d719b2cb-3d3a-4596-a3bc-dad00e67656f:db is authenticated (TIME_BASED_AUTHENTICATED_WRITE_ACCESS)
[*] Checking protections of UEFI variable d719b2cb-3d3a-4596-a3bc-dad00e67656f:dbx
[+] Variable d719b2cb-3d3a-4596-a3bc-dad00e67656f:dbx is authenticated (TIME_BASED_AUTHENTICATED_WRITE_ACCESS)

[*] Secure Boot appears to be disabled
[+] PASSED: All Secure Boot UEFI variables are protected

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.secureboot.variables
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.uefi.access_uefispec module

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

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.uefi.access_uefispec
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.uefi.access_uefispec
[*] running loaded modules ..

[*] running module: chipsec.modules.common.uefi.access_uefispec
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/uefi/access_uefispec.py
[x][ =======================================================================
[x][ Module: Access Control of EFI Variables
[x][ =======================================================================
[*] Testing UEFI variables ..
[*] Variable PlatformLangCodes (BS+RT)
[*] Variable PNP0F03_0_VV (BS+RT)
[*] Variable dbx (NV+BS+RT+TBAWS)
[*] Variable PchInitPei (NV+BS+RT)
[*] Variable PasswordInfo (NV+BS+RT)
[*] Variable AfterReadyToBoot (BS+RT)
[*] Variable SLP20Magic (NV+BS+RT)
[*] Variable ConOut (NV+BS+RT)
[*] Variable MemorySize (NV+BS+RT)
[*] Variable StdDefaults (NV+BS+RT)
[*] Variable FlashStatus (NV+BS+RT)
[*] Variable TcgInternalSyncFlag (NV+BS+RT)
[*] Variable AMITSESetup (NV+BS+RT)
[*] Variable BootOrder (NV+BS+RT)
[*] Variable LastBoot (NV+BS+RT)
[*] Variable PNP0530_0_VV (BS+RT)
[*] Variable Setup (NV+BS+RT)
[*] Variable db (NV+BS+RT+TBAWS)
[*] Variable NetworkStackVar (NV+BS+RT)
[*] Variable PNP0530_0_NV (NV+BS+RT)
[*] Variable UsbMassDevNum (BS+RT)
[*] Variable PNP0F03_0_NV (NV+BS+RT)
[*] Variable UUID (NV+BS+RT)
[*] Variable MaintenanceSetup (NV+BS+RT)
[*] Variable PreviousMemoryTypeInformation (NV+BS+RT)
[*] Variable CurrentSettingValues (NV+BS+RT)
[*] Variable ConOutDev (BS+RT)
[*] Variable CpuIdCkSum (NV+BS+RT)
[*] Variable CpuRatioLimit (NV+BS+RT)
[*] Variable MrcS3Resume (NV+BS+RT)
[*] Variable POSTCounter (NV+BS+RT)
[*] Variable UsbMassDevValid (BS+RT)
[*] Variable FlashInfoStructure (NV+BS+RT)
[*] Variable NVRAM Area (NV+BS+RT)
[*] Variable PK (NV+BS+RT+TBAWS)
[*] Variable FSCConfigVar (NV+BS+RT)
[*] Variable PegGen3PresetSearchData (NV+BS+RT)
[*] Variable BIOSVer (NV+BS+RT)
[*] Variable TpmDeviceSelectionUpdate (NV+BS+RT)
[*] Variable PKDefault (NV+BS+RT+TBAWS)
[!]   Extra attributes:NV+TBAWS
[*] Variable TrEEPhysicalPresence (NV+BS+RT)
[*] Variable NBPlatformData (BS+RT)
[*] Variable SetupPlatformData (BS+RT)
[*] Variable S3SS (BS+RT)
[*] Variable HiiDB (NV+BS+RT)
[*] Variable BootOptionSupport (BS+RT)
[*] Variable IcbdOcSetupData (NV+BS+RT)
[*] Variable MonotonicCounter (NV+BS+RT)
[*] Variable ConInDev (BS+RT)
