Srijeet's Ramp up on Undo

Daily Logs - Internship

3rd June

- Got introduced to the team and briefed about the project.
- Learnt how to setup VNC
- Got introductions about GNU Debugger and Undo Debugger
- Downloaded sample programs from GitHub and debugged them using GDB and UDB. (Debugged programs can be found in /nobackup/sriguha /podman-storage/test-gdb)
- Documentations in **Virtual Network Connection** and **GDB and UDB** sub-pages.

6th June

- Debugged another sample program using UDB that was given as an assignment.
- The program can be found in the above folder.

7th June

- Learnt about memory issues like memory leaks, memory corruption and dangling pointers
- Learnt about memory fencing.
- Got to know about Valgrind and ASAN.
- Documentation in the **Memory Issues** sub-page.

8th June

- Learnt about networking basics (Switches, Routers, Hubs, IP and MAC Addresses).
- Learnt about IP forwarding, IP routing and Border Gate Protocol.
- Documentation in **Networking** sub-page
- Tried to clone and build the open-source project : https://github.com/telefonicaid/netphony-network-protocols.
- Installed Maven and tried to build the code but failed as could not change default compiler of the environment from JRE to JDK.

9th June

- Tried to clone and build project : https://polaris-git.cisco.com/csg/polaris
- Received errors while generating SWIMS Tickets.
- Went through the wiki : IOSXE SWIMS Image Signing and subsequently mailed the SWIMS team for approval for ticket creation.
- Errors and the request can found in mail titled : "**Approval for SWIMS Ticket Creation for IOSXE Build".**

10th June

- Received approval for **SWIMS Ticket Creation** from the SWIMS Team.
- Went through the wiki: Getting Started · csg/polaris Wiki (cisco.com)to continue with the build process of polaris-dev code.
- Received errors regarding the installation and setup of **git-lfs**
- Could not log into my JFrog artifactory account to setup SSH keys and install git-lfs.

13th June

- Resolved the issues with login into my JFrog artifactory account.
- Inserted SSH keys and installed the **git-lfs** executable
- **exported the PATH=/auto/binos-tools/bin:/router/bin:/usr/cisco/bin:/ws/polaris_bc/tools/bin:/usr/share/Modules/bin:/usr/local/bin:/usr /bin:/usr/local/sbin:/usr/sbin:/bin:/opt/puppetlabs/bin:/usr/sbin:/sbin:/users/sriguha/bin/:.:$PATH** to setup correct path for executable binaries and setup files for building polaris-dev code.
- Successfully built the polaris-dev code in my workspace.

14th June

- Learnt about **versioning** of functions.
- created rudimentary codes for versioning of functions.
- Went through the polaris-dev code starting from the swift_undo_enabler.py script to understand the undo workflow and how to enable it manually during **gmk -f build process**.

15th June

- Task assigned : **Add LR_MANUAL flag to the test workflow to generate live-recordings (as per requirement) of the execution of test cases manually.**
- Added LR_MANUAL flag alongside existing LR_ARCHIVE flag (in swift_undo_enabler.py and launch_dynamic files in /polaris/binos/sample_code/ lua_runtime_test/test/scripts/launch_dynamic.sh and /polaris/binos/infra/orchestrator/sh/launch_dynamic.sh

- Checked the test logs to realize that the above files were **not being hit** during the build process of the test cases. **(from gmk -f command).**

INTERNSHIP TASK : Add LR_MANUAL flag to the test workflow to generate live-recordings (as per requirement) of the execution of test cases manually.

16th June

- Went through the entire logs generated due to the execution of **gmk -f** command to trace the flow of control in the scripts.
- Found difficulty in tracking the flow of control in the files.

20th June

- I went through the logs and the cisco wiki on host_ut, swift and the build process to gen more knowledge about the system.
- Created documentation regarding my understanding of the system and the test environment.

The documentation on Build Process and Test Environment can be found in this sub-page.

21st June - 23rd June

- Started to go to through the execution flow of **gmk -f test-aut-host_ut_cge7** command in an attempt o add the LR_MANUAL flag.
- Also checked how **LR_ARCHIVE** flag in the above command changes the workflow.
- Created the documentation of the flow of control in the build scripts.

Brief Documentation on control flow for execution of above command can be found in this sub-page

24th June

- Realized that a **patch is missing** in the **launch_dynamic.sh** script. So added the patch. The patch can be found in this **sub-page.**
- This patch actually enables us to **attach the live-recor**der to the execution of the test cases using the **LR_ARCHIVE** flag.
- **Also added the executables of live-record_x64, live-record_polaris, log_undo_usage, and gen-replay-archive.sh in mytoolchain folder.** R efer to above sub-page for more details.

All changes required in the current workspace to enable live-recorder can be found in this sub-page

27th June

- Went through the patch that was added.
- Also read about the executables that were given.
- Went through **Using the LiveRecorder tool — Undo.io** to learn more on live-recorder.

29th June - 30th June

- Tried to use live-recorder on scripts that I had written. Confirmed that **live-recorder is working fine** in my workspace.
- Ran **gmk -f LR_ARCHIVE=1 test-aut-host_ut_cge7** to confirm that undo recordings are getting generated in respective test case folders. The undo recordings were generated along with the logs to point at the correct directory where recordings are stored and the PID of the processes. All other logs and files that were generated have also been checked.
- Ran the undo recordings using udb command to check for the correctness of the undo recordings.
- Went through rest of the logs to check for other code flows.

Contents of odm_test__*odm_test_1*__multi after gmk command

```
bash-4.4$ pwd
/nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-host_ut_cge7/odm_test___odm_test_1___multi
bash-4.4$ ls
_BINOS_FRU_RP_0_0_2463193_127.1.1.1.create       odm_test-2463834-2022-07-19T20-35-36.543.undo
_BINOS_FRU_RP_0_0_2463193_127.1.1.1.remove       pid.2463293.___SWIFT_P_PROC___torchd__SWIFT_M_MUT__odm_test
btrace                                           pid.2463539.___SWIFT_P_PROC___odm_test__SWIFT_M_MUT__odm_test
local_BINOS_FRU_RP_0_0_2463440_127.1.1.2.create  pid.2463566.___SWIFT_P_PROC___odm_test__SWIFT_M_MUT__odm_test
local_BINOS_FRU_RP_0_0_2463440_127.1.1.2.remove  profraw
local_BINOS_FRU_RP_0_0_2463449_127.1.1.3.create  tmpfs
local_BINOS_FRU_RP_0_0_2463449_127.1.1.3.remove  torchd-2463414-2022-07-19T20-35-21.616.undo
logs                                             xunit
odm_test                                         xunit-incremental
odm_test-2463783-2022-07-19T20-35-32.687.undo
```

1st July

- Read about **postcommit** and **precommit** workflows

- Brief information about the same :
    - **precommit workflows :** Execution of test cases in manual build process. LR_MANUAL flag is supposed to generate the undo recordings in this case only.
    - **postcommit workflws** : Execution of the test cases in the automatic build process. This calls the unod_enable.py code to setup the flags and generate the undo recordings.

4th July - 8th July

- Added **LR_MANUAL flag** in the **/nobackup/sriguha/mytoolchain/gen-replay-archive.sh file.**
- Ran the command of **gmk -f LR_MANUAL=1 test-aut-host_ut_cge7** to execute the test cases and generate undo recordings.
- The processes and the functionality has remained very similar to the **LR_ARCHIVE flag**.
- Ran the undo recordings to ensure that the recordings generated are correct.
- Received similar logs as above and also the same files were generated as above.
- Went through the logs to ensure correctness of execution.
- Started **documentation on cisco-wiki.**

All changes required in the current workspace to enable live-recorder with LR_MANUAL flag can be found in this sub-page

11th July - 15th July

- Realized the test cases were **unexpectedly failing** despite the test cases passing using the UDB method.
- **Reviewed the logs** to great extent and tried to track down the problems.
- Tried reverting back to the original codes with **minimal changes to track the source of the problem**.
- Found the reasons for the problem but could not find how to solve the same.
- **Made some changes to start live-recorder without initialization of UDB.**

Further Documentation details the errors and issues that was encountered up till now and steps taken to debug the issue. The same can be found here.

18th July

- Started creating an **exhaustive documentation of all progress done in the internship period**.
- Updated and formalized **daily logs**.
- Started work on the presentation for final presentation.
- Also tried to **debug further the problems**.

