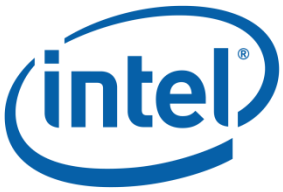

Host-based Firmware Analyzer User Guide

Contents

1 Introduction 1
1.1 Purpose of This Document 1
1.2 Terms 1
1.3 Related Information 2
1.4 Conventions Used in this Document 2
1.4.1 Typographic Conventions 2
2 How to Create New Case 4
2.1 Case Structure in Host-based Firmware Analyzer 4
2.1.1 Steps to Create Structure 4
2.1.2 Example 5
2.2 Add Case with AFL 5
2.2.1 Steps to Create Case 5
2.2.2 Example 6
2.3 Add Case with libFuzzer 7
2.3.1 Steps to Create Case 7
2.3.2 Example 7
2.4 Add Case with Peach 8
2.4.1 Steps to Create Case 8
2.4.2 Example 9
2.5 Add Case with KLEE 11
2.5.1 Steps to Create Case 11
2.5.2 Example 12
3 How to Check Result 13
3.1 Check AFL Result 13
3.2 Check libFuzzer Result 14
3.3 Check Peach Result 15
3.4 Check KLEE Result 17
4 How to Improve Code Coverage 19
4.1 Improve Code Coverage 19
5 How to Add Error Injection 20
5.1 Inject Error Path 20
5.2 Example 21
6 How to Select Test Method 22
6.1 Select Test Method 22

FIGURES

Figure 1. Structure of PartitionDxe 5
Figure 2. AFL Status 13
Figure 3. Debug Report 14
Figure 4. Code Coverage Report 14
Figure 5. libFuzzer Output 15
Figure 6. Peach is Running 16
Figure 7. Peach with an Error 17
Figure 8. KLEE is running 18
Figure 9. ".ktest" Files in "klee-out-x" 18
Figure 10. ".seed" Files 18

Revision History

Revision Number	Description	Revision Date

1 Introduction

1.1 Purpose of This Document

This document provides the methods of adding new cases into Host-based Firmware Analyzer. And this document also explains how to check test results, debug cases and add Error Injection into cases. At the end of this document, some advice about how to improve code coverage and how to choose test methods is given. Readers can follow similar steps to enable their own cases into Host-based Firmware Analyzer.

1.2 Terms

The following terms are used throughout this document to describe varying aspects of input localization:

OS

Operation system.

Fuzzer

Tools that can do fuzzing test.

AFL

American fuzzy lop. It is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.

libFuzzer

In-process, coverage-guided, evolutionary fuzzing engine.

Peach

Peach accomplishes fuzzing test by separating out modeling of the data and state systems being fuzzed and the actual fuzzing engine.

KLEE

KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure, and available under the UIUC open source license.

Sanitizer

AddressSanitizer (or ASan) is an open source programming tool by Google that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer (use-after-free).

Test seeds

The input files which can walk through almost all test logic and is used to be an initial template for fuzzer.

Test cases

The test cases under HBFA/UefiHostFuzzTestCasePkg/TestCase.

Test binary

The binary file built from test case and can be run in OS.

1.3 Related Information

The following publications and sources of information may be useful to you or referred to by this document:

- *Wikipedia*, a free-access, free content Internet encyclopedia, supported and hosted by the non-profit Wikimedia Foundation. Those who can access the site and follow its rules can edit most of its articles, https://www.wikipedia.org/.
- *American fuzzy lop (afl),* the official website of AFL, from where you can find all official documents about AFL, http://lcamtuf.coredump.cx/afl/.
- *libFuzzer,* the official page of libFuzzer, https://llvm.org/docs/LibFuzzer.html.
- *Peach Fuzzer,* provides tutorials information on how to get started fuzzing with Peach 3, http://community.peachfuzzer.com/v3/PeachQuickStart.html.
- *KLEE,* documentation for KLEE master branch, http://klee.github.io/.

1.4 Conventions Used in this Document

This document uses typographic and illustrative conventions described below.

1.4.1 Typographic Conventions

Typographic Convention	Typographic Convention Description
Plain Text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain Text (blue)	Any plain text that is underlined and in blue indicates an active link to the cross-reference.
Bold	In text, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.

Typographic Convention	Typographic Convention Description
BOLD Monospace	Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
Bold Monospace	Words in a **Bold Monospace** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink.
$(VAR)	This symbol VAR defined by the utility or input files.
Italic Monospace	In code or in text, words in *Italic Monospace* indicate placeholder names for variable information that must be supplied (i.e., arguments).

Note: *Due to management and files size considerations, only the first occurrence of the reference on each is an active link. Subsequent references on the same page will not be actively linked to the definition and will use the standard, none-underlined* ***BOLD Monospace*** *typeface. Find the first instance of the name (in the underlined* ***BOLD Monospace*** *typeface) on the page and click on the word to jump to the function or type definition.*

2
How to Create New Case

This chapter explains how to create new case into Host-based Firmware Analyzer with some meaningful examples and assumes that user have knowledge of EDKII and familiar with the source code you want to test.

2.1 Case Structure in Host-based Firmware Analyzer

2.1.1 Steps to Create Structure

Step 1. Assume that you want to construct a case into Host-based Firmware Analyzer, you should create a folder for your case under HBFA/UefiHostFuzzTestCasePkg/TestCase:

```
HBFA
└──────UefiHostFuzzTestCasePkg
    └──────TestCase
        ...
            └──────TestModuleFolder
                └──────TestXXX.c
                └──────TestXXX.inf
```

Step 2. And the files need to be added into this folder are:

1) You can create your own test logic in TestXXX.c:
`UefiHostFuzzTestCasePkg/TestCase/.../TestModuleFolder/TestXXX.c`
NOTE: If the target source file is Library/PPI/Protocol, you need to include the header file to call the target function.

