CHIPSEC

version 1.2.0

Platform Security Assesment Framework

June 08, 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 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 20 LXDE 64bit
- Fedora 21 LXDE 64bit
- Ubuntu 14.04 LTE 64bit

Creating a Live Linux image with CHIPSEC

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

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

or

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

> ***Note***
>
> When installing using a live image, it is difficult to actually update the kernel. Instead of doing this, you can simply 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)`.

5. Copy chipsec to the USB stick

Installing CHIPSEC

6. Build Linux driver for CHIPSEC
    - `cd source/drivers/linux`
    - `make`
7. Load CHIPSEC driver in running system
    - `cd source/drivers/linux`
    - (Optional) `chmod 755 run.sh`
    - `sudo ./run.sh` or `sudo make install`

OR

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

8. Run CHIPSEC

    `cd source/tool` `sudo python chipsec_main.py` or `sudo python chipsec_util.py`
9. Remove CHIPSEC driver after using

    `sudo make uninstall`

UEFI Shell Installation

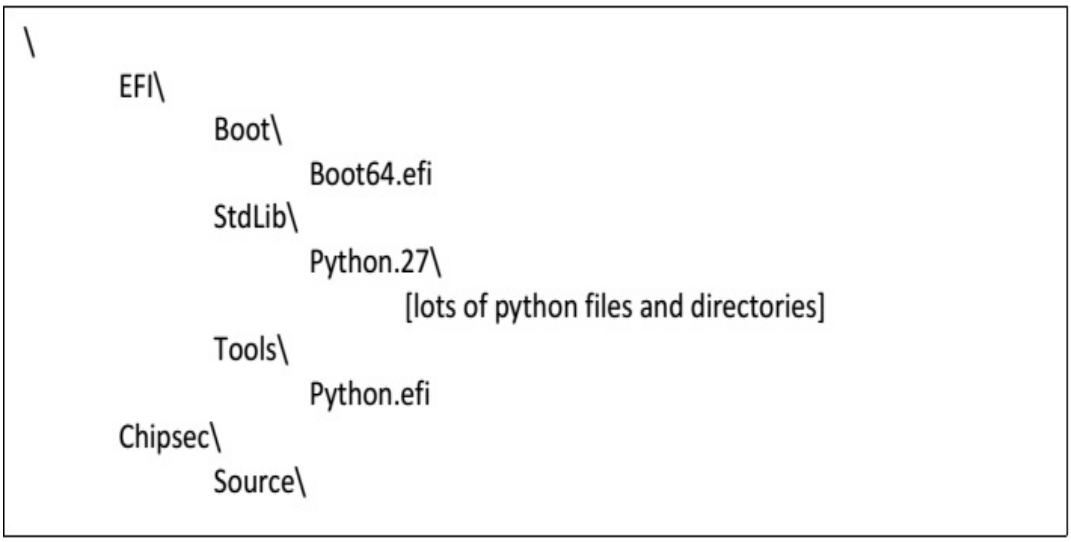
Building bootable USB thumb drive with UEFI Shell

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:



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 chipsec_main.py

```
################################################################################
## ##
## CHIPSEC: Platform Hardware Security Assessment Framework ##
## ##
################################################################################
[CHIPSEC] Version 1.2.0
[CHIPSEC] Arguments:

WARNING: ****************************************************************
WARNING: Chipsec should only be used on test systems!
WARNING: It should not be installed/deployed on production end-user systems.
WARNING: See WARNING.txt
WARNING: ****************************************************************

ERROR: could not locate driver file
'C:BuildAgentworkd5dd2e9ba98efba8artifactsPUBLICsourcetoolchipsechelperwinwin7_amd64chipsec_hlpr.sys'
ERROR: Could not start the OS Helper, are you running as Admin/root?
Message: "(6, 'QueryServiceStatus', 'The handle is invalid.')"
```

- 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 | IVT | AVN | HSW | 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 where 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 (security check or an exploit..):

- `# python chipsec_main.py -m common.bios_wp`
- `# python chipsec_main.py -m common.spi_desc`
- `# 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

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.

