Office of the Director of National Intelligence

STONESOUP

Securely Taking On Software of Uncertain Provenance

Intelligence Advanced Research Projects Activity

IARPA

BE THE FUTURE

Leading Intelligence Integration

STONESOUP Phase 3 Test and Evaluation Execution and Analysis System (TEXAS)

Communications API User Guide

12 December 2014

This report was prepared by TASC, Inc., Ponte Technologies LLC, and i_SW LLC. Supported by the Intelligence Advanced Research Projects Activity (IARPA), Research Operational Support Environment (ROSE) contract number 2011-110902-00005-002. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA or the U.S. Government.

Table of Contents

1 **OVERVIEW** **1**
1.1 SCOPE 1
1.2 TEST CASE FLOW 1
2 **COMMUNICATION SERVER** **3**
2.1 PROTOCOL 3
2.2 HTTP HEADER INFORMATION 3
2.2.1 "CONTENT-TYPE" HEADER 3
2.2.2 "ACCEPT" HEADER 4
2.2.3 "FROM" HEADER 4
2.3 ACTIONMESSAGETYPE 4
2.3.1 XML SCHEMA 5
2.3.2 JSON SCHEMA 5
2.3.3 URL 6
2.3.4 CLIENT 6
2.4 TECHNICALIMPACTMESSAGETYPE 7
2.4.1 XML SCHEMA 7
2.4.2 JSON SCHEMA 7
2.4.3 URL 8
2.4.4 CLIENT 8
2.5 LOGMESSAGETYPE 9
2.5.1 XML SCHEMA 9
2.5.2 JSON SCHEMA 9
2.5.3 URL 9
2.5.4 CLIENT 10
2.6 RESULTMESSAGETYPE 10
2.6.1 XML SCHEMA 11
2.6.2 JSON SCHEMA 11
2.6.3 URL 11
2.6.4 CLIENT 12
2.7 WEAKNESSMESSAGETYPE 13
2.7.1 XML SCHEMA 13
2.7.2 JSON SCHEMA 13
2.7.3 URL 14
2.7.4 CLIENT 14
2.8 RESPONSEMESSAGETYPE 14
2.8.1 XML SCHEMA 14
2.8.2 JSON SCHEMA 15
2.8.3 HTTP STATUS CODES 16
3 **ANALYSIS** **17**
3.1 ENVIRONMENT VARIABLES 17
3.2 ANALYZE.SH 20
3.2.1 PARAMETERS 20

3.3 PERFORMER_ANALYZE.SH 20
3.3.1 GENERAL GUIDANCE 20
3.3.2 FUNCTIONS 21
3.3.2.1 performer_set_flags 21
3.3.2.2 performer_build 21
3.3.2.3 performer_report 22

4 EXECUTION 23
4.1 ENVIRONMENT VARIABLES 24
4.2 EXECUTE.SH 25
4.2.1 PARAMETERS 25
4.3 PERFORMER_EXECUTE.SH 26
4.3.1 GENERAL GUIDANCE 27
4.3.2 FUNCTIONS 27
4.3.2.1 performer_execute 27
4.3.2.2 Signature 27
4.3.3 TIMEOUT REQUIREMENTS 27
4.3.4 REQUIRED MESSAGES 28
4.3.5 OPTIONAL MESSAGES 28
4.4 PERFORMER_SETUP.SH 28
4.4.2 OPTIONAL MESSAGES 29
4.5 PERFORMER_TEARDOWN.SH 29
4.5.1 OPTIONAL MESSAGES 29

5 INSTALLATION OF INTEGRATION 31

APPENDIX A: XML SCHEMA 1

APPENDIX B: JSON SCHEMAS 1
ACTION.JSON 1
IMPACT.JSON 1
LOG.JSON 2
RESULT.JSON 2
WEAKNESS.JSON 3

APPENDIX C: ANALYZE.SH 1

APPENDIX D: EXECUTE.SH 1

APPENDIX E: UTILS.SH 1

APPENDIX F: SAMPLE PERFORMER_ANALYZE.SH 1

APPENDIX G: SAMPLE PERFORMER_EXECUTE.SH 1

APPENDIX H: ACRONYMS 1

List of Figures

Figure 1: TEXAS Analysis Workflow....................17
Figure 2: TEXAS Execution Workflow 23

List of Tables

Table 1: Allowable Content-Type Header Values and Corresponding Message Encoding 3
Table 2: Allowable Accept Header Values and Corresponding Return Encoding 4
Table 3: Performer Controlled Environment Variables for Analysis 17
Table 4: Analysis Location Environment Variables 18
Table 5: Environment Variables Modified During Analysis 19
Table 6: performer_build Parameters 21
Table 7: Performer Controlled Environment Variables For Execution 24
Table 8: Execution Location Environment Variables 24
Table 9: Environment Variables Modified During Execution 25
Table 10: performer_execute Parameters 27

1 Overview

The Test & Evaluation eXecution and Analysis System (TEXAS) was designed to test a Performer technology's ability to detect and mitigate software vulnerabilities and exploit through static analysis and run time countermeasures. To accomplish this, the TEXAS software must interact with the Performer technology such that it can communicate required information to the Performer technology for analysis and execution, as well as tell it when it should be invoked on that information and be able to inform TEXAS of any findings. These interaction points between the Performer technology and the TEXAS system are collectively known as the Communications API. Implementation of the Communications API by a Performer technology allows it to be effectively managed by the TEXAS system during the execution of a test case.

1.1 Scope

This scope of this document includes the integration points between Performer technology and the Test & Evaluation eXecution and Analysis System (TEXAS).

1.2 Test Case Flow

Execution of test cases occurs in three main steps: Analysis, Execution, and Scoring. Performer technology must integrate with TEXAS software so that it can be invoked during the Analysis and Execution steps to process and protect the test program and the host environment.

In the Analysis step, the source code or binary of a program is analyzed looking for vulnerabilities and applying any mitigation necessary to harden the binary against those vulnerabilities. The output of the Analysis phase is a binary executable with hardening or identified vulnerabilities. After the Analysis step has been successfully run, the Execution and Scoring steps are run for each I/O pair defined in the test case's metadata. Execution involves invoking the binary created in Analysis with known inputs (both benign and exploiting). Performer technology may also monitor execution of the binary to apply diversification techniques or look for execution patterns indicative of an attack in progress or software vulnerability. The Scoring check is executed immediately following Execution and looks at the environment for known outputs, as defined in the metadata, of the I/O pair that was executed.

The Analysis, Execution, and Scoring steps of a test case are collectively known as a stage. Test cases are run through two stages. Stage 1 occurs without performer technology (only compiling the code) and confirms/documents program execution with benign and exploiting inputs and effectiveness of the fault injection on the base program. If Stage 1 is scored as completely valid, Stage 2 occurs with Performer Technology. Stage 2 tests the efficacy of the Performer Technology in mitigating the injected fault without altering the behavior of the program. Scoring of successful mitigation is an essential part of this operation and will be built in during design and development. The exact requirements for integrating with the TEXAS system as a Performer Technology during each step are described below.

2 Communication Server

The Communication Server is the component of the Communications API responsible for capturing messages sent by the Performer to the TEXAS system and logging them for evaluation in scoring or debugging. The Communications Server is capable of accepting and responding to a number of messaging formats including eXtensible Markup Language (XML) and JavaScript Object Notation (JSON). The exact methods of communication and messages are described below.

2.1 Protocol

The Communication Server is actually an embedded web server that accepts HTTP requests in a RESTful manner. For the purposes of the Communication Server the applicable HTTP VERBS perform the following actions:

- GET – Not Applicable, returns an error message
- POST – Validates the message is syntactically valid and adds it to the logging queue before returning a ResponseMessageType response to the client
- PUT – See POST
- OPTIONS - Returns the allowed HTTP VERBS for the given URL (OPTIONS, POST, and PUT in all cases)

2.2 HTTP Header Information

The Communication Server extracts important data from the HTTP Headers included on an HTTP Request to process the message and determine the best response to send. These HTTP Headers include: content-type, accept, and from. The purpose and use of each of these headers is described below.

2.2.1 "content-type" Header

The "content-type" HTTP header is required. The server uses it to determine the encoding of the message received. The specific allowable values are listed in **Table 1.**

Table 1: Allowable Content-Type Header Values and Corresponding Message Encoding

Content Type	Format of Message
application/json	JSON
text/javascript	JSON
text/xml	XML

Additionally the content-type HTTP header can contain information about the character set used in the encoding; accomplished by appending a ';' and then the charset information to the content type string. For example, 'application/json;charset=utf-8' tells the server the message is JSON and is encoded using a character set of 'utf-8'. The default charset assumed by the server is 'ISO-8859-1', the default for HTTP.

2.2.2 “accept” Header

The “accept” HTTP header is optional. If set, the server uses it to determine the encoding to use for responses. This header can be constructed similarly to the content-type header. Allowable values and the return format are listed in **Table 2.**

Table 2: Allowable Accept Header Values and Corresponding Return Encoding

Content Type	Format of Response Message
application/json	JSON
text/javascript	JSON
text/xml	XML
text/html	HTML

2.2.3 “from” Header

The “from” HTTP header is required and provides the server with information about who sent the request. This information is logged with the message to help in later analysis or debugging of the run. This header should be in the format of an email address that is further constrained for the purposes of this API to being:

```
<user>@<hostname>.local
```

2.3 ActionMessageType

The action message reports an action taken by the performer technology to mitigate or stop a perceived weakness from being exploited by malicious input. Currently, the actions are "Controlled Exit" and "Continued Execution". With an action, the negative technical impact and perceived target weakness may optionally be reported.

It is not required to report an action in this manner. An action may also be reported as part of the result message. This message is useful if multiple actions are taken by the performer technology, as only one result message may be reported.

2.3.1 XML Schema

```xml
<xs:complexType name="ActionMessageType">
    <xs:sequence>
        <xs:element maxOccurs="1" minOccurs="0" name="weakness"
            type="WeaknessMessageType"/>
        <xs:element maxOccurs="1" minOccurs="0" name="impact"
            type="TechnicalImpactMessageType"/>
        <xs:element maxOccurs="unbounded" minOccurs="0"
            name="additional_information" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="behavior" type="ActionEnumType" use="required"/>
</xs:complexType>
<xs:simpleType name="ActionEnumType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="NONE"/>
        <xs:enumeration value="CONTROLLED_EXIT"/>
        <xs:enumeration value="CONTINUED_EXECUTION"/>
        <xs:enumeration value="OTHER"/>
    </xs:restriction>
</xs:simpleType>
```

2.3.2 JSON Schema

```json
{
    "id": "urn:stonesoup:api:communications:action",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API - Action Message",
    "description": "An action taken by the proposed technology.",
    "properties": {
        "behavior": {
            "description": "The behavior of the proposed technology for this action.",
            "type": "string",
            "enum" : [ "NONE",
                "CONTROLLED_EXIT",
                "CONTINUED_EXECUTION",
                "OTHER"
            ]
        },
        "weakness": {
            "$ref": "urn:stonesoup:api:communications:weakness"
        },
        "impact": {
            "$ref": "urn:stonesoup:api:communications:impact"
        },
        "additional_information": {
            "description": "A list additional statements to log with the weakness.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "required": [ "behavior" ],
    "additionalProperties": false
}
```

2.3.3 URL

HTTP PUT or POST Requests with messages of this type can be sent to:

```
http://$SS_COMM_ADDR:$SS_COMM_PORT/action
```

2.3.4 Client

A helper script is provided that allows easy communication with the server through a command line script. The return code of the command will determine the success or failure of communication with the server. This script can be invoked using the following command.

```
usage: commapi_cli action [-h] -u URL [-l LOG_DIR] [-d] [-z] [-st] -b
                              BEHAVIOR [-w--weakness WEAKNESS] [-i IMPACT]
                              [-ai ADDITIONAL [ADDITIONAL ...]]
sends an action json command
optional arguments:
  -h, --help            show this help message and exit
  -u URL, --url URL     URL of the comms api server
  -l LOG_DIR, --logdir LOG_DIR
                        directory for log files
  -d, --debug           run in debug mode with more verbose logs
  -z, --nothing         generate json command but print them them to stdout
                        with out sending them to server
  -st, --stop           Stop on any error
  -b BEHAVIOR, --behavior BEHAVIOR
                        The behavior of the proposed technology for this
                        action
  -w --weakness WEAKNESS
                        An identified weakness in valid json format
  -i IMPACT, --impact IMPACT
                        An identified weakness in valid json format
  -ai ADDITIONAL [ADDITIONAL ...], --additional ADDITIONAL [ADDITIONAL ...]
                        A list additional statements to log with the action.
```

2.4 TechnicalImpactMessageType

The impact message is used to report a negative technical impact identified by the proposed technology. Reporting a negative technical impact is optional, but aids in evaluating the performer technology. A negative technical impact may also be reported as part of the action message, rather than as a separate message.

