Software

WRITING CHIPSEC MODULES & TOOLS

Module & Command Development

Erik Bjorge, Maggie Jauregui & Brian Richardson
Platform Armoring & Resiliency Team

OSFC 2018

Why are we here?

- Supporting CHIPSEC at Intel
- Help the CHIPSEC community to write more modules
- Improve the functionality of CHIPSEC

Agenda

- A Little History
- Architecture
- Modules (Tests & Tools)
- Utility Commands

CHIPSEC

CHIPSEC History

- CHIPSEC is a framework for analyzing the security of PC platforms including hardware, system firmware (BIOS/UEFI), and platform components.
- Originally developed by Yuriy Bulygin (@c7zero)
- First version of CHIPSEC was released in March 2014 at CanSecWest
- Currently used by firmware developers, system validation and system integrators

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

Running CHIPSEC

Boot to the USB drive

- Ubuntu 18.04 with CHIPSEC source
- Password: 0$fc2018

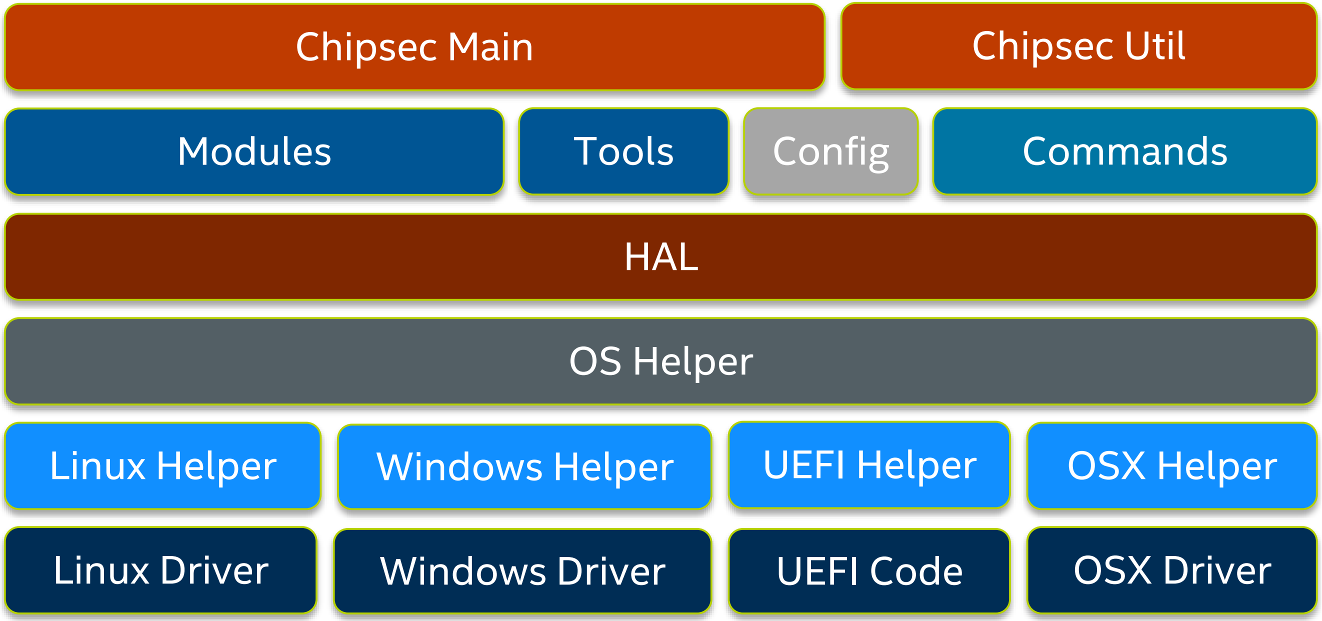
From a terminal:

```
cd ~/src/chipsec

python setup.py build_ext -i

sudo python chipsec_util.py platform

sudo python chipsec_main.py
```

Common Terms

- Device ID (DID)
- Hardware Abstraction Layer (HAL)
- Platform Controller Hub (PCH)
- Serial Peripheral Interface (SPI)
- System Management Mode (SMM)
- Unified Extensible Firmware Interface (UEFI)
- Vendor ID (VID)