[*] Variable Boot0001 (NV+BS+RT)
[*] Variable DmiData (NV+BS+RT)
[*] Variable ErrOut (NV+BS+RT)
[*] Variable PNP0303_0_NV (NV+BS+RT)
[*] Variable Events (NV+BS+RT)
[*] Variable MfgMode (NV+BS+RT)
[*] Variable HiiWhiteList (NV+BS+RT)
[*] Variable UsbSupport (NV+BS+RT)
[*] Variable CustomizationFlags (NV+BS+RT)
[*] Variable PchS3Peim (BS+RT)
[*] Variable Lang (NV+BS+RT)
[*] Variable PchInit (NV+BS+RT)
[*] Variable DriverHealthCount (BS+RT)
[*] Variable OsIndicationsSupported (BS+RT)
[*] Variable FPDT_Variable (BS+RT)
[*] Variable Guid1394 (NV+BS+RT)
[*] Variable BootCurrent (BS+RT)
[*] Variable Timeout (NV+BS+RT)
[*] Variable CPUS3APICID (NV+BS+RT)
[*] Variable SignatureSupport (BS+RT)
[*] Variable KEK (NV+BS+RT+TBAWS)
[*] Variable FSCDefConfigVar (NV+BS+RT)
[*] Variable MemoryTypeInformation (NV+BS+RT)
[*] Variable CoolingAssistSetup (NV+BS+RT)
[*] Variable SetupMode (BS+RT)
[*] Variable Boot0000 (NV+BS+RT)
[*] Variable ErrOutDev (BS+RT)
[*] Variable Boot0002 (NV+BS+RT)
[*] Variable Boot0003 (NV+BS+RT)
[*] Variable Boot0005 (NV+BS+RT)
[*] Variable Boot0006 (NV+BS+RT)
[*] Variable Boot0007 (NV+BS+RT)
[*] Variable Boot0008 (NV+BS+RT)
[*] Variable MemoryOverwriteRequestControl (NV+BS+RT)
[*] Variable FSCFanLabelVar (NV+BS+RT)
[*] Variable SecureBoot (BS+RT)
[*] Variable FirmwareId (NV+BS+RT)
[*] Variable EfiTime (NV+BS+RT)
[*] Variable ITKCompatibility (NV+BS+RT)
[*] Variable DriverHlthEnable (BS+RT)
[*] Variable GraphicalAssetInfo (NV+BS+RT)
[*] Variable Boot000B (NV+BS+RT)
[*] Variable UserOcSetup (NV+BS+RT)
[*] Variable ConIn (NV+BS+RT)
[*] Variable OcDefault (NV+BS+RT)
[*] Variable SecurityTokens (NV+BS+RT)
[*] Variable FastBootOption (NV+BS+RT)
[*] Variable PNP0303_0_VV (BS+RT)
[*] Variable OcCurrent (NV+BS+RT)
[*] Variable MemCeil. (NV+BS+RT)
[*] Variable PttInfoVariable (NV+BS+RT)
[*] Variable DimmSPDdata (NV+BS+RT)
[*] Variable SetupDefault (NV+BS+RT)
[*] Variable Boot000A (NV+BS+RT)
[*] Variable AcpiGlobalVariable (NV+BS+RT)
[*] Variable LangCodes (BS+RT)
[*] Variable MEFWVersion (NV+BS+RT)
[*] Variable SbAslBufferPtrVar (NV+BS+RT)
[*] Variable PlatformLang (NV+BS+RT)
[*] Variable RecoveryFile (NV+BS+RT)

[-] Variables with attributes that differ from UEFI spec:
    PKDefault

[-] FAILED: Some EFI variables were not protected according to spec.

[CHIPSEC] ***************************  SUMMARY  ***************************
```

```
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        0:
[CHIPSEC] Modules failed        1:
[-] FAILED: chipsec.modules.common.uefi.access_uefispec
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.uefi.s3bootscript module

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

References:

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

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

Attacks on UEFI Security by Rafal Wojtczuk and Corey Kallenberg.

Attacking UEFI Boot Script by Rafal Wojtczuk and Corey Kallenberg.

Exploiting UEFI boot script table vulnerability by Dmytro Oleksiuk.