2.4.1 XML Schema

```
<xs:complexType name="TechnicalImpactMessageType">
    <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="0"
            name="additional_information" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="effect" type="TechnicalImpactEnumType"
        use="required"/>
</xs:complexType>
<xs:simpleType name="TechnicalImpactEnumType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="NONE"/>
        <xs:enumeration value="UNSPECIFIED"/>
        <xs:enumeration value="READ_FILE"/>
        <xs:enumeration value="READ_APPLICATION_DATA"/>
        <xs:enumeration value="GAIN_PRIVILEGES"/>
        <xs:enumeration value="HIDE_ACTIVITIES"/>
        <xs:enumeration value="EXECUTE_UNAUTHORIZED_CODE"/>
        <xs:enumeration value="MODIFY_FILES"/>
        <xs:enumeration value="MODIFY_APPLICATION_DATA"/>
        <xs:enumeration value="BYPASS_PROTECTION_MECHANISM"/>
        <xs:enumeration value="ALTER_EXECUTION_LOGIC"/>
        <xs:enumeration value="UNEXPECTED_STATE"/>
        <xs:enumeration value="DOS_UNCONTROLLED_EXIT"/>
        <xs:enumeration value="DOS_AMPLIFICATION"/>
        <xs:enumeration value="DOS_INSTABILITY"/>
        <xs:enumeration value="DOS_BLOCKING"/>
        <xs:enumeration value="DOS_RESOURCE_CONSUMPTION"/>
    </xs:restriction>
</xs:simpleType>
```

2.4.2 JSON Schema

```
{
    "id": "urn:stonesoup:api:communications:impact",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API - Impact Message",
    "description": "An identified negative technical impact.",
    "properties": {
        "effect": {
            "description": "The negative technical impact.",
            "type": "string",
            "enum" : [ "NONE",
                "UNSPECIFIED",
                "READ_FILE",
                "READ_APPLICATION_DATA",
                "GAIN_PRIVILEGES",
                "HIDE_ACTIVITIES",
                "EXECUTE_UNAUTHORIZED_CODE",
                "MODIFY_FILES",
                "MODIFY_APPLICATION_DATA",
                "BYPASS_PROTECTION_MECHANISM",
                "ALTER_EXECUTION_LOGIC",
                "UNEXPECTED_STATE",
                "DOS_UNCONTROLLED_EXIT",
                "DOS_AMPLIFICATION",
                "DOS_INSTABILITY",
                "DOS_BLOCKING",
                "DOS_RESOURCE_CONSUMPTION"
            ]
        },
        "additional_information": {
            "description": "A list additional statements to log with the weakness.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "required": [ "effect" ],
    "additionalProperties": false
}
```

2.4.3 URL

HTTP PUT or POST Requests with messages of this type can be sent to:

```
http://$SS_COMM_ADDR:$SS_COMM_PORT/impact
```

2.4.4 Client

A helper script is provided that allows easy communication with the server through a command line script. The return code of the command also represents the success or failure of the communication with the server. This script can be invoked using the following command.

```
usage: commapi_cli impact [-h] -u URL [-l LOG_DIR] [-d] [-z] [-st] -e
                              EFFECT [-ai ADDITIONAL [ADDITIONAL ...]]

The negative technical impact.

optional arguments:
  -h, --help            show this help message and exit
  -u URL, --url URL     URL of the comms api server
  -l LOG_DIR, --logdir LOG_DIR
                        directory for log files
  -d, --debug           run in debug mode with more verbose logs
  -z, --nothing         generate json command but print them them to stdout
                        with out sending them to server
  -st, --stop           Stop on any error
  -e EFFECT, --effect EFFECT
                        The behavior of the proposed technology for this
                        action
  -ai ADDITIONAL [ADDITIONAL ...], --additional ADDITIONAL [ADDITIONAL ...]
                        A list additional statements to log with the impact.
```

2.5 LogMessageType

The log message is used to log arbitrary messages from the performer technology during analyze and execute invocations. It is optional to use and results will not be directly used during scoring unless a specific issue arises.

2.5.1 XML Schema

```
<xs:complexType name="LogMessageType">
    <xs:sequence>
        <xs:element maxOccurs="unbounded" minOccurs="1" name="statement"
            type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

2.5.2 JSON Schema

```
{
    "id": "urn:stonesoup:api:communications:log",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API",
    "description": "Information to log.",
    "properties": {
        "statements": {
            "description": "A list statements to log.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "required": [ "statements" ],
    "additionalProperties": false
}
```

2.5.3 URL

HTTP PUT or POST Requests with messages of this type can be sent to:

```
http://$SS_COMM_ADDR:$SS_COMM_PORT/log
```

2.5.4 Client

A helper script is provided that allows easy communication with the server through a command line script. The return code of the command also represents the success or failure of the communication with the server. This script can be invoked using the following command.

```
usage: commapi_cli log [-h] -u URL [-l LOG_DIR] [-d] [-z] [-st] -s
                           STATEMENTS

sends an log json command

optional arguments:
  -h, --help            show this help message and exit
  -u URL, --url URL     URL of the comms api server
  -l LOG_DIR, --logdir LOG_DIR
                        directory for log files
  -d, --debug           run in debug mode with more verbose logs
  -z, --nothing         generate json command but print them them to stdout
                        with out sending them to server
  -st, --stop           Stop on any error
  -s STATEMENTS, --statements STATEMENTS
                        A list statements to log
```

2.6 ResultMessageType

The result message is used to report the final result of an analyze or execute invocation and is a required message. Only one result message should ever be sent during a single workflow. This message indicates the end of reporting for the performer technology, and triggers the message server to shut down. Any following messages are not guaranteed to be successfully processed. The result message includes only a status enumeration an the raw process return code (i.e. 0-255). An execute result may also include the action taken by proposed technology, rather than reporting it as a separate message. An analyze result should not populate that element.

2.6.1 XML Schema

```
<xs:complexType name="ResultMessageType">
    <xs:sequence>
        <xs:element maxOccurs="1" minOccurs="0" name="action"
            type="ActionMessageType"/>
    </xs:sequence>
    <xs:attribute name="status" type="ResultEnumType"
         use="required"/>
    <xs:attribute name="return_code" type="xs:integer"
         use="required"/>
</xs:complexType>
<xs:simpleType name="ResultEnumType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="SUCCESS"/>
        <xs:enumeration value="SKIP"/>
        <xs:enumeration value="TIMEOUT"/>
        <xs:enumeration value="OTHER"/>
    </xs:restriction>
</xs:simpleType>
```

2.6.2 JSON Schema

```
{
    "id": "urn:stonesoup:api:communications:result",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API",
    "description": "Defines the JSON encoding for the messages supported by the
communications API",
    "properties": {
        "status": {
            "description": "The proposed technology status for completion.",
            "type": "string",
            "enum" : [ "SUCCESS",
                "SKIP",
                "TIMEOUT",
                "OTHER"
            ]
        },
        "return_code": {
            "description": "The exit code for the analyze operation (i.e. make/build)
or the execute of the test case application.",
            "type": "integer",
            "minimum": 0,
            "maximum": 255
        },
        "action": {
            "$ref": "urn:stonesoup:api:communications:action"
        }
    },
    "required": [ "status", "return_code" ],
    "additionalProperties": false
}
```

2.6.3 URL

HTTP PUT or POST Requests with messages of this type can be sent to:

```
http://$SS_COMM_ADDR:$SS_COMM_PORT/result
```

2.6.4 Client

A helper script is provided that allows easy communication with the server through a command line script. The return code of command will also represent the success or failure of the communication with the server. This script can be invoked using the following command.

```
usage: commapi_cli result [-h] -u URL [-l LOG_DIR] [-d] [-z] [-st] -s
                              STATUS -r RETURN_CODE [-a ACTION]

sends an result json command

optional arguments:
  -h, --help            show this help message and exit
  -u URL, --url URL     URL of the comms api server
  -l LOG_DIR, --logdir LOG_DIR
                        directory for log files
  -d, --debug           run in debug mode with more verbose logs
  -z, --nothing         generate json command but print them them to stdout
                        with out sending them to server
  -st, --stop           Stop on any error
  -s STATUS, --status STATUS
                        The proposed technology status for completion.
  -r RETURN_CODE, --rcode RETURN_CODE
                        The exit code for the analyze operation (i.e.
                        make/build) or the execute of the test case
                        application.
  -a ACTION, --action ACTION
                        An action taken by the proposed technology."
```

2.7 WeaknessMessageType

The weakness message is used to report an identified or perceived software vulnerability. Reporting a weakness is optional, but aids in evaluating the proposed technology. A weakness may also be reported as part of the action message, rather than as a separate message.

2.7.1 XML Schema

```
<xs:complexType name="WeaknessMessageType">
     <xs:sequence minOccurs="0">
        <xs:element maxOccurs="unbounded" minOccurs="0"
             name="additional_information" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="cwe" type="CWEType" use="optional"/>
    <xs:attribute name="file_name" type="xs:string" use="optional"/>
    <xs:attribute name="function_name" type="xs:string" use="optional"/>
    <xs:attribute name="line_number" type="xs:integer" use="optional"/>
</xs:complexType>
<xs:simpleType name="CWEType">
    <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]{3}-[0-9]{1,3}"/>
    </xs:restriction>
</xs:simpleType>
```

2.7.2 JSON Schema

```
{
    "id": "urn:stonesoup:api:communications:weakness",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API - Weakness Message",
    "description": "An identified weakness.",
    "properties": {
        "cwe": {
            "description": "The CWE or other weakness id.",
            "type": "string",
            "pattern": "[A-Z]{3}-[0-9]{1,3}"
        },
        "file_name": {
            "description": "The file name the weakness was found within.",
            "type": "string"
        },
        "function_name": {
            "description": "The function name the weakness was found within.",
            "type": "string"
        },
        "line_number": {
            "description": "The function name the weakness was found within.",
            "type": "integer",
            "minimum": 1
        },
        "additional_information": {
            "description": "A list additional statements to log with the weakness.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "additionalProperties": false
}
```

2.7.3 URL

HTTP PUT or POST Requests with messages of this type can be sent to:

```
http://$SS_COMM_ADDR:$SS_COMM_PORT/weakness
```

2.7.4 Client

A helper script is provided that allows easy communication with the server through a command line script. The return code of the command also represents the success or failure of the communication with the server. This script can be invoked using the following command.

```
usage: commapi_cli weakness [-h] -u URL [-l LOG_DIR] [-d] [-z] [-st]
                                [-c CWE] [-f FILE_NAME] [-fu FUNCTION_NAME]
                                [-li LINE_NUMBER]
                                [-ai ADDITIONAL [ADDITIONAL ...]]

sends an weakness json command

optional arguments:
  -h, --help            show this help message and exit
  -u URL, --url URL     URL of the comms api server
  -l LOG_DIR, --logdir LOG_DIR
                        directory for log files
  -d, --debug           run in debug mode with more verbose logs
  -z, --nothing         generate json command but print them them to stdout
                        with out sending them to server
  -st, --stop           Stop on any error
  -c CWE, --cwe CWE     The CWE or other weakness id.
  -f FILE_NAME, --file FILE_NAME
                        The file name the weakness was found within.
  -fu FUNCTION_NAME, --function FUNCTION_NAME
                        The function name the weakness was found within.
  -li LINE_NUMBER, --line LINE_NUMBER
                        The line the weakness was found within
  -ai ADDITIONAL [ADDITIONAL ...], --additional ADDITIONAL [ADDITIONAL ...]
                        A list additional statements to log with the weakness.
```

2.8 ResponseMessageType

The response message is used to report the success or failure in processing the received message (i.e. log, result, finish, etc.). This message provides an overall result, the HTTP status code, and an optional message providing any other useful information.

This message exists to support debugging integration with performer technologies. The HTTP status code encapsulated in the headers should be sufficient during runtime.

2.8.1 XML Schema

```
<xs:complexType name="ResponseMessageType">
    <xs:sequence>
        <xs:element maxOccurs="1" minOccurs="0" name="message"
             type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="result" type="ResponseResultEnumType"
        use="required"/>
    <xs:attribute name="code" type="xs:integer" use="required"/>
</xs:complexType>
```

2.8.2 JSON Schema

```
{
  "id" : "urn:stonesoup:api:communications:response",
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "STONESOUP Communications API",
  "description" : "Report success or failure from received message",
  "properties" : {
      "message" : {
          "description" : "The message to send as a response",
          "type" : "string"
      },
      "result" : {
          "description" : "The result of the response",
          "type" : "string",
          "enum" : [ "SUCCESS",
              "FAILURE"
          ]
      },
      "code" : {
          "description" : "HTTP response code",
          "type" : "integer"
      }
  },

