Security of BIOS/UEFI System Firmware from Attacker and Defender Perspectives

Section 3. Hands-On Learning of Platform Hardware and Firmware

Yuriy Bulygin *
Alex Bazhaniuk *
Andrew Furtak *
John Loucaides **

* Advanced Threat Research, McAfee
** Intel

License

Training materials are shared under Creative Commons “Attribution” license CC BY 4.0

Provide the following attribution:

Derived from “Security of BIOS/UEFI System Firmware from Attacker and Defender Perspective” training by Yuriy Bulygin, Alex Bazhaniuk, Andrew Furtak and John Loucaides available at https://github.com/advanced-threat-research/firmware-security-training

Section 3. Hands-On Learning of Platform Hardware and Firmware

3.0 Building and Installing CHIPSEC

Bootable Linux USB with CHIPSEC

Bootable Linux USB with CHIPSEC

Ubuntu bootable USB with CHIPSEC (includes all dependencies)

Building and running CHIPSEC:

```
# cd ~/Desktop/chipsec/source/
# git pull (you already have latest CHIPSEC on USB)

# python setup.py build_ext -i
# sudo python chipsec_util.py platform
```

Installing CHIPSEC on Windows

1. Install Python 2.7.x (http://www.python.org/download/)
2. Install additional packages for installed Python release

   ```
   pip install setuptools
   pip install pywin32
   ```

3. Build CHIPSEC Windows driver. Skip this step if you already have `chipsec_hlpr.sys` built
4. Copy CHIPSEC driver (chipsec_hlpr.sys) to proper path in CHIPSEC `\chipsec\chipsec\helper\win\win7_amd64` or `win7_x86`
5. Install CHIPSEC

   ```
   pip install chipsec
   ```

6. Turn off kernel driver signature checks in Windows 8, 8.1, 10 64-bit

Refer to CHIPSEC manual:
https://github.com/chipsec/chipsec/blob/master/chipsec-manual.pdf

3.1 Access to Hardware Resources

Hardware Configuration

CPU

1. x86 state: GPR (RAX, …), Control Registers (CRx), Debug Registers (DRx), etc.
2. CPU Model Specific Registers (MSR)

CPU and Chipset (SoC)

1. Processor I/O space: I/O ports and I/O BARs
2. PCIe devices configuration space
3. Memory-mapped PCIe configuration access a.k.a. Enhanced Configuration Access Mechanism (ECAM)
4. Memory-mapped I/O ranges
5. IOSF Message Bus registers

Processor I/O Space: I/O Ports and BARs

- Legacy I/O interface accessible via x86 IN and OUT assembly instructions
- Offset in I/O space is *I/O register* or *I/O port.* I/O space contains multiple *ranges* assigned to some device or controller
- I/O ranges can be *fixed*:
  - `60h/62h/64h/66h`: keyboard/embedded uController
  - `CF8h/CFCh`: PCIe devices CFG access
  - `CF9h`: platform reset
  - `B2h/B3h`: APMC/SMI
- Or *variable* (defined by I/O BAR registers)
  - SMBus
  - ACPI/PMBASE
  - GPIO
  - I/O Trap

Processor I/O Space: I/O Ports and BARs

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

```
# chipsec_util.py io 0xcf8 dword 0x80000000
[CHIPSEC] OUT 0x0CF8 <- 0x80000000 (size = 0x04)

# chipsec_util.py io 0xcfc dword
[CHIPSEC] IN 0x0CFC -> 0x0A048086 (size = 0x04)
```

PCIe Configuration Space Access

SW uses one these mechanisms to access config space:

1. Legacy configuration access via *control* **`CF8h`** & *data* **`CFCh`** processor I/O ports
   - PCI config register address

8 * 100h per device

```
bus << 16 | device << 11 | function << 8 | offset & ~3
```

32* 8 * 100h per bus

100h bytes of CFG header

   - **`CF8h`** ← **`1<<31 | bdf_address`**
   - Read data from or write data to port (**`CFCh + off[1:0]`**)

2. Extended (memory-mapped) config access (see later)

PCIe Configuration Space and Registers

- Enumerate all available PCIe devices:

```
# chipsec_util pci enumerate
[CHIPSEC] Enumerating available PCIe devices..
BDF     | VID:DID   | Vendor                                          | Device
-------------------------------------------------------------------------------
00:00.0 | 8086:0A04 | Intel Corporation                               |
00:02.0 | 8086:0A16 | Intel Corporation                               |
```