Usage:

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

Examples:

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.uefi.s3bootscript
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.uefi.s3bootscript
[*] running loaded modules ..

[*] running module: chipsec.modules.common.uefi.s3bootscript
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/uefi/s3bootscript.py
[x][ =======================================================================
[x][ Module: S3 Resume Boot-Script Protections
[x][ =======================================================================
[*] SMRAM: Base = 0x00000000DC000000, Limit = 0x00000000DCFFFFFF, Size = 0x01000000
[!] Found 1 S3 boot-script(s) in EFI variables
[*] Checking S3 boot-script at 0x00000000DBAB2000
[-] S3 boot-script is not in SMRAM
[*] Reading S3 boot-script from memory..
[*] Decoding S3 boot-script opcodes..
[*] Checking entry-points of Dispatch opcodes..
[-] Dispatch opcode (off 0x2037) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2097) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2238) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2250) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x29D8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2CF8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2D10) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2D28) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2D40) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2D58) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2D70) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2D88) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2DA0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2DB8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2DD0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2DE8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E00) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E18) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E30) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E48) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E60) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E78) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2E90) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2EA8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2EC0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2ED8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2EF0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F08) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F20) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F38) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F50) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F68) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F80) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2F98) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2FB0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2FC8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2FE0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x2FF8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3010) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3028) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3040) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3058) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3070) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3088) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x30A0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x30B8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x30D0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x30E8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3100) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3118) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3130) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3148) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3160) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3178) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3190) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x31A8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x31C0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x31D8) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x31F0) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3208) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3220) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3238) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3250) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3268) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3280) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x34AA) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x34C2) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x34DA) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x34F2) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x350A) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3522) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x353A) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3552) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x356A) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3582) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x359A) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x399B) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x39B3) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x39CB) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x39E3) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x39FB) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3A13) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3A2B) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x3A43) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x46B2) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4B51) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4B69) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4B81) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4B99) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4BB1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4BC9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4BE1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x4BF9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7169) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7181) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7199) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x71B1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x71C9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x71E1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x71F9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7211) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7729) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7809) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x78E9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x79A1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7A59) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7B11) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7BC9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7C59) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7C71) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7C89) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7CA1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7CB9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7CD1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7CE9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D01) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D19) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D31) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D49) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D61) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D79) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7D91) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7DA9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7DC1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7DD9) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7DF1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7E09) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7E21) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7E61) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7EA1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7EE1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7F21) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7F61) with entry-point 0x00000000DBB09260 > UNPROTECTED
[-] Dispatch opcode (off 0x7FA1) with entry-point 0x00000000DBB09260 > UNPROTECTED
[*] Found 134 Dispatch opcodes
[-] Entry-points of Dispatch opcodes in S3 boot-script are not in protected memory

[-] FAILED: S3 Boot-Script and Dispatch entry-points do not appear to be protected
[!] Additional testing of the S3 boot-script can be done using tools.uefi.s3script_modify

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        0:
[CHIPSEC] Modules failed        1:
[-] FAILED: chipsec.modules.common.uefi.s3bootscript
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.bios_kbrd_buffer module

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

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

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.bios_kbrd_buffer
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.bios_kbrd_buffer
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_kbrd_buffer
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/bios_kbrd_buffer.py
[x][ =======================================================================
[x][ Module: Pre-boot Passwords in the BIOS Keyboard Buffer
[x][ =======================================================================
[*] Keyboard buffer head pointer = 0x0 (at 0x41A), tail pointer = 0x0 (at 0x41C)
[*] Keyboard buffer contents (at 0x41E):
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
[*] Checking contents of the keyboard buffer..