  "required" : ["result", "code" ],
  "additionalProperties" : "false"
}
```

2.8.3 HTTP Status Codes

- 201 Created - Message processed and recorded.
- 400 – Bad Message
- 404 – Resource Not Found
- 405 – HTTP Verb is not supported via this API.
- 415 – Content type or Character encoding is not supported.

3 Analysis

The Analysis step of a test case run is where the binary is actually produced and analyzed by performer technology. **Figure 1: TEXAS Analysis Workflow** describes the overall workflow of the TEXAS system during Analysis. Of interest for this document are the actions and decision points that occur during the "Execute Analyze Script" action as these describe the actions and decisions made in the analyze.sh script.

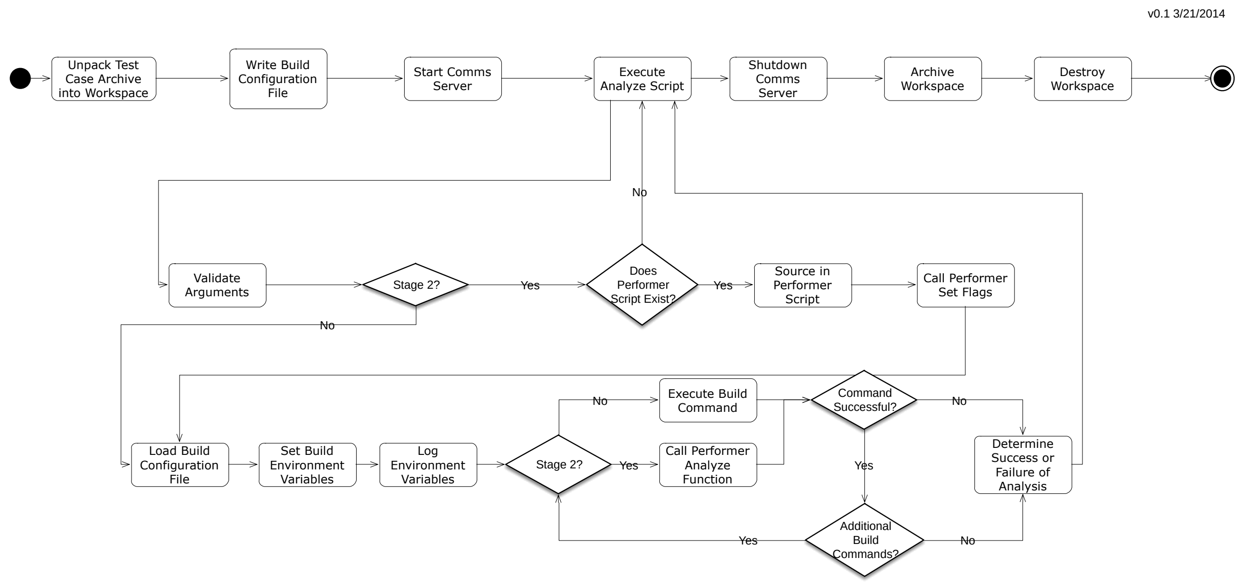
v0.1 3/21/2014
Unpack Test Case Archive into Workspace
Write Build Configuration File
Start Comms Server
Execute Analyze Script
Shutdown Comms Server
Archive Workspace
Destroy Workspace
No
Validate Arguments
Stage 2?
Yes
Does Performer Script Exist?
Yes
Source in Performer Script
Call Performer Set Flags
No
Execute Build Command
No
Command Successful?
No
Load Build Configuration File
Set Build Environment Variables
Log Environment Variables
Stage 2?
Yes
Call Performer Analyze Function
Yes
Determine Success or Failure of Analysis
Yes
Additional Build Commands?
No

Figure 1: TEXAS Analysis Workflow

3.1 Environment Variables

To run some of the commands needed to compile and run the test cases several environment variables need to be set during the testing process so these programs can be compiled and run from semi dynamic locations. Because Performers may need to modify some of the same environment variables for their technology, alternative performer controlled environment variables have been designated to allow modification in a controlled manner. These alternative environment variables are listed in **Table 3: Performer Controlled Environment Variables for Analysis.**

Table 3: Performer Controlled Environment Variables for Analysis

Variable	Description
SS_CC	This variable allows the performer to override what compiler the build process for the test case is using. The default compiler value is 'gcc'.
SS_LNK	This variable allows the performer to override what compiler the build process for the test case is using. The default value is '${SS_CC}'.
SS_LDFLAGS	This variable allows the performer to modify the global LDFLAGS environment variable in a way that will not affect the TEXAS system. The exact structure of the final LDFLAGS variable is described in Table 5: Environment Variables Modified During Analysis.
SS_CFLAGS	This variable allows the performer to modify the global CFLAGS environment variable in a way that

Variable	Description
	will not adversely affect the TEXAS system. The exact structure of the final CFLAGS variable is described in Table 5: Environment Variables Modified During Analysis.
SS_ANT_OPTS	This variable allows the performer to override the compiler or other ANT build options used to build Java test cases much like the SS_CC variable. Build commands are usually structured 'ant SS_ANT_OPTS ….'
SS_LIBS	This variable allows the performer to modify the global LIBS environment variable in a way that will not adversely affect the TEXAS system. The exact structure of the final LIBS variable is described in Table 5: Environment Variables Modified During Analysis.
SS_LDFLAGS	This variable allows the performer to modify the global LDFLAGS environment variable in a way that will not adversely affect the TEXAS system. The exact structure of the final LDFLAGs variable is described in Table 5: Environment Variables Modified During Analysis.

Environment variables that provide information about locations on the system used specifically in the test case metadata are also available. These variables are listed in **Table 4: Analysis Location Environment Variables.**

Table 4: Analysis Location Environment Variables

Name	Description
SS_TC_DEPS	This environment variable holds the absolute directory path where dependencies have been preinstalled. The test user must have read/execute access to the directory.
SS_TC_ROOT	This is the absolute path to the root of the unzipped test case directory on the Test Host. This is required for other environment variables below.
SS_TC_INSTALL	This is the relative path where the program is installed inside of the SS_TC_ROOT directory (i.e. the destination for make install). The absolute path of this directory is $SS_TC_ROOT/$SS_TC_INSTALL. All executable and library objects under this directory are considered valid locations for the weakness to be injected.
SS_TC_DATA	This is the relative path of the current working copy of the iodata directory. The absolute path of this directory is then be created by $SS_TC_ROOT/$SS_TC_DATA
SS_JAVA_MAIN	This environment variable contains a colon-delimited list of main classes used in the input/output pairs of a test case.
SS_JAVA_CLASSPATH	This environment variable contains the runtime class path used in input/output pairs of a test case.
SS_JAVA_EXTRA_CLASSPATH	This environment variable contains any extra classpath information that is not included on the runtime classpath by default.
SS_COMM_ADDR	This environment variable contains the hostname or ip address of the machine running the Communication Server. The default value is 'localhost'
SS_COMM_PORT	This environment variable contains the port on which the Communication Server is listening. The default value is '8886'
SS_LEARNING_ROOT	This environment variable contains an absolute path to the root directory that contains any learning information available for the test case. This directory is writeable and my be persisted depending on the mode of operation the test case is being run in.

Name	Description
STONESOUP_DISABLE_WEAKNESS	This environment variable is included to turn off injected weaknesses during analysis so that programs that call themselves during compilation do not block waiting for input from external sources like sockets or shared memory.

The environment variables listed in **Table 5: Environment Variables Modified During Analysis** are environment variables commonly utilized in the compilation of programs. These environment variables must be modified from their system defaults to support building with dependencies that are installed in non-default folders on the system. These variables should not be directly modified by the Performer but should instead be modified by setting the values of the environment variables listed in **Table 5: Environment Variables Modified During Analysis**. If the performer technology requires changes to any other system wide environment variable please contact the TEXAS development team immediately.

Table 5: Environment Variables Modified During Analysis

Variable	Value
PATH	${SS_BASE_PATH}:${PATH}
CFLAGS	${SS_BASE_CFLAGS} ${SS_CFLAGS}
CPPFLAGS	${SS_BASE_CPPFLAGS}
LIBS	${SS_BASE_LIBS} ${SS_LIBS}
LDFLAGS	${SS_BASE_LDFLAGS} ${SS_LDFLAGS}
PKG_CONFIG_PATH	${SS_BASE_PKGCONFIG}:${PKG_CONFIG_PATH}
SS_BASE_CFLAGS	${SS_BUILD_COMMAND_CFLAGS} -fno-stack-protector
SS_BASE_CPPFLAGS	${SS_BUILD_COMMAND_CPPFLAGS} -I${SS_TC_DEPS}/include
SS_BASE_LIBS_DIR	-L${SS_TC_DEPS}/lib -L${SS_TC_DEPS}/lib64
SS_BASE_LIBS	${SS_BUILD_COMMAND_LIBS}
SS_BASE_LIBS_RPATH	-Wl,-R${SS_TC_DEPS}/lib -Wl,-R${SS_TC_DEPS}/lib64
SS_BASE_PKGCONFIG	${SS_TC_DEPS}/lib:${SS_TC_DEPS}/lib/pkgconfig:${SS_TC_DEPS}/lib64:${SS_TC_DEPS}/lib64/pkgconfig
SS_BASE_PATH	${SS_TC_DEPS}/bin:${SS_TC_DEPS}/sbin
SS_BASE_LDFLAGS	${SS_BUILD_COMMAND_LDFLAGS} ${SS_BASE_LIBS_DIR} ${SS_BASE_LIBS_RPATH} -Wl,-z,execstack
SS_BUILD_COMMAND_LDFLAGS	LDFLAGS from the testcase metadata, otherwise blank
SS_BUILD_COMMAND_CPPFLAGS	CPPFLAGS from the testcase metadata, otherwise blank
SS_BUILD_COMMAND_CFLAGS	CFLAGS from the testcase metadata, otherwise blank
SS_BUILD_COMMAND_LIBS	LIBS from the testcase metadata, otherwise blank

3.2 analyze.sh

The analyze.sh script is the portion of TEXAS software responsible for managing the compilation/analysis portion of a test case. This script actually executes the build commands for the test program to compile it into a binary, either with or without Performer Technology. Performers integrate into this script by writing their own performer_analyze.sh script, detailed in section **performer_analyze.sh.** The full analyze.sh script is available in Appendix C: analyze.sh.

3.2.1 Parameters

```
usage: analyze.sh [OPTIONS] -d COMMAND_CWD [-f COMMAND_FILE]
This script provides an interface through which a target application source or binary
can be analyzed (and created if building the source).
OPTIONS:
   -d      Working directory from which COMMAND_FILE will execute.
   -f      Configuration file containing build commands.  If this is not
           provided, the data is expected to be provided via stdin.

           The configuration format closely follows that of INI, with the
           exception that comments are denoted with '#'.  Below is an
           example:
             # This is a config for build commands
             [buildcommands]
             # Configure the build
             command=./configure --prefix="$SS_TC_ROOT/$SS_TC_INSTALL"
             # Build and then install
             command=make all
             command=make install
   -g      Debug flag.  RESERVED FOR PERFORMER USE.
   -h      Show this message.
   -n      Do nothing.  Parse and print commands.
   -p      Run with performer technology.
   -r      Run in release mode. This mode should represent the most production-
           like version of the performer technology. RESERVED FOR PERFORMER USE.
   -t      Test case name.
   -v      Run in verbose mode.
```

3.3 performer_analyze.sh

The performer_execute.sh script is the main integration point for Performers during the Analysis step in running a test case. Implementing this script in concert with implementation of appropriate messaging to the Communications Server constitutes integration into the Analysis step of a test case in TEXAS.

3.3.1 General Guidance

In order to support correct scoring via standard out and standard error of the run command, performer scripts should not print to standard out or standard error unless the debug flag is enabled. Furthermore, in release mode the performer should operate in as close to the same manner they would expect to deploy into an operational production environment.

3.3.2 Functions

Implementation of this script requires implementation of any required functions called from the analyze.sh script. These functions are all denoted by the prefix 'performer_' in the name of the function. These functions are not called when the analyze.sh script is executed with the –p flag to enable Stage 2.

3.3.2.1 performer_set_flags

The performer_set_flags function is intended to allow the Performer to set any of the environment variables listed in **Table 3: Performer Controlled Environment Variables for Analysis** before they are used in their global counterparts listed in **Table 5: Environment Variables Modified During Analysis.** This function must be implemented, with an empty function representing a valid instance.

3.3.2.1.1 Signature

```
performer_set_flags
```

3.3.2.1.2 Optional Messages

The following messages may be sent to the Communications Server during execution of the performer_set_flags function.

- **LogMessageType** – This message may be sent any time the performer wishes to log something for later debugging or analysis purposes.

3.3.2.2 performer_build

This function is called for every specified build command for the test program.

3.3.2.2.1 Signature

```
performer_build "${COMMAND_EXEC_CWD}" "${BUILD_COMMAND}"
```

Table 6: performer_build Parameters

Parameter	Description
COMMAND_EXEC_CWD	The path the BUILD_COMMAND should be executed in. It is expected that the "any" implementation of this method will change directory to this directory before executing the BUILD_COMMAND
BUILD_COMMAND	This is the actual command to be executed to build the test program (i.e. make install). Implementations may find it useful to execute the change to the COMMAND_EXEC_CWD and the BUILD_COMMAND a sub-process.

3.3.2.2.2 Optional Messages

The following messages may be sent to the Communications Server during execution of the performer_build function.

- **ActionMessageType –** This message should be sent any time the performer takes an action such as mitigating a vulnerability or initiating a controlled exit of the program. Information about the suspected weakness can also be included in this message.

- **LogMessageType** – This message may be sent any time the performer wishes to log something for later debugging or analysis purposes.
- **TechnicalImpactMessageType -** This message may be sent any time a suspected technical impact is detected.
- **WeaknessMessageType** – This message may be sent any time a suspected weakness is discovered. The same information can be conveyed using an ActionMessageType.

3.3.2.3 performer_report

This function is called once after all build commands have been executed.

3.3.2.3.1 Signature