- Reading from/writing to PCIe device's configuration space:

```
# chipsec_util.py pci <bus> <dev> <fun> <off> <width> [value]

# chipsec_util.py pci 0 0 0 0x0 dword

[CHIPSEC] reading PCI B/D/F 0/0/0, off 0x00: 0xA048086

# chipsec_util.py pci 0 0x1F 0 0xDC byte

[CHIPSEC] reading PCI B/D/F 0/31/0, off 0xDC: 0x2A
```

Extended PCIe Configuration

- Enhanced Configuration Access Mechanism (ECAM) allows accessing PCIe extended configuration space (4kB) beyond PCI configuration space (256 bytes)
- To access entire PCIe extended configuration space CPU reserves memory-mapped range in physical addressable memory (MMCFG)
  - Range is re-locatable (e.g. PCIEXBAR register in B.D:F 0.0:0 on Core/Xeon, ECBASE msgbus register on Atom, MSR on AMD APUs…)
- All access to MMCFG range is mapped to PCI configuration cycles
- MMCFG is split into consecutive 4kB large chunks, each is extended CFG header per bus/device/function
- Access is done at memory offset within MMCFG range

```
MMCFG offset =

bus*32*8*1000h + dev*8*1000h + fun*1000h + offset
```

ECAM (MMCFG) Address Mapping

Memory Address	PCI Express Configuration Space
A[(20+n-1):20]	Bus Numbers 1 ≤ n ≤ 8
A[19:15]	Device Number
A[14:12]	Function Number
A[11:8]	Extended Register Number
A[7:2]	Register Number
A[1:0]	Along with size of the access, used to generate Byte Enable

Memory Mapped PCIe Configuration

chipsec_util.py mmio with 'MMCFG' BAR

```
# chipsec_util.py mmio list
------------------------------------------------------------------------------
 MMIO Range   | BAR          | Base             | Size     | En? | Description
------------------------------------------------------------------------------
...
MMCFG        | 00:00.0 + 60 | 00000000F8000000 | 00001000 | 1   | PCI Express Range
...

# chipsec_util.py mmio read MMCFG 0x0 0x4
[CHIPSEC] Read MMCFG + 0x0: 0x0A048086
```

 # chipsec_util.py mmcfg <b> <d> <f> <off> <width> [value]

```
# chipsec_util.py mmcfg
[CHIPSEC] Memory Mapped Config Base: 0x00000000F8000000

# chipsec_util.py mmcfg 0 0 0 0x0 dword
[CHIPSEC] reading MMCFG register (00:00.0 + 0x00): 0xA048086

# chipsec_util.py mmcfg 0 0 0 0xF80DC byte
[CHIPSEC] reading MMCFG register (00:00.0 + 0xF80DC): 0x2A
```

It doesn't work at MinnowBoard, because Bay Trail has different interface for `MMCFG` (use address `0xE0000000` as `MMCFG`)

Memory Mapped I/O Registers

- Devices may have more registers than I/O and PCIe CFG spaces can fit so BIOS may reserve physical address ranges for devices
- Ranges are defined by Base Address Registers (BAR). MMIO registers are offsets off of base of MMIO ranges
- Any access to such MMIO range is forwarded to the device which owns this range (local in the CPU or over a system bus to chipset) rather than decoded to DRAM
- `mmio` command in CHIPSEC can be used to list predefined MMIO BARs, dump entire BAR, and read/write MMIO registers

```
# chipsec_util.py mmio list
-------------------------------------------------------------------------------
 MMIO Range   | BAR          | Base             | Size     | En? | Description
-------------------------------------------------------------------------------
 GTTMMADR     | 00:02.0 + 10 | 00000000F0000000 | 00001000 | 1   | Graphics Translation Table Range
 SPIBAR       | 00:1F.0 + F0 | 00000000FED1F800 | 00000200 | 1   | SPI Controller Register Range
 HDABAR       | 00:03.0 + 10 | 0000007FFFFFF000 | 00001000 | 1   | HD Audio Controller Register Range
 GMADR        | 00:02.0 + 18 | 00000000E0000000 | 00001000 | 1   | Graphics Memory Range
 DMIBAR       | 00:00.0 + 68 | 00000000FED18000 | 00001000 | 1   | Root Complex Register Range
 MMCFG        | 00:00.0 + 60 | 00000000F8000000 | 00001000 | 1   | PCI Express Register Range
 RCBA         | 00:1F.0 + F0 | 00000000FED1C000 | 00004000 | 1   | PCH Root Complex Register Range
 MCHBAR       | 00:00.0 + 48 | 00000000FED10000 | 00008000 | 1   | Host Memory Mapped Register Range
...

# chipsec_util.py mmio read|write|dump <BAR_name> <off> <width> [value]

# chipsec_util.py mmio read SPIBAR 0x78 4
[CHIPSEC] Read SPIBAR + 0x78: 0x8FFF0F40
```