19th July

- Completed the documentations on all progress made in the internship period.
- Continued with the final internship presentation.
- **Debugged the problems to now allow the live-recorder to record and generate the test recordings and the test cases passes.**
- **Completed the basic internship tasks** (further refinement is required).
- **Demonstrated the capability of generated undo recordings while passing the test cases.**

The complete addition to the workspace can be found here at the bottom of the document.

20th July

- Reviewed the presentation and made minor changes to the documentation.
- Cleaned up the scripts a bit.

Virtual Network Connection

Virtual Network Connection

Server – Client Model (VNC)

Server – Client Model: Server is a piece of hardware or software that provides facilities to other systems via the internet or through a direct connection. Client basically sends a request, and the Server processes the request and responds back to the client.

VNC (Virtual Network Connection): It uses a client server model where the server can broadcast the entire screen to the client. One server can broadcast the screen to multiple clients and one client can see the screen of multiple servers. All commands, touches, mouse movements, keystrokes are transmitted to the server from the client and the server processes it and responds back with the result.

So VNC has 2 parts to it: **VNC Server** and **VNC Client**

VNC Server: Software which allows us to broadcast the screen onto the internet.

VNC Client: It is the software which allows us to act as a client and use the server.

VNC Server-client model uses **Remote Framebuffer (RFB)** protocol for transmitting the information.

Now we have several of such systems which uses the client server model in the market. These include VDI, RDS and VNC.

VNC – Virtual Network Connection – It can be used to broadcast any server on any client. Means a windows server can be serving to a mac client. **Uses Remote Framebuffer (RFB) protocol.**

RDS – Remote Desktop Services – It can be used to broadcast a windows server on a windows client only. Uses **Remote Desktop Protocol (RDP).**

VDI – **Virtual Desktop Infrastructure** – It can be used to broadcast a windows server on any client. Uses **Remote Desktop Protocol (RDP).**

So we have a **Linux server based in California** and we can use a part of the memory for my intern purposes. The IP address of the server is **64.104.135. 159 (alias. bgl-ads-5285).** To use that server remotely we need to **ssh** into that server and then use it as our own system **The server since is a linux server has VNC server software installed in it. We basically need to install the VNC Client software (VNC Viewer) and start connecting to it and using it.**

Software's that we are using for the same are :

PuTTY: This software provides a terminal window to remotely ssh into a server. So basically, the client has it installed. It also provides a terminal window and all other software required for a VNC connection**. Same can be done using the command prompt of the windows as well.**

VNC Viewer: It helps us to visualize the remote server or the system. The Linux system can be visualized like the Virtual Machine and the keystrokes etc. can be seen.

VNC Commands

- **ssh IP_Address :** This helps us to ssh into the IP address server and start using the system.
- **vncserver :** It gives details of the vnc server like when it was last accessed, Its IP address, how many desktops are being currently accessed and so on.

GDB and UDB

GNU Debugger (GDB) and UNDO Debugger

Official Website: GDB Documentation (sourceware.org)

Official Documentation: Top (Debugging with GDB) (sourceware.org)

YouTube tutorial Playlist: (110) introduction to GDB a tutorial - Harvard CS50 - YouTube

Introduction

GNU Debugger (GDB) is a software which helps to run the program step by step. Command line-based software. Also helps in forcing values and helps in debugging.

Some **basic commands** that we often use while using the GDB are:

Starting GDB

We need to first create the executable of the file that we had created. For that we use **make file_name.extension** to make the file. We donot use the compiler directly because it does not give the -o file. Now we need to make a map between the executable and the files. Fot that we use **make_executable_line -g** command. Now we are all set for running it on gdb. **UDB is an aliased command of /auto/binos-tools/bin/udb**

- **Gdb executable_name** : It allows us to ru the executable on the gdb
- **Udb executable_name :** It allows us to run the executable on the udb

Creating Break Points

Helps to create break points so when we run the program the GDB stops the executions just before the break points. So, when we are doing next or step it stops the execution just before the break point. We can create multiple break points.

- **Break func_name** : inserts a break point just after the function call.
- **Break file_name:line_num** : inserts a break point just before the particular line. Note the GDB must be in that file while we are executing so we must go to that file by doing step.
- **(In case of function overloading in the first case refer to documentation for resolving break discrepancies)**
- **Info break:** Lists all the break points.
- **Delete :** Deletes all the break points
- **Delete br_num :** deletes that particular breakpoint number.

Starting program execution

It helps to start the program running after it has been invoked on the gdb debugger.

- **Start :** It starts running the program from the main and then we need to do run / step to the first break point.
- **Run** : It starts the program and stops the execution just before the first break point.

Running the Program

It helps to run or step the program to the next execution line.

- **Step** : It allows us to step through every line of the file. In case there exists a function call directing to a different file or some other executable, it transfers control to that file as well.
- **Next** : It allows to run through every line of the high level code in the main file. In case of a function call, it just skips over it.
- **Reverse-next** : reverses the instruction
- **Finish** : Finish the execution of the program
- **Continue** : Go on to the next iteration

Printing and Forcing Variable values

This allows us to print values of the variables in the middle of execution.

- **Print var_name** : It allows us to print a variable value in tme middle of execution.
- **Print var_name = value** : It allows us to force store the variable.
- **info locals** : It allows us to print all the local variables of the function.
- **Info variables** : It allows us to print all the variables outside the scope of the function.

Watching Variables

It helps to watch the variables when it is altered / read / goes beyond a value etc.

- **Watch var_name** : stops execution when the given variables’ value is changed
- **Watch -r var_name** : Watches if the var_name’s location is altered.

- **Watch var_name if (condition)** : Stops the execution if the condition is met.

Memory Issues

Memory Issues

Memory Leaks

Reference : What Is a Memory Leak and How Do They Happen? (makeuseof.com)

Reference : Top 20+ Memory Leak Detection Tools for Java and C++ (softwaretestinghelp.com)

- It happens when we **allocate memory in the heap of the program space**, and we forget to free it. Memory allocated dynamically in the heap space **remains allocated** until we either purposefully free it or the program comes to an end.
- Program allocates more memory that it will ever need.
- We have allocated a memory and lost the pointer or reference to it. So now we cannot access the memory.

This problem is more applicable in servers or for applications which do not shut down for a long time. Memory leaks can consume the systems memory and slow down the system / reduce system performance.

Results of memory leaks: Reduced system performance, reduced memory available. Next time the application tries to access memory, the OS refuses to provide it as it has exceeded the given memory space, so application crashes.

Preventing memory leaks: Freeing up dynamically allocated memory after use. We can also **use garbage collectors** which help to free up unused dynamically allocated memory. Many programming languages use automatic garbage collectors for this purpose.

Memory Corruption

Reference : Memory corruption - Wikipedia

When a memory location has been changed unintentionally by a program, it is called memory corruption. If the same memory location is again later used, then it causes wrong results and may cause system crashes.

Causes of memory leaks: It can be caused due to multiple reasons like:

- Using memory before initializing it. It causes wrong values to be taken up and therefore wrong results.
- Using pointers to access / modify memory. The pointers may be a dangling pointer or may have been freed before itself and thus has garbage values or is being used by some other program. Modifying / accessing it causes errors. Errors may be caused if the memory that has been freed up is again freed up.
- Accessing / modifying memory out of bounds. It may happen due to faulty termination condition that we access out of bounds memory locations.

Memory Fencing

Reference: https://kuafu1994.github.io/MoreOnMemory/memory-fence.html

Reference: (Linux documentation)

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/memory-barriers.txt?id=HEAD

Memory fences are types of instruction that puts a constraint on the reordering of instructions during their execution on the processor. For any processor, all memory access-related instructions before the memory fence instruction are executed before the memory access-related instructions after the memory fence.

This allows the compiler to reorder the memory access instructions before and after the fence only. This is very useful when an application is running multiple parallel threads and each such thread is trying to access/modify shared memory variables.

Memory fencing provides a barrier and prevents out-of-order memory access across the fence instruction. The example is given in the first reference.

[Note the compiler/processor can reorder instructions where the memory access of one instruction does not affect the other instruction. But altering this order can affect the execution of other threads in the same process].

