# Using Infer to Find Bugs: Lab Directed Training

Shane Magrath

July 7, 2020

# 1 Introduction

## 1.1 Overview

*Infer* is a static analysis tool whose purpose is to find bugs in code that normal code compilation won't find. Infer is owned by Facebook who open-sourced the tool in 2015. Since then the tool has been used by quite a few Internet scale companies such as Uber, Spotify, Mozilla as well as Facebook to find thousands of bugs a month in their code bases. It has a good reputation.

*Infer* has a significant foundation in advanced program analysis techniques developed by academia. This includes theories such as *Abstract Interpretation* and *Separation Logic*. You do not need to understand these theories to use the tool but they are the reason why the tool is effective in finding non-trivial bugs that the compiler doesn't catch. The cost however for this ability is that the tool can take considerable resources to run in terms of memory and compute power.

*Infer* can analyse C, C++, Objective-C and Java code. Not surprisingly these are languages that matter a lot to Facebook: mobile applications and backend server systems. In this training we focus on C and C++ code-bases. To be able to analyse code with Infer you must be able to compile and build it. The reason for this is that Infer captures many of the front-end compilation data-structures in order for it to perform the advanced analyses it applies.

The bug reports generated by a static analysis tool such as Infer differ from the bugs found by other techniques from dynamic analysis. With dynamic analysis techniques such as fuzzing, the bugs mostly are true positives. The reason is simple: an actual execution that results in a failure reveals that the bug is not just theoretically possible but actually exists.

In contrast static analysis attempts to consider not just the space of *actual* executions but the much larger space of *all possible* executions. However to make this computation feasible static analysis techniques use **over-approximation strategies** to reduce the size of the analysis space. The price you pay for computability is a reduction in the precision of the analysis. It is important to understand this when inspecting Infer's bug reports.

## 1.2 Learning Outcomes

Upon completion of this course, the student will be able to:

- Understand static analysis and its applications and limitations to bug finding
- Install Infer on a Linux machine
- Run Infer against a C/C++ source code targets
- Enable different Infer checkers, including *Inferbo* and *Quandary*
- Generate Infer HTML and JSON bug reports
- Read and understand Infer bug reports for code audit purposes
- Understand and develop Infer linters for syntactic bug finding using Infer's AL language
- How to generate Call Flow Graphs (CFGs) and other front end compilation data for target code understanding and debugging

- How to use Infer with CMake using a *compilation database*

## 1.3 Pre-requisites

- Knowledge and some proficiency in C and C++ coding, compilation and debugging
- Knowledge of C/C++ bug classes and Code Review principles and practice
- Knowledge of Linux Shell commands and operating system operation

## 1.4 Lab Exercises

- **Lab ONE**: Basic analysis of a target
- **Lab TWO**: *Inferbo*: Buffer Overflow and Integer Bugs
- **Lab THREE**: *Quandary*:
  Static Taint Analysis and Command Injection Bugs
- **Lab FOUR**: *Linters*: How to write a linter in AL

# 2 Getting Started

## 2.1 Infer Installation Using Docker

This section we are going to install Infer and the three audit targets using Docker. The purpose of using Docker is to avoid spending unnecessary time on resolving configuration and dependency issues with the tool and the various targets. However in the appendix we show how to install and configure Infer natively for a Linux system.

These notes are on how to use Docker to do this lab.

**Complete the Following: Install and Run the Infer Docker**

From the root of the lab repo:

```
# Create our lab directory to share files with host
mkdir ./lab

# Install and configure Docker on Ubuntu
sudo apt install docker.io
sudo usermod -aG docker ${USER}
su - ${USER}
git clone https://github.com/DSTCyber/infer-training
cd infer-training

# Build a docker image from the Dockerfile in sub-dir ‘lab‘
# Tag it infer-training
docker build -t infer-training .

# Create a new container based on the image above
# mount the host sub-dir ‘content‘ into the container
docker create -it -v ‘pwd‘/lab:/root/lab --name infer infer-training

# Boot the container
docker start infer

# Run a terminal in the container
docker exec -it infer /bin/bash

# In the Docker terminal run the configure script.
# This script pulls down and "patches" the audit targets.
cd lab
./configure.sh
```

You now have a running Docker container with Infer and the three audit targets installed and configured along with their dependencies. Also, the host and the Docker container share a common sub-directory on which allows files to be transferred to the host machine. This shared directory is the `./lab` directory of this training repo. The main reason for doing this is so we can use the host OS’s web browser to view the Infer reports and other data files. The Docker image has no GUI support.

When you're finished with your Docker image complete the following:

```
# docker stop infer
# docker rm infer
# docker rmi infer-training
```

This will stop the Docker container and delete the Infer image from the host machine.

## 3 Audit Targets

We will use the following three Linux games to explore Infer.

- **Angband** - `https://rephial.org/`
- **Skynet** - `https://github.com/cloudwu/skynet`
- **BSDGames** - `https://github.com/vattam/BSDGames`

The targets were chosen because together they exhibit a good diversity of bugs and have modest build and analysis requirements. Some targets because of size and complexity issues require large amounts of memory (i.e. more than 16 GB) and compute time (i.e. more than 10 minutes) which is not appropriate for the typical resources available for this lab.

The Docker container has the build systems, source code and library dependencies already installed. However we have included in the Appendix the process to build the targets from scratch for the purpose of being analysed by Infer.

Infer uses Clang and therefore the targets must be buildable with Clang. Moreover Infer hooks several build tools such as `make`, however it can take a bit of effort to get Infer to work with tools like `CMake`. We look at how this down in the appendix.

In practice you should not under-estimate how much effort is required to properly prepare a target for Infer. Your goal is to maximise analysis coverage which means that you need to configure the target build system to use as many features and libraries as possible. Recursively, you should consider analysing the source code of the library dependencies as well.