CPU Model Specific Registers (MSR)

- CPU contains many Model Specific Registers (MSR) to enable/disable/configure various features & read statuses
- MSRs can be architectural (e.g. IA32_APIC_BASE) and specific to some CPU models (e.g. LBR_TO/FROM_MSR)
- MSRs can be per logical CPU, core or entire CPU package

- Reading from / writing to CPU MSRs:

```
# chipsec_util.py msr <msr> [eax] [edx] [cpu_id]
```

- Specifying `cpu_id` allows access to MSRs of specific logical CPU
  - When omitted the command reads/writes MSR for all logical CPU in the package

IA-32 Control Registers (CR)

- x86 CPU CRs control behavior/state of the logical CPU
- Example:
  - CR0.PG - paging enabled
  - CR0.PE - protection enabled
  - CR3 (PDBR) - physical address of page directory
  - CR4.SMEP / CR4.SMAP

- Reading from / writing to CRs:

```
# chipsec_util.py cpu cr <cpu_id> <cr_number> [value]
```

- Access to CRs is per logical CPU, you need to specify the # of the logical CPU (cpu_id) in CHIPSEC

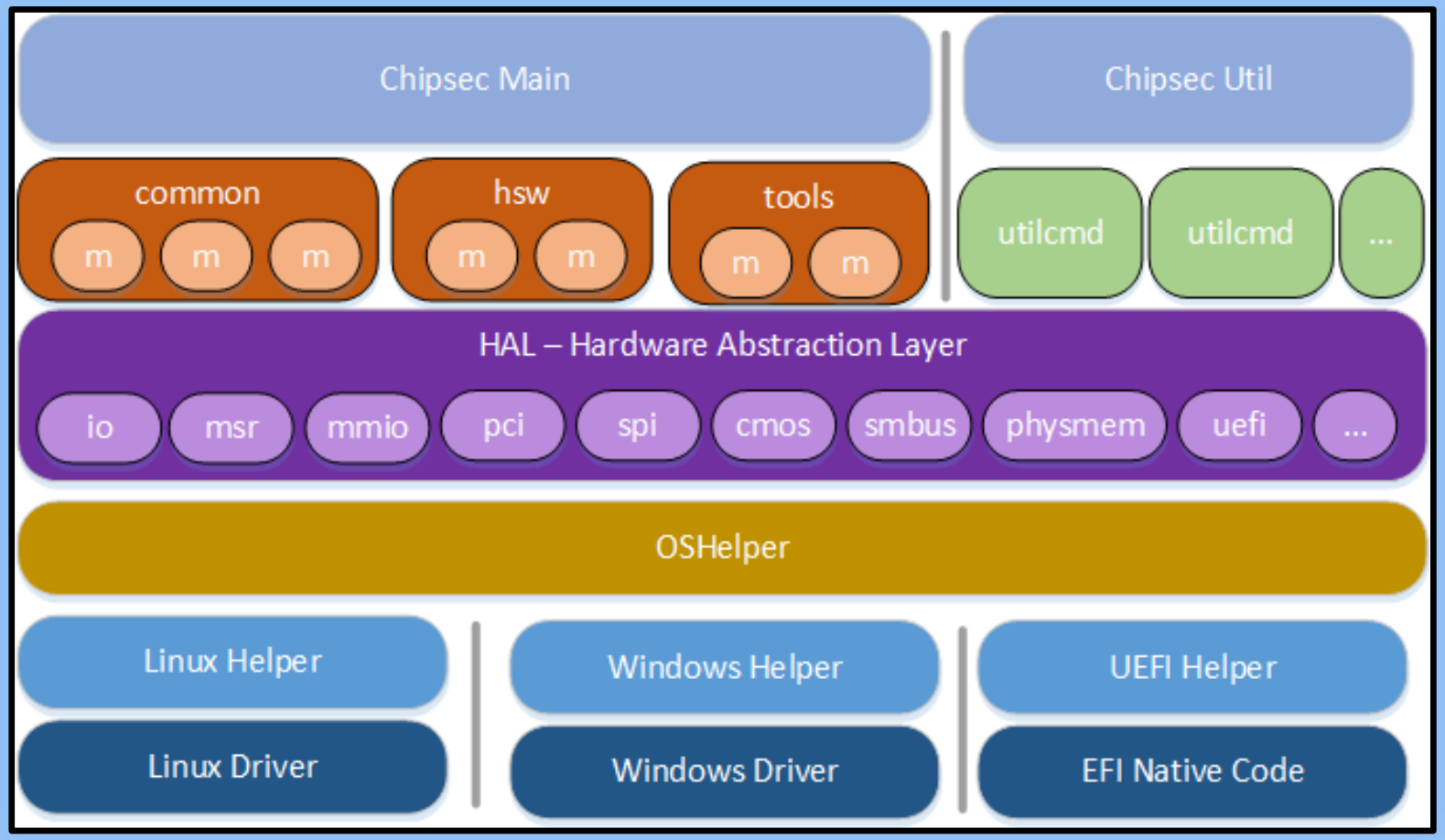
Exercise 3.1

Read BIOS/SPI Security Configuration

Exercise 3.2

Access Hardware Resources

3.2 Overview of Open Source CHIPSEC Framework



*Other names and brands may be claimed as the property of others.