Valgrind

Reference: https://valgrind.org/ (Official site)

The Valgrind distribution currently includes seven production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and two different heap profilers

Address Sanitizer

AddressSanitizer — Clang 15.0.0git documentation (llvm.org)

Networking

Networking

Nodes -> sub-network -> LAN ->WAN

Components of the network

Switches: It connects all the computers in the sub-network. Any packet that it receives from one of the computers/nodes, its destination MAC address is checked and if the computer is in the sub-network, the packet is forwarded to the node. If the MAC address is not found in the sub-net, then it is sent to the default router connected to the switch which then routes it to the next router in the Wide Area Network based on its entries in its routing table.

Routers: Routers connect multiple Local Area Networks on the Wide Area Network or the internet. Each Local Area Network has a router that connects it to the internet. Any cachet that it receives from the multiple switches connected router, is then forwarded to the other router based on the IP address found from the routing table and the protocol used.

Hubs: It is used to broadcast a message in the sub-network to all the nodes.

Each LAN has multiple switches. Each switch corresponds to one sub-network. Each LAN has one router which connects the LAN to the WAN.

IP and MAC Addresses

IP Addresses : (IPv4): These are 32-bit numbers divided into sets of 8-bit numbers ranging from 0 to 255. Each node in the network is located by its IP Address. These are the logical address and is based on the LAN and the sun-net the node is connected to. If the node is connected to some other network then the IP Addresses change.

Routers have 2 IP Addresses. One for receiving data packets and one for sending data packets. Switches/hubs do not have IP Addresses.

No 2 nodes in the LAN can have the same IP Addresses. Multiple nodes on the internet can have the same IP Addresses. The differentiation is made using private and public IP Addresses.

Public and Private IP Addresses: Each router has a public and private IP address. The public IP addresses are unique throughout the internet. The **Netw ork address Translation** helps to convert the private to public IP addresses. Whenever a packet is received at the router, the router changes the source IP address from the node to that of the private IP address of the router which is then translated to public. When a response to the packet is received, it forwards it to the node after checking the source from the **connection tracking table**.

Netmask: Each sub-net has a netmask. This netmask is also a 32-bit IP address. The netmask helps us to determine the range of IP addresses possible in the sub-net. To find the range of IP addresses, all the bits of IP addresses corresponding to 1 in the netmask are constant whereas those that are 0 are changing. The first few bits that are not changing are the **Network address** and the remaining bits corresponding to 0 in netmask are **the host address**

IP Addresses : (IPv6): 128 bits and has 8 sections of 16 bits each. Represented in hex instead of decimal. Netmasks are not used here. Instead, we use / (no. of bits in network address from the start). Since the number of IP addresses is large, the network address translation is no longer required and each device can have its IP address.

IP Routing

IP routing is a method by which we transmit the IP packets from one router to another over the wide-area network. The steps involved are :

- The packet is made by the node. It has the source IP and MAC address and the destination IP and MAC address. The packet has some security features and checksums to see if the packet is corrupted or not.
- The packet is then sent to the switch of the sub-net. The connection between the node and switch is usually direct. The switch checks the MAC address of the packet and checks through the MAC address table if the node belongs to the same subnet. If it does, it does the checksums to check if the packet is corrupted or not. If not it sends the packet to the node.
- Now if the MAC address does not match, it sends the packet to the default router or the router determined by the router table. The router now goes for a checksum to see if the packet is corrupted or not. If not then dumps the source IP address and logs it to its connection tracking table. It then attaches its IP address to the packet. Then based on the protocol used, it decides which router to relay the packet next. The router IO address is found from its router table.
- After the packet reaches the destination router, it is sent to the switch based on the IP address of the destination mentioned in the packet. The switch then sends it to the node with the matching MAC address.

The same process is followed when sending a response.

IP Forwarding

IP forwarding basically determines the way by which the packet will be forwarded from one router to another on the internet from the source to the destination. The routing tables in each router determine which router the packet needs to be forwarded to for shortest path movement. The routing table has IP addresses range of connected routers, netmasks, connection interface. The next router the packet needs to hop is found from the longest matching mask.

Border Gate Protocol

It is a protocol to determine to which router the packet needs to be forwarded to when multiple routers are connected to it.

Here the routers exchange routing tables between each other. This allows the router how to move to the destination router and in how many hops. It also know the options it have

Build Process and Test Environment

Documentation on SWIFT

Reference : SWIFT - SW Integration and Functional Test framework for Polaris processes - POLARISENG polarisenG - IT Wiki (cisco.com)

SWIFT stands for **Software Integration and Function Test** framework. It helps to test parts of the polaris-dev code or modules of the code without involving all the modules of the code. So SWIFT framework as mentioned in the later parts of the document will often be used for testing the build process and any changes that may be made in the scripts involving the build process.

The SWIFT framework makes use of the **Orchestrator Infrastructure** to orchestrate the execution of test cases and the parts of polaris-dev code. The orchestrator helps us to test all the **test cases in a loop** and only stop if there exists some error or if any test case fails.

NOTE:

- The orchestrator will test cases and print logs in appropriate places only if the **test cases have not been previously tested** before.
- The directory where the orchestrator scripts and files are present are : **polaris/binos/infra/orchestrator/**
- Whenever we run try to execute the test cases, the orchestrator checks the **/polaris/binos/mgmt/odm/test/_gen_<test dir>** directory if the logs of the test cases are present or not. If the logs are present then the orchestrator just skips over the test cases and runs the test cases whose logs are not available in the above mentioned directory.
- Here we will usually test the **_gen_swift_aut-host_ut_cge7** test cases so all further documentation is with respect to same.

Running and Implementing SWIFT test cases

Reference : SWIFT - SW Integration and Functional Test framework for Polaris processes - POLARISENG polarisenG - IT Wiki (cisco.com)

Reference : Running SWIFT Test Cases - POLARISENG polarisenG - IT Wiki (cisco.com)

The orchestrator framework is responsible for running the test cases. The target test cases and the directory where the logs and other stuff is to be stored is mentioned in the command to the orchestrator for running the test cases. As mentioned in the references the orchestrator can run the SWIFT test cases either on the **host-machine** or the **target-machine**. For the rest of the documentation, **we run the SWIFT test cases in host based testing.**

For **Host-based** testing, we use a special **host_ut** target for running and compiling the test cases on the build machine. Therefore the test target as **_gen_swift_aut-host_ut_cge7** is used for compiling and running the test cases. To run the SWIFT test cases on the host-machine with **_gen_swift_aut-host_ut_cge7** we can use the **gmk -f test-aut-host_ut_cge7 c**ommand which needs to be run form the **/polaris/binos/mgmt/odm/test** directory only after removing the existing above directory so as to force the orchestrator to run the test cases on the target machine.

Therefore the commands that needs to be run to run the test cases are :

Host testing of SWIFT test cases

```
cd polaris/binos/mgmt/odm/test/
rm -rf  test-aut-host_ut_cge7
gmk -f test-aut-host_ut_cge7
```

The gmk -f command builds the target SWIFT test cases in the host machine and runs all the test cases in loop until there is an error or all the test cases passes. The logs and other stuff regarding the test cases can be found in :