[+] PASSED: Keyboard buffer looks empty. Pre-boot passwords don't seem to be exposed

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.bios_kbrd_buffer
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.bios_smi
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
```

```
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.bios_smi
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_smi
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/bios_smi.py
[x][ =======================================================================
[x][ Module: SMI Events Configuration
[x][ =======================================================================
[+] SMM BIOS region write protection is enabled (SMM_BWP is used)

[*] Checking SMI enables..
    Global SMI enable: 1
    TCO SMI enable   : 1
[+] All required SMI events are enabled

[*] Checking SMI configuration locks..
[+] TCO SMI configuration is locked (TCO SMI Lock)
[+] SMI events global configuration is locked (SMI Lock)

[+] PASSED: All required SMI sources seem to be enabled and locked

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.bios_smi
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.bios_ts module

BIOS Boot Hijacking and VMware Vulnerabilities Digging - Sun Bing

Checks for BIOS Top Swap Mode

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.bios_ts
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.bios_ts
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_ts
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/bios_ts.py
[x][ =======================================================================
[x][ Module: BIOS Interface Lock and Top Swap Mode
[x][ =======================================================================
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE           = 0 << BIOS Write Enable
    [01] BLE              = 1 << BIOS Lock Enable
    [02] SRC              = 2 << SPI Read Configuration
    [04] TSS              = 0 << Top Swap Status
    [05] SMM_BWP          = 1 << SMM BIOS Write Protection
[*] BIOS Top Swap mode is disabled
[*] BUC = 0x00000000 << Backed Up Control (RCBA + 0x3414)
```

```
    [00] TS               = 0 << Top Swap
[*] RTC version of TS = 0
[*] GCS = 0x00000061 << General Control and Status (RCBA + 0x3410)
    [00] BILD             = 1 << BIOS Interface Lock Down
    [10] BBS              = 0 << Boot BIOS Straps

[+] PASSED: BIOS Interface is locked (including Top Swap Mode)

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.bios_ts
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.bios_wp module

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

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

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

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

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

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.bios_wp
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.bios_wp
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_wp
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/bios_wp.py
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)
```

```
    [00] BIOSWE           = 0 << BIOS Write Enable
    [01] BLE              = 1 << BIOS Lock Enable
    [02] SRC              = 2 << SPI Read Configuration
    [04] TSS              = 0 << Top Swap Status
    [05] SMM_BWP          = 1 << SMM BIOS Write Protection
[+] BIOS region write protection is enabled (writes restricted to SMM)

[*] BIOS Region: Base = 0x00180000, Limit = 0x007FFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (78)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0

[!] None of the SPI protected ranges write-protect BIOS region

[+] PASSED: BIOS is write protected

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.bios_wp
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.smm module

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

In 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.

This CHIPSEC module simply reads SMRAMC and checks that D_LCK is set.

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.smm
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.smm
[*] running loaded modules ..

[*] running module: chipsec.modules.common.smm
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/smm.py
[x][ =======================================================================
[x][ Module: Compatible SMM memory (SMRAM) Protection
[x][ =======================================================================
[*] PCI0.0.0_SMRAMC = 0x1A << System Management RAM Control (b:d.f 00:00.0 + 0x88)
    [00] C_BASE_SEG       = 2 << SMRAM Base Segment = 010b
    [03] G_SMRAME         = 1 << SMRAM Enabled
    [04] D_LCK            = 1 << SMRAM Locked
    [05] D_CLS            = 0 << SMRAM Closed
    [06] D_OPEN           = 0 << SMRAM Open
[*] Compatible SMRAM is enabled
[+] PASSED: Compatible SMRAM is locked down

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.smm
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.smrr module

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

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.smrr
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.smrr
[*] running loaded modules ..

[*] running module: chipsec.modules.common.smrr
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/smrr.py
[x][ =======================================================================
[x][ Module: CPU SMM Cache Poisoning / System Management Range Registers
[x][ =======================================================================
[+] OK. SMRR range protection is supported

[*] Checking SMRR range base programming..
[*] IA32_SMRR_PHYSBASE = 0xDC000006 << SMRR Base Address MSR (MSR 0x1F2)
    [00] Type             = 6 << SMRR memory type
    [12] PhysBase         = DC000 << SMRR physical base address
[*] SMRR range base: 0x00000000DC000000
[*] SMRR range memory type is Writeback (WB)
[+] OK so far. SMRR range base is programmed

[*] Checking SMRR range mask programming..
[*] IA32_SMRR_PHYSMASK = 0xFF000800 << SMRR Range Mask MSR (MSR 0x1F3)
    [11] Valid            = 1 << SMRR valid
    [12] PhysMask         = FF000 << SMRR address range mask
[*] SMRR range mask: 0x00000000FF000000
[+] OK so far. SMRR range is enabled