In the following subsections we will complete the build configuration for the targets. We could have scripted this work however this is not helpful to you. Part of using Infer effectively is understanding and controlling the build process of the audit targets intimately.

### 3.1 Angband

In the Docker terminal:

**Complete the Following: Verify you can build Angband**

```
$ cd angband-4.2.0
$ CC=clang CXX=clang++ ./configure --enable-sdl2
$ make
```

The build should complete successfully with no errors.

## 3.2 Skynet

We are going to build `Skynet`. I've always wanted to build Skynet. I've always wanted to ride a Harley-Davidson, wear a leather jacket and wield a sawn-off shotgun. I look cool in sun-glasses too.

To build Skynet we need to explicitly identify that it is a `linux` build configuration. You will need to remember this throughout the course.

In the Docker terminal:

**Complete the Following: Verify you can build Skynet**

```
$ cd skynet
$ CC=clang CXX=clang++ make linux
```

The build should complete successfully with no errors.

## 3.3 BSDGames

We will need to manually edit (using Vim) the `Makeconfig` file to use Clang instead of GCC. The usual method of setting environment variables is ineffective.

In the Docker terminal:

**Complete the Following: Verify you can build BSDGames**

```
$ cd bsdgames
$ ./configure
$ vim ./Makeconfig
# --> change the compilers from GCC/G++ to Clang/Clang++
$ make
```

The build should complete successfully with no errors however you will see plenty of warnings. It's old code.

## 4 Lab One: Basic Usage of Infer

The purpose of this lab is to:

- Verify that your Infer installation is working properly by producing *basic reports*.
- Generate both HTML and JSON bug reports.
- Examine the basic report in a browser and gain familiarity with what basic bug classes Infer finds and how they are reported.
- Gain proficiency understanding Infer's reported bugs.

**Complete the Following: Basic Infer Usage**

**For each of the three targets perform the following operations:**

1. In the repo root directory, type:
   `make clean`, and then
   `infer run --keep-going -- make -j6`
   The option `--keep-going` forces the process to keep going when Infer encounters errors. Always recommended. Also, always make sure you have a `make clean` state before analysis. Infer analysis can take some time to complete. Typically:
   - **Angband**: about 7 minutes
   - **SkyNet**: about 1 minute
   - **BSDGames**: about 3 minutes
2. When the Infer analysis is complete you will see on screen a summary. Verify that it looks something like the following:

```
...too many issues to display (limit=10 exceeded), please see
/home/dstcyber/workspace/audit/BSDGames/infer-out/bugs.txt
or run ‘infer-explore‘ for the remaining issues.

Summary of the reports

UNINITIALIZED_VALUE: 100
DEAD_STORE: 37
NULL_DEREFERENCE: 34
RESOURCE_LEAK: 10
MEMORY_LEAK: 5
USE_AFTER_FREE: 1
```

3. Verify that Infer has created a sub-directory containing all its analysis:
   `# cd ./infer-out`
   Have a look in this directory. If you want to re-run the analysis I tend to

just delete the `./infer-out` directory tree and start again. This is not always necessary and in fact it can save considerable time if you don't but don't be afraid to delete it if you want to start again.

4. Inspect the JSON report file:
   `# vim ./infer-out/report.json`
   You may want to examine this in VSCode which can pretty print format the JSON data.
5. Generate a HTML report file:
   `# infer explore --html`
   You should see something like the following:

   ```
   Detected GitHub project vattam/BSDGames
   Saved html report in:
   ./infer-out/report.html/index.html
   ```

6. Open the HTML report in a browser **from your host**:
   `# xdg-open ./infer-out/report.html/index.html`

When you open the report in a browser you should see something similar to Figure 1. It's not a very attractive looking report :-)

## 4.1 Examining Bug Reports

It is very common to get a large number of bug reports when using Infer. It is important to have a strategy when deciding which reports to look at. Your strategy will depend on what your goals are and will improve with experience in using the tool. Not all bug reports are of equal value.

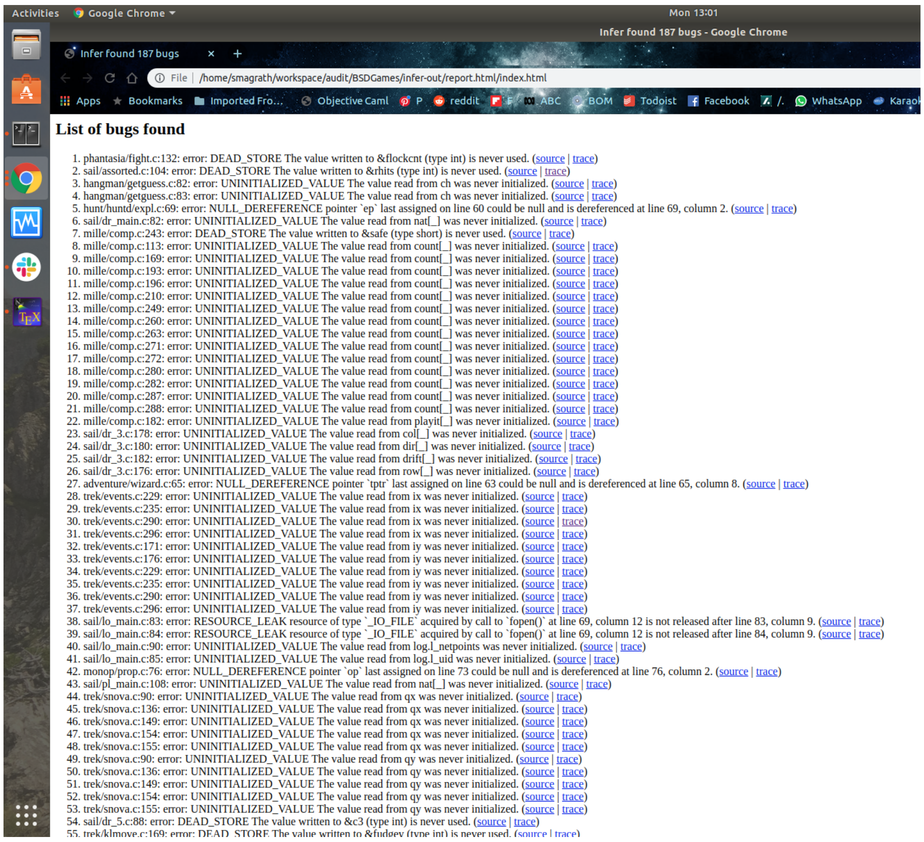
Your strategy may include factors such as:

- Which bug classes are of the most interest to you;
- Which functions and translation units (files) are most important;
- How much time you have :-)

In general, your analysis process will need to examine the *trace* of a bug report. The trace is a *almost* human readable description of why Infer thinks it has found a bug. When looking at the trace the most important parts are the *first few lines and the last few lines.* The first line declares what the bug is and where it is. The last few lines point to the specific problem point. If the trace has reasonable length then the intermediate parts of the trace are the execution path which causes the problem.

**Exercise: Click on a sample of the following bug classes:**

- `DEAD_STORE`
- `NULL_DEREFERENCE`



**List of bugs found**

```
 1. phantasia/fight.c:132: error: DEAD_STORE The value written to &flockcnt (type int) is never used. (source | trace)
 2. sail/assorted.c:104: error: DEAD_STORE The value written to &rhits (type int) is never used. (source | trace)
 3. hangman/getguess.c:82: error: UNINITIALIZED_VALUE The value read from ch was never initialized. (source | trace)
 4. hangman/getguess.c:83: error: UNINITIALIZED_VALUE The value read from ch was never initialized. (source | trace)
 5. hunt/huntd/expl.c:69: error: NULL_DEREFERENCE pointer `ep` last assigned on line 60 could be null and is dereferenced at line 69, column 2. (source | trace)
 6. sail/dr_main.c:82: error: UNINITIALIZED_VALUE The value read from nat[_] was never initialized. (source | trace)
 7. mille/comp.c:243: error: DEAD_STORE The value written to &safe (type short) is never used. (source | trace)
 8. mille/comp.c:113: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
 9. mille/comp.c:169: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
10. mille/comp.c:193: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
11. mille/comp.c:196: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
12. mille/comp.c:210: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
13. mille/comp.c:249: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
14. mille/comp.c:260: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
15. mille/comp.c:263: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
16. mille/comp.c:271: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
17. mille/comp.c:272: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
18. mille/comp.c:280: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
19. mille/comp.c:282: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
20. mille/comp.c:287: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
21. mille/comp.c:288: error: UNINITIALIZED_VALUE The value read from count[_] was never initialized. (source | trace)
22. mille/comp.c:182: error: UNINITIALIZED_VALUE The value read from playit[_] was never initialized. (source | trace)
23. sail/dr_3.c:178: error: UNINITIALIZED_VALUE The value read from col[_] was never initialized. (source | trace)
24. sail/dr_3.c:180: error: UNINITIALIZED_VALUE The value read from dir[_] was never initialized. (source | trace)
25. sail/dr_3.c:182: error: UNINITIALIZED_VALUE The value read from drift[_] was never initialized. (source | trace)
26. sail/dr_3.c:176: error: UNINITIALIZED_VALUE The value read from row[_] was never initialized. (source | trace)
27. adventure/wizard.c:65: error: NULL_DEREFERENCE pointer `tptr` last assigned on line 63 could be null and is dereferenced at line 65, column 8. (source | trace)
28. trek/events.c:229: error: UNINITIALIZED_VALUE The value read from ix was never initialized. (source | trace)
29. trek/events.c:235: error: UNINITIALIZED_VALUE The value read from ix was never initialized. (source | trace)
30. trek/events.c:290: error: UNINITIALIZED_VALUE The value read from ix was never initialized. (source | trace)
31. trek/events.c:296: error: UNINITIALIZED_VALUE The value read from ix was never initialized. (source | trace)
32. trek/events.c:171: error: UNINITIALIZED_VALUE The value read from iy was never initialized. (source | trace)
33. trek/events.c:176: error: UNINITIALIZED_VALUE The value read from iy was never initialized. (source | trace)
34. trek/events.c:229: error: UNINITIALIZED_VALUE The value read from iy was never initialized. (source | trace)
35. trek/events.c:235: error: UNINITIALIZED_VALUE The value read from iy was never initialized. (source | trace)
36. trek/events.c:290: error: UNINITIALIZED_VALUE The value read from iy was never initialized. (source | trace)
37. trek/events.c:296: error: UNINITIALIZED_VALUE The value read from iy was never initialized. (source | trace)
38. sail/lo_main.c:83: error: RESOURCE_LEAK resource of type `_IO_FILE` acquired by call to `fopen()` at line 69, column 12 is not released after line 83, column 9. (source | trace)
39. sail/lo_main.c:84: error: RESOURCE_LEAK resource of type `_IO_FILE` acquired by call to `fopen()` at line 69, column 12 is not released after line 84, column 9. (source | trace)
40. sail/lo_main.c:90: error: UNINITIALIZED_VALUE The value read from log.l_netpoints was never initialized. (source | trace)
41. sail/lo_main.c:85: error: UNINITIALIZED_VALUE The value read from log.l_uid was never initialized. (source | trace)
42. monop/prop.c:76: error: NULL_DEREFERENCE pointer `op` last assigned on line 73 could be null and is dereferenced at line 76, column 2. (source | trace)
43. sail/pl_main.c:108: error: UNINITIALIZED_VALUE The value read from nat[_] was never initialized. (source | trace)
44. trek/snova.c:90: error: UNINITIALIZED_VALUE The value read from qx was never initialized. (source | trace)
45. trek/snova.c:136: error: UNINITIALIZED_VALUE The value read from qx was never initialized. (source | trace)
46. trek/snova.c:149: error: UNINITIALIZED_VALUE The value read from qx was never initialized. (source | trace)
47. trek/snova.c:154: error: UNINITIALIZED_VALUE The value read from qx was never initialized. (source | trace)
48. trek/snova.c:155: error: UNINITIALIZED_VALUE The value read from qx was never initialized. (source | trace)
49. trek/snova.c:90: error: UNINITIALIZED_VALUE The value read from qy was never initialized. (source | trace)
50. trek/snova.c:136: error: UNINITIALIZED_VALUE The value read from qy was never initialized. (source | trace)
51. trek/snova.c:149: error: UNINITIALIZED_VALUE The value read from qy was never initialized. (source | trace)
52. trek/snova.c:154: error: UNINITIALIZED_VALUE The value read from qy was never initialized. (source | trace)
53. trek/snova.c:155: error: UNINITIALIZED_VALUE The value read from qy was never initialized. (source | trace)
54. sail/dr_5.c:88: error: DEAD_STORE The value written to &c3 (type int) is never used. (source | trace)
55. trek/klmove.c:169: error: DEAD_STORE The value written to &fudgev (type int) is never used. (source | trace)
```