Structure

chipsec_main.py	runs modules (see modules dir below)
chipsec_util.py	runs manual utilities (see utilcmd dir below)
/chipsec	
/cfg	platform specific configuration
/hal	all the HW stuff you can interact with
/helper	support for OS/environments
/modules	modules (tests/tools/PoCs) go here
/utilcmd	utility commands for chipsec_util

OS/Environment Specific Helpers

CHIPSEC supports Windows, Linux and UEFI shell environment.

OS/environment specific helpers for:

- Windows : `helper\win\win32helper.py`
- Linux : `helper\linux\helper.py`
- UEFI (shell): `helper\efi\efihelper.py`

Abstracts for support various OS/environments, wrapper around platform specific code that invokes kernel driver implemented in: `helper/oshelper.py`

Call path:

`Module (or util)` → `HAL component` → `oshelper`

→ `helper [Linux]` → `OS native code [LKM]`

Helper Code Example

helper/oshelper.py

```
    # Read/Write CR registers
    def read_cr(self, cpu_thread_id, cr_number):
        return self.helper.read_cr( cpu_thread_id, cr_number )
```

OS independent function which all HAL components invoke (read_cr)

helper\linux\helper.py

```
def IOCTL_RDCR():   return _IOCTL_BASE + 0x10
. . .
class LinuxHelper:
     def __init__(self):
        import platform
        self.os_system  = platform.system()
        self.os_machine = platform.machine()
. . .
        self.init()
. . .
    def read_cr(self, cpu_thread_id, cr_number):
        self.set_affinity(cpu_thread_id)
        cr = 0
        in_buf = struct.pack( "3"+_PACK, cpu_thread_id, cr_number, cr)
        unbuf = struct.unpack("3"+_PACK, fcntl.ioctl( _DEV_FH, IOCTL_RDCR(), in_buf ))
        return (unbuf[2])
```

IOCTL invoking handler in the kernel module (“read CR”)

OS specific (Linux) helper class specific to each OS

read_cr function in Linux helper invokes IOCTL within the kernel module

Detecting the Platform

- Each platform supports its own set of hardware resources (interfaces, configuration registers)
- Different modules may be applicable to only specific platforms or family of platforms
- To support above, CHIPSEC detects the platform it's running on
- Detection is done using Host Controller Device ID (b.d:f 00.00:0). Supported DIDs are in `chipsec/chipset.py`
  - **Tip:** to add support of a new platform, add its description in `Chipset_Dictionary`. Alternatively, create custom_chipsets.py, add `my_dict` with additional DIDs and add them to the main dictionary using `chipset.Chipset_Dictionary.update(my_dict)`
- After detection, configuration (XMLs) for the detected platform is initialized (`chipset.init_xml_configuration`)

Detecting the Platform

```
# chipsec_util.py platform
```

```
Supported platforms:

DID     | Name           | Code   | Long Name
------------------------------------------------------------------------------
...
 0xa04  | Haswell        | hsw    | 4th Generation Core Processor (Haswell U/Y)
 0xc08  | Haswell        | hsw    | Intel Xeon Processor E3-1200 v3 (Haswell CPU, C220 Series PCH)
 0xa08  | Haswell        | hsw    | 4th Generation Core Processor (Haswell U/Y)
 0xa00  | Haswell        | hsw    | 4th Generation Core Processor (Haswell U/Y)
 0xc00  | Haswell        | hsw    | Desktop 4th Generation Core Processor (Haswell CPU / Lynx Point PCH)
 0xc04  | Haswell        | hsw    | Mobile 4th Generation Core Processor (Haswell M/H / Lynx Point PCH)
 0x158  | Ivy Bridge     | ivb    | Intel Xeon Processor E3-1200 v2 (Ivy Bridge CPU, C200/C216 Series PCH)
 0x150  | Ivy Bridge     | ivb    | Desktop 3rd Generation Core Processor (Ivy Bridge CPU / Panther Point PCH)
 0x154  | Ivy Bridge     | ivb    | Mobile 3rd Generation Core Processor (Ivy Bridge CPU / Panther Point PCH)
...

Platform: 4th Generation Core Processor (Haswell U/Y)
          VID: 8086
          DID: 0A04
```