Logs for gmk -f test cases

```
bash-4.4$ cd polaris/binos/mgmt/odm/test/_gen_swift_aut-host_ut_cge7
#this directory has all the test cases and the corresponding logs. Doing an ls in this directory gives:
bash-4.4$ ls
odm_test___odm_test_10___multi  sanity_test_odm_test___odm_test_10___multi.log
sanity_test_odm_test___odm_test_26___multi.time
odm_test___odm_test_11___multi  sanity_test_odm_test___odm_test_10___multi.time
sanity_test_odm_test___odm_test_27___multi.log
odm_test___odm_test_12___multi  sanity_test_odm_test___odm_test_11___multi.log
sanity_test_odm_test___odm_test_27___multi.time
odm_test___odm_test_13___multi  sanity_test_odm_test___odm_test_11___multi.time
sanity_test_odm_test___odm_test_28___multi.log
odm_test___odm_test_14___multi  sanity_test_odm_test___odm_test_12___multi.log
sanity_test_odm_test___odm_test_28___multi.time
odm_test___odm_test_15___multi  sanity_test_odm_test___odm_test_12___multi.time
sanity_test_odm_test___odm_test_29___multi.log
odm_test___odm_test_16___multi  sanity_test_odm_test___odm_test_13___multi.log
sanity_test_odm_test___odm_test_29___multi.time
odm_test___odm_test_17___multi  sanity_test_odm_test___odm_test_13___multi.time
```

```
sanity_test_odm_test___odm_test_2___multi.log
odm_test___odm_test_18___multi  sanity_test_odm_test___odm_test_14___multi.log
sanity_test_odm_test___odm_test_2___multi.time
odm_test___odm_test_19___multi  sanity_test_odm_test___odm_test_14___multi.time
sanity_test_odm_test___odm_test_30___multi.log
odm_test___odm_test_1___multi   sanity_test_odm_test___odm_test_15___multi.log
sanity_test_odm_test___odm_test_30___multi.time
odm_test___odm_test_20___multi  sanity_test_odm_test___odm_test_15___multi.time
sanity_test_odm_test___odm_test_31___multi.log
odm_test___odm_test_21___multi  sanity_test_odm_test___odm_test_16___multi.log
sanity_test_odm_test___odm_test_31___multi.time
odm_test___odm_test_22___multi  sanity_test_odm_test___odm_test_16___multi.time
sanity_test_odm_test___odm_test_32___multi.log
odm_test___odm_test_23___multi  sanity_test_odm_test___odm_test_17___multi.log
sanity_test_odm_test___odm_test_32___multi.time
odm_test___odm_test_24___multi  sanity_test_odm_test___odm_test_17___multi.time
sanity_test_odm_test___odm_test_33___multi.log
odm_test___odm_test_25___multi  sanity_test_odm_test___odm_test_18___multi.log
sanity_test_odm_test___odm_test_33___multi.time
odm_test___odm_test_26___multi  sanity_test_odm_test___odm_test_18___multi.time
sanity_test_odm_test___odm_test_34___multi.log
odm_test___odm_test_27___multi  sanity_test_odm_test___odm_test_19___multi.log
sanity_test_odm_test___odm_test_34___multi.time
odm_test___odm_test_28___multi  sanity_test_odm_test___odm_test_19___multi.time
sanity_test_odm_test___odm_test_35___multi.log
odm_test___odm_test_29___multi  sanity_test_odm_test___odm_test_1___multi.log
sanity_test_odm_test___odm_test_35___multi.time
odm_test___odm_test_2___multi   sanity_test_odm_test___odm_test_1___multi.time
sanity_test_odm_test___odm_test_3___multi.log
odm_test___odm_test_30___multi  sanity_test_odm_test___odm_test_20___multi.log
sanity_test_odm_test___odm_test_3___multi.time
odm_test___odm_test_31___multi  sanity_test_odm_test___odm_test_20___multi.time
sanity_test_odm_test___odm_test_4___multi.log
odm_test___odm_test_32___multi  sanity_test_odm_test___odm_test_21___multi.log
sanity_test_odm_test___odm_test_4___multi.time
odm_test___odm_test_33___multi  sanity_test_odm_test___odm_test_21___multi.time
sanity_test_odm_test___odm_test_5___multi.log
odm_test___odm_test_34___multi  sanity_test_odm_test___odm_test_22___multi.log
sanity_test_odm_test___odm_test_5___multi.time
odm_test___odm_test_35___multi  sanity_test_odm_test___odm_test_22___multi.time
sanity_test_odm_test___odm_test_6___multi.log
odm_test___odm_test_3___multi   sanity_test_odm_test___odm_test_23___multi.log
sanity_test_odm_test___odm_test_6___multi.time
odm_test___odm_test_4___multi   sanity_test_odm_test___odm_test_23___multi.time
sanity_test_odm_test___odm_test_7___multi.log
odm_test___odm_test_5___multi   sanity_test_odm_test___odm_test_24___multi.log
sanity_test_odm_test___odm_test_7___multi.time
odm_test___odm_test_6___multi   sanity_test_odm_test___odm_test_24___multi.time
sanity_test_odm_test___odm_test_8___multi.log
odm_test___odm_test_7___multi   sanity_test_odm_test___odm_test_25___multi.log
sanity_test_odm_test___odm_test_8___multi.time
odm_test___odm_test_8___multi   sanity_test_odm_test___odm_test_25___multi.time
sanity_test_odm_test___odm_test_9___multi.log
odm_test___odm_test_9___multi   sanity_test_odm_test___odm_test_26___multi.log
sanity_test_odm_test___odm_test_9___multi.time
```

Integration of UDB and live-recorder_x64 to the SWIFT host_ut test cases

Reference : UNDO for HostUT - POLARISENG polarisenG - IT Wiki (cisco.com)

We want to attach UDB or live-recorder to the SWIFT test cases. So that if any errors are seen it can be easily debugged.

In this internship perspective, we want to attach live-recorder_x64 to the execution of SWIFT test cases so that test recordings are generated irrespective of whether the test cases pass or not. **The recordings for each separate test cases need to be stored in a designated directory so that we can replay the recordings later to debug the test cases.**

The recordings technically are supposed to be generated **odm_test__odm_test_<num>__multi** directory.

Some existing flags that have been integrated in the test process are:

LR_ARCHIVE : this flags automatically attaches UDB or live-recorder_x64 (after the patch) in the test process for automatic builds

LR_MANUAL: this flag needs to be added to attach live_recorder_x64 in the test process for manual testing.

Control Flow for gmk -f test-aut-host_ut_cge7 command

As stated before when the above command is executed from the polaris/binos/mgmt/odm/test/ directory, it helps to execute the _gen_swift_aut-host_ut_cge7 test cases on the host machine. Through the logs we can track the flow of control. The scripts accessed in order are :

- **polaris/binos/build/scripts/gmk.local** : This is basically a bootstrap program and calls the next script. Essentially it has no features except to process the command given. This is the first script that is called immediately after the command gets executed.
- **/polaris/binos/build/scripts/gmk.local :** This is the main program that starts the build process and the execution of the test cases. Some major functions of this script are :
    - It parses the command line for flags in the command and separates them and calls respective functions for the same. **(parse_cmdline() function)**
    - It also creates the log file where all the logs are stored. **(generate_build_log_path() function)**
    - Exports the required environment variables. (**process_environment() function**)
    - Checks for memory requirements and other system requirements (**calc_jobs_max() and get_os_version() function**)
    - Perform build using the **mcp_ios_precommit aut-test** command (**perform_build() function**)
    - Perform build using the gmk -f command (**perform_ibc_build() function**)
    - We are concerned with the above function for tracking the flow of the command and tracked the flow of control from this script to another from this function.

part of script that starts gmk -f build process

```
# The following build command gets invoked when user runs:
    # mcp_ios_precommit from ios/sys/ or gmk -f from Component or binos/ dir
    print "MAKE CMD ($$): [@args]\n";
    my $ret = system(@args);

    # FIXME TIME_TARGET is a hack to allow
    if ($BUILD_TYPE{host_sdk}) {
        $cmdStr = "'TIME_TARGET=sdk-host' ".$cmdStr;
    } elsif (%targets) {
        $cmdStr = "'TIME_TARGET=".join(":", sort keys %targets)."' ".$cmdStr;
    }
    if ($ret == 0) {
        $exit_text_string =
            qq($PROG ($$): SUCCESS ($target_archs) [cmd: $cmdStr]\n);
    } else {
        my $errmsg = get_error_message($CHILD_ERROR, $ERRNO);
        $exit_text_string =
            qq($PROG ($$): FAILED ($target_archs) [cmd: $cmdStr] : $errmsg\n);
        $build_failed = 1;
        die "$PROG ($$): FAILED";
```

gmk -f test-aut-host_ut_cge7 command with LR_ARCHIVE flag

If the command is run without the flag, this section is never executed. However, this script this encountered due to the flag.

After some surveying the code, we come across **/polaris/binos/infra/orchestrator/sh/launch_dynamic.sh :** This script is in which we are interested in. This is responsible for **attaching UDB or live-record** to the execution of the test cases. Note the script in the **opengrok does not have the patch** added mentioned in the daily logs. The patch and other changes in the workspace has been mentioned in other sub-pages. Going through this script we encounter some important parts of the script:

- **enable_debug_for_program ()** : function checks if the current workflow is a postcommit or a pre-commit workflow. In case of postcommit workflow, In case it is a post-commit workflow, it calls some other functions to attach UDB or live-recorder.
- **binos_exec () :** This function is actually responsible for calling the scripts for attaching the UDB or the live-recorder. Here, since the patch has not yet been added, the UDB script is being called. When we add the LR_ARCHIVE flag this part is particularly interesting inside the function.