Figure 1: Basic Infer Report for BSDGames

- `UNINITIALIZED_VALUE`
- `RESOURCE_LEAK`
- `NULL_DEREFERENCE`
- `MEMORY_LEAK`
- `USE_AFTER_FREE`

**For each of the bug classes consider the following:**

1. Is the bug report correct?
2. If not, can you think of a reason why it failed?
   This is an important skill for learning how to quickly dismiss bug reports.

In particular examine the following interesting *specific* bugs:

- See the whiteboard :-)

However it is important that you learn how to:

1. efficiently triage the report when you have a lot of records;
2. efficiently follow the reasoning of an Infer report trace.

## 4.2 What Have We Learnt So Far?

1. How to install and run Infer using the default checkers;
2. How to check the Infer man page and online documentation;
3. The location and contents of the Infer analysis in `./infer-out`
4. How to find the JSON bug report;
5. How to generate a HTML report of the bugs found;
6. How to read the bug traces for a variety of bug classes;

## 4.3 What Next?

In the next lab we will be looking at Integer bugs and Buffer-overflows.

## 5 Lab Two: Inferbo

The purpose of this lab is to use Infer to find:

- **Integer Bugs -** overflows, sign vs unsigned bugs, sign extension bugs etc;
- **Buffer Overflows**: memory corruption bugs

Infer uses an experimental checker called "Inferbo" to do this analysis. This checker uses Abstract Interpretation Theory to analyse the effects of integer expressions and their application to memory structures such as arrays.

### 5.1 Running Inferbo

Like all checkers , we can run *Inferbo* as an additional analysis *or* on its own.

- As an **additional** checker:
  `# infer run --bufferoverun -- make`
- **Inferbo only**:
  `# infer run --bufferoverrun-only -- make`

**Exercise: Working with Inferbo**

For each of the targets run Inferbo on its own and generate HTML bug reports. Open each of the bug reports and examine the various bug *classes*.

**Remember:**

- Delete any previous Infer data:
  `# rm -rf ./infer-out`
  Otherwise the results of previous analysis may leak into the reports.
- Clean the build:
  `# make clean`
  Otherwise Infer won't analyze anything.

You should see a summary like:

```
...

Summary of the reports

BUFFER_OVERRUN_L3: 14
BUFFER_OVERRUN_L2: 9
INFERBO_ALLOC_MAY_BE_BIG: 9
INTEGER_OVERFLOW_L2: 8
INFERBO_ALLOC_IS_ZERO: 4
INFERBO_ALLOC_MAY_BE_NEGATIVE: 1
INTEGER_OVERFLOW_R2: 1
BUFFER_OVERRUN_L1: 1
```

Notice that the various bug classes have suffixes like “L1” and “L3”. I'm not exactly sure what they mean but I strongly suspect that they relate to specific class variations of a bug that is being reported.

A few specific things to know:

- **Understanding Traces**: The first few lines of a `BUFFER_OVERRUN` trace are very important. It tells you why it thinks there is a memory error. For example:
  `Offset: [3, 18] Size: 16` is interpreted as the buffer is sized 16 but the index range is predicted by analysis to vary between 3 and 18.
- **Confusing arithmetic**: Some maths expressions seem to confuse Infer and therefore the range calculations seem to be in error. In particular bit operations and union structures seem a problem.
- **Precision**: Remember this is *static* analysis. Runtime execution may be protective of the problem identified. However, a lot of bug reports would be eliminated if the source code had proper bounds checking in place.

In particular examine the following interesting *specific* bugs:

- See whiteboard :-)

## 5.2 What have we learnt so far?

In my experience the Inferbo reports are always worth thinking through. They often can be eliminated with simple bounds checking at the identified problem site.

## 5.3 What Next?

The next section we will look at *Quandary* - a static source-sink or taint checker.

## 6 Lab Three: Quandary

Quandary is an experimental checker in Infer. It performs *static taint analysis*, also called *source-sink* analysis. The essential idea is that you can specify a *data-flow* between a

- **source** function - that is the return value of the function;
- **sink** function: to a specific argument or any argument of that function.

This is a really interesting checker from a security auditing perspective. For example, it allows you to specify a test from attacker-controlled data (for example, environment variables) to security sensitive functions (for example, `execve` functions).

For technical reasons this is also an impressive checker because it does this analysis *statically* and efficiently.