```
performer_report
```

3.3.2.3.2 Required Messages

The following messages must be sent to the Communication Server during execution of the performer_report function.

- **ResultMessageType** – This should be sent as a final method in the performer_report function.

3.3.2.3.3 Optional Messages

The following messages may be sent to the Communications Server during execution of the performer_analyze function.

- **ActionMessageType –** This message should be sent any time the performer takes an action such as mitigating a vulnerability or initiating a controlled exit of the program. Information about the suspected weakness can also be included in this message.
- **LogMessageType** – This message may be sent any time the performer wishes to log something for later debugging or analysis purposes.
- **TechnicalImpactMessageType -** This message may be sent any time a suspected technical impact is detected.
- **WeaknessMessageType** – This message may be sent any time a suspected weakness is discovered. The same information can be conveyed using an ActionMessageType.

4 Execution

The Execution step of a test case run is where the binary created during the Analysis step is actually invoked with benign and exploiting inputs. **Figure 2: TEXAS Execution** describes the overall workflow of the TEXAS system during an Execution step. Of particular interest for this document are the actions and decision points that occur during the Execute Script action as these describe the actions and decisions made in the execute.sh script.

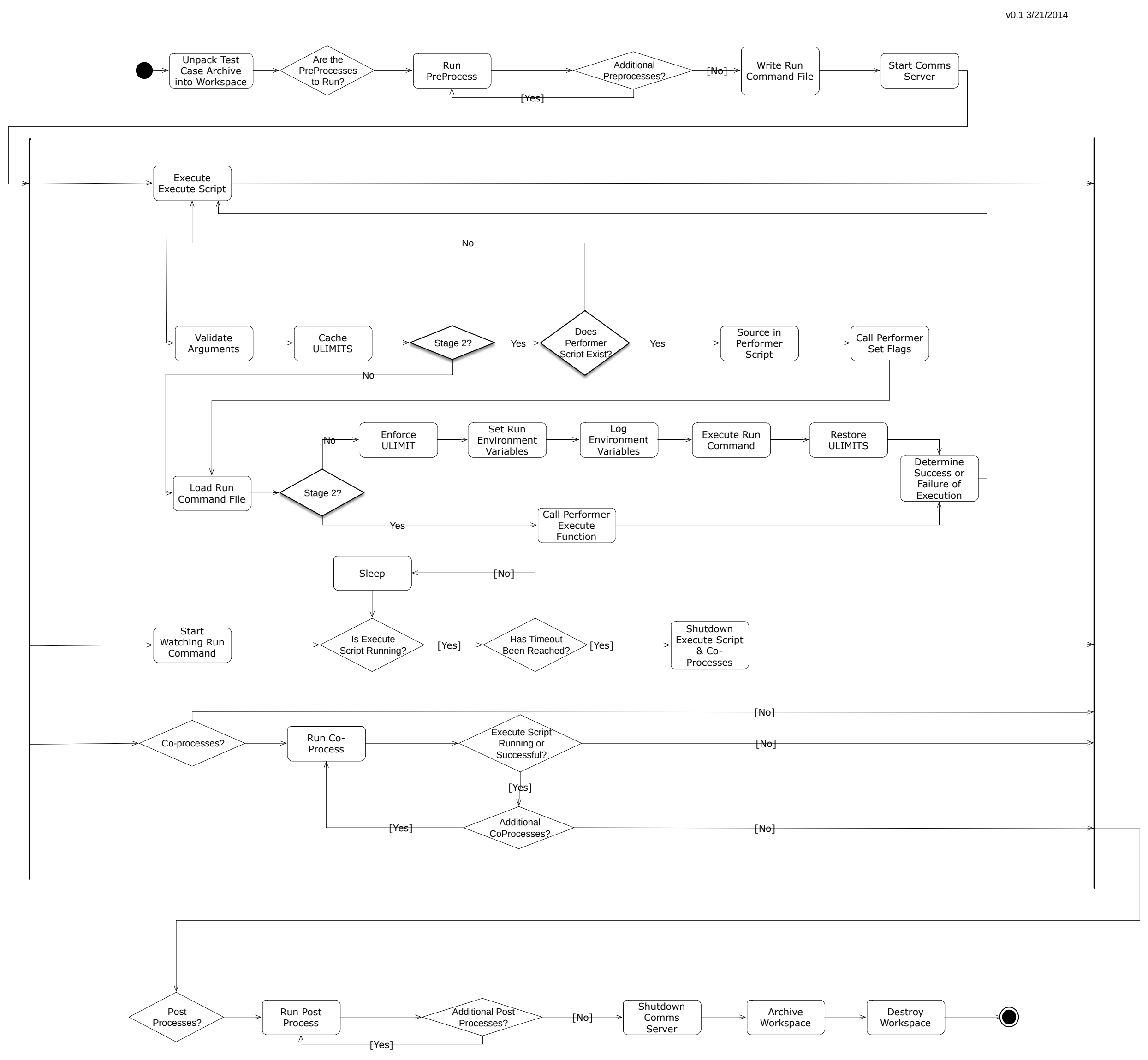

Figure 2: TEXAS Execution Workflow

4.1 Environment Variables

To invoke the test program correctly from within the environment, several environment variables need to be set during the testing process. Because Performers may need to modify some of the same environment variables for their technology, alternative performer controlled environment variables have been designated to allow modification in a controlled manner. These alternative environment variables are listed in Table 7: Performer Controlled Environment Variables For Execution**.**

Table 7: Performer Controlled Environment Variables For Execution

Variable	Description
SS_LD_PRELOAD	This variable allows the performer to add information to the global LD_PRELOAD environment variable without impacting changes made by the TEXAS system.

Like the Analysis step, environment variables that provide information about locations on the system used specifically in the test case metadata are also available. These variables are listed in **Table 8: Execution Location Environment Variables.**

Table 8: Execution Location Environment Variables

Name	Description
SS_TC_DEPS	This environment variable holds the absolute path of the directory where the dependencies have been preinstalled. The test user must have read/execute access to the directory.
SS_TC_ROOT	This is the absolute path to the root of the unzipped test case directory on the Test Host. This is required for other environment variables below.
SS_TC_INSTALL	This is the relative path where the program is installed to in the SS_TC_ROOT directory (i.e. the destination for make install). The absolute path of this directory is $SS_TC_ROOT/$SS_TC_INSTALL. All executable and library objects under this directory are considered valid locations for the weakness to be injected.
SS_TC_DATA	This is the relative path of the current working copy of the iodata directory. The absolute path of this directory would then be created by $SS_TC_ROOT/$SS_TC_DATA
SS_JAVA_MAIN	This environment variable contains a colon-delimited list of main classes used in the input/output pairs of a test case.
SS_JAVA_CLASSPATH	This environment variable contains the runtime class path used in the input/output pairs of a test case.
SS_COMM_ADDR	This environment variable contains the hostname or ip address of the machine running the Communication Server. The default value is ‘localhost’
SS_COMM_PORT	This environment variable contains the port on which the Communication Server is listening. The default value is ‘8886’
SS_LEARNING_ROOT	This environment variable contains an absolute path to the root directory that contains any learning information available for the test case. This directory is writeable and my be persisted depending on the mode of operation the test case is being run in.

The environment variables listed in the **Table 9: Environment Variables Modified During Execution** are environment variables commonly utilized in the compilation of programs. These environment variables must be modified from their system defaults to support building with dependencies installed in non-default folders on the system. These variables should not be directly modified by the Performer but instead modified by setting the values of the environment variables listed in **Table 7: Performer Controlled Environment Variables For Execution**. If the performer technology requires changes to any other system wide environment variable please contact the TEXAS development team.

Table 9: Environment Variables Modified During Execution

Variable	Value
PATH	${SS_TC_PATH}
LD_PRELOAD	${SS_BASE_LD_PRELOAD} ${SS_LD_PRELOAD}
SS_BASE_LD_PRELOAD	${SS_RUN_COMMAND_LD_PRELOAD}
SS_TC_DEPS_PATH	${SS_TC_DEPS}/bin:${SS_TC_DEPS}/sbin
SS_TC_PATH	${SS_TC_ROOT}/${SS_TC_INSTALL}/bin:${SS_TC_ROOT}/${SS_TC_INSTALL}/sbin: ${SS_TC_DEPS_PATH}:${PATH}
SS_TC_LIBDIR_NAME	lib$([[$(uname -m) == x86_64]] && echo 64)

4.2 execute.sh

The execute.sh script is the portion of TEXAS software responsible for actually executing the vulnerable program, with or without the Performer Technology protecting it. Performers actually integrate into this script by writing their own performer_execute.sh script, detailed in **performer_execute.sh.** The full execute.sh script is available in **Appendix D: execute.sh.**

4.2.1 Parameters

```
usage: execute.sh [OPTIONS] -d COMMAND_CWD [-f COMMAND_FILE]
This script provides an interface through which a target application can be executed.
Commands will be evaluated by the SHELL to simulate executing what would otherwise be
provided by a user on the command line.
OPTIONS:
   -d      Working directory from which COMMAND_FILE will execute.
   -f      Configuration file containing run commands.  If this is not
           provided, the data is expected to be provided via stdin.

           The configuration format closely follows that of INI, with the
           exception that comments are denoted with '#'.  Below is an
           example:
             # This is a config for run commands
             [runcommands]
             # Run the target application
             command=$SS_TC_ROOT/$SS_TC_INSTALL/bin/example -h

   -g      Debug flag.  RESERVED FOR PERFORMER USE.
   -h      Show this message.
   -l      Set ULIMITs for the target executable.  This flag may be
           provided multiple times.  In most cases, a number is
           expected, but the special string "ulimited" is also supported.
```

```
            The settings take the format:
              KEY=VALUE
            The following ULIMITs are supported:
              RLIMIT_AS      - Maximum size of virtual memory.
              RLIMIT_NOFILE  - Maximum number of open file descriptors.
              RLIMIT_NPROC   - Maximum number of processes that may be
                               created.
              RLIMIT_CPU     - Maximum CPU time (in seconds).
            Refer to http://linux.die.net/man/2/getrlimit for more
            information of the flags and their effect on the process.
    -n      Do nothing.  Parse and print commands.
    -p      Run with performer technology.
    -r      Run in release mode. This mode should represent the most production-
            like version of the performer technology. RESERVED FOR PERFORMER USE.
    -t      Test case name.
    -u      Run as different user.
    -v      Run in verbose mode.
```

4.3 performer_execute.sh

The performer_execute.sh script is the main integration point for Performers during the Execution step in running a test case. Implementation of this script in concert with implementation of appropriate messaging to the Communications Server constitutes integration into the Execution step of a test case in TEXAS.

4.3.1 General Guidance

In order to support correct scoring via standard out and standard error of the run command, performer scripts should not print to standard out or standard error unless the debug flag is enabled. Furthermore, in release mode the performer should operate in as close to the same manner they would expect to deploy into an operational production environment.

4.3.2 Functions

Implementation of this script requires the implementation of any required functions called from the execute.sh script. These functions are all denoted by the prefix 'performer_' in the name of the function. These functions are not called when the execute.sh script is executed with the –p flag to enable Stage 2.

4.3.2.1 performer_execute

This function provides necessary information to the Performer to actually invoke the test program from the correct folder with the correct command line inputs. The execute.sh script calls this method during Stage 2 when the command would expect to be invoked. Calling out to this function provides the Performer the ability to modify the command to actually invoke their technology before or during the command execution.

4.3.2.2 Signature

```
performer_execute "${COMMAND_EXEC_CWD}" "${BASE_RUN_COMMAND}"
   "${RUN_COMMAND}"
```

Table 10: performer_execute Parameters

Parameter	Description
COMMAND_EXEC_CWD	The path the RUN_COMMAND should be executed in. It is expected that implementation of this method will change directory to this directory before executing the RUN_COMMAND
BASE_RUN_COMMAND	In the event the –u flag is set on invocation of the execute script, this parameter contains the commands necessary to switch execution of the actual RUN_COMMAND to execute as a different user. For example, "-u tm" will result in this parameter being set to "sudo -n -E -u tm -s --"
RUN_COMMAND	This is the actual command that invokes the vulnerable test program as well as any required arguments. Implementations are expected to execute the RUN_COMMAND in conjunction with the BASE_RUN_COMMAND (i.e. eval "${BASE_RUN_COMMAND}" "${RUN_COMMAND}"). Implementations may find it useful to run this command in a sub shell so they can execute other commands after evaluating the RUN_COMMAND and then bubble the return code back up to the calling function.

4.3.3 Timeout Requirements

Execution of a test case is subject to a timeout if the program runs too long. This functionality is supported by use of writing a PID file for the shell executing the run command to a file. This is accomplished through two method calls in the performer_execute method before execution of the run command.

```
# Capture the subshell pid, so we can kill the children later.
# By default, the children of the shell will be killed, but
# the actual shell pid will not receive a kill signal
ensure_shell_pid
printf "%s" "${BASHPID}" > "${KILLTREE_PID_FILE}"
```

This method of timing out the process allows TEXAS to recursively kill the children of the captured PID while not killing the actual shell PID, thus returning control back to the shell where any cleanup or analysis can be completed before the script exits cleanly. If multiprocessing is utilized, the PID captured can be a dummy PID that the script blocks on before cleanup. Writing out the PID and responding to timeouts cleanly is required for integration.

4.3.4 Required Messages

The following messages must be sent to the Communication Server during the execution of the performer_execute function.

- **ResultMessageType** – This should be sent as the final action of the performer_execute function.

4.3.5 Optional Messages

The following messages may be sent to the Communications Server during execution of the performer_execute function.

- **ActionMessageType** – This message should be sent any time the performer takes an action such as mitigating a vulnerability or initiating a controlled exit of the program. Information about the suspected weakness can be included in this message.
- **LogMessageType** – This message may be sent any time the performer wishes to log something for later debugging or analysis purposes.
- **TechnicalImpactMessageType -** This may be sent any time a suspected technical impact is detected.
- **WeaknessMessageType** – This may be sent any time a suspected weakness is discovered. The same information can be conveyed using an ActionMessageType.

4.4 performer_setup.sh

The performer_setup.sh script is a script called directly before any pre-processes are executed. This script is only executed when TEXAS is running with Performer Technology. This script is intended to allow any necessary setup to be completed before the actual execution of the test case. Environment variables listed in **Table 8: Execution Location Environment Variables** are available. The default script is not included in the Appendix section because they are blank.

4.4.2 Optional Messages

The following messages may be sent to the Communications Server during execution of the performer_setup function.

- **LogMessageType** – This message may be sent any time the performer wishes to log something for later debugging or analysis purposes.

4.5 performer_teardown.sh

The performer_setup.sh script is a script called directly after any post-processes are executed. This script is only executed when TEXAS is running with Performer Technology. This script is intended to allow any necessary teardown of the environment to be completed before the scoring and archival of the test case. Environment variables listed in Table 8: Execution Location Environment Variables are available. The default script is not included in the Appendix section because they are blank.

4.5.1 Optional Messages

The following messages may be sent to the Communications Server during execution of the performer_teardown function.

- **LogMessageType** – This message may be sent any time the performer wishes to log something for later debugging or analysis purposes.

5 Installation of Integration

Integrating a new Performer Technology with TEXAS is straightforward. The Performer must implement the performer_analyze.sh and performer_execute.sh scripts. Installation is then achieved by placing those scripts into the same directory the analyze.sh and execute.sh scripts are installed on a Test Host, normally /opt/stonesoup/texas_scripts. The scripts can also be symlinked into the directory as well, if that is more convenient. These scripts will not be overwritten by subsequent installations.

Appendix A: XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    targetNamespace="urn:stonesoup:api:communications"
xmlns="urn:stonesoup:api:communications">
    <xs:element name="log" type="LogMessageType">
        <xs:annotation>
            <xs:documentation>
                The log message is used to log arbitrary messages from the
                proposed technology during analyze and execute invocations.
                It is optional to use and the results will not be directly
                used during scoring unless a specific issue arises.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="result" type="ResultMessageType">
        <xs:annotation>
            <xs:documentation>
                The result message is used to report the final result of an
                analyze or execute invocation.  It is a required message.
                Only one result message should ever be sent during a single
                workflow.  This message will indicate the end of reporting
                for the proposed technology, and will trigger the message
                server to shutdown.  Any following messages are not
                guaranteed to be successfully processed.

                The result message includes only a status enumeration an
                the raw process return code (i.e. 0-255).  An execute result
                may also include the action taken by proposed technology,
                rather than reporting it as a separate message.  An analyze
                result should not populate that element.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="weakness" type="WeaknessMessageType">
        <xs:annotation>
            <xs:documentation>
                The weakness message is used to report an identified or
                perceived software vulnerability.  Reporting a weakness is
                optional, but will aid in evaluating the proposed technology.

                A weakness may also be reported as part of the action message,
                rather than as a separate message.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="impact" type="TechnicalImpactMessageType">
        <xs:annotation>
            <xs:documentation>
                The impact message is used to report a negative technical
                impact identified by the proposed technology.  Reporting
                negative technical impacts is optional, but will aid in
                evaluating the proposed technology.

                A negative technical impact may also be reported as part of
                the action message, rather than as a separate message.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="action" type="ActionMessageType">
        <xs:annotation>
```

```
            <xs:documentation>
                The action message is used to report an action taken by the
                proposed technology to mitigate or stop a preceived weakness
                from being exploited by a malicious input.  Currently, the
                actions are "Controlled Exit" and "Continued Execution".
                With an action, the negative technical impact and perceived
                target weakness may optionally be reported.

                It is not required that you report an action in this manner.
                An action may also be reported as part of the result message.
                However, this message may be useful if multiple actions are
                taken by the proposed technology, as only one result may be
                reported.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="response" type="ResponseMessageType">
        <xs:annotation>
            <xs:documentation>
                The response message is used to report the success or failure
                in processing the received message (i.e. log, result, finish,
                etc.).  This message will provide av overall result, the HTTP
                status code, and an optional message providing any other
                useful information.

                This message exists to support debugging integration with
                performer technologies.  The HTTP status code encapsulated in
                the headers should be sufficient during runtime.

                HTTP Status Codes:
                    201 Created - Message processed and recorded.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="messages" type="ReceivedMessagesType">
        <xs:annotation>
            <xs:documentation>
                Provides a root element to log all received messages.
                This element should not be used or implemented by
                performers directly.  This is reserved for the T&amp;E
                team use only.
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:complexType name="ReceivedMessageType">
        <xs:choice>
            <xs:element name="log" type="LogMessageType"/>
            <xs:element name="result" type="ResultMessageType"/>
            <xs:element name="action" type="ActionMessageType"/>
            <xs:element name="weakness" type="WeaknessMessageType"/>
            <xs:element name="impact" type="TechnicalImpactMessageType"/>
        </xs:choice>
        <xs:attribute name="received" type="xs:dateTime" use="required"/>
        <xs:attribute name="hostname" type="xs:string"/>
    </xs:complexType>
    <xs:complexType name="ReceivedMessagesType">
        <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded" name="message"
                type="ReceivedMessageType"/>
        </xs:sequence>
        <xs:attribute name="hostname" type="xs:string" use="required"/>
        <xs:attribute name="created" type="xs:dateTime" use="required"/>
    </xs:complexType>
    <xs:complexType name="ResultMessageType">
        <xs:sequence>
            <xs:element maxOccurs="1" minOccurs="0" name="action"
type="ActionMessageType"/>
        </xs:sequence>
        <xs:attribute name="status" type="ResultEnumType" use="required"/>
        <xs:attribute name="return_code" type="xs:integer" use="required"/>
    </xs:complexType>
    <xs:complexType name="WeaknessMessageType">
        <xs:sequence minOccurs="0">
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="additional_information"
                type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="cwe" type="CWEType" use="optional"/>
        <xs:attribute name="file_name" type="xs:string" use="optional"/>
        <xs:attribute name="function_name" type="xs:string" use="optional"/>
        <xs:attribute name="line_number" type="xs:integer" use="optional"/>
    </xs:complexType>
    <xs:complexType name="TechnicalImpactMessageType">
        <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="additional_information"
                type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="effect" type="TechnicalImpactEnumType" use="required"/>
    </xs:complexType>
    <xs:complexType name="LogMessageType">
        <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="1" name="statement"
type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ActionMessageType">
        <xs:sequence>
            <xs:element maxOccurs="1" minOccurs="0" name="weakness"
type="WeaknessMessageType"/>
            <xs:element maxOccurs="1" minOccurs="0" name="impact"
type="TechnicalImpactMessageType"/>
            <xs:element maxOccurs="unbounded" minOccurs="0"
name="additional_information"
                type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="behavior" type="ActionEnumType" use="required"/>
    </xs:complexType>
<xs:complexType name="ResponseMessageType">
    <xs:sequence>
        <xs:element maxOccurs="1" minOccurs="0" name="message" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="result" type="ResponseResultEnumType" use="required"/>
    <xs:attribute name="code" type="xs:integer" use="required"/>
</xs:complexType>
    <xs:simpleType name="ResultEnumType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="SUCCESS"/>
            <xs:enumeration value="SKIP"/>
            <xs:enumeration value="TIMEOUT"/>
            <xs:enumeration value="OTHER"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ActionEnumType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="NONE"/>
            <xs:enumeration value="CONTROLLED_EXIT"/>
            <xs:enumeration value="CONTINUED_EXECUTION"/>
            <xs:enumeration value="OTHER"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="TechnicalImpactEnumType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="NONE"/>
            <xs:enumeration value="UNSPECIFIED"/>
            <xs:enumeration value="READ_FILE"/>
            <xs:enumeration value="READ_APPLICATION_DATA"/>
            <xs:enumeration value="GAIN_PRIVILEGES"/>
            <xs:enumeration value="HIDE_ACTIVITIES"/>
            <xs:enumeration value="EXECUTE_UNAUTHORIZED_CODE"/>
            <xs:enumeration value="MODIFY_FILES"/>
            <xs:enumeration value="MODIFY_APPLICATION_DATA"/>
            <xs:enumeration value="BYPASS_PROTECTION_MECHANISM"/>
            <xs:enumeration value="ALTER_EXECUTION_LOGIC"/>
            <xs:enumeration value="UNEXPECTED_STATE"/>
            <xs:enumeration value="DOS_UNCONTROLLED_EXIT"/>
            <xs:enumeration value="DOS_AMPLIFICATION"/>
            <xs:enumeration value="DOS_INSTABILITY"/>
            <xs:enumeration value="DOS_BLOCKING"/>
            <xs:enumeration value="DOS_RESOURCE_CONSUMPTION"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="CWEType">
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Z]{3}-[0-9]{1,3}"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ResponseResultEnumType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="SUCCESS"/>
            <xs:enumeration value="ERROR"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Appendix B: JSON Schemas

action.json