2) INF file for your test module:
`UefiHostFuzzTestCasePkg/TestCase/.../TestModuleFolder/TestXXX.inf`
NOTE: `ToolChainHarnessLib` should be added into `[LibraryClasses]` section.

Step 3. Append your test module to `[Components]` section in UefiHostFuzzTestCasePkg.dsc:
`UefiHostFuzzTestCasePkg/TestCase/.../TestModuleFolder/TestXXX.inf`
NOTE: If a target function is not in any header file, you need to add the target INF file to `<LibraryClasses>` of the `TestXXX.inf`. For example:

```
UefiHostFuzzTestCasePkg/TestCase/MdeModulePkg/Library/DxeCapsuleLibFmp/Te
stDxeCapsuleLibFmp.inf {
    <LibraryClasses>
     NULL|MdeModulePkg/Library/DxeCapsuleLibFmp/DxeCapsuleLib.inf
}
```

Step 4. Then you can build your test module, and try to run it in OS environment.

2.1.2 Example

Take PartitionDxe for example in HBFA/UefiHostFuzzTestCasePkg/TestCase/MdeModulePkg/Universal/Disk/PartitionDxe/, the structure is shown as following:

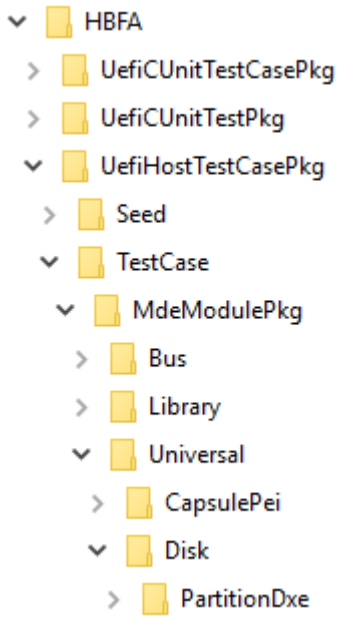

Figure 1. Structure of PartitionDxe

And the files under PartitionDxe are:

```
// Where the test logic in
UefiHostFuzzTestCasePkg/TestCase/.../PartitionDxe/TestPartition.c
// INF file
UefiHostFuzzTestCasePkg/TestCase/.../PartitionDxe/TestPartition.inf
```

2.2 Add Case with AFL

2.2.1 Steps to Create Case

Step 1. You need to realize `FixBuffer()`, `GetMaxBufferSize()` and `RunTestHarness()` and enable your test logic to figure out a way to call those API that has external input with TestBuffer in TestXXX.c:

```
/* The FixBuffer function is aimed to fix some bits in the TestBuffer so
that the TestBuffer can pass basic check and touch deeper paths. It's
optional. */
VOID
FixBuffer (
  UINT8                     *TestBuffer
  )
{
}
```

```
UINTN
EFIAPI
GetMaxBufferSize (
  VOID
  )
{
  return TOTAL_SIZE;
}

VOID
EFIAPI
RunTestHarness(
  IN VOID  *TestBuffer,
  IN UINTN TestBufferSize
  )
{
  FixBuffer(TestBuffer);

  // test logic that use TestBuffer as an input to try to call tested
API.
  TestLogicFunc(TestBuffer, TestBufferSize);
}
```

Step 2. Create some test seeds that can walk through your test logic and trigger enough code paths. You can create your test seeds based on your knowledge about tested code, or you can try to enable KLEE (detail is in 2.5) to generate some test seeds for you.

Step 3. Then you can build test module and run test case with AFL (in Linux) or VS2015x86 (in Windows).

2.2.2 Example

We have an example of AFL in HBFA/UefiHostFuzzTestCasePkg/TestCase/MdeModulePkg/Universal/Disk/PartitionDxe/TestPartition.c:

```
#define TOTAL_SIZE    (512 * 1024)
#define BLOCK_SIZE    (512)
#define IO_ALIGN      (1)

VOID
FixBuffer (
  UINT8                   *TestBuffer
  )
{
}
UINTN
EFIAPI
GetMaxBufferSize (
  VOID
  )
{
  return TOTAL_SIZE;
}

VOID
```

```
EFIAPI
RunTestHarness(
  IN VOID  *TestBuffer,
  IN UINTN TestBufferSize
  )
{
  EFI_LBA                StartingLBA;
  EFI_LBA                EndingLBA;
  EFI_BLOCK_IO_PROTOCOL  *BlockIo;
  EFI_DISK_IO_PROTOCOL   *DiskIo;

  FixBuffer (TestBuffer);

  DiskStubInitialize (TestBuffer, TestBufferSize, BLOCK_SIZE, IO_ALIGN,
&BlockIo, &DiskIo);

  // fuzz function:
  // buffer overflow, crash will be detected at place.
  // only care about security, not for function bug.
  //
  // try to separate EFI lib, use stdlib function.
  // no asm code.
  FindUdfFileSystem (
            BlockIo,
            DiskIo,
            &StartingLBA,
            &EndingLBA
            );

  DiskStubDestory();
}
```

And you can also find some examples of PartitonDxe's seeds in HBFA/UefiHostFuzzTestCasePkg/Seed/UDF/Raw/Partition.

2.3 Add Case with libFuzzer

2.3.1 Steps to Create Case

Step 1. When we test with libFuzzer, the test logic use the same code as AFL in TestXXX.c, if you haven't added case with AFL, you can refer to 2.2 and write test logich, or you have already written test logic, just skip this step.

Step 2. Create some test seeds that can walk through your test logic and trigger enough code paths. You can create your test seeds based on your knowledge about tested code, or you can try to enable KLEE (detail is in 2.5) to generate some test seeds for you.

Step 3. Then you can build test module and run test case with LIBFUZZER (in Linux) or LIBFUZZERWIN (in Windows).

2.3.2 Example

Same as 2.2.2.

2.4 Add Case with Peach

2.4.1 Steps to Create Case

Step 1. When we test with peach, the test logic use the same code as AFL in TestXXX.c, if you haven't added case with AFL, you can refer to 2.2 and write test logic for Peach, or you have already written test logic for AFL, just skip this step.