  (this is where we have changes in the patch)

Executed for LR_ARCHIVE flag

```
    elif [ "$LR_ARCHIVE" ] && [ "$CAN_LR" == "1" ]; then

        # check whether to enable debug for this MUT
        enable_debug_for_program

        # run test case with undo in the background
        /auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh $@ &

        BINOS_PID=$!
        PROC_PATH=${FULL_PROC_PATH%/*}
        PROC=${FULL_PROC_PATH##${PROC_PATH}/}
        binos_echo ${PROC}, ${FULL_PROC_PATH}
        PID_FILE=pid.${BINOS_PID}.___SWIFT_P_PROC___${PROC}__SWIFT_M_MUT__${PROG_NAME}
        rm -f `pwd`/${PID_FILE}
        binos_echo "${PROG_NAME}:${FULL_PROC_PATH}:${RESOLVED_PROCESS}" > `pwd`/${PID_FILE}

        # Get the parent process id and process group id. If they are equal
        # then that is the group id of the top-level test. That group
        # id can be used to cleanup any forked children of the test
        local psinfo=( $(ps xo pid,ppid,pgid | grep "^\s*$BINOS_PID\s") )
        if [ "${psinfo[1]}" == "${psinfo[2]}" ]; then
            BINOS_PGID=${psinfo[2]}
            BINOS_PGNAME=`ps --no-headers -p $BINOS_PGID | awk '{print $4}'`
            BINOS_GROUP_CLEANUP=1
            binos_echo "SWIFT process group: $BINOS_PGID"
        fi

        trap "kill_module_under_test \"${BINOS_PGNAME}\" ${BINOS_PID}" SIGHUP SIGINT SIGTERM SIGKILL
        wait $BINOS_PID
        rc=$?
        binos_echo "wait returned $rc"
```

- As can be seen from above **/auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh** calls the script to start the UDB and attach it to the currently running process.
- On going through the above script, we can see how UDB is getting attached. Some important functions that are in use in the above script are :
    - **prepare_environment()** : function creates and prepares the environment for attaching UDB. It records the start time, UDB_pid, storage locations and other details.
    - **generate_working_paths()**; function generates where all the data is going to be stored and prepares files for the same.
    - **udb_handle_SIGTERM()**: traps the end signal of the udb function and successfully detaches it.
    - **run_udb():** initialised the udb, attached the udb to currently executing process, waits to trap the end of udb and safely detaches udb from the build process.

    run_udb function has a change

```
run_udb() {
    # Run UDB
    # We turn snapshot target down and reduce the snapshot rate
    # to reduce memory overhead - we aren't reversing so there's
    # no need for snapshots.
    # Finally, we use a special keyfile that doesn't
    # need access to the license server.
    lr_log "Recording test..."

    export UNDO_snapshot_interval=$((2^20))
    export UNDO_keyfile=$UDB_KEY

    local UDB=$(which udb)
    if ! [[ $UDB ]]; then
        UDB=/auto/binos-tools/bin/udb
    fi

    lr_log "Starting $UDB"

    if [[ "$LR_SKIP_DL" == "1" ]]; then
        $UDB -batch -quiet --defer-recording \
             -x $UDB_INIT_SCRIPT \
             --max-snapshots 4 \
             --args "$@" &
        UDB_PID=$!
    else
        $UDB -batch -quiet \
             -x $UDB_INIT_SCRIPT \
             --max-snapshots 4 \
             --args "$@" &
        UDB_PID=$!
    fi


    lr_log "Waiting for UDB exit"
    # Install a sigterm handler that knows how to shut
    # down the debuggee
    trap udb_handle_SIGTERM SIGTERM

    # Wait for UndoDB to exit
    if wait $UDB_PID; then
        lr_log "UDB exited normally"
    else
        lr_log "UDB exited uncleanly"
        echo 1 > $EXIT_CODE_FILE
    fi

    # Ensure the debuggee has terminated
    if [[ -e $START_TIME_FILE ]]; then
        local DEBUGGEE_START_TIME=$(<$START_TIME_FILE)
        lr_log "Waiting for debuggee exit (debuggee started at $DEBUGGEE_START_TIME)"
        while true; do
            local J=$(cut -f 22 -d ' ' /proc/$(<$PID_FILE)/stat || true)
            lr_log "PID $(<$PID_FILE) start time = $J"
            if [[ $J != $DEBUGGEE_START_TIME ]]; then
                # J is empty (because the file is missing)
                # or is different because the PID has been recycled.
                # Either way, the debuggee exited.
                break
            fi
            sleep 1
        done
    else
        lr_log "Warning: No start time file $START_TIME_FILE"
    fi

    lr_log "Debuggee exited"
}
```

Workspace Additions

This documentation includes all the changes that have been made in my workspace to accommodate the live-recorder feature using LR_MANUAL flag.

- The following patch has been added in **/nobackup/sriguha/polaris/binos/infra/orchestrator/sh/launch_dynamic.sh script**. This patch makes changes in the orchestrator infrastructure to change how undo is attached to executing test cases. process. Now instead of calling the **gen-replay-archive** script from **/auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh** (which basically initiates and attached the udb to the current process) the launch_dynamic calls the **/nobackup/sriguha/mytoolchain/gen-replay-archive.sh** (which basically initiates and attaches the live-recorder). The change can be done manually as the changes are small.

x.diff (for attaching live-recorder with LR_ARCHIVE flag)

```
 diff --git a/binos/infra/orchestrator/sh/launch_dynamic.sh b/binos/infra/orchestrator/sh/launch_dynamic.sh
index 41428c609f4a..9e4a79bdea17 100755
 --- a/binos/infra/orchestrator/sh/launch_dynamic.sh
 +++ b/binos/infra/orchestrator/sh/launch_dynamic.sh
@@ -487,7 +487,12 @@ function binos_exec () {
        enable_debug_for_program

        # run test case with undo in the background
-        /auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh $@ &
+        #/auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh $@ &
+        set_ulimit_unlimited
+#        /nobackup/sriguha/polaris/binos/linkfarm/sdks-aut-host_ut_cge7/iosxe/x86-64-sdk-generic/sdk/sysroots
/x86_64-xesdk-linux/usr/lib/python3.8/site-packages/gen-replay-archive.sh $@ &
+#        export PYTHONPATH="/nobackup/sriguha/polaris/binos/linkfarm/sdks-aut-host_ut_cge7/iosxe/x86-64-sdk-
generic/sdk/sysroots/x86_64-xesdk-linux/usr/lib/python3.8/site-packages"
+        /nobackup/sriguha/mytoolchain/gen-replay-archive.sh $@ &
+        export PYTHONPATH="/ws/afaizalb-bgl/lib"

         BINOS_PID=$!
         PROC_PATH=${FULL_PROC_PATH%/*}
```

- The above patch calls scripts in /nobackup/sriguha/mytoolchain directory. So we need to **first make the following directory. this can be done using the following commands.**

Creating mytoolchain directory and copying gen-replay-archive.sh

```
cd /nobackup/sriguha/
mkdir mytoolchain
cp /auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh /nobackup/sriguha/mytoolchain/gen-replay-
archive.sh
```

- Now we make alterations in the **gen-replay-archive.sh** script in **mytoolchain** directory. This change allows **LR_ARCHIVE flag to attach live-recorder** to the executing test cases. **Replace the entire run_udb() function in the script with this change**