```
{
    "id": "urn:stonesoup:api:communications:action",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API - Action Message",
    "description": "An action taken by the proposed technology.",
    "properties": {
        "behavior": {
            "description": "The behavior of the proposed technology for this action.",
            "type": "string",
            "enum" : [ "NONE",
                "CONTROLLED_EXIT",
                "CONTINUED_EXECUTION",
                "OTHER"
            ]
        },
        "weakness": {
            "$ref": "urn:stonesoup:api:communications:weakness"
        },
        "impact": {
            "$ref": "urn:stonesoup:api:communications:impact"
        },
        "additional_information": {
            "description": "A list additional statements to log with the weakness.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "required": [ "behavior" ],
    "additionalProperties": false
}
```

impact.json

```
{
    "id": "urn:stonesoup:api:communications:impact",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API - Impact Message",
    "description": "An identified negative technical impact.",
    "properties": {
        "effect": {
            "description": "The negative technical impact.",
            "type": "string",
            "enum" : [ "NONE",
                "UNSPECIFIED",
                "READ_FILE",
                "READ_APPLICATION_DATA",
                "GAIN_PRIVILEGES",
                "HIDE_ACTIVITIES",
                "EXECUTE_UNAUTHORIZED_CODE",
                "MODIFY_FILES",
                "MODIFY_APPLICATION_DATA",
                "BYPASS_PROTECTION_MECHANISM",
                "ALTER_EXECUTION_LOGIC",
                "UNEXPECTED_STATE",
                "DOS_UNCONTROLLED_EXIT",
                "DOS_AMPLIFICATION",
                "DOS_INSTABILITY",
                "DOS_BLOCKING",
                "DOS_RESOURCE_CONSUMPTION"
            ]
        },
        "additional_information": {
            "description": "A list additional statements to log with the weakness.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "required": [ "effect" ],
    "additionalProperties": false
}
```

log.json

```
{
    "id": "urn:stonesoup:api:communications:log",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API",
    "description": "Information to log.",
    "properties": {
        "statements": {
            "description": "A list statements to log.",
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            }
        }
    },
    "required": [ "statements" ],
    "additionalProperties": false
}
```

result.json

```
{
    "id": "urn:stonesoup:api:communications:result",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API",
    "description": "Defines the JSON encoding for the messages supported by the
communications API",
    "properties": {
        "status": {
            "description": "The proposed technology status for completion.",
            "type": "string",
            "enum" : [ "SUCCESS",
                "SKIP",
                "TIMEOUT",
                "OTHER"
            ]
        },
        "return_code": {
            "description": "The exit code for the analyze operation (i.e. make/build)
or the execute of the test case application.",
            "type": "integer",
            "minimum": 0,
            "maximum": 255
        },
        "action": {
            "$ref": "urn:stonesoup:api:communications:action"
        }
    },
    "required": [ "status", "return_code" ],
    "additionalProperties": false
}
```

weakness.json

```
{
    "id": "urn:stonesoup:api:communications:weakness",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "STONESOUP Communications API - Weakness Message",
    "description": "An identified weakness.",
    "properties": {
        "cwe": {
            "description": "The CWE or other weakness id.",
            "type": "string",
            "pattern": "[A-Z]{3}-[0-9]{1,3}"
        },
        "file_name": {
            "description": "The file name the weakness was found within.",
            "type": "string"
        },
        "function_name": {
            "description": "The function name the weakness was found within.",
            "type": "string"
        },
        "line_number": {
            "description": "The function name the weakness was found within.",
            "type": "integer",
            "minimum": 1
        },
        "additional_information": {
            "description": "A list additional statements to log with the weakness.",
            "type":"array",
            "minItems":1, "items": {
                "type": "string"
            }
        }
    },
    "additionalProperties": false
}
```

Appendix C: analyze.sh

This is a draft script. Not all functionality is necessarily present. The most up-to-date versions of this script can be obtained by contacting the TEXAS Development Team.

```
#!/bin/bash
# -----------------------------Copyright-------------------------------------
# NOTICE
#
# This software (or technical data) was produced for the U. S.
# Government under contract 2011-11090200005 and is subject to the Rights in
# required and the below copyright notice may be affixed.
#
# Copyright (c) 2014. All Rights Reserved.
# -----------------------------Copyright--------------------------------------

#=== Section ==============================================================
#
# Globals
#
#==========================================================================
SCRIPT_DIR=$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )
CONFIG_SECTION_NAME="buildcommands"
PERFORMER_SCRIPT_FILE="${SCRIPT_DIR}/performer_analyze.sh"


# Default Environment Variables (Test Case Required)
SS_BASE_CFLAGS="${SS_BUILD_COMMAND_CFLAGS} -fno-stack-protector"
SS_BASE_CPPFLAGS="${SS_BUILD_COMMAND_CPPFLAGS} -I${SS_TC_DEPS}/include"
SS_BASE_LIBS="${SS_BUILD_COMMAND_LIBS}"
SS_BASE_LIBS_DIR="-L${SS_TC_DEPS}/lib -L${SS_TC_DEPS}/lib64"
SS_BASE_LIBS_RPATH="-Wl,-R${SS_TC_ROOT}/${SS_TC_INSTALL}/lib -Wl,-
R${SS_TC_ROOT}/${SS_TC_INSTALL}/lib64 -Wl,-R${SS_TC_DEPS}/lib -Wl,-
R${SS_TC_DEPS}/lib64"
SS_BASE_PKGCONFIG="${SS_TC_DEPS}/lib:${SS_TC_DEPS}/lib/pkgconfig:${SS_TC_DEPS}/lib64:$
{SS_TC_DEPS}/lib64/pkgconfig"
SS_BASE_PATH="${SS_TC_DEPS}/bin:${SS_TC_DEPS}/sbin"
SS_BASE_LDFLAGS="${SS_BUILD_COMMAND_LDFLAGS} ${SS_BASE_LIBS_DIR} ${SS_BASE_LIBS_RPATH}
-Wl,-z,execstack"
PATH="${SS_BASE_PATH}:${PATH}"

# Preserve existing PKG_CONFIG_PATH, if set
if [[ -z "${PKG_CONFIG_PATH}" ]]
then
    PKG_CONFIG_PATH="${SS_BASE_PKGCONFIG}"
else
    PKG_CONFIG_PATH="${SS_BASE_PKGCONFIG}:${PKG_CONFIG_PATH}"
fi

# Export the Environment Variables not set by performers
```

```
export PKG_CONFIG_PATH="${PKG_CONFIG_PATH}"
export PATH="${PATH}"


# Default Environment Variables (Provided by performers)
SS_CC="gcc"
SS_LNK="${SS_CC}"
SS_LDFLAGS=""
SS_LIBS=""
SS_CFLAGS=""
SS_ANT_OPTS=""

#=== Section ==================================================================
#
# Functions
#
#==============================================================================
#### Function #################################################################
# Print script usage.
# Globals:
#   $0 - Script name
# Arguments:
#   None
# Returns:
#   None
###############################################################################
function print_usage() {
    cat << EOF
usage: $0 [OPTIONS] -d COMMAND_CWD [-f COMMAND_FILE]

This script provides an interface through which a target application source
or binary can be analyzed (and created if building the source).  This

OPTIONS:
   -d      Working directory from which COMMAND_FILE will execute.

   -f      Configuration file containing build commands.  If this is not
           provided, the data is expected to be provided via stdin.

           The configuration format closely follows that of INI, with the
           exception that comments are denoted with '#'.  Below is an
           example:

             # This is a config for build commands
             [buildcommands]
             # Configure the build
             command=./configure --prefix="$SS_TC_ROOT/$SS_TC_INSTALL"
             # Build and then install
             command=make all
             command=make install

   -g      Debug flag.  RESERVED FOR PERFORMER USE.

   -h      Show this message.

   -n      Do nothing.  Parse and print commands.

   -p      Run with performer technology.

   -r      Run in release mode. This mode should represent the most production-
           like version of the performer technology. RESERVED FOR PERFORMER USE.

   -t      Test case name.

   -v      Run in verbose mode.
EOF
}

#=== Section ==================================================================
#
# Main Script
#
#==============================================================================
# Load the utilities, mostly functions
source "${SCRIPT_DIR}/utils.sh"

# Ensure this is run as root.  We will drop privileges when the commands
# are execute, if required.
if [[ "$EUID" != "0" ]]
then
    print_error "This must be run as root."
    exit 1
fi

# Ensure this is a linux based platform, or exit
if [[ "$OSTYPE" != linux* ]]
then
    print_error "$OSTYPE" "is not a supported platform." \
        "Only specific Linux distributions are supported."
    exit 1
fi

# Parse the arguments
while getopts "d:f:ghnprvt:" OPTION
do
    case $OPTION in
        h)
            print_usage
            exit 0
            ;;
        g)
            MODE_DEBUG="yes"
            ;;
        r)
            MODE_RELEASE="yes"
            ;;
        n)
            MODE_DRY_RUN="yes"
            ;;
        p)
            MODE_PERFORMER="yes"
            ;;
        v)
            MODE_VERBOSE="yes"
            ;;
        t)
            TESTCASE_NAME="${OPTARG}"
            ;;
        f)
            COMMAND_FILE="${OPTARG}"
            COMMAND_FILE_STDIN="no"
            ;;
        d)
            COMMAND_EXEC_CWD="${OPTARG}"
            ;;
        ?)
            print_usage
            exit 1
            ;;
    esac
done

# Apply defaults to all arguments not specified
MODE_DEBUG="${MODE_DEBUG:-no}"
MODE_RELEASE="${MODE_RELEASE:-no}"
MODE_DRY_RUN="${MODE_DRY_RUN:-no}"
MODE_PERFORMER="${MODE_PERFORMER:-no}"
MODE_VERBOSE="${MODE_VERBOSE:-no}"
TESTCASE_NAME="${TESTCASE_NAME:-UNKNOWN}"
COMMAND_FILE_STDIN="${COMMAND_FILE_STDIN:-yes}"
COMMAND_FILE="${COMMAND_FILE:-/dev/stdin}"
COMMAND_EXEC_CWD="${COMMAND_EXEC_CWD:-}"
ANALYZE_SUCCESS="yes"