- Additional command-line options:
  - `--platform (-p)`: use it if you know the platform but CHIPSEC doesn't auto detect the platform
  - `--ignore_platform (-i)`: avoid platform detection when using platform agnostic functionality (e.g. access to UEFI variables)

HW Abstraction Layer (HAL)

- HAL is the set of components providing access to various hardware resources on the platform
- HAL components abstract HW access specific to OS environment and expose it via a set of common APIs consumed by all modules in any OS environment
- HAL components are OS unaware and invoke common helper functions from OS agnostic `oshelper.py`
- HAL components can be *basic primitives* or *complex*
  - Basic primitive HAL components provide access to basic HW resource (Example: CPU I/O, MMIO, etc.)
  - Complex HAL components use basic primitive HAL to implement access to higher level HW resources (Example: SPI access is implemented via MMIO access)

HW Abstraction Layer (HAL)

File name	Description
`hal/pci.py`	Access to PCIe configuration space
`hal/physmem.py`	Access to physical memory
`hal/msr.py`	Access to CPU resources (for each CPU thread): Model Specific Registers (MSR), IDT/GDT
`hal/mmio.py`	Access to MMIO (Memory Mapped IO) BARs and Memory-Mapped PCI Configuration Space (MMCFG)
`hal/spi.py`	Access to SPI Flash parts
`hal/ucode.py`	Microcode update specific functionality
`hal/io.py`	Access to Port I/O Space
`hal/smbus.py`	Access to SMBus Controller in the PCH
`hal/uefi.py`	Main UEFI component using platform specific and common UEFI functionality
`hal/uefi_common.py`	Common UEFI functionality (EFI variables, db/dbx decode, etc.)
`hal/uefi_platform.py`	Platform specific UEFI functionality (parsing platform specific EFI NVRAM, capsules, etc.)
`hal/interrupts.py`	CPU Interrupts specific functions (SMI, NMI)
`hal/cmos.py`	CMOS memory specific functions (dump, read/write)
`hal/cpuid.py`	CPUID information
`hal/spi_descriptor.py`	SPI Flash Descriptor binary parsing functionality

HAL: Basic HW Access

1. Basic HAL primitives is a set of HAL components which provide access to basic HW resources which are used to access any other HW resources
2. Each basic HAL primitive has its own OS native functions (e.g. in kernel module) implementing access to corresponding HW resource specific to that OS
3. Basic HAL primitives are IO, MEM, MSR, MMIO, PCIE, CR, CPUID, etc.
4. All other HAL components are complex and can be implemented using the above set of basic HAL primitives
5. Basic primitives can be accessed through Chipset instance:

```
cs       = chipsec.chipset.cs()
pci_devs = cs.pci.enumerate_devices()
```

HAL Example: SPI Flash Memory Access

- SPI Flash Memory Access is a HAL component which implements access to system SPI flash memory devices
- Current SPI HAL implementation uses *hardware sequencing* access (which predefines SPI flash opcodes and can operate in descriptor mode only)
- Exposes the following API:
  - `read_spi, write_spi, erase_spi_block` – access to SPI flash
  - `get_SPI_regions` – returns SPI flash regions
  - `get_SPI_Protected_Range` – returns SPI flash protected ranges
  - `display_SPI_Flash_Descriptor` – decodes SPI flash descriptor
- Accessed through `chipsec_util spi/decode` commands

HAL Example: CPU Configuration Access

- SPI Flash Memory Access is a HAL components which implement access to CPU HW resources (MSR, descriptor tables, microcode updates, CPUID, CR, interrupts ..)
- Provide the following API:
  - `msr.read_msr, msr.write_msr` – access to CPU MSRs
  - `msr.get_IDTR, msr.get_GDTR` – read IDT/GDT
  - `ucode.ucode_update_id` – read microcode update ID
  - `cpuid.cpuid` – read CPU CPUID
  - `interrupts.send_SMI_APMC` – send SMI through port B2h
  - `cr.read_cr, cr.write_cr` – access to CPU Control Registers
- Accessed through `chipsec_util spi/cr/ucode/cupid/smi/idt/gdt` commands