### 6.1 Running Quandary

Like all checkers , we can run *Quandary* as an additional analysis *or* on its own.

- As an **additional** checker:
  `# infer run --quandary --keep-going -- make`
- **Inferbo only**:
  `# infer run --quandary-only --keep-going -- make`

**Exercise: Working with Quandary - Part One**

For each of the targets run Quandary on its own and generate HTML bug reports. Open each of the bug reports and examine the various bug *classes*. Start with `BSDGames` in this example.

**Remember:**

- Delete any previous Infer data:
  `# rm -rf ./infer-out`
  Otherwise the results of previous analysis may leak into the reports.
- Clean the build:
  `# make clean`
  Otherwise Infer won't analyze anything.

You should see a summary like:

```
...
Summary of the reports

SHELL_INJECTION: 4
```

By default, Quandary looks for data-flows between environment variables (as returned by `gentenv()`) and the `execve()` functions. These data-flows represent command injection vulnerabilities if there is no sanitisation of the environment variable.

**Exercise: Working with Quandary - Part Two**

In this example we are going to define our own source-sink data-flow we are interested in.

- Create the following JSON file: `.inferconfig` in the root of the target repo.
- Insert the following contents:

```
{
    "quandary-sources": [
    {
        "procedure": "getenv",
        "kind": "Logging"
    }
    ],
    "quandary-sinks": [
    {
        "procedure": "atoi",
        "kind": "Logging"
     }
     ]
}
```

- Re-run the Quandary analysis:
  `# infer run --keep-going --quandary-only -- make`

**Remember:**

- Delete any previous Infer data:
  `# rm -rf ./infer-out`
  Otherwise the results of previous analysis may leak into the reports.
- Clean the build:
  `# make clean`
  Otherwise Infer won't analyze anything.

You should see a summary like:

```
...
Summary of the reports

SHELL_INJECTION: 4
QUANDARY_TAINT_ERROR: 3
```

In this example we have picked three examples of a data-flow between an environment variable and a function we defined as interesting: `atoi()`. This is an interesting flow because the function `atoi()` has no error handling and fails silently. If you click on the trace for these reports you should indeed see an actual data-flow analysis that reveals a bug. In fact any use of the function `atoi()` should be considered a bug however when confronted with lots of bugs these ones found by Quandary should be prioritised.

It is also possible to be specific about *which argument* of the sink function you want to trace. By default, a data-flow to *any argument* of the sink function triggers a report.

## 6.2 Review

What have we learnt so far?

- Quandary is a static taint tracing checker;
- Quandary finds by default potential *command injection* vulnerabilities;
- How to create our own source-sink analyses using the JSON file `.inferconfig`

## 6.3 What Next?

In the next section we are going to learn how to use Infer's *linting* capabilities.

# 7 Lab Four: Finding Syntactic Bugs

`Linting` is the process of finding *syntactic* problems in code. For example, any use of the function `strcpy()` is now considered unsafe and should be replaced with a more robust string copy method. Searching for uses of this function is linting.

Infer has a robust linting capability. It uses a domain specific language called "AL" which is very advanced. We are only going to cover the surface of what you can do with AL. In particular, you should look at the fairly extensive documentation on AL here: `https://fbinfer.com/docs/linters.html` and at the linters that come with Infer by default - from the root of the Infer installation find them as follows: `find . -name "*.al"`.

## 7.1 Creating a linter: Part One

In this section we are going to create a very simple linter using AL to find the unsafe functions `atoi()` and `strcpy()`. These are just two obvious choices of potentially many functions you would want to audit.

- Create the following file: `mylinter.al`
- Add the following content: we will go through what this means presently.

```
DEFINE-CHECKER UNSAFE_FUNCTION_USAGE = {
LET isatoi = call_function(REGEXP("\\batoi"));
SET report_when =
isatoi;
SET message = "Unsafe function usage - atoi()";
SET suggestion = "See Reference - TODO";
SET severity = "ERROR";
};

DEFINE-CHECKER UNSAFE_FUNCTION_USAGE = {
LET isstrcpy = call_function(REGEXP("\\bstrcpy"));
SET report_when =
isstrcpy;
SET message = "Unsafe function usage - strcpy()";
SET suggestion = "See Reference - TODO";
SET severity = "ERROR";
};
```

- For each of the targets, run Infer as follows using the command below. Remember to `make clean` and remove any prior `infer-out` directory.

```
# infer run --keep-going --linters-only --linters-def-file mylinter.al -- make
```

You should see something like the following:

```
...
Summary of the reports

UNSAFE_FUNCTION_USAGE: 21
```

Create a HTML report and have look at some of the traces.

## 7.2 Understanding AL

AL is a pattern matching language that operates on Clang's *Abstract Syntax Tree* (AST). The type of pattern matching is called *Computational Tree Logic* (CTL). CTL is a way of specifying pattern matching on tree structures. See "AL Formulas" at `https://fbinfer.com/docs/linters.html` for a detailed description of how to do this.

The AST is a tree of the syntactic structure of the compiled source code produced by Clang. Therefore we need to specify the syntactic pattern we are interested in finding in the AST in terms of AL formulas. We will look at this in a moment.

Linting in AL consists of specifying one or more checkers. A checker consists of

- One or more **formulas** that are triggered by a syntactic pattern in the AST;
- A **report rule** which causes a bug report to be raised. The report rule consists of a combination of formulas that when true trigger the report;
- A **message** which summarises the issue;
- A **suggestion** of what to do about the issue;
- A **severity** to indicate how serious the issue is.