Replacement of run_udb() in gen-replay-archive.sh script

```
run_udb() {
    # Run UDB
    # We turn snapshot target down and reduce the snapshot rate
    # to reduce memory overhead - we aren't reversing so there's
    # no need for snapshots.
    # Finally, we use a special keyfile that doesn't
    # need access to the license server.
    lr_log "Recording test..."

    export UNDO_snapshot_interval=$((2^20))
    export UNDO_keyfile=$UDB_KEY

    local UDB=$(which udb)
    if ! [[ $UDB ]]; then
```

```
        UDB=/auto/binos-tools/bin/udb
    fi

    UDB="/nobackup/sriguha/mytoolchain/live-record_x64"
    lr_log "Starting $UDB"
    $UDB $@ &



    if [[ "$LR_SKIP_DL" == "1" ]]; then
        # $UDB -batch -quiet --defer-recording \
        #      -x $UDB_INIT_SCRIPT \
        #      --max-snapshots 4 \
        #      --args "$@" &
        UDB_PID=$!
    else
        # $UDB -batch -quiet \
        #      -x $UDB_INIT_SCRIPT \
        #      --max-snapshots 4 \
        #      --args "$@" &
            UDB_PID=$!
    fi


    lr_log "Waiting for UDB exit"
    # Install a sigterm handler that knows how to shut
    # down the debuggee
    trap udb_handle_SIGTERM SIGTERM

    #Wait for UndoDB to exit
    if wait $UDB_PID; then
        lr_log "UDB exited normally"
    else
        lr_log "UDB exited uncleanly"

        echo 1 > $EXIT_CODE_FILE
    fi

    # Ensure the debuggee has terminated
    if [[ -e $START_TIME_FILE ]]; then
        local DEBUGGEE_START_TIME=$(<$START_TIME_FILE)
        lr_log "Waiting for debuggee exit (debuggee started at $DEBUGGEE_START_TIME)"
        while true; do
            local J=$(cut -f 22 -d ' ' /proc/$(<$PID_FILE)/stat || true)
            lr_log "PID $(<$PID_FILE) start time = $J"
            if [[ $J != $DEBUGGEE_START_TIME ]]; then
                # J is empty (because the file is missing)
                # or is different because the PID has been recycled.
                # Either way, the debuggee exited.
                break
            fi
            sleep 1
        done
    else
        lr_log "Warning: No start time file $START_TIME_FILE"
    fi

    lr_log "Debuggee exited"
}
```

- **The above script tries to attach live-recorder_x64 from mytoolchain directory. So we need to have the same in mytoolchain directory as well. For this please refer this documentation.**

Up till now gmk -f command with LR_ARCHIVE flag must generate undo recordings in the said location. Now we need a similar functionality in LR_MANUAL flag as well.

- To get the LR_MANUAL functionality, we need to make some changes in the **/nobackup/sriguha/polaris/binos/infra/orchestrator/sh /launch_dynamic.sh script**. We need to add an addition elif condition below the LR_ARCHIVE condition:
- The final **binos_exec()** function looks something like this : (after adding the LR_MANUAL Flag)

final binos_exec function after change

```
function binos_exec () {

    fix_silent $@
    # Setup gdb if run via debug dynamic script
    if [[ "$GDB_PROG" == "$PROG_NAME" ]]; then
        export GDB=$GDB
    else
        export -n GDB
    fi

    ip_assign_init

    trap binos_exec_cleanup SIGHUP SIGINT SIGTERM SIGKILL
    trap port_ip_cleanup EXIT

    pre_init

    # Check if program is an executable or a script. The loader is not needed
    # for scripts.
    PROG_FILETYPE=$(file --mime-type -bL "$1" | sed 's#/.*##g')
    if [ "$PROG_FILETYPE" == "text" ]; then
        export LOADER=""
        export GDB=""
        export COVERAGE_TOOL="$CUSTOM_COVERAGE_TOOL"
    fi

# when RUN_FOREGROUND is set, the process on host_ut keeps control of console
#  this is simply done by not going in background (i.e.: removing the "&" at
#  the end of the command line)
    local rc

    # We can only LR things which have properly-configured RPATH
    # If not, ldd won't be able to find
    if ldd $1 | grep -q "not found"; then
       echo "Cannot record $(basename $1)"
    else
       CAN_LR=1
    fi

    # start execute test case
    if [ "$GDB" ]; then
        $CUSTOM_PREFIX $STRACE $GDB --args $LOADER "$@"

    elif [ "$LR_ARCHIVE" ] && [ "$CAN_LR" == "1" ]; then

        # introduce LR_MANUAL as 1 in or
        # check whether to enable debug for this MUT
        enable_debug_for_program

        binos_echo "SRIJEET: after LR_ARCHIVE if condt."
        binos_echo "$LR_ARCHIVE"

        # run test case with undo in the background
        set_ulimit_unlimited
        #/auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh $@ &
#         /nobackup/sriguha/polaris/binos/linkfarm/sdks-aut-host_ut_cge7/iosxe/x86-64-sdk-generic/sdk/sysroots
/x86_64-xesdk-linux/usr/lib/python3.8/site-packages/gen-replay-archive.sh $@ &
#         export PYTHONPATH="/nobackup/sriguha/polaris/binos/linkfarm/sdks-aut-host_ut_cge7/iosxe/x86-64-sdk-
generic/sdk/sysroots/x86_64-xesdk-linux/usr/lib/python3.8/site-packages"
        /nobackup/sriguha/mytoolchain/gen-replay-archive.sh $@ &
        export PYTHONPATH="/ws/afaizalb-bgl/lib"

        BINOS_PID=$!
        PROC_PATH=${FULL_PROC_PATH%/*}
        PROC=${FULL_PROC_PATH##${PROC_PATH}/}
        binos_echo ${PROC}, ${FULL_PROC_PATH}
        PID_FILE=pid.${BINOS_PID}.___SWIFT_P_PROC___${PROC}__SWIFT_M_MUT__${PROG_NAME}
        rm -f `pwd`/${PID_FILE}
        binos_echo "${PROG_NAME}:${FULL_PROC_PATH}:${RESOLVED_PROCESS}" > `pwd`/${PID_FILE}

        # Get the parent process id and process group id. If they are equal
        # then that is the group id of the top-level test. That group
        # id can be used to cleanup any forked children of the test
        local psinfo=( $(ps xo pid,ppid,pgid | grep "^\s*$BINOS_PID\s") )
        if [ "${psinfo[1]}" == "${psinfo[2]}" ]; then
            BINOS_PGID=${psinfo[2]}
            BINOS_PGNAME=`ps --no-headers -p $BINOS_PGID | awk '{print $4}'`
            BINOS_GROUP_CLEANUP=1
            binos_echo "SWIFT process group: $BINOS_PGID"
        fi

        trap "kill_module_under_test \"${BINOS_PGNAME}\" ${BINOS_PID}" SIGHUP SIGINT SIGTERM SIGKILL
        wait $BINOS_PID
        rc=$?
        binos_echo "wait returned $rc"

    #SRIJEET: Added elif condition below
    elif [ "$LR_MANUAL" ]; then

        # introduce LR_MANUAL as 1 in or
        # check whether to enable debug for this MUT
        #enable_debug_for_program

        binos_echo "SRIJEET: after LR_MANUAL if condt."
        binos_echo "$LR_MANUAL"

        # run test case with undo in the background
        #set_ulimit_unlimited
        #/auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh $@ &
#        /nobackup/sriguha/polaris/binos/linkfarm/sdks-aut-host_ut_cge7/iosxe/x86-64-sdk-generic/sdk/sysroots
/x86_64-xesdk-linux/usr/lib/python3.8/site-packages/gen-replay-archive.sh $@ &
#        export PYTHONPATH="/nobackup/sriguha/polaris/binos/linkfarm/sdks-aut-host_ut_cge7/iosxe/x86-64-sdk-
generic/sdk/sysroots/x86_64-xesdk-linux/usr/lib/python3.8/site-packages"
         /nobackup/sriguha/mytoolchain/gen-replay-archive.sh $@ &
        export PYTHONPATH="/ws/afaizalb-bgl/lib"
        # $UDB $@ &

        # /nobackup/sriguha/mytoolchain/live-record_x64 --pid $!

        BINOS_PID=$!
        PROC_PATH=${FULL_PROC_PATH%/*}
        PROC=${FULL_PROC_PATH##${PROC_PATH}/}
        binos_echo ${PROC}, ${FULL_PROC_PATH}
        PID_FILE=pid.${BINOS_PID}.___SWIFT_P_PROC___${PROC}__SWIFT_M_MUT__${PROG_NAME}
        rm -f `pwd`/${PID_FILE}
        binos_echo "${PROG_NAME}:${FULL_PROC_PATH}:${RESOLVED_PROCESS}" > `pwd`/${PID_FILE}

        # Get the parent process id and process group id. If they are equal
        # then that is the group id of the top-level test. That group
        # id can be used to cleanup any forked children of the test
        local psinfo=( $(ps xo pid,ppid,pgid | grep "^\s*$BINOS_PID\s") )
        if [ "${psinfo[1]}" == "${psinfo[2]}" ]; then
            BINOS_PGID=${psinfo[2]}
            BINOS_PGNAME=`ps --no-headers -p $BINOS_PGID | awk '{print $4}'`
            BINOS_GROUP_CLEANUP=1
            binos_echo "SWIFT process group: $BINOS_PGID"
        fi

        trap "kill_module_under_test \"${BINOS_PGNAME}\" ${BINOS_PID}" SIGHUP SIGINT SIGTERM SIGKILL
        wait $BINOS_PID
        rc=$?
        binos_echo "wait returned $rc"
    #SRIJEET : till here added

    else
        if [ "$RUN_FOREGROUND" ]; then
            $CUSTOM_PREFIX $STRACE $COVERAGE_TOOL $LOADER "$@"
            rc=$?
        else
            # check whether to enable debug for this MUT
            enable_debug_for_program
            $CUSTOM_PREFIX $STRACE $COVERAGE_TOOL $LOADER "$@" &
            BINOS_PID=$!
            PROC_PATH=${FULL_PROC_PATH%/*}
            PROC=${FULL_PROC_PATH##${PROC_PATH}/}
            log_debug "${PROC}, ${FULL_PROC_PATH}"
            PID_FILE=pid.${BINOS_PID}.___SWIFT_P_PROC___${PROC}__SWIFT_M_MUT__${PROG_NAME}
            rm -f `pwd`/${PID_FILE}
            echo "${PROG_NAME}:${FULL_PROC_PATH}:${RESOLVED_PROCESS}" > `pwd`/${PID_FILE}

            # Get the parent process id and process group id. If they are equal
            # then that is the group id of the top-level test. That group
            # id can be used to cleanup any forked children of the test
            local psinfo=( $(ps xo pid,ppid,pgid | grep "^\s*$BINOS_PID\s") )
            if [[ "${psinfo[1]}" == "${psinfo[2]}" && ! -z "${psinfo[2]}" ]]; then
                BINOS_PGID=${psinfo[2]}
                BINOS_PGNAME=`ps --no-headers -p $BINOS_PGID | awk '{print $4}'`
                BINOS_GROUP_CLEANUP=1
                log_debug "SWIFT process group: $BINOS_PGID"
            fi

            trap "kill_module_under_test \"${BINOS_PGNAME}\" ${BINOS_PID}" SIGHUP SIGINT SIGTERM SIGKILL
            wait $BINOS_PID
            rc=$?
            log_debug "wait returned $rc"
        fi
    fi

    cleanup_at_binos_exec_exit $rc ${BINOS_PGNAME}
    return $rc
}
```