HAL Example: UEFI

- UEFI HAL components implements functionality to work with UEFI interfaces and structures
  - Dumping UEFI Variables at run-time through UEFI API
  - Extracting UEFI Variables from NVRAM store in SPI memory dump
  - Decoding certificates/hashes from UEFI variables
  - Parsing UEFI Volumes with executables from SPI memory dump
  - Extracting and decoding S3 resume boot script
- Common UEFI API consumed by modules is exposed through `chipsec.hal.uefi`
- UEFI functionality can be common for all UEFI based firmware or can depend on BIOS implementation or UEFI version
  - Common UEFI functionality is in `hal/uefi_common.py`
  - BIOS dependent UEFI functionality is in `hal/uefi_platform.py`
- Accessed through `chipsec_util uefi` command

Platform Configuration

- Each platform (chipset, CPU, devices) has it's own configuration defined by registers/ranges in I/O, MMIO, PCIe CFG, MSR spaces...
  - The same register may be defined at different offsets, even in different places on different platforms
  - The definition of the register may change (bits, masks ..)
  - Definitions of I/O or MMIO ranges change (location, size ..)
  - Each platform may have its own set of internal devices or controllers
- We don't want to re-write modules for every new platform
- It would be nice to be able do this (regardless of where register is):

```
reg = read_register( "MY_REGISER" )
reg_field = read_register_field( "MY_REGISER", "MY_FIELD" )
```

- CHIPSEC does that using configuration described in XML files for each platform, or per-feature, or common (`chipsec/cfg` directory)
- Look for these lines in the output:

```
[*] loading common platform config from '..\chipsec/cfg\common.xml'..
[*] loading 'hsw' platform config from '..\chipsec/cfg\hsw.xml'..
```

Platform Configuration: IO, MMIO…

- Internal PCIe devices (devices, controllers, interfaces ..)

```
<pci>
  <device name='HOSTCTRL' bus='0x0' dev='0x0' fun='0x0'../>
```

- Memory Mapped I/O ranges (BARs)

```
<mmio>
  <bar name='SPIBAR'.. reg='0xF0' enable_bit='0' ../>
```

- Legacy port I/O ranges (BARs)

```
<io>
  <bar name='ABASE'.. reg='0x40' ../>
```

- Memory ranges

```
<memory>
  <range name='LEGACY' address='0x00' size='0x100000' ../>
```

Platform Configuration: Registers

- Configuration registers

```
<registers>

  <register name='BC' type='pcicfg' desc='BIOS Control'>
    <field name='BIOSWE' bit='0' />
    <field name='BLE' bit='1' />
    ..
    <field name='SMM_BWP' bit='5' />
  </register>
  <register name='HSFS' type='mmio'>
    ..
    <field name='FLOCKDN' bit='15'/>
  </register>
  <register name='IA32_SMRR_PHYSMASK' type='msr'>
    <field name='Valid' bit='11'/>
  </register>

</registers>
```

Platform Configuration: “Controls”

- **Controls** are important hardware lock bits, hardware protection enables, etc.

```
<control name='FlashLock' register='HSFS' field='FLOCKDN'/>
```

- Modules can read the value of controls on any platform by the name regardless of where this control is (which register)

```
flock = chipsec.chipset.get_control(self.cs, 'FlashLockDown')
```

CHIPSEC Has Two Entry-Points

1. **chipsec_main.py** (module launcher)
   - Runs modules/tools automatically in a “security regression suite” mode
   - Runs only modules applicable to current platform

   ```
   chipsec_main.py [--type BIOS]
   ```

   - Or individually via “`--module`” command-line option

   ```
   chipsec_main.py --module common.bios_wp
   ```

2. **chipsec_util.py** (manual utilities)
   - Provides manual access to HW resources (io, mem, pci ..)
   - Individual utility commands are in `utilcmd/*_cmd.py`

   ```
   chipsec_util.py spi dump spi.bin
   ```

   (e.g. `spi` util command is implemented in `spi_cmd.py` file)

Useful Options

- `-a` (`--module_args`): Specifies arguments to each module individually
- `-n` (`--no_driver`): Tells CHIPSEC to not launch Windows kernel driver (in case module doesn't need it)
- `-x` (`--xml`):Outputs result in JUnit compatible XML form which may be useful to integrate in validation env.
- `-t` (`--module_type`): Run only modules of a specific type. Examples: `BIOS, SMM, SECUREBOOT, HWCONFIG`
  - New types may be defined in `chipsec/module_common.py`
- `-v` (`--verbose`): Logs all output from HAL components, helpers, dumps buffers, logs exception back-traces, etc.