Lets analyse the following:

```
DEFINE-CHECKER UNSAFE_FUNCTION_USAGE = {
LET isstrcpy = call_function(REGEXP("\\bstrcpy"));
SET report_when =
isstrcpy;
SET message = "Unsafe function usage - strcpy()";
SET suggestion = "See Reference - Please read:
https://stackoverflow.com/questions/41869803/what-is-the-best-alternative-to-strncpy";
SET severity = "ERROR";
};
```

Observe the following:

- We create the checker `UNSAFE_FUNCTION_USAGE`
- We create a simple formula `isstrcpy`
- We specify an AST pattern using `call_function` which matches Clang's AST node for function call sites;
- We further specify a regex for function names which in our case matches `strcpy`
- We specify a report to be raised when the `isstrcpy` formulae is satisfied;
- The remaining lines are straight forward

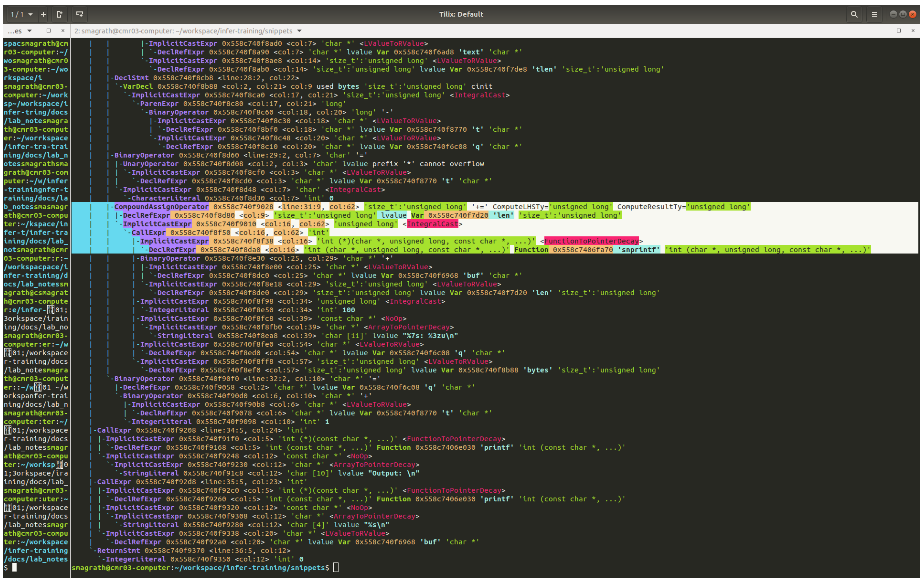
This example demonstrates the basic structure of a linting checker. We will now look at a more involved example.

## 7.3 Creating a Linter: Part Two

In this section you will create a linter for the "unsafe pointer accumulation using `snprintf()`" bug. Consider the following code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define SIZE 1024

/* Snippet to demonstrate the snprintf() pointer accumulation bug */
int main(int argc, char** argv) {
    char* buf = malloc(SIZE);
    char* text = NULL;
    char* q = text; //indexing ptr into text
    size_t len = 0; //of bytes written to buf
    size_t tlen = 0; //length on source text

    text = malloc(SIZE);
    strcpy(text, "the quick brown fox jumped over the lazy dog");
    q = text;
    tlen = strlen(text);
    bool islast = false;
    printf("Input String = \"%s\"", text);

    while (!islast && ((q-text) < tlen) && (len < SIZE)) {
        char* t = index(q, ' '); //find the space
        if (t == NULL) {
            islast = true; //means we are at the last word
            t = text + tlen; //point at the end of the string
        }
        size_t bytes = (t-q); //bytes to write
        *t = '\0'; //re-write the space so we can use the word
        //BUG: Unsafe pointer accummulation using snprintf()
        len += snprintf(buf+len, 100, "%7s: %3zu\n", q, bytes);
        q = t + 1; //next word
    }
    printf("Output: \n");
    printf("%s\n", buf);
    return 0;
}
```

The bug we are interesting in finding involves this line:
`len += snprintf(buf+len, 100, "%7s: %3zu\n", q, bytes);`
The source of the bug is assuming that the `snprintf()` function returns the number of bytes written to the buffer. This function does not make that guarantee - if the size of the string exceeds 100 bytes in this example then the return value is the number of bytes that *would* have been written - a larger


Tilix: Default
2: smagrath@cmr03-computer: ~/workspace/infer-training/snippets