# Validate arguments
if [[ -z "${COMMAND_EXEC_CWD}" ]]
then
```

```
    print_error "Build directory not specified."
    exit 1
elif [[ ! -d "${COMMAND_EXEC_CWD}" ]]
then
    print_error "Build directory does not exist:" \
        "${COMMAND_EXEC_CWD}"
    exit 1
fi

if [[ "${MODE_VERBOSE}" == "yes" ]]
then
    echo "Beginning analysis of Test Case." "ID:" "${TESTCASE_NAME}"

    echo "Analysis settings:"
    printf "    %s:   %s\n" "Run Directory" "${COMMAND_EXEC_CWD}"
    printf "    %s:           %s\n" "Stage" "$(if [[ "${MODE_PERFORMER}" == "no" ]];
then echo "STAGE 1"; else echo "STAGE 2"; fi)"
    printf "    %s:           %s\n" "Debug" "${MODE_DEBUG}"
    printf "    %s:         %s\n" "Release" "${MODE_RELEASE}"
    printf "    %s:         %s\n" "Dry Run" "${MODE_DRY_RUN}"
    printf "    %s:    %s\n" "Command File" "${COMMAND_FILE}"
    printf "    %s:            %s\n" "Path" "${PATH}"
    printf "    %s:    %s\n" "Package Path" "${PKG_CONFIG_PATH}"
    printf "\n"
fi


if [[ "${MODE_VERBOSE}" == "yes" ]]
then
    echo "Analysis Base Flag Settings:"
    printf "    %s      %s\n" "Base CFlags" "${SS_BASE_CFLAGS}"
    printf "    %s    %s\n" "Base CPPFlags" "${SS_BASE_CPPFLAGS}"
    printf "    %s     %s\n" "Library Dirs" "${SS_BASE_LIBS_DIR}"
    printf "    %s   %s\n" "Base Libraries" "${SS_BASE_LIBS}"
    printf "    %s   %s\n" "Libraries Path" "${SS_BASE_LIBS_RPATH}"
    printf "    %s   %s\n" "Package Config" "${SS_BASE_PKGCONFIG}"
    printf "    %s        %s\n" "Base Path" "${SS_BASE_PATH}"
    printf "    %s     %s\n" "Base LDFlags" "${SS_BASE_LDFLAGS}"
    printf "\n"
fi


# Check if we are running with performer technology (STAGE 2).
# If so, then source in the performer functions.
if [[ "${MODE_PERFORMER}" == "yes" ]]
then
    # Ensure the file exists
    if [[ ! -f "${PERFORMER_SCRIPT_FILE}" || ! -r "${PERFORMER_SCRIPT_FILE}" ]]
    then
```

```
        print_error "ERROR: Performer source file not found or accessible."
        exit 1
    fi

    source "${PERFORMER_SCRIPT_FILE}"

    # Load the performer build flags
    performer_set_flags
fi



# Parse and load the build commands from the config file.
declare -a BUILD_COMMANDS=( )
BUILD_COMMAND=""
while read BUILD_COMMAND || [[ -n "$BUILD_COMMAND" ]]
do
    BUILD_COMMANDS+=("${BUILD_COMMAND}")
done < <(parse_config_commands "${CONFIG_SECTION_NAME}" "${COMMAND_FILE}")

# Ensure build commands were parsed
if [[ "${#BUILD_COMMANDS[@]}" -eq 0 ]]
then
    print_error "ERROR: No build commands provided."
    exit 1
fi

# Set all of the build Environment Variables.  Performers are able to
# set the SS_CC, SS_LNK, SS_CFLAGS, and SS_LDFLAGS variables.  Those values
# are then appended to the base variables required to perform a normal
# build of the application.
export SS_CC="${SS_CC}"
export SS_LNK="${SS_LNK}"
export CFLAGS="${SS_BASE_CFLAGS} ${SS_CFLAGS}"
export CPPFLAGS="${SS_BASE_CPPFLAGS}"
export LDFLAGS="${SS_BASE_LDFLAGS} ${SS_LDFLAGS}"
export LIBS="${SS_BASE_LIBS} ${SS_LIBS}"


if [[ "${MODE_VERBOSE}" == "yes" ]]
then

      echo "Environment Variables:"
      printf "    %s:           %s\n" "SS_CC" "${SS_CC}"
      printf "    %s:          %s\n" "SS_LNK" "${SS_LNK}"
      printf "    %s:      %s\n" "SS_LDFLAGS" "${LDFLAGS}"
      printf "    %s:       %s\n" "SS_CFLAGS" "${CFLAGS}"
      printf "    %s:     %s\n" "SS_ANT_OPTS" "${SS_ANT_OPTS}"
      printf "    %s:      %s\n" "SS_TC_DEPS" "${SS_TC_DEPS}"
```

```
        printf "    %s:      %s\n" "SS_TC_ROOT" "${SS_TC_ROOT}"
        printf "    %s:   %s\n" "SS_TC_INSTALL" "${SS_TC_INSTALL}"


fi

# Execute the build commands
BUILD_COMMAND_RETURN_CODE="0"
for BUILD_COMMAND in "${BUILD_COMMANDS[@]}"
do
        if [[ "${MODE_VERBOSE}" == "yes" ]]
        then
        printf "Running command: "
        eval echo $(string_escape "${BUILD_COMMAND}")
        printf "\n"
    fi

    # CD to the target build directory and execute the command.  The command
    # is evaluated to expand variables and process quotes as though the user
    # had typed it at an interactive shell prompt.
    #
    # NOTE: We do this all in a sub-shell, so we do not have to deal with
    #       tracking and restoring the original working directory.
    if [[ "${MODE_DRY_RUN}" == "no" ]]
    then
        if [[ "${MODE_PERFORMER}" == "no" ]]
        then
            # ********************
            #       STAGE 1
            # ********************
            # We are not running with performer technology.  We will simply
            # execute the command as defined.
            ( cd "${COMMAND_EXEC_CWD}" && eval "${BUILD_COMMAND}" )

        else
            # ********************
            #       STAGE 2
            # ********************
            # We are running with performer technology.
            ( performer_build "${COMMAND_EXEC_CWD}" "${BUILD_COMMAND}" )

        fi
        BUILD_COMMAND_RETURN_CODE="$?"
        # Stop if the command failed, no need to execute the remaining
        if [[ "${BUILD_COMMAND_RETURN_CODE}" -ne 0 ]]
        then

            ANALYZE_SUCCESS="no"
            print_error "ERROR: Previous command failed.  Return code:"\
                    "${BUILD_COMMAND_RETURN_CODE}"
            break

         fi
     else
       printf "%s\n" "$(eval echo "${BUILD_COMMAND}")"
     fi
done

if [[ "${MODE_PERFORMER}" == "yes" ]]
then
       performer_report
fi

if [[ "${MODE_VERBOSE}" == "yes" ]]
then
       printf "Completed analysis of Test Case.  Result: "
       if [[ "${ANALYZE_SUCCESS}" == "yes" ]]
       then
       printf "SUCCESS\n"
       else
       printf "FAILED\n"
       fi
fi

# Bubble up the run command return code
exit "${BUILD_COMMAND_RETURN_CODE}"
```

Appendix D: execute.sh

This is a draft script. Not all functionality is necessarily present. The most up-to-date versions of this script can be obtained by contacting the TEXAS Development Team.

```
#!/bin/bash
# -----------------------------Copyright-------------------------------------
# NOTICE
#
# This software (or technical data) was produced for the U. S.
# Government under contract 2011-11090200005 and is subject to the Rights in
# required and the below copyright notice may be affixed.
#
# Copyright (c) 2014. All Rights Reserved.
# -----------------------------Copyright--------------------------------------

#=== Section ===============================================================
#
# Globals
#
#==========================================================================
SCRIPT_DIR=$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )
CONFIG_SECTION_NAME="runcommands"
PERFORMER_SCRIPT_FILE="${SCRIPT_DIR}/performer_execute.sh"
DEFAULT_SHELL="/bin/bash"

# These are the list of ULIMITs that will be supported.  We'll use the Linux
# API defs as keys, since they provide a much better set of documentation
# regarding what each does.
declare -r ULIMITS_SUPPORTED=("RLIMIT_AS" \
                              "RLIMIT_NOFILE" \
                              "RLIMIT_NPROC" \
                              "RLIMIT_CPU")
# Stores the original ULIMITs, these will be reapplied after execution
declare -a ULIMITS_ORIGINAL
# Stores the user specified ULIMITs to enforce during execution
declare -a ULIMITS_ENFORCE

# NOTE: this will only work for detecting 64-bit on an AMD/Intel arch.
SS_TC_LIBDIR_NAME="lib$([[ $(uname -m) == x86_64 ]] && echo 64)"

# Build the PATH to use for this test case execution
# Set the PATH for the dependencies
SS_TC_DEPS_PATH="${SS_TC_DEPS}/bin:${SS_TC_DEPS}/sbin"
# Set the PATH for the test case installation
SS_TC_PATH="${SS_TC_ROOT}/${SS_TC_INSTALL}/bin:${SS_TC_ROOT}/${SS_TC_INSTALL}/sbin"
# Bring all PATHs together
SS_TC_PATH+=":${SS_TC_DEPS_PATH}:${PATH}"
```

```
# Build the LD_PRELOAD library path.  This is required
# for base programs that fork in order to ensure libraries are loaded
# correctly.
# NOTE: This should be addressed with performers to ensure that
#       their proposed technology does not require LD_PRELOAD
#       in order to override libc implementations, such as malloc.
SS_BASE_LD_PRELOAD="${SS_RUN_COMMAND_LD_PRELOAD}"

#=== Section ================================================================
#
# Functions
#
#============================================================================
#### Function ###############################################################
# Print script usage.
# Globals:
#   $0 - Script name
# Arguments:
#   None
# Returns:
#   None
#############################################################################
function print_usage() {
    cat << EOF
usage: $0 [OPTIONS] -d COMMAND_CWD [-f COMMAND_FILE]

This script provides an interface through which a target application can be
executed.  Commands will be evaluated by the SHELL to simulate executing
what would otherwise be provided by a user on the command line.

OPTIONS:
   -d      Working directory from which COMMAND_FILE will execute.

   -f      Configuration file containing run commands.  If this is not
           provided, the data is expected to be provided via stdin.

           The configuration format closely follows that of INI, with the
           exception that comments are denoted with '#'.  Below is an
           example:

             # This is a config for run commands
             [runcommands]
             # Run the target application
             command=$SS_TC_ROOT/$SS_TC_INSTALL/bin/example -h

   -g      Debug flag.  RESERVED FOR PERFORMER USE.

   -h      Show this message.
```

```
    -l      Set ULIMITs for the target executable.  This flag may be
            provided multiple times.  In most cases, a number is
            expected, but the special string "ulimited" is also supported.
            The settings take the format:
              KEY=VALUE

            The following ULIMITs are supported:
              RLIMIT_AS      - Maximum size of virtual memory.
              RLIMIT_NOFILE  - Maximum number of open file descriptors.
              RLIMIT_NPROC   - Maximum number of processes that may be created.
              RLIMIT_CPU     - Maximum CPU time (in seconds).

            Refer to http://linux.die.net/man/2/getrlimit for more information
            of the flags and their effect on the process.

    -n      Do nothing.  Parse and print commands.

    -p      Run with performer technology.

    -r      Run in release mode. This mode should represent the most production-
            like version of the performer technology. RESERVED FOR PERFORMER USE.

    -t      Test case name.

    -u      Run as different user.

    -v      Run in verbose mode.
EOF
}


#### Function ################################################################
# Parse RLIMIT_AS ulimit
# Globals:
#   None
# Arguments:
#   ulimit_value to set for RLIMIT_AS takes the form "[0-9]+(KB)"
#       - If KB is found at the end of the string, it is stripped and
#         considered to be in kilobytes.
#       - Otherwise it should be in bytes.
# Returns:
#   A number (in kilobytes) to be set as the ulimit value for RLIMIT_AS (-Sv)
##############################################################################
function parse_rlimit_as(){
    local ulimit_value="${1}"

    local ulimit_type=${ulimit_value:(-2)}
    kb_val=1024
    case $ulimit_type in
```

```
        [Ee][Dd])
                echo "${ulimit_value}"
                return 0
                ;;
        [Kk][Bb])
                echo `expr match "$ulimit_value" '\(^[0-9]*\)'`
                return 0
                ;;
        [0-9][Bb])
                echo $(expr `expr match "$ulimit_value" '\(^[0-9]*\)'` / $kb_val)
                return 0
                ;;
        [0-9]*)
                echo `expr $ulimit_value / $kb_val`
                return 0
                ;;
    esac
    return 1
}
#### Function ##############################################################
# Parse ulimit key value pair.
# Globals:
#   ULIMITS_ENFORCE
# Arguments:
#   ulimit setting string.  takes the form ULIMIT_KEY=VALUE
# Returns:
#   None
###########################################################################
function parse_ulimit_arg() {
    local ulimit_setting="${1}"


    # Get everything before the "=" separator.
    local ulimit_setting_key=$(parse_ulimit_key "${ulimit_setting}")


    # Ensure the key is all caps
    # The pipe is very inefficient, but we want to support older versions of Bash
    ulimit_setting_key=$(echo "${ulimit_setting_key}" | tr '[:lower:]' '[:upper:]')


    if [[ $(is_ulimit_supported "${ulimit_setting_key}") == "yes" ]]
    then
        # Store the ULIMIT to enforce.
        ULIMITS_ENFORCE+=("${ulimit_setting}")

        return 0
    else
        print_error "The ULIMIT \"${ulimit_setting_key}\" is not supported."
        return 1
    fi
}

#### Function ##############################################################
# Parse ulimit key from the ulimit key-value pair string
# Globals:
#   None
# Arguments:
#   ulimit setting string.  takes the form ULIMIT_KEY=VALUE
# Returns:
#   ulimit setting key
###############################################################################
function parse_ulimit_key() {
    local ulimit_setting="${1}"

    # Get everything before the "=" separator.
    local ulimit_setting_key=${ulimit_setting%%=*}

    # Ensure the key is all caps
    # The pipe is very inefficient, but we want to support older versions of Bash
    ulimit_setting_key=$(echo "${ulimit_setting_key}" | tr '[:lower:]' '[:upper:]')
    echo "${ulimit_setting_key}"
}

#### Function ##############################################################
# Parse ulimit key from the ulimit key-value pair string
# Globals:
#   None
# Arguments:
#   ulimit setting string.  takes the form ULIMIT_KEY=VALUE
# Returns:
#   ulimit setting value
###############################################################################
function parse_ulimit_value() {
       local ulimit_setting="${1}"