[*] Verifying that SMRR range base & mask are the same on all logical CPUs..
```

```
[CPU0] SMRR_PHYSBASE = 00000000DC000006, SMRR_PHYSMASK = 00000000FF000800
[CPU1] SMRR_PHYSBASE = 00000000DC000006, SMRR_PHYSMASK = 00000000FF000800
[CPU2] SMRR_PHYSBASE = 00000000DC000006, SMRR_PHYSMASK = 00000000FF000800
[CPU3] SMRR_PHYSBASE = 00000000DC000006, SMRR_PHYSMASK = 00000000FF000800
[+] OK so far. SMRR range base/mask match on all logical CPUs
[*] Trying to read memory at SMRR base 0xDC000000..
[+] PASSED: SMRR reads are blocked in non-SMM mode

[+] PASSED: SMRR protection against cache attack is properly configured

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.smrr
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.common.spi_desc module

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

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.spi_desc
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.spi_desc
[*] running loaded modules ..

[*] running module: chipsec.modules.common.spi_desc
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/spi_desc.py
[x][ =======================================================================
[x][ Module: SPI Flash Region Access Control
[x][ =======================================================================
[*] FRAP = 0x00004A4B << SPI Flash Regions Access Permissions Register (SPIBAR + 0x50)
    [00] BRRA             = 4B << BIOS Region Read Access
    [08] BRWA             = 4A << BIOS Region Write Access
    [16] BMRAG            = 0 << BIOS Master Read Access Grant
    [24] BMWAG            = 0 << BIOS Master Write Access Grant
[*] Software access to SPI flash regions: read = 0x4B, write = 0x4A

[+] PASSED: SPI flash permissions prevent SW from writing to flash descriptor

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.spi_desc
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

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 MITRE's Copernicus

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

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m common.spi_lock
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.common.spi_lock
[*] running loaded modules ..

[*] running module: chipsec.modules.common.spi_lock
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/common/spi_lock.py
[x][ =======================================================================
[x][ Module: SPI Flash Controller Configuration Lock
[x][ =======================================================================
[*] HSFS = 0xE008 << Hardware Sequencing Flash Status Register (SPIBAR + 0x4)
    [00] FDONE            = 0 << Flash Cycle Done
    [01] FCERR            = 0 << Flash Cycle Error
    [02] AEL              = 0 << Access Error Log
    [03] BERASE           = 1 << Block/Sector Erase Size
    [05] SCIP             = 0 << SPI cycle in progress
    [13] FDOPSS           = 1 << Flash Descriptor Override Pin-Strap Status
    [14] FDV              = 1 << Flash Descriptor Valid
    [15] FLOCKDN          = 1 << Flash Configuration Lock-Down
[+] PASSED: SPI Flash Controller configuration is locked

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.common.spi_lock
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

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.smm_ptr module

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

A tool to test SMI handlers for pointer validation vulnerabilities

Usage:

```
chipsec_main -m tools.smm.smm_ptr [ -a <mode>,<config_file>|<smic_start:smic_end>,<si ze>,<ad
```

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

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

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

Where

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

chipsec.modules.tools.uefi.s3script_modify module

Note

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

Examples:

```
chipsec_main.py -m tools.uefi.s3script_modify -a <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 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 dispatch
```

The modify_dispatch 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 dispatch_ep
```

The modify_dispatch_ep 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.modules.tools.vmm.cpuid_fuzz module

Simple CPUID VMM emulation fuzzer

Usage:

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

chipsec.modules.tools.vmm.iofuzz module

Simple port I/O VMM emulation fuzzer

Usage:

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

chipsec.modules.tools.vmm.msr_fuzz module

Simple CPU Module Specific Register (MSR) VMM emulation fuzzer

Usage:

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

chipsec.modules.tools.vmm.pcie_fuzz module

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

Usage:

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

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 -l pcie_overlap_fuzz.log
```

chipsec.modules.tools.vmm.venom module

QEMU VENOM vulnerability DoS PoC test Module is based on http://bluereader.org/article/41518389 which is based on PoC by Marcus Meissner (https://marc.info/?l=oss-security&m=143155206320935&w=2)

Usage:

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

chipsec.modules.module_template module

Template for a new module

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m module_template
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.module_template
[*] running loaded modules ..