This will provide the functionality to generate the undo recordings using LR_MANUAL flag.

Issues and Debugging Efforts

The above additions and patches creates the undo recordings but does not pass all the test cases. The individual test cases pass but they fail eventually. The logs for running the test cases are :

Logs for running with LR_MANUAL flag

```
[2022/07/20 01:09:32.020] Sanity test starting
[2022/07/20 01:09:32.020] SRIJEET: printing args :
[2022/07/20 01:09:32.020] Namespace(args=' ', chasfs='../../odm_test_chasfs.txt', command='./linkfarm/aut-
host_ut_cge7/usr/binos/bin/odm_test_multi_dynamic', coverage='../../_gen_coverage_aut-host_ut_cge7
/odm_test___odm_test_1___multi', cyan_modules=None, cyan_plat=None, dryrun=0, log='./mgmt/odm/test
/_gen_swift_aut-host_ut_cge7/sanity_test_odm_test___odm_test_1___multi.log', loops=1, multi=True,
mut='odm_test', nocoverage=False, output='./mgmt/odm/test/_gen_swift_aut-host_ut_cge7
/odm_test___odm_test_1___multi', root='/nobackup/sriguha/polaris/binos', solo=None, suppressions=None,
tag='odm_test_1', timeout=600, valgrind=None)
[2022/07/20 01:09:32.020] SRIJEET: inside enable_undo_workflow() in swift_undo_enabler.py script
[2022/07/20 01:09:32.020] Undo turned off in the current workflow ...
[2022/07/20 01:09:32.020] Sanity test starting (1): /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_prog_bin-
aut-host_ut_cge7/odm_test_multi_dynamic --- ---b ---n ---s ---t odm_test_1 ---x ./xunit ---o ./logs ----
coverage ../../_gen_coverage_aut-host_ut_cge7/odm_test___odm_test_1___multi/mut
[2022/07/20 01:09:32.020] keepalive file ./mgmt/odm/test/_gen_swift_aut-host_ut_cge7
/odm_test___odm_test_1___multi/logs/keepalive
SRIJEET: start of launch_dynamic.sh
SRIJEET: start of launch_dynamic.sh
22/07/20 06:39:32 *T*[@/nobackup/sriguha/polaris/binos/infra/cyan/lua/utility/cyanTarget.lua:16]Initialize cyan
target
22/07/20 06:39:32 *I*[@/nobackup/sriguha/polaris/binos/infra/cyan/lua/utility/cyanTarget.lua:55]Apply DB for
host, version NO_VERSION, bay 0, launched by /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_prog_bin-aut-
host_ut_cge7/odm_test_multi_dynamic
SRIJEET: after LR_MANUAL if condt.
1
torchd, /nobackup/sriguha/polaris/binos/./linkfarm/aut-host_ut_cge7/usr/binos/bin/torchd
gen-replay-archive.sh is being called
Arguments : /nobackup/sriguha/polaris/binos/./linkfarm/aut-host_ut_cge7/usr/binos/bin/torchd --- ---b ---n ---s
---t odm_test_1 ---x ./xunit ---o ./logs ----coverage ../../_gen_coverage_aut-host_ut_cge7
/odm_test___odm_test_1___multi/mut --- ---m
SWIFT process group: 2717369
{
    "source": "com.cisco.polaris.undo",
    "dataVolume": "small",
    "startTime": 1658279372000,
    "dataKey": "sriguha_1658279372000",
    "data": {
        "dataVersion": 1,
        "recordingPath": "unknown",
        "tool": "gen_replay_archive",
        "userId": "sriguha",
        "server": "bgl-ads-5285",
        "program": "torchd",
        "branch": "polaris_dev",
        "progPath": "/nobackup/sriguha/polaris/binos/./linkfarm/aut-host_ut_cge7/usr/binos/bin/torchd",
        "os": "XE"
    },
    "event": "swift_usage"
}
[LR_ARCHIVE:2717469:torchd] Preparing environment
[LR_ARCHIVE:2717469:torchd] Processing parameters
[LR_ARCHIVE:2717469:torchd] LR_ALWAYS_SAVE = 0
[LR_ARCHIVE:2717469:torchd] LR_ARCHIVE_SOURCE = 1
[LR_ARCHIVE:2717469:torchd] LR_SKIP_DL = 1
[LR_ARCHIVE:2717469:torchd] Generating paths
[LR_ARCHIVE:2717469:torchd] Preparing to record
Executing ------------------- run_udb-----------------
[LR_ARCHIVE:2717469:torchd] Recording test...
which: no udb in (/nobackup/sriguha/polaris/binos/linkfarm/host/bin:/router/bin:/bin:/usr/bin:/sbin:/usr/sbin)
[LR_ARCHIVE:2717469:torchd] Starting /nobackup/sriguha/mytoolchain/live-record_x64
[LR_ARCHIVE:2717469:torchd] Waiting for UDB exit
live-record: Termination recording will be written to /nobackup/sriguha/polaris/binos/mgmt/odm/test
```

```
/_gen_swift_aut-host_ut_cge7/odm_test___odm_test_1___multi/torchd-2717783-2022-07-20T06-39-38.919.undo
live-record: Maximum event log size is 1G
cdef failed. Ignored ...nos/linkfarm/aut-host_ut_cge7/lua/ffi_utils/ffi_extn.lua:781: '<identifier>' expected
near '<eof>'
orchestrator_sb.lua: ORCH2.0 ORCH_APP is: 'swift'
orchestrator_app.lua: ORCH2.0 ORCH_APP is: 'swift'
      INFO :  Loading Application 'swift'
ffi.load AUT_STUB is        table: 0x40f4b348
INFO: BP inited: false
(debug) LUA_PATH/LUA_CPATH are adjusted (PD): aut-host_ut_cge7
(debug) PI_LIB_FARM: /nobackup/sriguha/polaris/binos/linkfarm/host_ut_cge7/usr/binos/lib64
(debug) PI_BIN_FARM: /nobackup/sriguha/polaris/binos/linkfarm/host_ut_cge7/usr/binos/bin
(debug) PD_LIB_FARM: /nobackup/sriguha/polaris/binos/linkfarm/aut-host_ut_cge7/usr/binos/lib64
(debug) PD_BIN_FARM: /nobackup/sriguha/polaris/binos/linkfarm/aut-host_ut_cge7/usr/binos/bin
(error) Cannot find target32 for: host_ut_cge7
(debug) LIB Search Path: /nobackup/sriguha/polaris/binos/linkfarm/host_ut_cge7/usr/binos/lib64:/nobackup/sriguha
/polaris/binos/linkfarm/aut-host_ut_cge7/usr/binos/lib64
(debug) BIN Search Path: /nobackup/sriguha/polaris/binos/linkfarm/host_ut_cge7/usr/binos/bin:/nobackup/sriguha
/polaris/binos/linkfarm/aut-host_ut_cge7/usr/binos/bin
(debug) parse_config_file
(info) Skipping app_init odm_test
SWIFT_MUT_DEP set "$SWIFT_MUT_DEP odm_test chasfs crimson_ios mock_interpose orch_assist orchestrator_lib
tdllib"
SWIFT_MUT_DEP_FFI set "$SWIFT_MUT_DEP_FFI odm_test_prog_ffi_binding_gen.lua chasfs_lib_ffi_binding_gen.lua
crimson_ios_lib_ffi_binding_gen.lua mock_interpose_lib_ffi_binding_gen.lua orch_assist_lib_ffi_binding_gen.lua
orchestrator_lib_lib_ffi_binding_gen.lua tdllib_lib_ffi_binding_gen.lua"
Coro for Suite... odm_test_1
Suite Policy start odm_test_1
Started suite odm_test_1
Waiting for suite done 1 1 0
failed to get q entry queue_get_entry QUEUE_TIMED_OUT Suite_Done_Q

Waiting for suite done 1 1 0
failed to get q entry queue_get_entry QUEUE_TIMED_OUT Suite_Done_Q

Waiting for suite done 1 1 0
failed to get q entry queue_get_entry QUEUE_TIMED_OUT Suite_Done_Q

Waiting for suite done 1 1 0
failed to get q entry queue_get_entry QUEUE_TIMED_OUT Suite_Done_Q

Waiting for suite done 1 0 0
Done all suites table: 0x40ccdce8 table: 0x40ccdd10
Found test report nil
Test report done

============================================================
Summary of the completed test suite :'odm_test_1'

TestCase          Result          Elapsed Time    Reason
___________________________________________________________________
perform_test_case FAILED          0.00 ms         odm_test1_localrp001:Client Aborted!
____________________________________________________________
Summary - Total Tests: 1, Total Elapsed Time: 48637.97 ms
Passed Tests: 0, Failed Tests: 1, Skipped Tests: 0, Runtime Errors: 0

Failed Tests : {
 "odm_test_1.perform_test_case"
}

Log file is ./logs/orch-odm_test_1-2022-7-20-06:39:41-log.txt

============================================================
------------------------------------------------------------
Final summary - Total Tests: 1, Total Elapsed Time: 48637.97 ms
Passed Tests: 0, Failed Tests: 1, Skipped Tests: 0, Runtime Errors: 0

Orchestrator Completed
Logs location: /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-host_ut_cge7
/odm_test___odm_test_1___multi/logs
```

```
Sanity Test Failed:
Failed tests {
  "odm_test_1.perform_test_case"
}
live-record: Saving to '/nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-host_ut_cge7
/odm_test___odm_test_1___multi/torchd-2717783-2022-07-20T06-39-38.919.undo'...
live-record: Termination recording written to /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-
host_ut_cge7/odm_test___odm_test_1___multi/torchd-2717783-2022-07-20T06-39-38.919.undo
live-record: Detaching...
[LR_ARCHIVE:2717469:torchd] UDB exited uncleanly
[LR_ARCHIVE:2717469:torchd] Warning: No start time file /nobackup/sriguha/polaris/binos/mgmt/odm/test
/_gen_swift_aut-host_ut_cge7/odm_test___odm_test_1___multi/lr-working-dir-dnpMHhKtvy.tmp/start_time.txt
[LR_ARCHIVE:2717469:torchd] Debuggee exited
[LR_ARCHIVE:2717469:torchd] No replay archive generated
[LR_ARCHIVE:2717469:torchd] Cleaning up working dir /nobackup/sriguha/polaris/binos/mgmt/odm/test
/_gen_swift_aut-host_ut_cge7/odm_test___odm_test_1___multi/lr-working-dir-dnpMHhKtvy.tmp...
wait returned 1
WARNING: Process 'odm_test_multi_' (or one of its child processes) might have crashed.
WARNING: Core file if generated can be located at /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-
host_ut_cge7/odm_test___odm_test_1___multi directory.
ERROR: ip address cleanup error with return code 1
***WARNING: NODE: 1, MUT_TAG: odm_test___odm_test_1___multi, or CONTEXT: _BINOS_FRU_RP_0_0 not holding IP
address
[2022/07/20 01:10:35.020] Sanity test duration: 62 secs
[2022/07/20 01:10:35.020] Sanity test failed with 1
```