    # Get everything after the "=" separator.
    local ulimit_setting_value=${ulimit_setting#*=}

    ulimit_value=("$(string_trim ${ulimit_setting_value})")

    if [[ -z "${ulimit_value}" ]]
    then
        print_error "No value provided for ULIMIT" \
            "\"${ulimit_key}\"."
        return 1
    fi
    echo "${ulimit_value}"

}

#### Function ################################################################
# Check if ULIMIT is supported.
# Globals:
#   ULIMITS_SUPPORTED
# Arguments:
#   ulimit setting key (name).
# Returns:
#   yes/no if supported
##############################################################################
function is_ulimit_supported() {
    local ulimit_setting_key="${1}"
    local ulimit_supported=""
    local is_supported_result="no"

    for ulimit_supported in "${ULIMITS_SUPPORTED[@]}"
    do
        if [[ "${ulimit_supported}" == "${ulimit_setting_key}" ]]
        then
            is_supported_result="yes"
            break
        fi
    done

    echo "${is_supported_result}"
}

#### Function ################################################################
# Enforce the specified ULIMIT
# Globals:
#   MODE_ENFORCE_ULIMIT
# Arguments:
#   ulimit_key
#   ulimit_value
# Returns:
#   None
##############################################################################
function set_ulimit() {
    local ulimit_key="${1}"
    local ulimit_value="${2}"

    if [[ "${MODE_ENFORCE_ULIMIT}" == "yes" ]]
    then
        case "${ulimit_key}" in
            "RLIMIT_AS")
```

```
                parsed_value=$(parse_rlimit_as "${ulimit_value}")
                if [[ "$?" -ne 0  ]]
                then
                    return 1
                fi
                ulimit -Sv "${parsed_value}"
                if [[ "${MODE_VERBOSE}" == "yes" ]]
                then
                    echo "Setting RLIMIT_AS to" "${ulimit_value}"
                    ulimit -Sv
                fi

                ;;
            "RLIMIT_NOFILE")
                ulimit -Sn "${ulimit_value}"
                if [[ "${MODE_VERBOSE}" == "yes" ]]
                then
                    echo "Setting RLIMIT_NOFILE to" "# ${ulimit_value}"
                    ulimit -Sn
                fi

                ;;
            "RLIMIT_NPROC")
                ulimit -Su "${ulimit_value}"
                if [[ "${MODE_VERBOSE}" == "yes" ]]
                then
                    echo "Setting RLIMIT_NPROC to" "${ulimit_value}"
                    ulimit -Su
                fi

                ;;
            "RLIMIT_CPU")
                ulimit -St "${ulimit_value}"
                if [[ "${MODE_VERBOSE}" == "yes" ]]
                then
                    echo "Setting RLIMIT_CPU to" "${ulimit_value}"
                    ulimit -St
                fi

                ;;
            *)
                print_error "Error: Unimplemented ULIMIT" \
                    "option \"${ulimit_key}\"."
                echo "Setting failed"
                return 1
                ;;
        esac
    fi
}
```

```
#### Function ###############################################################
# Report the specified ULIMIT
# Globals:
#   MODE_ENFORCE_ULIMIT
# Arguments:
#   ulimit_key
#   ulimit_value
# Returns:
#   None
##############################################################################
function get_ulimit() {
    local ulimit_key="${1}"

    case "${ulimit_key}" in
        "RLIMIT_AS")
            ulimit -Sd
            ;;
        "RLIMIT_NOFILE")
            ulimit -Sn
            ;;
        "RLIMIT_NPROC")
            ulimit -Su
            ;;
        "RLIMIT_CPU")
            ulimit -St
            ;;
        *)
            print_error "Error: Unimplemented ULIMIT" \
                "option \"${ulimit_key}\"."
            return 1
            ;;
    esac
}

#### Function ###############################################################
# Enforce the specified ULIMITS
# Globals:
#   ULIMITS_ENFORCE
#   MODE_ENFORCE_ULIMIT
# Arguments:
#   None
# Returns:
#   None
##############################################################################
function enforce_ulimits() {
    local ulimit_key=""

    if [[ "${MODE_ENFORCE_ULIMIT}" == "yes" ]]
    then
        for ulimit_setting in "${ULIMITS_ENFORCE[@]}"
        do

            set_ulimit "$(parse_ulimit_key "${ulimit_setting}")" "$(parse_ulimit_value
"${ulimit_setting}")"

            # check result
            if [[ "$?" -ne 0 ]]
            then
                return 1
            fi
        done
    fi

    return 0
}

#### Function ##################################################################
# Restores the ULIMITs to the original runtime values.
# Globals:
#   ULIMITS_ORIGINAL
#   ULIMITS_ENFORCE
#   MODE_ENFORCE_ULIMIT
# Arguments:
#   None
# Returns:
#   None
################################################################################
function restore_ulimits() {
    local ulimit_key=""

    if [[ "${MODE_ENFORCE_ULIMIT}" == "yes" ]]
    then
        for ulimit_setting in "${ULIMITS_ORIGINAL[@]}"
        do
            set_ulimit "$(parse_ulimit_key "${ulimit_setting}")" "$(parse_ulimit_value
"${ulimit_setting}")"

            # check result
            if [[ "$?" -ne 0 ]]
            then
                return 1
            fi
        done
    fi

    return 0
}
```

```
#### Function ###############################################################
# Cache the original ULIMITs for this process
# Globals:
#   ULIMITS_ORIGINAL
# Arguments:
#   None
# Returns:
#   None
#############################################################################
function cache_ulimits() {
    local ulimit_key=""

    for ulimit_key in "${ULIMITS_SUPPORTED[@]}"
    do
       ulimit_value=$(get_ulimit "${ulimit_key}")
        # Report the current ULIMIT so that we can restore it later
        ULIMITS_ORIGINAL+=("${ulimit_key}=${ulimit_value}")

        # check result
        if [[ "$?" -ne 0 ]]
        then
            return 1
        fi
    done

    return 0
}

#### Function ###############################################################
# Sets additional environment variables to use for the execution of the
# RUN_COMMAND in the subshell.  Typically this is just PATH and LD_PRELOAD.
# Globals:
#   SS_TC_PATH
#   SS_LD_PRELOAD
# Arguments:
#   None
# Returns:
#   None
#############################################################################
function set_run_environment() {
    PATH="${SS_TC_PATH}"
    LD_PRELOAD="${SS_BASE_LD_PRELOAD} ${SS_LD_PRELOAD}"

    export PATH
    export LD_PRELOAD
}

#=== Section ================================================================
#
# Main Script
#
#==============================================================================
# Load the utilities, mostly functions
source "${SCRIPT_DIR}/utils.sh"

# Ensure this is run as root.  We will drop privileges when the commands
# are execute, if required.
if [[ "$EUID" != "0" ]]
then
    print_error "ERROR: This must be run as root."
    exit 1
fi

# Ensure this is a linux based platform, or exit
if [[ "$OSTYPE" != linux* ]]
then
    print_error "ERROR:" "$OSTYPE" "is not a supported platform." \
        "Only specific Linux distributions are supported."
    exit 1
fi

# Parse the arguments
while getopts "hd:f:gl:npt:u:vr" OPTION
do
    case $OPTION in
        h)
            print_usage
            exit 0
            ;;
        g)
            MODE_DEBUG="yes"
            ;;
        r)
            MODE_RELEASE="yes"
            ;;
        n)
            MODE_DRY_RUN="yes"
            ;;
        p)
            MODE_PERFORMER="yes"
            ;;
        v)
            MODE_VERBOSE="yes"
            ;;
        u)
            MODE_RUN_AS_USER="yes"
            RUN_AS_USERNAME="${OPTARG}"
            ;;
        l)
            MODE_ENFORCE_ULIMIT="yes"
            parse_ulimit_arg "${OPTARG}"
            ;;
        t)
            TESTCASE_NAME="${OPTARG}"
            ;;
        f)
            COMMAND_FILE="${OPTARG}"
            COMMAND_FILE_STDIN="no"
            ;;
        d)
            COMMAND_EXEC_CWD="${OPTARG}"
            ;;
        ?)
            print_usage
            exit 1
            ;;
    esac
done

# Apply defaults to all arguments not specified
MODE_DEBUG="${MODE_DEBUG:-no}"
MODE_RELEASE="${MODE_RELEASE:-no}"
MODE_DRY_RUN="${MODE_DRY_RUN:-no}"
MODE_PERFORMER="${MODE_PERFORMER:-no}"
MODE_RUN_AS_USER="${MODE_RUN_AS_USER:-no}"
MODE_ENFORCE_ULIMIT="${MODE_ENFORCE_ULIMIT:-no}"
MODE_VERBOSE="${MODE_VERBOSE:-no}"
RUN_AS_USERNAME="${RUN_AS_USERNAME:-}"
TESTCASE_NAME="${TESTCASE_NAME:-UNKNOWN}"
COMMAND_FILE_STDIN="${COMMAND_FILE_STDIN:-yes}"
COMMAND_FILE="${COMMAND_FILE:-/dev/stdin}"
COMMAND_EXEC_CWD="${COMMAND_EXEC_CWD:-}"
KILLTREE_PID_FILE="${KILLTREE_PID_FILE:-$SS_TC_ROOT/rc_parent.pid}"


if [[ "${MODE_VERBOSE}" == "yes" ]]
then
    echo "Environment Variables:"
    printf "    %s:   %s\n" "SS_TC_DEPS_PATH" "${SS_TC_DEPS_PATH}"
    printf "    %s:        %s\n" "SS_TC_PATH" "${SS_TC_PATH}"
    printf "    %s:     %s\n" "SS_LD_PRELOAD" "${SS_LD_PRELOAD}"
    printf "    %s:%s\n" "SS_BASE_LD_PRELOAD" "${SS_BASE_LD_PRELOAD}"
    printf "    %s:              %s\n" "PATH" "${PATH}"
    printf "\n"
fi
```

```
#explain options to user
if [[ "${MODE_VERBOSE}" == "yes" ]]
then
    echo "Commands to Execute:"
    printf "    %s: %s\n" "Debug" "${MODE_DEBUG}"
    printf "    %s: %s\n" "Release" "${MODE_RELEASE}"
    printf "    %s: %s\n" "Dry Run" "${MODE_DRY_RUN}"
    printf "    %s: %s\n" "Performer Run" "${MODE_PERFORMER}"
    printf "    %s: %s\n" "Run as User" "${MODE_RUN_AS_USER}"
    printf "    %s: %s\n" "Enforce Ulimits" "${MODE_ENFORCE_ULIMIT}"
    printf "    %s: %s\n" "Verbose" "${MODE_VERBOSE}"
    printf "    %s: %s\n" "Username" "${RUN_AS_USERNAME:-root}"
    printf "    %s: %s\n" "Testcase name" "${TESTCASE_NAME}"
    printf "    %s: %s\n" "Using STDIN" "${COMMAND_FILE_STDIN}"
    printf "    %s: %s\n" "Filename" "${COMMAND_FILE}"
    printf "    %s: %s\n" "Killtree pid file" "${KILLTREE_PID_FILE}"
    printf "\n"
fi

# Validate arguments
if [[ -z "${COMMAND_EXEC_CWD}" ]]
then
    print_error "Run directory not specified."
    exit 1
elif [[ ! -d "${COMMAND_EXEC_CWD}" ]]
then
    print_error "Run directory does not exist:" \
        "${COMMAND_EXEC_CWD}"
    exit 1
fi

# Setup base command if we are running as a different user.  The default
# is to just run the command as the current user.
BASE_RUN_COMMAND=""
if [[ "${MODE_RUN_AS_USER}" == "yes" ]]
then
    # To run as a different user, the script will sudo the command.  The
    # following options are used:
    #     -n    run non-interactive
    #     -E    preserve the environment
    #     -u    target username
    #     -s    run in shell (uses SHELL env or /etc/passwd shell)
    #     --    stop processing arguments, pass them to command
    BASE_RUN_COMMAND+=" sudo -n -E -u ${RUN_AS_USERNAME} -s --"
fi

# Setup the ulimit defaults by grabbing the current settings
cache_ulimits
```

```
if [[ "$?" -ne 0 ]]
then
    print_error "ERROR: Failed to cache current process ULIMITs."
    exit 1
fi

if [[ "${MODE_VERBOSE}" == "yes" ]]
then
    echo "Beginning execute of Test Case." "ID:" "${TESTCASE_NAME}"

    echo "Execution settings:"
    printf "    %s:   %s\n" "Run Directory" "${COMMAND_EXEC_CWD}"
    printf "    %s:           %s\n" "Stage" "$(if [[ "${MODE_PERFORMER}" == "no" ]];
then echo "STAGE 1"; else echo "STAGE 2"; fi)"
    printf "    %s:          %s\n" "Run As" "${RUN_AS_USERNAME:-root}"
    printf "    %s:           %s\n" "Debug" "${MODE_DEBUG}"
    printf "    %s:         %s\n" "Release" "${MODE_RELEASE}"
    printf "    %s:         %s\n" "Dry Run" "${MODE_DRY_RUN}"
    printf "    %s:          %s\n" "Limits" "${MODE_ENFORCE_ULIMIT}"
    if [[ "${MODE_ENFORCE_ULIMIT}" == "yes" ]]
    then
        for ulimit_setting in "${ULIMITS_ENFORCE[@]}"
        do

            printf "        %s: %s\n" "$(parse_ulimit_key ${ulimit_setting})"
"$(parse_ulimit_value ${ulimit_setting})"
        done
    fi
fi

# Check if we are running with performer technology (STAGE 2).
# If so, then source in the performer functions.
if [[ "${MODE_PERFORMER}" == "yes" ]]
then
    # Ensure the file exists
    if [[ ! -f "${PERFORMER_SCRIPT_FILE}" || ! -r "${PERFORMER_SCRIPT_FILE}" ]]
    then
        print_error "ERROR: Performer source file not found or not accessible."
        exit 1
    fi

    source "${PERFORMER_SCRIPT_FILE}"
fi

# Ensure the specified command file exists
if [[ "${COMMAND_FILE_STDIN}" == "no" ]]
then
    if [[ ! -f "${COMMAND_FILE}" || ! -r "${COMMAND_FILE}" ]]
    then
```

```
        print_error "ERROR: Specified command file not found or not accessible."
        exit 1
    fi
fi

# Parse and load the build commands from the config file.
declare -a RUN_COMMANDS=( )
RUN_COMMAND=""
while read RUN_COMMAND || [[ -n "$RUN_COMMAND" ]]
do
    RUN_COMMANDS+=("${RUN_COMMAND}")
done < <(parse_config_commands "${CONFIG_SECTION_NAME}" "${COMMAND_FILE}")