Step 2. Different from above two fuzzing tools, peach generate fuzzing inputs by PIT file which define the DataModel and other requirements of inputs, please read http://community.peachfuzzer.com/v3/PeachQuickStart.html and learn how to write a PIT file, and create PIT file for your test module. Here are some points you need to pay attention to when create PIT file:

1) PIT file needs to put under PeachDataModel folder in your test folder, and PIT file should be named the same as INF file:

```
TestModuleFolder
└──────TestedSrcCode.c
└──────TestXXX.c
└──────TestXXX.inf
└──────PeachDataModule
    └──────TestXXX.xml
```

2) Important parts in PIT file:

a) DataModel:

```
  <DataModel name="VARIABLE_STORE_HEADER">
    <Blob name="Signature" length="16"
value="782cf3aa7b949a43a1802e144ec37792" valueType="hex"
mutable="false"/>
    <Number name="Size" size="32" value="00001000"
valueType="hex"/>
    ......
  </DataModel>
```

b) Executable and Arguments

```
  <!-- Setup windows agent that will monitor for faults -->
  <Agent name="WinAgent">
    <Monitor class="WindowsDebugger">
      <Param name="CommandLine" value="TestModule fuzzfile.bin"/>
    </Monitor>

  </Agent>

  <!-- Setup linux agent that will monitor for faults -->
  <Agent name="LinuxAgent">
    <Monitor class="LinuxDebugger">
      <Param name="Executable" value="TestModule" />
      <Param name="Arguments" value="fuzzfile.bin" />
    </Monitor>

  </Agent>

  <Test name="Default">
    <Agent ref="WinAgent" platform='windows'/>
```

```
    <Agent ref="LinuxAgent" platform="linux"/>
    <StateModel ref="State"/>

    <!-- Configure our publisher with correct filename to write
  too -->
    <Publisher class="File">
      <Param name="FileName" value="fuzzfile.bin" />
    </Publisher>

    <!-- Configure a logger to store collected information -->
    <Logger class="Filesystem">
      <Param name="Path" value="logtest" />
    </Logger>
  </Test>
```

Step 3. Then you can build test module and run test case with Peach.

2.4.2 Example

For test logic, you can refer to the example in 2.2.2, and for PIT file, we have an example in HBFA/UefiHostFuzzTestCasePkg/TestCase/MdeModulePkg/Bus/Usb/UsbBusDxe/PeachDataModule/TestUsb.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://peachfuzzer.com/2012/Peach ../peach.xsd">

  <!-- Define our file format DDL -->

  <DataModel name="USB_CONFIG_DESCRIPTOR">
    <Number name="Length" size="8" value="09" valueType="hex"/>
    <Number name="DescriptorType" size="8" value="02" valueType="hex"
mutable="false"/>
    <Number name="TotalLength" size="16" value="0022" valueType="hex"/>
    <Number name="NumInterfaces" size="8" value="01" valueType="hex"
mutable="false"/>
    <Number name="ConfigurationValue" size="8" value="01" valueType="hex"
mutable="false"/>
    <Number name="Configuration" size="8" value="00" valueType="hex"
mutable="false"/>
    <Number name="Attributes" size="8" value="A0" valueType="hex"
mutable="false"/>
    <Number name="MaxPower" size="8" value="32" valueType="hex"
mutable="false"/>
  </DataModel>

  <DataModel name="USB_INTERFACE_DESCRIPTOR">
    <Number name="Length" size="8" value="09" valueType="hex"/>
    <Number name="DescriptorType" size="8" value="04" valueType="hex"
mutable="false"/>
    <Number name="InterfaceNumber" size="8" value="00" valueType="hex"
mutable="false"/>
    <Number name="AlternateSetting" size="8" value="00" valueType="hex"
mutable="false"/>
    <Number name="NumEndpoints" size="8" value="01" valueType="hex"
mutable="false"/>
```

```
    <Number name="InterfaceClass" size="8" value="03" valueType="hex"
mutable="false"/>
    <Number name="InterfaceSubClass" size="8" value="01" valueType="hex"
mutable="false"/>
    <Number name="InterfaceProtocol" size="8" value="01" valueType="hex"
mutable="false"/>
    <Number name="Interface" size="8" value="00" valueType="hex"
mutable="false"/>
  </DataModel>

  <DataModel name="EFI_USB_HID_CLASS_DESCRIPTOR">
    <Number name="DescriptorType" size="8" value="22" valueType="hex"
mutable="false"/>
    <Number name="DescriptorLength" size="16" value="0041"
valueType="hex"/>
  </DataModel>

  <DataModel name="EFI_USB_HID_DESCRIPTOR">
    <Number name="Length" size="8" value="09" valueType="hex"/>
    <Number name="DescriptorType" size="8" value="21" valueType="hex"
mutable="false"/>
    <Number name="BcdHID" size="16" value="0110" valueType="hex"
mutable="false"/>
    <Number name="CountryCode" size="8" value="00" valueType="hex"
mutable="false"/>
    <Number name="NumDescriptors" size="8" value="01" valueType="hex"/>
    <Block ref="EFI_USB_HID_CLASS_DESCRIPTOR"/>
  </DataModel>

  <DataModel name="USB_ENDPOINT_DESCRIPTOR">
    <Number name="Length" size="8" value="07" valueType="hex"/>
    <Number name="DescriptorType" size="8" value="05" valueType="hex"
mutable="false"/>
    <Number name="EndpointAddress" size="8" value="81" valueType="hex"
mutable="false"/>
    <Number name="Attributes" size="8" value="03" valueType="hex"
mutable="false"/>
    <Number name="MaxPacketSize" size="16" value="0008" valueType="hex"
mutable="false"/>
    <Number name="Interval" size="8" value="0A" valueType="hex"
mutable="false"/>
  </DataModel>