CHIPSEC Architecture



CHIPSEC Architecture

Modules & Tools

- Implementation of tests or other functionality for `chipsec_main`

Configuration Files

- Provide a human readable abstraction for registers in the system

Commands

- Implement functionality of `chipsec_util`

HAL

- Useful abstractions for common tasks such as accessing the SPI

OS Helpers & Drivers

- Provides a translation layer to convert a common interface to OS specific driver calls

CHIPSEC_MAIN Program Flow

1. Load OS Specific Driver
2. Detect Platform
3. Load Modules
4. Load Configuration Files
5. Run Loaded Modules
6. Report Results

Platform Detection

- Uses PCI VID and DID to detect processor and PCH
  - Processor 0:0.0
  - PCH 0:31.0
- Chip information located in chipsec/chipset.py
  - Currently requires VID of 0x8086
  - DID is used as the lookup key
- Select a specific platform using the -p flag
- Ignore the platform specific registers using the -i flag

Configuration Files

- Broken into common and platform specific configuration files
- Used to define controls, registers and bit fields
- Common files always loaded first so the platform files can override values
- Correct platform configuration files loaded based off of platform detection

Configuration File Examples

```
<mmio>

    <bar name="SPIBAR" bus="0" dev="0x1F" fun="5" reg="0x10" width="4" mask="0xFFFFF000"
    size="0x1000" desc="SPI Controller Register Range" offset="0x0"/>

</mmio>

<registers>

    <register name="BC" type="pcicfg" bus="0" dev="0x1F" fun="5" offset="0xDC" size="4"
    desc="BIOS Control">

        <field name="BIOSWE" bit="0" size="1" desc="BIOS Write Enable" />

        …

        <field name="BILD" bit="7" size="1" desc="BIOS Interface Lock Down"/>

    </register>

</registers>

<controls>

    <control name="BiosInterfaceLockDown" register="BC" field="BILD" desc="BIOS Interface
    Lock-Down"/>

</controls>
```

Register Interfaces

- Used to access controls, registers and fields based on the human readable name
- Enables test code to be portable when registers move or are renamed
- Controls allow for mapping different register names to a common control name
- Interfaces exist for reading and writing as well as checking for existence

Register Interface Summary

Control Access:

- `is_control_defined, get_control, set_control`

Register Access:

- `is_register_defined, read_register, write_register, print_register`

Field Access:

- `register_has_field, read_register_field, write_register_field, get_register_field_mask, get_register_field, set_register_field`

Note: Only commonly used interfaces listed

Logging Interface

- CHIPSEC defines its own logging interface
  - Used for display to terminal
  - Used to write to different log file types
- Provides color text output to the console
  - Linux support without additional modules
  - Windows color console support requires additional python modules
- Should be used to display output instead of `print()`

Logging Interface Summary

`log`

- Logs the specific string same as a print

`log_*`

- Prepends formatted text to the provided string
- log_warning will prepend the string with "**`[!] WARNING:`**" in yellow

`log_*_check`

- Used to log the overall result for the module
- Always called once (and only once) in a module
- Also used to finalize XML log entry

HAL Overview

`cpu`

- Access to processor registers and special instructions like cupid

`mmio`

- Direct or register based access to MMIO regions

`pci`

- Access to PCI devices and Option ROM information

`spi`

- Simplifies accessing the SPI flash and enumerating different regions

`uefi`

- Access to UEFI functionality such as variables, system tables or compression

* Many more exist in the chipsec/hal directory

Return Values

PASSED - Test detected mitigation

FAILED - Test failed to detect mitigation

WARNING - Test results require manual investigation

INFORMATION - Test output is informational only

SKIPPED - Test not implemented for current platform (test not run)

NOTAPPLICABLE - Test does not apply to current platform (test not run)

ERROR - The test generated an exception

Modules (Tests & Tools)

- Test Modules
  - Verify a specific vulnerability has been mitigated
  - Do not modify the system configuration
  - Enumerated and run automatically by **`chipsec_main`**
- Tool Modules
  - Allowed to modify the state of the system
  - May be destructive to the system
  - Must be run manually via command line parameter
  - Stored in the chipsec/modules/tools directory
- All module classes are derived from **`BaseModule`**
- Only difference between tests and tools is where the file is stored

Module Interfaces

```
__init__(self)
```

- Initialize your modules class state if needed

```
is_supported(self)
```

- Determines if the module should be run on the current platform

```
run(self, module_argv)
```

- Entry point for the actual test or tool
- Modules can accept arguments
- Return value determines the exit state of the module
  - Pass, Failure, Warning, etc.