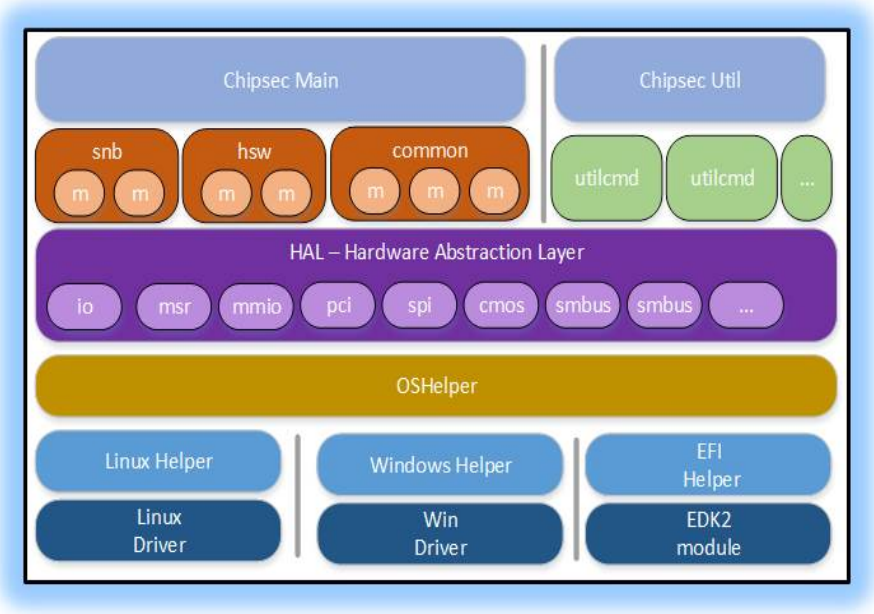
Copy your module into the chipsec/modules/ directory structure

- 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.modules.common.uefi.access_uefispec module

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

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

chipsec.modules.common.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.modules.common.bios_smi module

Setup for Failure: Defeating SecureBoot by Corey Kallenberg, Xeno Kovah, John Butterworth, Sam Cornwell

Checks for SMI events configuration

chipsec.modules.common.bios_ts module

BIOS Boot Hijacking and VMware Vulnerabilities Digging - Sun Bing

Checks for BIOS Top Swap Mode

chipsec.modules.common.bios_wp module

Black Hat USA 2013 BIOS Security by MITRE (Kovah, Butterworth, Kallenberg)

NoSuchCon 2013 BIOS Chronomancy: Fixing the Static Root of Trust for Measurement by MITRE (Kovah, Butterworth, Kallenberg)

Checks if BIOS Write Protection HW mechanisms are enabled

chipsec.modules.common.smm module

CanSecWest 2006 Security Issues Related to Pentium System Management Mode by Duflot

Common checks for protection of compatible System Management Mode (SMM) memory (SMRAM)

chipsec.modules.common.smrr module

Attacking SMM Memory via Intel CPU Cache Poisoning by ITL (Rutkowska, Wojtczuk)

Getting into the SMRAM: SMM Reloaded by Duflot, Levillain, Morin, Grumelard

Checks for SMRR configuration to protect from SMRAM cache attack

chipsec.modules.common.spi_desc module

Parsing of SPI descriptor access permissions is implemented in "ich_descriptors_tool" which is part of open source flashrom

Checks SPI Flash Region Access Permissions programmed in the Flash Descriptor

chipsec.modules.common.spi_lock module

FLOCKDN is in flashrom and MITRE's Copernicus

Checks that the SPI Flash Controller configuration is locked. If it is not locked other Flash Program Registers could be written.

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>
- `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.module_template module

Template for a new module

chipsec.modules.remap module

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

Check Memory Remapping Configuration

chipsec.modules.smm_dma module

Programmed I/O accesses: a threat to Virtual Machine Monitors? by Lioc Duflot & Laurent Absil

Check SMM memory (SMRAM) is properly configured to protect from DMA attacks.

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.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.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.cpuid module

CPUID information

usage:

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

chipsec.hal.cr module

Access to CR registers

usage:

```
>>> read_cr( 0 )
>>> write_cr( 4, 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.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.cpuid_cmd module

`cpuid` (`argv`)

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

Examples:

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

chipsec.utilcmd.cr_cmd module

`crx` (`argv`)

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

Examples:

```
>>> chipsec_util cr 0 0
>>> chipsec_util cr 0 4 0x0
```

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

Examples:

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

chipsec.utilcmd.mem_cmd module

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

`mem` (argv)

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

Examples:

```
>>> chipsec_util mem 0x0 0x41E 0x20
>>> chipsec_util mem 0x0 0xA0000 4 0x9090CCCC
>>> chipsec_util mem 0x0 0xFED40000 0x4
>>> chipsec_util mem allocate 0x1000
```

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-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]
```

For a list of fw types run:

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

Examples:

```
>>> chipsec_util uefi var-list
>>> chipsec_util uefi var-read db D719B2CB-3D3A-4596-A3BC-DAD00E67656F db.bin
>>> chipsec_util uefi var-write db D719B2CB-3D3A-4596-A3BC-DAD00E67656F db.bin
>>> chipsec_util uefi 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
```

Auxiliary components

`bist.cmd`	built-in self test for various basic HW functionality to make sure it's not broken
`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