  <DataModel name="FULL_USB_CONFIG_DESCRIPTOR">
    <Block ref="USB_CONFIG_DESCRIPTOR"/>
    <Block ref="USB_INTERFACE_DESCRIPTOR"/>
    <Block ref="EFI_USB_HID_DESCRIPTOR"/>
    <Block ref="USB_ENDPOINT_DESCRIPTOR"/>
  </DataModel>

  <!-- Define a simple state machine that will write the file and
    then launch a program using the FileWriter and DebuggerLaucher
publishers -->
  <StateModel name="State" initialState="Initial">
    <State name="Initial">

      <!-- Write out contents of file.  The publisher attribute matches
      the name we provide for the publisher in the Test section. -->
      <Action type="output">
```

```
        <DataModel ref="FULL_USB_CONFIG_DESCRIPTOR" />
      </Action>

      <!-- Close file -->
      <Action type="close" />

      <!-- Launch the file consumer -->
      <Action type="call" method="ScoobySnacks" publisher="Peach.Agent"/>

    </State>
  </StateModel>

  <!-- Setup windows agent that will monitor for faults -->
  <Agent name="WinAgent">
    <Monitor class="WindowsDebugger">
      <Param name="CommandLine" value="TestModule fuzzfile.bin"/>
    </Monitor>

  </Agent>

  <!-- Setup linux agent that will monitor for faults -->
  <Agent name="LinuxAgent">
    <Monitor class="LinuxDebugger">
      <Param name="Executable" value="TestModule" />
      <Param name="Arguments" value="fuzzfile.bin" />
    </Monitor>

  </Agent>

  <Test name="Default">
    <Agent ref="WinAgent" platform='windows'/>
    <Agent ref="LinuxAgent" platform="linux"/>
    <StateModel ref="State"/>

    <!-- Configure our publisher with correct filename to write too -->
    <Publisher class="File">
      <Param name="FileName" value="fuzzfile.bin" />
    </Publisher>

    <!-- Configure a logger to store collected information -->
    <Logger class="Filesystem">
      <Param name="Path" value="logtest" />
    </Logger>
  </Test>