`is_supported` Guidance

Reduce maintenance…

- Check to see if registers are defined
- Check for PCI device types or classes
- Check CPUID or specific feature support
- Avoid checking for a specific platform if possible
  - Checking for a class of processor like all Atom processors is fine

```
def is_supported( self ):
    supported = self.cs.helper.EFI_supported()
    if not supported: self.logger.log_skipped_check( "OS does not support UEFI Runtime API" )
    return supported
```

`run` Guidance

- Call `self.logger.start_test()` early in execution
  - This will display the test header
- Try to map test code to a single vulnerability
  - May require multiple mitigations
  - Not always logical to do this
- Log intermediate results if required
- Log final result of module with `log_*_check`
  - Called once per execution of the module

Example Module

The goal is to generate a new informational module to gather useful data about the host processor and display it to the user.

- Processor brand string
- Family, model and stepping
- Microcode revision

Full source in chipsec/modules/common/cpu/cpu_info.py on USB drive

Initial Template

```
class cpu_info(BaseModule):
    def __init__(self):
        BaseModule.__init__(self)

    def is_supported(self):
        return True


    def run(self, module_argv):
        # Log the start of the test
        self.logger.start_test('Current Processor Information')

        return ModuleResult.INFORMATION
```

Collect & Display Brand String

```
# Get processor brand string
brand = ''
for eax_val in [0x80000002, 0x80000003, 0x80000004]:
  regs = self.cs.cpu.cpuid(eax_val, 0)
  for i in range(4):
    brand += struct.pack('<I', regs[i])
self.logger.log('[*] Processor: {}'.format(brand))
```

Collect & Display More Data

```
# Get microcode revision

microcode_rev =
self.cs.read_register_field('IA32_BIOS_SIGN_ID', 'Microcode')

self.logger.log('[*]            Microcode:
{:08X}'.format(microcode_rev))


self.logger.log_information_check('Current information
displayed')


return ModuleResult.INFORMATION
```

Module Output

```
[*] running module: chipsec.modules.common.cpu.cpu_info
[x][ =============================================================
[x][ Module: Current Processor Information
[x][ =============================================================
[*] Processor: Intel(R) Core(TM) i7-6770HQ CPU @ 2.60GHz
[*]            Family: 06 Model: 5E Stepping: 3
[*]            Microcode: 000000C2
[#] INFORMATION: Current information displayed
```

```
Command Line:
sudo python chipsec_main.py -m common.cpu.cpu_info
```

CHIPSEC Commands

- Run using **`chipsec_util`**
- Provide interactive access to system components from command line
  - Most support read/write access
  - Can be destructive
- Useful when doing research or other investigations
- Command classes are derived from **`BaseCommand`**
- Command line parameters available in **`self.argv`**
- Files in the chipsec/utilcmd directory

Command Interfaces

`requires_driver(self)`

- Used to determine if the OS specific driver is required to run the command

`run(self)`

- Main entry point to perform the command and display the results

`commands`

- Dictionary to map command names to class implementation

Command Example

```
class PlatformCommand(BaseCommand):

    def requires_driver(self):

        return True

    def run(self):

        try:
            print_supported_chipsets()
            self.logger.log("")
            self.cs.print_chipset()
            self.cs.print_pch()

        except UnknownChipsetError, msg:
            self.logger.error( msg )

commands = { 'platform': PlatformCommand }
```

Summary

Now that you have the basics, start writing new modules and commands

Submit pull requests and issues on GitHub

https://github.com/chipsec/chipsec

Contact the Intel CHIPSEC team

chipsec@intel.com

Legal Notice

No computer system can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel, the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

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

Software