The Meat: CHIPSEC Modules

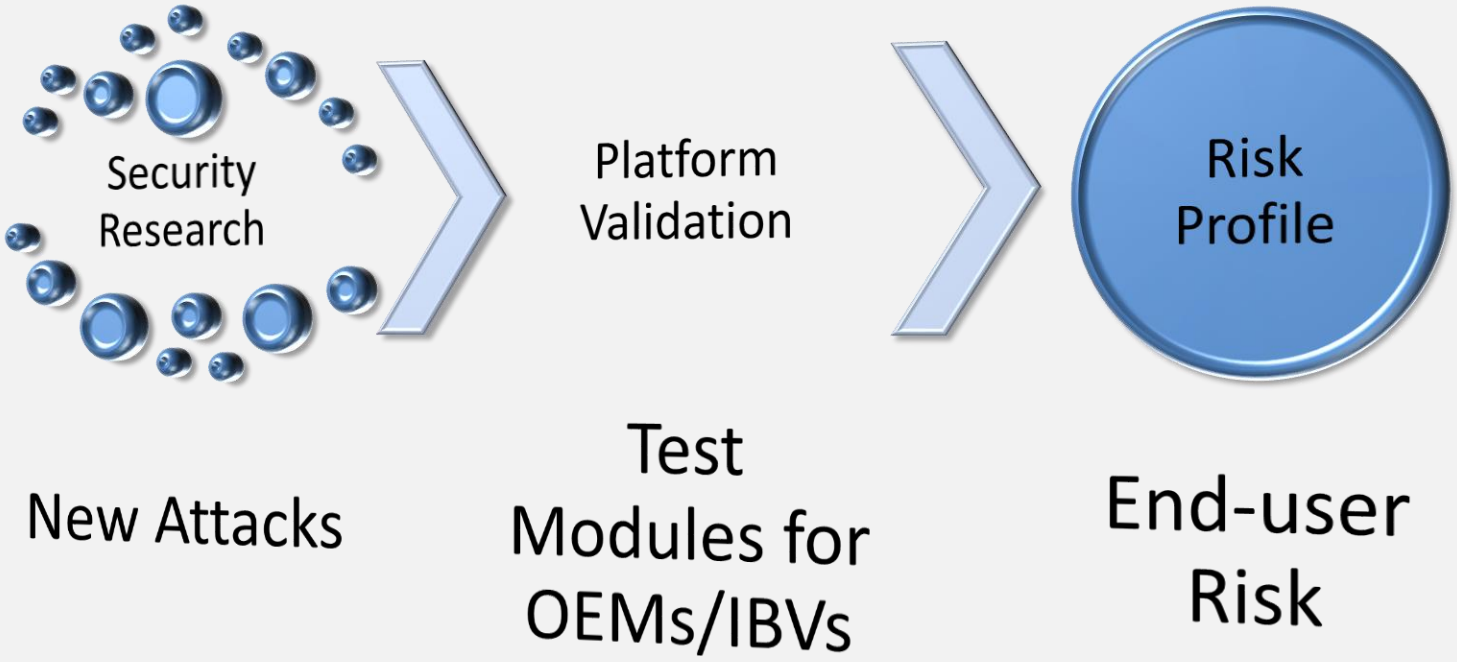
Modules encapsulate the main functionality of CHIPSEC:

1. Tests for known vulnerabilities in firmware
2. Tests for insufficient or incorrectly configured hardware protections
3. Hardware/firmware-level security tools
   - Fuzzing tools for firmware interfaces/formats
   - Manual security checkers (e.g. TE checker, DMA dumper)
   - Reside in `modules/tools` directory are not launched automatically (only through `-m` command-line option)
4. PoC exploit modules demonstrating vulnerabilities

The Meat: CHIPSEC Modules

- All modules reside in `chipsec/modules` directory
- Modules can be specific to one or more platforms or common for all supported platforms
  - Modules in `modules/<platform_code>` directory will only be executed on `<platform_code>` platform
  - Modules in `modules/common` directory will always be executed
- Modules can implement `is_supported` function which can further check for supported platforms, OS environments (legacy vs UEFI boot), etc.

Raising the Bar for Platform Security



Empowering End-Users to Make a Risk Decision

Source: https://cansecwest.com/slides/2014/Platform%20Firmware%20Security%20Assessment%20wCHIPSEC-csw14-final.pdf