</Peach>
<!-- end -->
```

2.5 Add Case with KLEE

2.5.1 Steps to Create Case

Step.1 KLEE is a tool to analyze code and generate test seed. When build with KLEE, KLEE will help to generate seeds that suit for your code. The test logic that KLEE run is the same as the logic in 2.2.

Step.2 Then you can build test module and generate test seeds with KLEE (Linux only).

2.5.2 Example

Same as 2.2.2.

3
How to Check Result

This chapter assumes that you have already know how to build and run test module and introduces how to check results after running test module.

3.1 Check AFL Result

1) After starting to run test module with AFL, you will get AFL status screen like below:

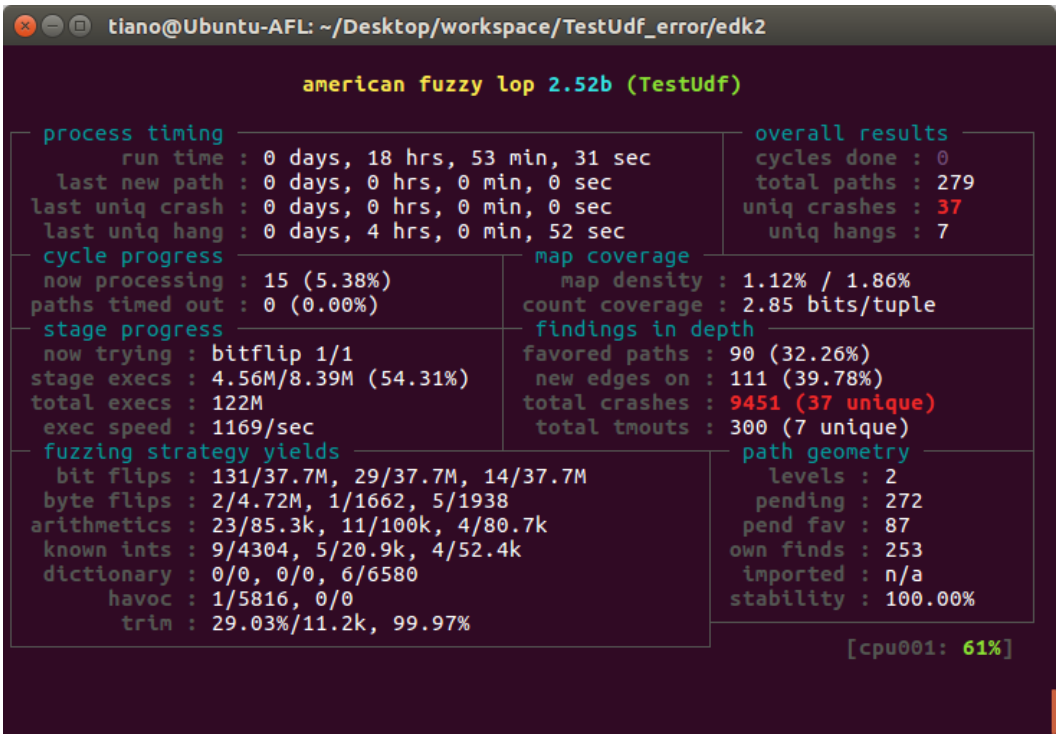

Figure 2. AFL Status

In this window, you can get summary information of the test status, you can get meaning of each item by referring to https://github.com/nccgroup/TriforceAFL/blob/master/docs/status_screen.txt.

2) And if you find there are crashes or hangs detected when running AFL, you can get debug report for every crash and every hang by the commands that generate test report, the report is shown as below:

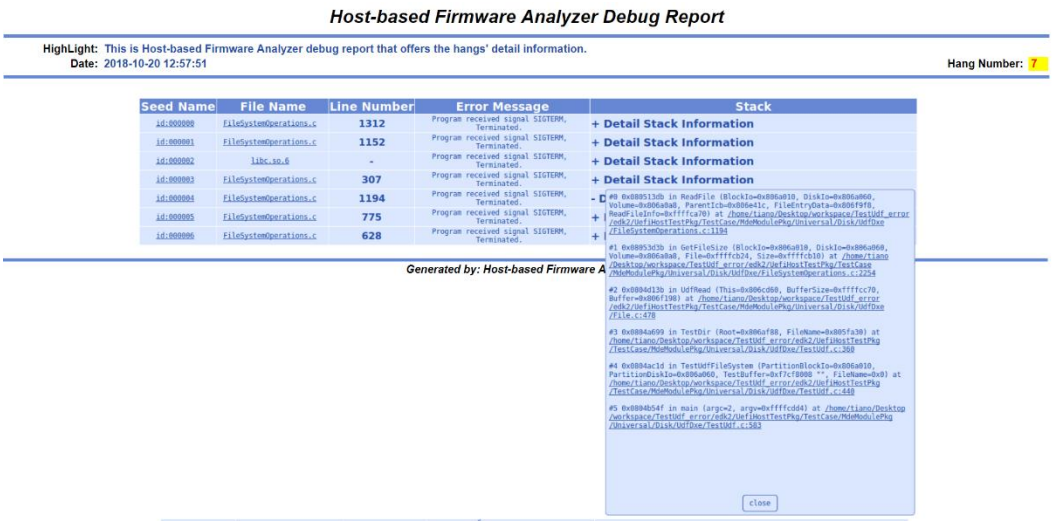

Figure 3. Debug Report

And you also can get into the folder to see the input binary which triggers this issue by pressing seed name.