[*] running module: chipsec.modules.module_template
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/module_template.py
Skipping module chipsec.modules.module_template since it is not supported in this platform

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        0:
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 1:
[*] SKIPPED: chipsec.modules.module_template
[CHIPSEC] *****************************************************************
```

chipsec.modules.remap module

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

Check Memory Remapping Configuration

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m remap
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.remap
[*] running loaded modules ..

[*] running module: chipsec.modules.remap
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/remap.py
[x][ =======================================================================
[x][ Module: Memory Remapping Configuration
[x][ =======================================================================
[*] Registers:
[*]   TOUUD     : 0x000000021FE00001
[*]   REMAPLIMIT: 0x000000021FD00001
[*]   REMAPBASE : 0x00000001FF000001
[*]   TOLUD     : 0xDF200001
[*]   TSEGMB    : 0xDC000001

[*] Memory Map:
[*]   Top Of Upper Memory: 0x000000021FE00000
[*]   Remap Limit Address: 0x000000021FDFFFFF
[*]   Remap Base Address : 0x00000001FF000000
[*]   4GB                : 0x0000000100000000
[*]   Top Of Low Memory  : 0x00000000DF200000
[*]   TSEG (SMRAM) Base  : 0x00000000DC000000

[*] checking memory remap configuration..
[*]   Memory Remap is enabled
[+]   Remap window configuration is correct: REMAPBASE <= REMAPLIMIT < TOUUD
[+]   All addresses are 1MB aligned
[*] checking if memory remap configuration is locked..
[+]   TOUUD is locked
[+]   TOLUD is locked
[+]   REMAPBASE and REMAPLIMIT are locked

[+] PASSED: Memory Remap is configured correctly and locked

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.remap
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

chipsec.modules.smm_dma module

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

DMA attacks were discussed in Programmed I/O accesses: a threat to Virtual Machine Monitors? and System Management Mode Design and Security Issues. This is also discussed in Summary of Attack against BIOS and Secure Boot .

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.

```
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.2
[CHIPSEC] Arguments: --failfast -m smm_dma
****** Chipsec Linux Kernel module is licensed under GPL 2.0

[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: 4th Generation Core Processor (Haswell U/Y)
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0A04

[+] loaded chipsec.modules.smm_dma
[*] running loaded modules ..

[*] running module: chipsec.modules.smm_dma
[*] Module path: <chipsec_path>/source/tool/chipsec/modules/smm_dma.py
[x][ =======================================================================
[x][ Module: SMM TSEG Range Configuration Check
[x][ =======================================================================
[*] TSEG      : 0x00000000DC000000 - 0x00000000DCFFFFFF (size = 0x01000000)
[*] SMRR range: 0x00000000DC000000 - 0x00000000DCFFFFFF (size = 0x01000000)

[*] checking TSEG range configuration..
[+] TSEG range covers entire SMRAM
[+] TSEG range is locked
[+] PASSED: TSEG is properly configured. SMRAM is protected from DMA attacks