Summary of Modules in CHIPSEC

Issue	CHIPSEC Module	References
SMRAM Locking	`common.smm`	CanSecWest 2006
BIOS Keyboard Buffer Sanitization	`common.bios_kbrd_buffer`	DEFCON 16
SMRR Configuration	`common.smrr`	ITL 2009, CanSecWest 2009
BIOS Protection	`common.bios_wp`	BlackHat USA 2009, CanSecWest 2013, Black Hat 2013, NoSuchCon 2013
SPI Controller Locking	`common.spi_lock`	Flashrom, Copernicus
BIOS Interface Locking	`common.bios_ts`	PoC 2007
Secure Boot variables with keys and configuration are protected	`common.secureboot.variables`	UEFI 2.4 Spec , All Your Boot Are Belong To Us (here & here)
Memory remapping attack	`remap`	Preventing and Detecting Xen Hypervisor Subversions
DMA attack against SMRAM	`smm_dma`	Programmed I/O accesses: a threat to VMM?, System Management Mode Design and Security Issues
SMI suppression attack	`common.bios_smi`	Setup for Failure: Defeating Secure Boot
Access permissions to SPI flash descriptor	`common.spi_desc`	Flashrom
Access permissions to UEFI variables defined in UEFI Spec	`common.uefi.access_uefispec`	UEFI 2.4 Spec
Module to detect PE/TE Header Confusion Vulnerability	`tools.secureboot.te`	All Your Boot Are Belong To Us
Module to detect SMI input pointer validation vulnerabilities	`tool.smm.smm_ptr`	CanSecWest 2015

3.3 Developing Modules in CHIPSEC

Module template

```
from chipsec.module_common import *
_MODULE_NAME = 'module_template'

class module_template (BaseModule):

    def __init__(self):
        BaseModule.__init__(self)

    def check_something( self ):
        self.logger.start_test( "Module Template" )
        self.logger.log_passed_check( "Test P          )
        return ModuleResult.PASSED

    def is_supported(self):
        return False

    # ------------------------------------------------------------------------
    # run( module_argv )
    # Required function: run here all tests from this module
    # ------------------------------------------------------------------------

    def run( self, module_argv  ):
        return self.check_something()
```

Inherits BaseModule template

Return result

FAILED
PASSED
WARNING
SKIPPED
DEPRECATED
ERROR

Check if this module can run on this platform/OS

Module starts here. Can pass arguments to each module

Example: common.spi_lock

```
from chipsec.module_common import *
TAGS = [MTAG_BIOS]

class spi_lock(BaseModule):

    def __init__(self):
        BaseModule.__init__(self)

    def is_supported(self):
        return (self.cs.get_chipset_id() in \
    [chipsec.chipset.CHIPSET_FAMILY_CORE,chipsec.chipset.CHIPSET_FAMILY_XEON])

    def check_spi_lock(self):
        self.logger.start_test( "SPI Flash Controller Configuration Lock" )

        spi_lock_res = ModuleResult.FAILED
        hsfs_reg = chipsec.chipset.read_register( self.cs, 'HSFS' )
        chipsec.chipset.print_register( self.cs, 'HSFS', hsfs_reg )
        flockdn = chipsec.chipset.get_register_field( self.cs, 'HSFS', hsfs_reg, 'FLOCKDN' )

        if 1 == flockdn:
            spi_lock_res = ModuleResult.PASSED
            self.logger.log_passed_check( "SPI Flash Controller configuration is locked" )
        else:
            self.logger.log_failed_check( "SPI Flash Controller configuration is not locked" )

        return spi_lock_res

    def run( self, module_argv ):
        return self.check_spi_lock()
```

Type of the check (e.g. BIOS security)

Checks SPI controller configuration is locked down

Logging

- Output module's result:

```
log_passed/failed/skipped/warn_check()
```

- Various output modes:

```
log(),error(),warning(),log_bad(),log_good(), log_important()
```

- Turning VERBOSE output mode. Verbose mode logs everything from HAL, OS helpers etc.

```
self.logger.VERBOSE = True
# chipsec_main.py -m common.spi_lock --verbose
```

- Turn on/off logging by HAL components

```
self.logger.HAL
```

- Utility logging (ON by default when CHIPSEC_UTIL is used)

```
self.logger.UTIL_TRACE
```

- Flushing log output to a file (what if a fuzzer crashes OS?)

```
self.logger.flush()
self.logger.set_always_flush( True )
```

3.4 Developing Fuzzers for the System Firmware

Passing arguments to CHIPSEC modules

More complex modules (e.g. tools, fuzzers, PoCs..) may define module specific command-line arguments to be passed by CHIPSEC via “-a” option:

```
# chipsec_main -m tools.fuzzer -a rnd,1000,0xDEADBEEF
```

```
def run( module_argv ):
    logger.start_test( "Some fuzzer" )

    if len(module_argv) > 2:
        _mode     = module_argv[0]
        _attempts = int(module_argv[1])
        _address  = int(module_argv[2],16)

    fuzz( _mode, _attempts, _address )

    return ModuleResult.PASSED
```

Training materials are available on Github

https://github.com/advanced-threat-research/firmware-security-training

Yuriy Bulygin	@c7zero
Alex Bazhaniuk	@ABazhaniuk
Andrew Furtak	@a_furtak
John Loucaides	@JohnLoucaides