3) If you want to see how many lines and functions of code are covered, you can re-run test seeds under "queue" in output folder with binary build with GCC5, and use code coverage tool to catch the test code coverage, the code coverage report is like below:

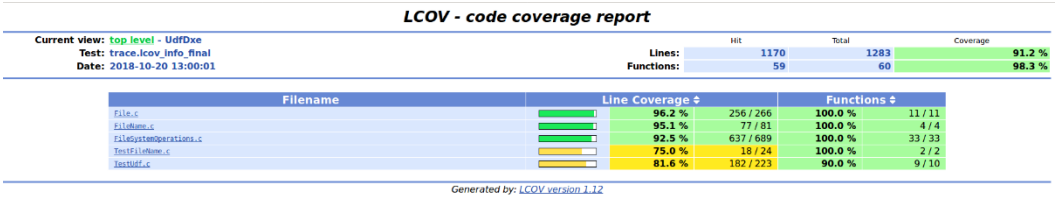

Figure 4. Code Coverage Report

3.2 Check libFuzzer Result

When running libFuzzer, you will see the running status in terminal like below:

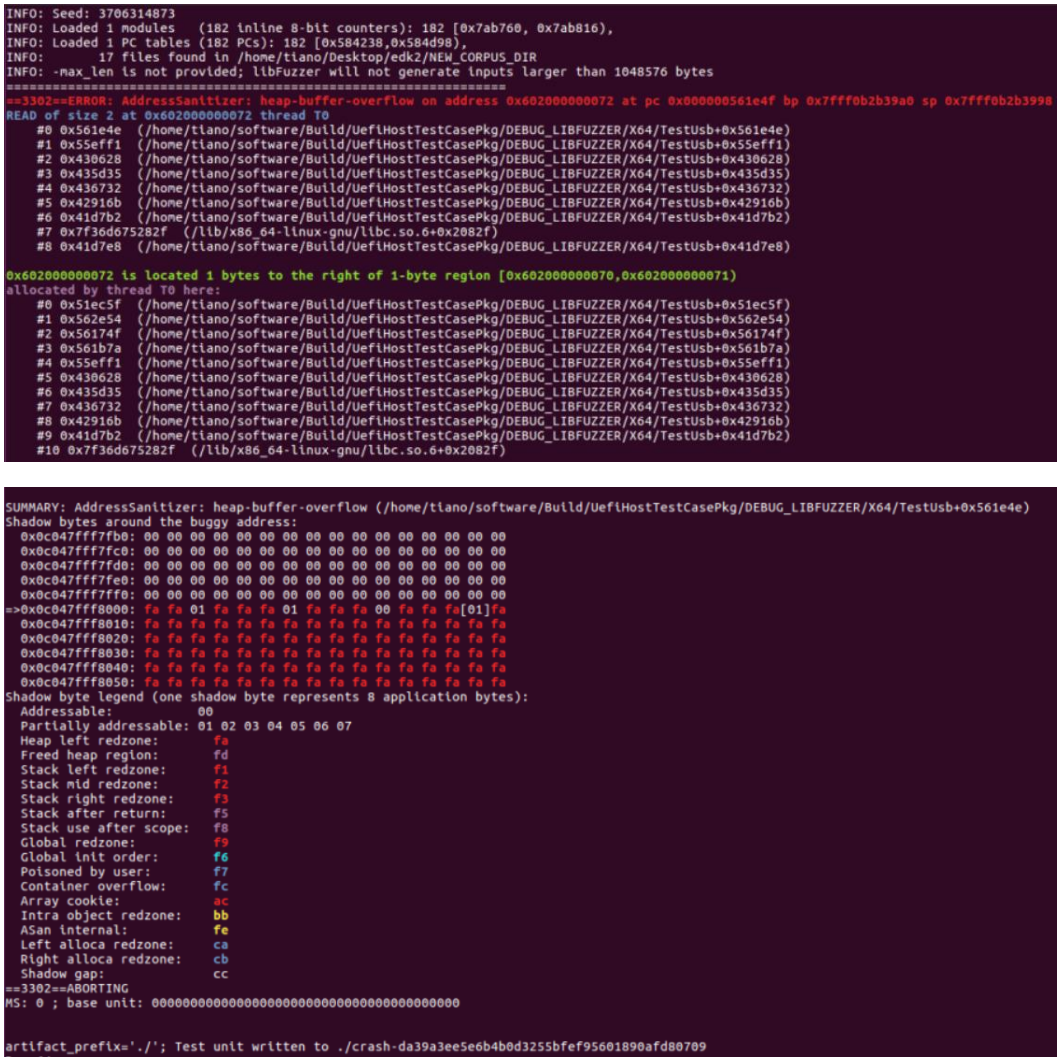

Figure 5. libFuzzer Output

From the output of this example, you can get below information:

1) The test is initialized by seeds under folder /home/tiano/Desktop/edk2/NEW_CORPUS_DIR

2) You can get same result by rerunning –seed=3706314873.

3) And Sanitizer catch an error of buffer overflow and output stack information.

4) Before aborting the process, the crash seed has been saved.

3.3 Check Peach Result

When running Peach, you will see the running status in terminal like below:

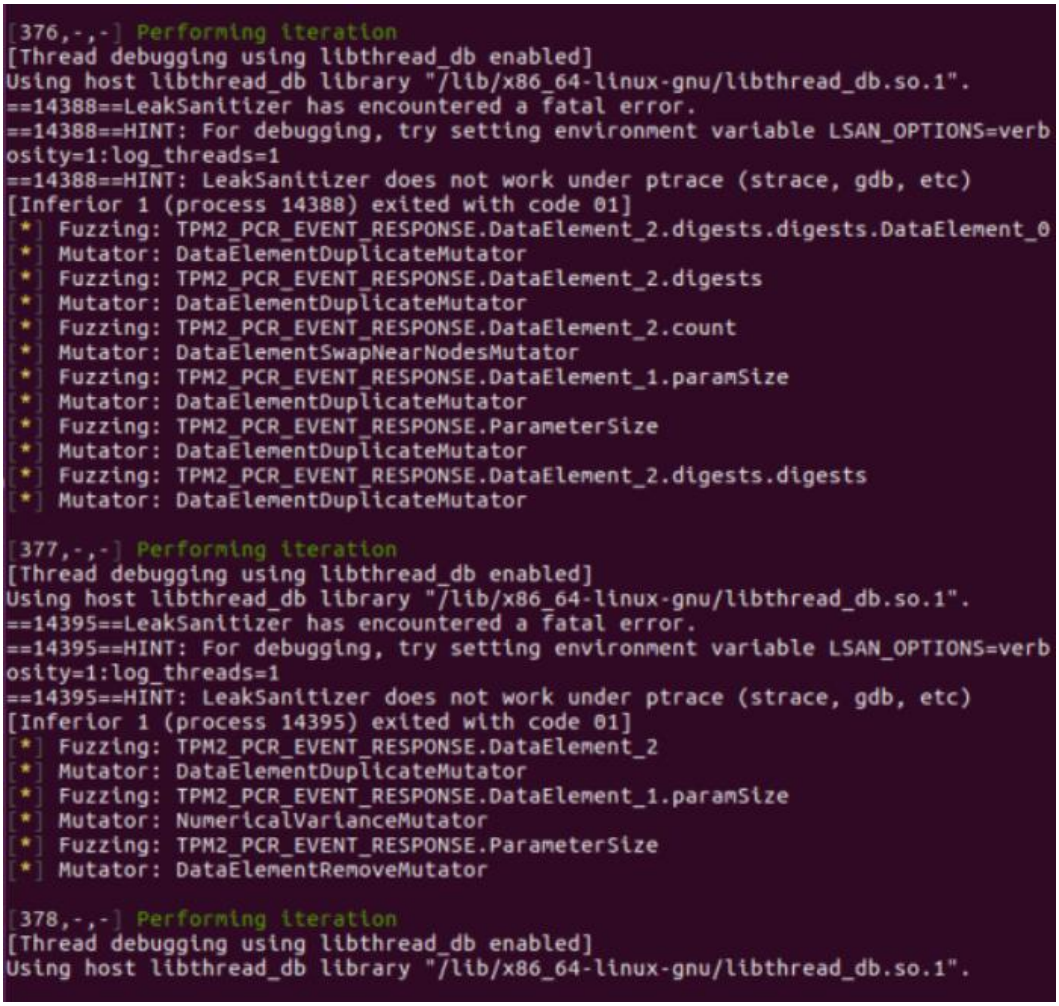

Figure 6. Peach is Running

And Sanitizer will also catch memory corruption bugs and output stack information like below.

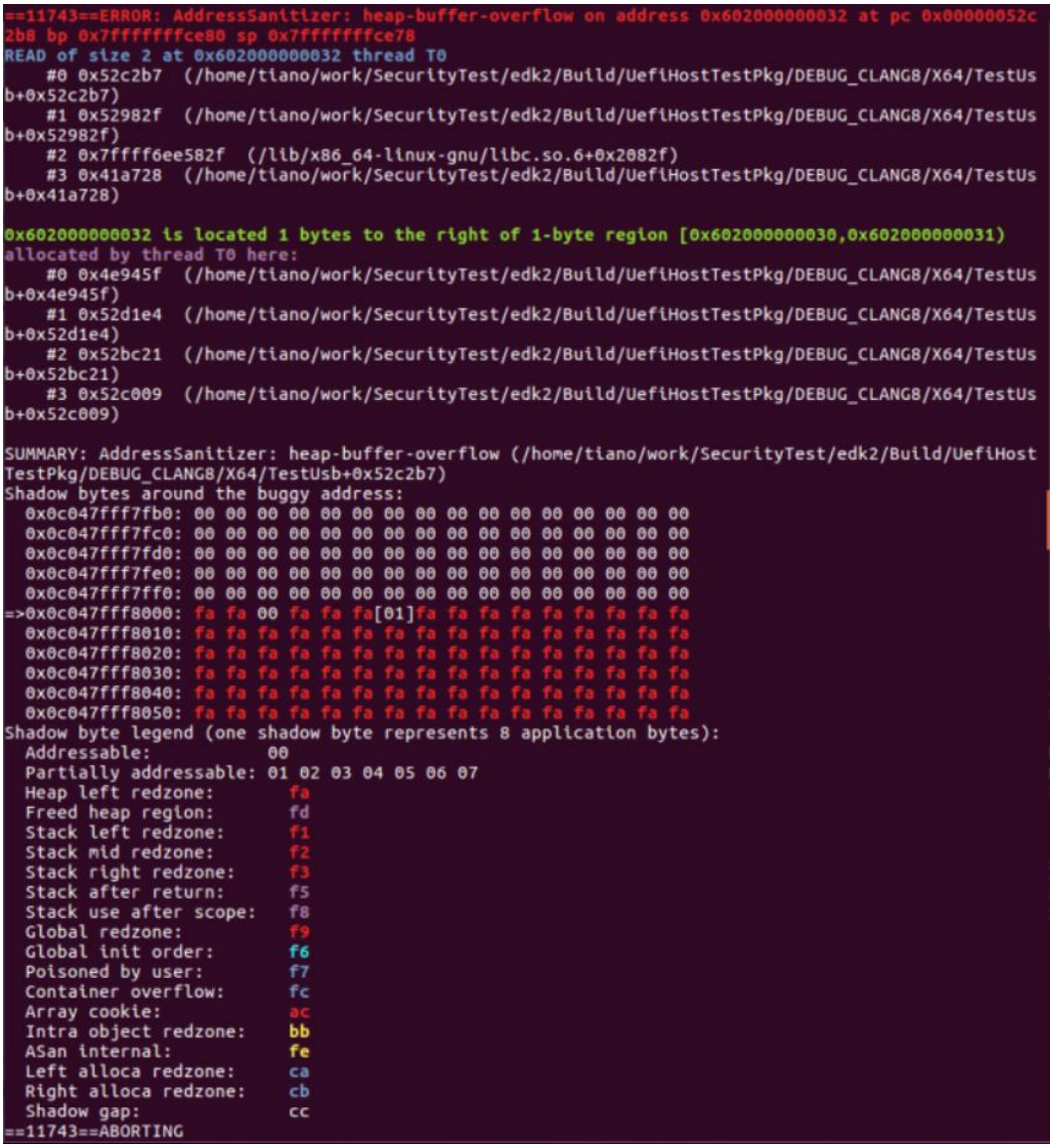

Figure 7. Peach with an Error

3.4 Check KLEE Result

You can see the below output when KLEE is running.