[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        1:
[+] PASSED: chipsec.modules.smm_dma
[CHIPSEC] Modules failed        0:
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
```

Platform Configuration

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

chipsec.cfg.avn.xml module

Reference: Intel(R) Atom(TM) Processor C2000 Product Family for Microserver, September 2014

URL: http://www.intel.com/content/www/us/en/processors/atom/atom-c2000-microserver-datasheet.html

chipsec.cfg.bdw.xml module

XML configuration for Broadwell

chipsec.cfg.byt.xml module

XML configuration for Baytrail

Reference: Intel(R) Atom(TM) Processor E3800 Product Family Datasheet September 2014, Revision 3.5

chipsec.cfg.chipsec_cfg.xsd module

PCI

chipsec.cfg.common.xml module

Common xml configuration file

chipsec.cfg.hsw.xml module

XML configuration file for Haswell

chipsec.cfg.hsx.xml module

XML configuration file for Haswell Server

chipsec.cfg.iommu.xml module

XML configuration file for Intel VT-d

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

chipsec.cfg.ivt.xml module

XML configuration file for Ivytown

chipsec.cfg.jkt.xml module

XML configuration file for Jaketown

chipsec.cfg.skl.xml module

XML configuration file for Skylake

chipsec.cfg.template.xml module

Template for XML configuration file, this first comment will show in the documentation

OS/Environment Helpers

chipsec.helper.efi.efihelper module

On UEFI use the efi package functions

chipsec.helper.linux.helper module

Linux helper

chipsec.helper.win.win32helper module

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

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

chipsec.helper.oshelper module

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

HW Abstraction Layer (HAL)

Components responsible for access to hardware (Hardware Abstraction Layer)

chipsec.hal.acpi module

HAL component providing access to and decoding of ACPI tables

chipsec.hal.acpi_tables module

HAL component decoding various ACPI tables

chipsec.hal.cmos module

CMOS memory specific functions (dump, read/write)

usage:

```
>>> dump()
>>> read_byte( offset )
>>> write_byte( offset, value )
```

chipsec.hal.cpu module

CPU related functionality

chipsec.hal.cpuid module

CPUID information

usage:

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

chipsec.hal.hal_base module

Base for HAL Components

chipsec.hal.interrupts module

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

usage:

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

chipsec.hal.io module

Access to Port I/O

usage:

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

chipsec.hal.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.iommu module

Access to IOMMU engines

chipsec.hal.mmio module

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

usage:

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

Access MMIO by BAR name:

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

Access Memory Mapped Config Space:

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

DEPRECATED: Access MMIO by BAR id:

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

chipsec.hal.msr module

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

usage:

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

chipsec.hal.pci module

Access to PCIe configuration spaces of I/O devices

usage:

```
>>> read_pci_dword( 0, 0, 0, 0x88 )
>>> write_pci_dword( 0, 0, 0, 0x88, 0x1A )
```

chipsec.hal.pcidb module

Note

THIS FILE WAS GENERATED

Auto generated from:

http://www.pcidatabase.com/vendors.php?sort=id http://www.pcidatabase.com/reports.php?type=csv

chipsec.hal.physmem module

Access to physical memory

usage:

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

DEPRECATED

```
>>> read_phys_mem( 0xf0000, 0x100 )
>>> write_phys_mem_dword( 0xf0000, 0xdeadbeef )
>>> read_phys_mem_dword( 0xfed40000 )
```

chipsec.hal.smbus module

Access to SMBus Controller

chipsec.hal.spd module

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

References:

http://www.jedec.org/sites/default/files/docs/4_01_02R19.pdf
http://www.jedec.org/sites/default/files/docs/4_01_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
http://www.simmtester.com/page/news/showpubnews.asp?num=184
http://www.simmtester.com/page/news/showpubnews.asp?num=153
http://www.simmtester.com/page/news/showpubnews.asp?num=101
http://en.wikipedia.org/wiki/Serial_presence_detect

chipsec.hal.spi module

Access to SPI Flash parts

usage:

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

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

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

Approximate performance (on 2 core HT Sandy Bridge CPU 2.6GHz): SPI read: ~25 sec per 1MB (DBC=64) SPI write: ~140 sec per 1MB (DBC=4)

chipsec.hal.spi_descriptor module

SPI Flash Descriptor binary parsing functionality

usage:

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

chipsec.hal.spi_uefi module

SPI UEFI Region parsing

usage:

```
>>> parse_uefi_region_from_file( filename )
```

chipsec.hal.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.uefi module

Main UEFI component using platform specific and common UEFI functionality

chipsec.hal.uefi_common module

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

chipsec.hal.uefi_platform module

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

Utility command-line scripts

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

Warning

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

Note

All numeric values in the instructions are in hex.

chipsec.utilcmd.acpi_cmd module

Command-line utility providing access to ACPI tables

`acpi` `(argv)`

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

Examples:

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

chipsec.utilcmd.chipset_cmd module

usage as a standalone utility:

```
>>> chipsec_util platform
```

`platform` (argv)

chipsec_util platform

chipsec.utilcmd.cmos_cmd module

`cmos` (argv)

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

Examples:

```
>>> chipsec_util cmos dump
>>> chipsec_util cmos rl 0x0
>>> chipsec_util cmos wh 0x0 0xCC
```

chipsec.utilcmd.cpu_cmd module

`cpu_cmd` (argv)

```
>>> chipsec_util cpu info
>>> chipsec_util cpu cr <cpu_id> <cr_number> [value]
```

Examples:

```
>>> chipsec_util cpu info
>>> chipsec_util cpu cr 0 0
>>> chipsec_util cpu cr 0 4 0x0
```

chipsec.utilcmd.cpuid_cmd module

`cpuid` (argv)

```
>>> chipsec_util cpuid <eax> [ecx]
```

Examples:

```
>>> chipsec_util cpuid 40000000
```

chipsec.utilcmd.decode_cmd module

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

Examples:

chipsec_util decode spi.bin vss

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

decode (argv)

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

For a list of fw types run:

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

Examples:

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

chipsec.utilcmd.desc_cmd module

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

gdt (argv)

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

Examples:

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

idt (argv)

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

Examples:

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

ldt (argv)

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

Examples:

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

chipsec.utilcmd.interrupts_cmd module

nmi (argv)

```
>>> chipsec_util nmi
```

Examples:

```
>>> chipsec_util nmi
```

smi (argv)

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

Examples:

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

chipsec.utilcmd.io_cmd module

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

port_io (argv)

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

Examples:

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

chipsec.utilcmd.iommu_cmd module

Command-line utility providing access to IOMMU engines

iommu_cmd (argv)

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

Examples:

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

chipsec.utilcmd.mem_cmd module

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

mem (argv)

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

Examples:

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

chipsec.utilcmd.mmcfg_cmd module

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

mmcfg (argv)

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

Examples:

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

chipsec.utilcmd.mmio_cmd module

mmio (argv)

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

Examples:

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

chipsec.utilcmd.msr_cmd module

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

msr (argv)

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

Examples:

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

chipsec.utilcmd.pci_cmd module

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

pci (argv)

```
>>> 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
```

chipsec.utilcmd.smbus_cmd module

smbus (argv)

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

Examples:

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

chipsec.utilcmd.spd_cmd module

spd (argv)

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

Examples:

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

chipsec.utilcmd.spi_cmd module

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

Warning

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

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

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

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

spi (argv)

```
>>> 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.utilcmd.spidesc_cmd module

spidesc (argv)

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

Examples:

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

chipsec.utilcmd.ucode_cmd module

ucode (argv)

```
>>> 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.

uefi (argv)

```
>>> chipsec_util uefi var-list
>>> chipsec_util uefi var-find <name>|<GUID>
>>> chipsec_util uefi var-read|var-write|var-delete <name> <GUID> <efi_variable_file>
>>> chipsec_util uefi nvram[-auth] <fw_type> [rom_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> <bios_rom> <modified_bios_rom> <uefi_file>
```

For a list of fw types run:

```
>>> chipsec_util uefi types
```

Examples:

```
>>> chipsec_util uefi var-list
>>> chipsec_util uefi var-find PK
>>> chipsec_util uefi var-read db D719B2CB-3D3A-4596-A3BC-DAD00E67656F db.bin
>>> chipsec_util uefi var-write db D719B2CB-3D3A-4596-A3BC-DAD00E67656F db.bin
>>> chipsec_util uefi var-delete db D719B2CB-3D3A-4596-A3BC-DAD00E67656F
>>> chipsec_util uefi nvram fwtype bios.rom
>>> chipsec_util uefi nvram-auth fwtype bios.rom
>>> chipsec_util uefi decode uefi.bin fwtype
>>> 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_file.raw uefi_file.bin
>>> chipsec_util uefi replace  AAAAAAAA-BBBB-CCCC-DDDD-EEEEEEEEEEEE bios.bin modified_bios.bin uefi_file.bin
```

Auxiliary components

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

Executable build scripts

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