# Ensure run commands were parsed
if [[ "${#RUN_COMMANDS[@]}" -eq 0 ]]
then
    print_error "ERROR: No run commands provided."
    exit 1
elif [[ "${#RUN_COMMANDS[@]}" -ne 1 ]]
then
    print_error "ERROR: Only one run command is supported."
    exit 1
fi

# Execute the build commands
RUN_COMMAND_RETURN_CODE="0"
for RUN_COMMAND in "${RUN_COMMANDS[@]}"
do
    if [[ "${MODE_VERBOSE}" == "yes" ]]
    then
        printf "Running command: "
       eval echo $(string_escape "${RUN_COMMAND}")
       printf "\n"
    fi

    # CD to the target run directory and execute the command.  The command
    # is evaluated to expand variables and process quotes as though the user
    # had typed it at an interactive shell prompt.
    #
    # NOTE: We do this all in a sub-shell, so we do not have to deal with
    #       tracking and restoring the original working directory.
    if [[ "${MODE_DRY_RUN}" == "no" ]]
    then
        if [[ "${MODE_PERFORMER}" == "no" ]]
        then
            # ********************
            #       STAGE 1
            # ********************
            # We are not running with performer technology.  We will simply
            # execute the command as defined.

            enforce_ulimits

            # Stop if ULIMITs were not set
            if [[ "$?" -ne 0 ]]
            then
                RUN_COMMAND_RETURN_CODE="$?"
                print_error "ERROR: Failed to enforce ULIMITs for execution."
                break
            fi

            # Execute the actual command in the target CWD.  This is all done in a
            # subshell to avoid dealing with restoring the PWD, affect environment
            # variables, etc.  The command return code will be passed up.
            (
                # Capture the subshell pid, so we can kill the children later.
                # By default, the children of the shell will be killed, but
                # the actual shell pid will not receive a kill signal
                ensure_shell_pid
                printf "%s" "${BASHPID}" > "${KILLTREE_PID_FILE}"

                cd "${COMMAND_EXEC_CWD}"

                # Ensure cd executed
                if [[ "$?" -ne 0 ]]
                then
                    print_error "ERROR: Failed to set the working directory" \
                        "prior to command execution"
                    exit 1
                fi

                # Sets any runtime environment flags
                set_run_environment

                # When executing, we use eval because the RUN_COMMAND is
                # provided as a string that a user would enter on the
                # command line.  Thus, we need quotes and environment variables
                # processed correctly.
                eval "${BASE_RUN_COMMAND}" "${RUN_COMMAND}"

                exit "$?"
            )

            RUN_COMMAND_RETURN_CODE="$?"

            # We do not log an error message here because the output error
            # should simply be that of the RUN_COMMAND_RETURN_CODE
            restore_ulimits

            # Stop if ULIMITs were not restored
            if [[ "$?" -ne 0 ]]
            then
                RUN_COMMAND_RETURN_CODE="$?"
                print_error "ERROR: Failed to restore ULIMITs post execution."
                break
            fi
        else
            # ********************
            #       STAGE 2
            # ********************
            # We are running with performer technology.
            # NOTE: ULIMITs should be set by the performers after
            (
             # Capture the subshell pid, so we can kill the children later.
                # By default, the children of the shell will be killed, but
                # the actual shell pid will not receive a kill signal
                ensure_shell_pid
                printf "%s" "${BASHPID}" > "${KILLTREE_PID_FILE}"

             performer_execute "${COMMAND_EXEC_CWD}" "${BASE_RUN_COMMAND}"
"${RUN_COMMAND}"
            )


            RUN_COMMAND_RETURN_CODE="$?"

            # Stop if the command failed, no need to execute the remaining
            if [[ "${RUN_COMMAND_RETURN_CODE}" -ne 0 ]]
            then
                # We do not log an error message here because the output error
                # should simply be that of the RUN_COMMAND_RETURN_CODE
                break
            fi
        fi
    else
       printf "%s\n" "$(eval echo "${RUN_COMMAND}")"
    fi
done

if [[ "${MODE_VERBOSE}" == "yes" ]]
then
    printf "Completed execute of Test Case.  Result: "
    if [[ "${RUN_COMMAND_RETURN_CODE}" -eq 0 ]]
    then
        printf "SUCCESS\n"
    else
```

```
        printf "FAILED\n"
    fi
fi

# Bubble up the run command return code
exit "${RUN_COMMAND_RETURN_CODE}"
```

Appendix E: utils.sh

```
#!/bin/bash
# -----------------------------Copyright-------------------------------------
# NOTICE
#
# This software (or technical data) was produced for the U. S.
# Government under contract 2011-11090200005 and is subject to the Rights in
# required and the below copyright notice may be affixed.
#
# Copyright (c) 2014. All Rights Reserved.
# -----------------------------Copyright--------------------------------------

# The following is required to get consistent behavior for conditionals
# involving regular expressions.  Newer versions of Bash do not like quotes
# around the regular expression, whereas older versions require it.  Output
# is redirected to hide error messages for older version of bash.
#
# NOTE: This may not be required for these interfaces
shopt -s compat31 1>/dev/null 2>&1

#=== Section ===============================================================
#
# Globals
#
#==========================================================================
# NONE

#=== Section ===============================================================
#
# Functions
#
#==========================================================================
#### Function ##############################################################
# Get user confirmation.
# Globals:
#   None
# Arguments:
#   Confirmation question or message
# Returns:
#   "yes" = yes
#   "no" = no
##########################################################################
function confirm() {
    while true; do
        read -p "$1 [y/n] " yn < /dev/tty
        case $yn in
            [Yy] | [Yy][Ee][Ss] )
                echo "yes"
                break
                ;;
            [Nn] | [Nn][Oo] )
                echo "no"
                break
                ;;
        esac
    done
}

#### Function ################################################################
# Print an error message to stderr.
# Globals:
#   None
# Arguments:
#   Strings to print
# Returns:
#   None
###############################################################################
function print_error() {
    echo "$@" >&2
}

#### Function ################################################################
# Trim leading and trailing whitespace from a string.
# Globals:
#   None
# Arguments:
#   String to trim
# Returns:
#   None
###############################################################################
function string_trim() {
    local input_string="${1}"
    shopt -s extglob
    input_string="${input_string##*([[:space:]])}"
    input_string="${input_string%%*([[:space:]])}"
    shopt -u extglob
    echo "${input_string}"
}

#### Function ################################################################
# Escape select bash characters
# Globals:
#   None
# Arguments:
#   Original string
# Returns:
#   Escaped string
```

```
################################################################################
function string_escape() {
    local input_string="${1}"
    input_string="${input_string//&/\&}"
    input_string="${input_string//|/\|}"
    input_string="${input_string//;/\;}"
    input_string="${input_string//\`/\\\`}"
    input_string="${input_string//(/\(}"
    input_string="${input_string//)/\)}"
    input_string="${input_string//\"/\\\"}"
    input_string="${input_string//!/\!}"
    input_string="${input_string//[/\[}"
    input_string="${input_string//]/\]}"
    input_string="${input_string//>/\>}"
    input_string="${input_string//</\<}"
    echo "${input_string}"
}

#### Function ##################################################################
# Parse the section name from the config.
# Globals:
#   None
# Arguments:
#   String to parse
# Returns:
#   None
################################################################################
function parse_config_section_header() {
    local input_string="${1}"
    shopt -s extglob
    input_string="${input_string##*(\[)}"
    input_string="${input_string%%*(\])}"
    shopt -u extglob
    echo "${input_string}"
}

#### Function ##################################################################
# Parse a target section and return all values for the entries with key
# "command".
# Globals:
#   NONE
# Arguments:
#   config_section_name to parse
#   config_file to parse
# Returns:
#   All values for command entries.  The values are separated by newlines.
################################################################################
function parse_config_commands() {
    # Store the parameters
    local config_section_target="${1}"
    local config_file="${2}"

    # Other variables used during execution
    local config_section_current=""
    local config_entry_key=""
    local config_entry_value=""
    local config_line=""

    # The bash read command will return immediately when EOF is found.
    # Thus, we need to check if the last line contained any data.  If
    # it did, then we process it as normal.
    while read config_line || [[ -n "$config_line" ]]
    do
        # Trim whitespace
        config_line=$(string_trim "${config_line}")

        # Skip all comment lines
        if [[ "${config_line}" =~ "^[#]" ]]
        then
            # Don't update state or counts, just skip
            continue
        fi

        # Look for the start of a new section
        if [[ "${config_line}" =~ "^\[[-[:alnum:]]+\]$" ]]
        then
            # Found a new section.

            # Next reset the variables
            config_entry_key=""
            config_entry_value=""

            # Get the section name and update the parser state
            config_section_current=$(parse_config_section_header \
                "${config_line}")

            # skip to next line
            continue
        fi

        # Check if we are actively parsing the target section
        if [[ "${config_section_current}" == "${config_section_target}" ]]
        then
            # Looking for config entries
            if [[ "${config_line}" =~ "^[[:alnum:]]*=" ]]; then

                # Get everything before the "=" separator.
                config_entry_key=${config_line%%=*}
                # Get everything after the "=" separator.
                config_entry_value=${config_line#*=}

                # Currently only load and return entries with the
                # "command" key.  The key is found via a case
                # insensitive comparison.
                case "${config_entry_key}" in
                    [Cc][Oo][Mm][Mm][Aa][Nn][Dd] )
                        echo "${config_entry_value}"
                        ;;
                    * )
                        # Catch unsupported keys and print the error
                        print_error "Unrecognized configuration setting:" \
                            "${config_entry_key}"
                        ;;
                esac
            fi
        fi
    done < "${config_file}"
}

#### Function ################################################################
# Ensure BASHPID variable exists.
# Globals:
#   BASH_VERSINFO - Check major version
#   BASHPID - set if < bash 4
# Arguments:
#   NONE
# Returns:
#   NONE
##############################################################################
function ensure_shell_pid() {
    if [[ "${BASH_VERSINFO[0]}" -lt 4 ]]
    then
        BASHPID="$(bash -c 'printf $PPID')"
    fi
}
```

Appendix F: Sample performer_analyze.sh

```
#!/bin/bash
# -----------------------------Copyright-------------------------------------
# NOTICE
#
# This software (or technical data) was produced for the U. S.
# Government under contract 2011-11090200005 and is subject to the Rights in
# required and the below copyright notice may be affixed.
#
# Copyright (c) 2014. All Rights Reserved.
# -----------------------------Copyright--------------------------------------

#Globals like SS_CC can also be set globally outside of the function
SS_ANT_OPTS=""

#### Function ################################################################
# Sets any flags relevant to the performer.
# Globals:
#   SS_CC
#   SS_LNK
#      SS_CFLAGS
#      SS_LDFLAGS
#      SS_ANT_OPTS
# Arguments:
#   None
# Returns:
#   None
##############################################################################
function performer_set_flags() {

       if [[ "${SS_TEST}" == "yes" ]]; then
              SS_CC="${SS_TEST_CC}"
              SS_LNK="${SS_TEST_LNK}"
              SS_LDFLAGS="${SS_TEST_LDFLAGS}"
              SS_CFLAGS="${SS_TEST_CFLAGS}"
              SS_ANT_OPTS="${SS_TEST_ANT_OPTS}"
       fi

}

#### Function ################################################################
# Execute the BUILD_COMMAND
# Globals:
#   None
# Arguments:
#   Build command to execute and the directory to execute it in.
# Returns:
#   Return code of the execution of the build command.
```

```
###############################################################################
function performer_build() {

       ( cd "${1}" && eval "${2}" )

}

function performer_report(){
    echo "SUCCESS"
}
```

Appendix G: Sample performer_execute.sh

```
#!/bin/bash
# -----------------------------Copyright--------------------------------------
# NOTICE
#
# This software (or technical data) was produced for the U. S.
# Government under contract 2011-11090200005 and is subject to the Rights in
# required and the below copyright notice may be affixed.
#
# Copyright (c) 2014. All Rights Reserved.
# -----------------------------Copyright---------------------------------------

#Globals like SS_CC can also be set globally outside of the function
SS_ANT_OPTS=""



#### Function ###############################################################
# Execute the RUN_COMMAND
# Globals:
#   None
# Arguments:
#   Base run command to change user if necessary, the run command to execute and
#   the directory to execute it in.
# Returns:
#   Return code of the execution of the build command.
###############################################################################
function performer_execute() {

       enforce_ulimits

       # Stop if ULIMITs were not set
       if [[ "$?" -ne 0 ]]
       then
              RUN_COMMAND_RETURN_CODE="$?"
              print_error "ERROR: Failed to enforce ULIMITs for execution."
              break
       fi

       # Execute the actual command in the target CWD.  This is all done in a
       # subshell to avoid dealing with restoring the PWD, affect environment
       # variables, etc.  The command return code will be passed up.
       (

        # Capture the subshell pid, so we can kill the children later.
        # By default, the children of the shell will be killed, but
        # the actual shell pid will not receive a kill signal
        ensure_shell_pid
```

```
         printf "%s" "${BASHPID}" > "${KILLTREE_PID_FILE}"

              cd "${COMMAND_EXEC_CWD}"

              # Ensure cd executed
              if [[ "$?" -ne 0 ]]
              then
                     print_error "ERROR: Failed to set the working directory" \
                            "prior to command execution"
                     exit 1
              fi

              # Sets any runtime environment flags
              set_run_environment

              # When executing, we use eval because the RUN_COMMAND is
              # provided as a string that a user would enter on the
              # command line.  Thus, we need quotes and environment variables
              # processed correctly.
              eval "${BASE_RUN_COMMAND}" "${RUN_COMMAND}"

              exit "$?"
       )

       RUN_COMMAND_RETURN_CODE="$?"

       # We do not log an error message here because the output error
       # should simply be that of the RUN_COMMAND_RETURN_CODE

       restore_ulimits

       # Stop if ULIMITs were not restored
       if [[ "$?" -ne 0 ]]
       then
              RUN_COMMAND_RETURN_CODE="$?"
              print_error "ERROR: Failed to restore ULIMITs post execution."
              break
       fi

       return "${RUN_COMMAND_RETURN_CODE}"
}
```

Appendix H: Acronyms

Acronym	Acronym Definition
JSON	JavaScript Object Notation
HTTP	HyperText Transfer Protocol
TEXAS	Test & Evaluation eXecution and Analysis System
XML	eXtensible Markup Language