```
KLEE: output directory is "/home/tiano/Desktop/TestNewVer/edk2/Build/UefiHostTestCasePkg/R
ELEASE_KLEE/IA32/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignmen
t of 8.
```

Figure 8. KLEE is running

You can also get the seeds generated by KLEE from folder "klee-out-x" of which the path is also shown in above picture, all the seeds are named end with ".ktest":

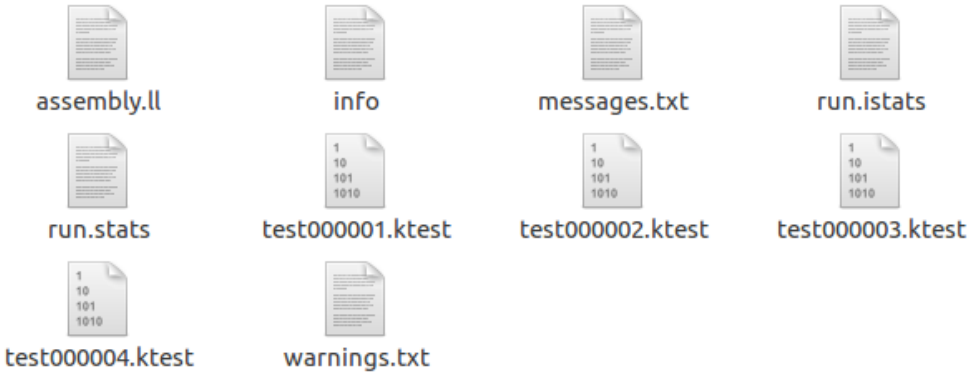

Figure 9. ".ktest" Files in "klee-out-x"

As ".ktest" file has KLEE header ahead of binary, you can use transfer-ktest-to-seed-tool.py to remove KLEE header, and get pure seed named end with ".seed" classified by buffer name you defined in your test logic.

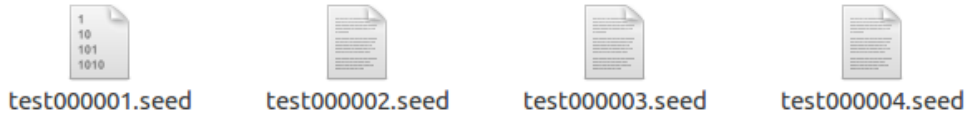

Figure 10. ".seed" Files

You can also check the quality of seeds by testing the code coverage after running test binary with them. If code coverage is too low, you can refer to chapter 4.1 and try to improve code coverage of seeds.

4
How to Improve Code Coverage

If you find the code coverage of your test module cannot meet your requirement, this chapter may give you some advice to improve it.

4.1 Improve Code Coverage

Actually, the final code coverage depends on the test seeds you input at first, to improve the quality of input seeds, you could try the following thing:

1) If you generate seeds by KLEE, you should know that KLEE will overburden when you use it to generate seeds suit for very complex test logic. You can get which path is not covered from code coverage report, and analyze why this happens, and try to adjust your test logic:

> a) Simple the test logic and add more test case. Cut down the complexity that KLEE calculate, and KLEE will give you a good feedback.
>
> b) Reduce the buffer size that KLEE need to generate. If these is a path cannot be covered by KLEE seeds, you can give a seed that can get into this path, and let KLEE generate the remained part of buffer.

2) If the code coverage has been already very high, but you still want to improve it to even higher, in this situation, it is not a good choose to rely on tools to improve code coverage, you know that fuzzers are not such smart to know where need to be fuzzed to covered these parts, and you don't know how much time fuzzers need to fuzz correct parts, so the efficiency way is creating test seeds to cover un-covered part yourself.

3) If there are still some error status can't be covered in your test, Error Injection probably a good way for you to have a try, please refer to Chapter 6.1.

5
How to Add Error Injection

This chapter explains the steps to add Error Injection into your module.

5.1 Inject Error Path

Step.1 Create the InstrunmentHookLibTestXXX under your test module folder:

```
TestModuleFolder
     └──────TestedSrcCode.c
     └──────TestXXX.c
     └──────TestXXX.inf
     └──────InstrunmentHookLibTestXXX
```

And the files need to be added into this folder are:

1) The source file that define which function is hooked to return error status or be set specific input:

```
.../TestModuleFolder/InstrunmentHookLibTestXXX/InstrunmentHookLibTestXXX.c
```

2) INF file for InstrunmentHookLib module:

```
.../TestModuleFolder/InstrunmentHookLibTestXXX/InstrunmentHookLibTestXXX.inf
```

Step.2 Append your InstrunmentHookLib module to `[Components]` section in UefiHostFuzzTestCasePkg.dsc under your test module:

```
UefiHostFuzzTestCasePkg/TestCase/.../TestModuleFolder/TestXXX.inf {
!if $(TEST_WITH_INSTRUMENT)
    <BuildOptions>
      MSFT:  *_*_*_CC_FLAGS = "-DTEST_WITH_INSTRUMENT=TRUE"
      GCC:*_*_*_CC_FLAGS = "-DTEST_WITH_INSTRUMENT=TRUE"
    <LibraryClasses>

InstrumentHookLib|UefiHostFuzzTestCasePkg/TestCase/.../TestModuleFolder/InstrumentHookLibTestXXX/InstrumentHookLibTestXXX.inf
!endif
  }
```

Step.3 Enable below `<BuildOptions>` for the code you want to hock.

```
.../XXX.inf {
    <BuildOptions>
      MSFT:  *_*_*_CC_FLAGS = /Gh /GH /Od /GL-
      GCC:*_*_*_CC_FLAGS = -O0 -finstrument-functions
  }
```

Step.4 Write Error Injection Profile:

1) Create ".ini" file like below: (test.ini)

```
########################
```

```
# CallErrorCount=N means which call returns error. N means the Nth call
returns error.
#   N start from 1. CallErrorCount = 0 means disable.
# ReturnValue=X means when error happens, which value is returned.
########################

[AllocateZeroPool]
  CallErrorCount = 1
  ReturnValue = 0

[ReadBlocks]
  CallErrorCount = 1
  ReturnValue = EFI_DEVICE_ERROR

[ReadDisk]
  CallErrorCount = 1
  ReturnValue = EFI_DEVICE_ERROR
########################
```

2) Run test binary as format:

```
<TestBinary> <TestSeed> <TestIni>
```

5.2 Example

We have an example of Error Injection in HBFA/UefiHostFuzzTestCasePkg/TestCase/MdeModulePkg/Universal/Disk/PartitionDxe/InstrumentHookLibTestPartition:

1) There are three function in this example, we use Error Injection to return error status:

```
GLOBAL_REMOVE_IF_UNREFERENCED FUNC_HOOK mFuncHook[] = {
  {"AllocateZeroPool", (UINTN)AllocateZeroPool, (UINTN)CommonEnter,
(UINTN)CommonExit},
  {"ReadBlocks",       (UINTN)ReadBlocks,       (UINTN)CommonEnter,
(UINTN)CommonExit},
  {"ReadDisk",         (UINTN)ReadDisk,         (UINTN)CommonEnter,
(UINTN)CommonExit},
};
```

2) And we write script to automatic create ".ini" file, you can refer to CreateErrorInjectionProfile.py.

6
How to Select Test Method

This chapter gives you some advice to select test method.

6.1 Select Test Method

We integrate several tools in Host-based Firmware Analyzer to make your test easier and faster, some tools are used to do fuzzing test, some tools are used to generate test seeds, and some tools are used to catch abnormal status or check whether specific status can be returned. If you wonder how to choose the method, the first thing you need to know is what you want to use the method for:

1) Do fuzzing test:

AFL, libFuzzer and Peach are all fuzzers to do fuzzing test. Here is a matrix for comparison of these three methods, you can choose fuzzing tool according this matrix:

Fuzzing Methods	Pros	Cons
AFL	a) Use some novel algorithms such as compile-time instrumentation and genetic algorithms to discover test cases. b) The efficiency of AFL is really high, run test cases 1000+/sec c) Designed for practicality, and after downloading AFL, very little configuration is requires before you use it.	a) Cannot work with Sanitizer to detect some kinds of security issue such as buffer overflow. b) Need user provide test seeds before starting fuzzing test. c) For some input that have complex structure, AFL may waste a lot of time to fuzz unimportant part.
libFuzzer	a) Can be used combine with Sanitizer to detect buffer overflow.	a) Need user provide test seeds before starting fuzzing test. b) For some input that have complex structure, libFuzzer may waste a lot of time to fuzz unimportant part.
Peach	a) Do not need user provide test seeds before starting fuzzing test. b) Can create DataModel to generate test inputs which is formatted with specific complex structure. c) Can be used combine with Sanitizer to detect buffer overflow.	a) Need extra effort to create DataModel.

Note: As the algorithms these three tools use is different, the superiorities of detecting issues are varying from issue type, if you try more tools, you have more chances to detect issues.

2) Create test seeds:

You can use KLEE to help you generate most of seeds, and improve seeds quality manually to cover the paths that KLEE seeds not cover.

3) Catch error:

Sanitizer is a good tool to catch special issue like buffer overflow, using Sanitizer along with other fuzzing tools can make test more efficiency.

4) Return some special error status:

These is a situation that some error status is hard to occur in real test, such as `EFI_OUT_OF_RESOURCE` in `AllocatePool()`, you can use Error Injection to produce this error status when call this function and check whether tested code works correctly when this error return.