From the above logs we can make certain observations :

-
  ```
  UDB exited uncleanly
  Warning: No start time file /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-host_ut_cge7
  /odm_test___odm_test_1___multi/lr-working-dirdnpMHhKtvy.tmp/start_time.txt
  Debuggee exited
  No replay archive generated
  Cleaning up working dir /nobackup/sriguha/polaris/binos/mgmt/odm/test/_gen_swift_aut-host_ut_cge7
  /odm_test___odm_test_1___multi/lr-working-dir-dnpMHhKtvy.tmp...
  ```
- Corresponding from the code we can see that :
  - UDB should never have started so the log that **UDB exited uncleanly should not be present**
  - Replay archive is generated only by UDB so that too should not have **generated and thus the logs should not be there.**

-
  ```
  SOURCE_LIST_FILE=$WORKING_DIR/sources.txt
  EXIT_CODE_FILE=$WORKING_DIR/exit_code.txt
  PRETTY_NAME_FILE=$WORKING_DIR/pretty_name.txt
  RECORDING_FILE=$WORKING_DIR/recording.undo
  PID_FILE=$WORKING_DIR/pid.txt
  START_TIME_FILE=$WORKING_DIR/start_time.txt
  #these files have been initialized for UDB and have no work in live-recorder. (these lines are present in
  the gen-replay-archive.sh)
  ```
- Corresponding from the logs we can see that :
  - The **existance and the contents of the files are checked throughout the code. Since these files have only been initialized and not created so the checks give false.**
  - Since the launch_dynamic is built such that if the **undo exit is not successful then the test cases are reported as false.**
  - Since the live-recorder is correctly working and UDB has never started, **UDB can never exit correctly, so naturally the files are not correctly created, so the exit status is false.**

Conclusions

- Since the gen-replay-archive.sh has been taken from the **/auto/binos-tools/bin/binos-tools/undo-utils/gen-replay-archive.sh** which has been made exclusively for attaching UDB to the executing test cases. **The current gen-replay-archive.sh also does all the initialization and checks for the attaching and clean exit of the UDB to check if the test cases are working fine.**
- Since the gen-replay-archive.sh never starts UDB because it starts live-recorder, **the files are never created, so the checks for UDB naturally fails.**
- Since the UDB clean exit is not detected, the test cases fail. But actually the test cases have actually worked fine and live-recorder have generated the recordings.
- **We need to find a way to prevent the gen-replay-archive from reporting an unclean exit of UDB to launch_dynamic.sh.**

Further Additions to Current Workspace

- We prevent the gen-replay-archive.sh from reporting wrong values. **So that is why we make a small change in the gen-replay-archive.sh :**

changes in gen-replay-archive.sh at the bottom

```
if [[ -e $EXIT_CODE_FILE ]]; then
    rc=$(cat $EXIT_CODE_FILE)
else
    rc=0
        #note only the above value has changed from 1 to 0
fi
```

This change prevents the script from returning a error value to launch_dynamic.sh file..

- **These passes all the test cases and generates undo recordings and they are stored in the respective directories.**

Further Work

- **Create a separate gen-replay-archive to be called exclusively by the LR_MANUAL flag.**
- **Remove all UDB initializations and checks so that the UDB checks never occur.**
- **Update the launch_dynamic.sh for the above.**