```
|   |       |-ImplicitCastExpr 0x558c740f8ad0 <col:7> 'char *' <LValueToRValue>
|   |       | `-DeclRefExpr 0x558c740f8a90 <col:7> 'char *' lvalue Var 0x558c740f6ad8 'text' 'char *'
|   |       `-ImplicitCastExpr 0x558c740f8ae8 <col:14> 'size_t':'unsigned long' <LValueToRValue>
|   |         `-DeclRefExpr 0x558c740f8ab0 <col:14> 'size_t':'unsigned long' lvalue Var 0x558c740f7de8 'tlen' 'size_t':'unsigned long'
|   |-DeclStmt 0x558c740f8cb8 <line:28:2, col:22>
|   | `-VarDecl 0x558c740f8b88 <col:2, col:21> col:9 used bytes 'size_t':'unsigned long' cinit
|   |   `-ImplicitCastExpr 0x558c740f8ca0 <col:17, col:21> 'size_t':'unsigned long' <IntegralCast>
|   |     `-ParenExpr 0x558c740f8c80 <col:17, col:21> 'long'
|   |       `-BinaryOperator 0x558c740f8c60 <col:18, col:20> 'long' '-'
|   |         |-ImplicitCastExpr 0x558c740f8c30 <col:18> 'char *' <LValueToRValue>
|   |         | `-DeclRefExpr 0x558c740f8bf0 <col:18> 'char *' lvalue Var 0x558c740f8770 't' 'char *'
|   |         `-ImplicitCastExpr 0x558c740f8c48 <col:20> 'char *' <LValueToRValue>
|   |           `-DeclRefExpr 0x558c740f8c10 <col:20> 'char *' lvalue Var 0x558c740f6c08 'q' 'char *'
|   |-BinaryOperator 0x558c740f8d60 <line:29:2, col:7> 'char' '='
|   | |-UnaryOperator 0x558c740f8d08 <col:2, col:3> 'char' lvalue prefix '*' cannot overflow
|   | | `-ImplicitCastExpr 0x558c740f8cf0 <col:3> 'char *' <LValueToRValue>
|   | |   `-DeclRefExpr 0x558c740f8cd0 <col:3> 'char *' lvalue Var 0x558c740f8770 't' 'char *'
|   | `-ImplicitCastExpr 0x558c740f8d48 <col:7> 'char' <IntegralCast>
|   |   `-CharacterLiteral 0x558c740f8d30 <col:7> 'int' 0
|   |-CompoundAssignOperator 0x558c740f9028 <line:31:9, col:62> 'size_t':'unsigned long' '+=' ComputeLHSTy='unsigned long' ComputeResultTy='unsigned long'
|   | |-DeclRefExpr 0x558c740f8d80 <col:9> 'size_t':'unsigned long' lvalue Var 0x558c740f7d20 'len' 'size_t':'unsigned long'
|   | `-ImplicitCastExpr 0x558c740f9010 <col:16, col:62> 'unsigned long' <IntegralCast>
|   |   `-CallExpr 0x558c740f8f50 <col:16, col:62> 'int'
|   |     |-ImplicitCastExpr 0x558c740f8f38 <col:16> 'int (*)(char *, unsigned long, const char *, ...)' <FunctionToPointerDecay>
|   |     | `-DeclRefExpr 0x558c740f8da0 <col:16> 'int (char *, unsigned long, const char *, ...)' Function 0x558c7406fa70 'snprintf' 'int (char *, unsigned long, const char *, ...)'
|   |     |-BinaryOperator 0x558c740f8e30 <col:25, col:29> 'char *' '+'
|   |     | |-ImplicitCastExpr 0x558c740f8e00 <col:25> 'char *' <LValueToRValue>
|   |     | | `-DeclRefExpr 0x558c740f8dc0 <col:25> 'char *' lvalue Var 0x558c740f6968 'buf' 'char *'
|   |     | `-ImplicitCastExpr 0x558c740f8e18 <col:29> 'size_t':'unsigned long' <LValueToRValue>
|   |     |   `-DeclRefExpr 0x558c740f8de0 <col:29> 'size_t':'unsigned long' lvalue Var 0x558c740f7d20 'len' 'size_t':'unsigned long'
|   |     |-ImplicitCastExpr 0x558c740f8f98 <col:34> 'unsigned long' <IntegralCast>
|   |     | `-IntegerLiteral 0x558c740f8e50 <col:34> 'int' 100
|   |     |-ImplicitCastExpr 0x558c740f8fc8 <col:39> 'const char *' <NoOp>
|   |     | `-ImplicitCastExpr 0x558c740f8fb0 <col:39> 'char *' <ArrayToPointerDecay>
|   |     |   `-StringLiteral 0x558c740f8ea8 <col:39> 'char [11]' lvalue "%7s: %3zu\n"
|   |     |-ImplicitCastExpr 0x558c740f8fe0 <col:54> 'char *' <LValueToRValue>
|   |     | `-DeclRefExpr 0x558c740f8ed0 <col:54> 'char *' lvalue Var 0x558c740f6c08 'q' 'char *'
|   |     `-ImplicitCastExpr 0x558c740f8ff8 <col:57> 'size_t':'unsigned long' <LValueToRValue>
|   |       `-DeclRefExpr 0x558c740f8ef0 <col:57> 'size_t':'unsigned long' lvalue Var 0x558c740f8b88 'bytes' 'size_t':'unsigned long'
|   `-BinaryOperator 0x558c740f90f0 <line:32:2, col:10> 'char *' '='
|     |-DeclRefExpr 0x558c740f9058 <col:2> 'char *' lvalue Var 0x558c740f6c08 'q' 'char *'
|     `-BinaryOperator 0x558c740f90d0 <col:6, col:10> 'char *' '+'
|       |-ImplicitCastExpr 0x558c740f90b8 <col:6> 'char *' <LValueToRValue>
|       | `-DeclRefExpr 0x558c740f9078 <col:6> 'char *' lvalue Var 0x558c740f8770 't' 'char *'
|       `-IntegerLiteral 0x558c740f9098 <col:10> 'int' 1
|-CallExpr 0x558c740f9208 <line:34:5, col:24> 'int'
| |-ImplicitCastExpr 0x558c740f91f0 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
| | `-DeclRefExpr 0x558c740f9168 <col:5> 'int (const char *, ...)' Function 0x558c7406e030 'printf' 'int (const char *, ...)'
| `-ImplicitCastExpr 0x558c740f9248 <col:12> 'const char *' <NoOp>
|   `-ImplicitCastExpr 0x558c740f9230 <col:12> 'char *' <ArrayToPointerDecay>
|     `-StringLiteral 0x558c740f91c8 <col:12> 'char [10]' lvalue "Output: \n"
|-CallExpr 0x558c740f92d8 <line:35:5, col:23> 'int'
| |-ImplicitCastExpr 0x558c740f92c0 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
| | `-DeclRefExpr 0x558c740f9260 <col:5> 'int (const char *, ...)' Function 0x558c7406e030 'printf' 'int (const char *, ...)'
| |-ImplicitCastExpr 0x558c740f9320 <col:12> 'const char *' <NoOp>
| | `-ImplicitCastExpr 0x558c740f9308 <col:12> 'char *' <ArrayToPointerDecay>
| |   `-StringLiteral 0x558c740f9280 <col:12> 'char [4]' lvalue "%s\n"
| `-ImplicitCastExpr 0x558c740f9338 <col:20> 'char *' <LValueToRValue>
|   `-DeclRefExpr 0x558c740f92a0 <col:20> 'char *' lvalue Var 0x558c740f6968 'buf' 'char *'
`-ReturnStmt 0x558c740f9370 <line:36:5, col:12>
  `-IntegerLiteral 0x558c740f9350 <col:12> 'int' 0
smagrath@cmr03-computer:~/workspace/infer-training/snippets$
```

Figure 2: AST Screen shot

number. Because we are in a loop the pointer value into `buf` is now wrong and we potentially could be writing into memory outside of `buf`.

We could just grep for `+= snprintf(` but this is not as robust nor integrated with Infer's bug reports. Therefore, we will make a linter to search for this bug pattern.

### 7.3.1 Step One: Understanding the AST

The first thing we need to do is understand the specific AST for this bug pattern. We can get Clang to print the AST when it compiles the snippet above:

```
# clang -c -Xclang -ast-dump -fsyntax-only test_2.c
```

You should see something similar to Figure 2. The part of the AST we are interested is highlighted. Study this closely and note the following:

- The `+=` accumulation operation is a `CompoundAssignOperator` node in the AST.
- The function call involves `CallExpr` node with a child node `DeclRefExpr` whose value is `snprintf`.
- Understand the parent-child relationships here in the AST.

The next step is to specify a matching pattern that captures this syntactic structure.

### 7.3.2 Step Two: Writing the Linter in AL

There are various ways of writing this linter. We are going to write a simple version that over-approximates. That is, it can (and does) raise false positives. A more precise linter could be written but as we will see this has its own problems.

Add the following code to your `mylinter.al` file.

```
DEFINE-CHECKER UNSAFE_SNPRINTF_ACCUMMULATION = {
    LET issnprintf = call_function(REGEXP("\\bsnprintf"));
    LET isaccum = is_node("CompoundAssignOperator");
    SET report_when =
        issnprintf AND isaccum
    HOLDS-EVENTUALLY;
    SET message = "Possible Unsafe pointer accummulation using snprintf()";
    SET suggestion = "See Reference - See
    https://access.redhat.com/blogs/766093/posts/1976193";
    SET severity = "ERROR";
};
```

Note the following:

1. We identify the `snprintf` function using an AL *predicate* called `call_function`. This selects function call nodes in the AST. We further filter using a regular expression.

2. We specifically select for `CompoundAssignOperator` nodes in the AST.

3. The particular tree pattern we want is where the call function is a child of the `CompoundAssignOperator` node. We can do this with the `HOLDS-EVENTUALLY` CTL operator. Refer to the "AL Formulas" section of `https://fbinfer.com/docs/linters.html`. In practice when developing linters you will need to become proficient with these formulas and linter debugging.

4. Is this pattern completely correct to catch the bug we are after?

**Exercise: Part Two: Linting for Unsafe Pointer Accumulation**

1. Use Infer with the updated linter file for the code snippet.
   Verify that it identifies the problem line in the bug snippet.
   `# infer run --linters-def-file ../mylinter.al -- clang -Wall -o test_2 test_2.c`

2. For each of the targets run the linter and examine the bug reports:
   `# infer run --keep-going --linters-only --linters-def-file ../mylinter.al -- make`

3. You should find instances of false and true positives.
   Can you explain why the false positive occurred?

4. What are the limits of linting? Could you write a version of the `snprintf()` bug program that would not be caught by the linter?

We are really only providing an introduction to Infer's linting capabilities here. You should examine the pre-provided linters that come with Infer to learn more advanced and interesting examples. To find these linters simply `grep -rn '*.al'` in the Infer installation.

### 7.4 Review

What have we learnt so far?

- We learnt that linters are used to find syntactic bugs;
- We have developed some simple linters using AL;
- We have used Clang to generate AST information for code we are interested in;
- We used the linters to find syntactic bugs in our targets.

# 8 Final Notes

The point of these lab notes is to give you an overview of Infer and some proficiency in using it to audit source code targets. You should develop a sense of Infer's strengths and weaknesses and understand its place amongst other tools for source code auditing. Static analysis is just one useful technique for finding bugs in software. Being able to use Infer well is an important and worthwhile skill to develop.

## 8.1 Using Infer with CMake

Getting Infer to work with the CMake build system can be difficult at first. The basic approach is to get CMake to generate a compilation database which Infer can then use.

1. Edit the `CmakeLists.txt` file to add the following:
   ```
   set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
   ```
2. Run the CMake build:
   ```
   # CC=clang CXX=clang++ cmake -DCMAKE_EXPORT_COMPILE=1
   ```
   This will export the JSON file `compile_commands.json`
3. Run the following commands:
   ```
   # infer capture -- make
   # infer analyze
   # infer explore --html
   ```
   Note: The Infer "run" mode is simply the "capture & analyze" modes;

## 8.2 Further Work

- How to write *models* of functions for use by Infer;
- How to write custom *checkers* in OCaml for use in Infer;
- How to build Infer from source;
- How to integrate into a continuous integration pipeline;
- Microsoft builds.

## A Installing Infer

We will download and install Infer without using Docker. We assume you are using Ubuntu as your OS environment.

We are going to install Infer. Normally you would follow the instructions on Infer's "Getting Started" website - `https://fbinfer.com/docs/getting-started.html` - however in my experience the documented process never worked.

**Complete the Following: Installing Infer**

1. Download the version 0.17.0 tarball from Github:
   `https://github.com/facebook/infer/releases`
2. Extract it in a home directory
   `# cd ~/bin && tar xvf infer-0.17.0.tar.gz`
3. Add the Infer binary to the PATH environment in `.bashrc`

   ```
   PATH=~/bin/infer/bin:$PATH
   export PATH
   ```

4. Test that your installation runs. In a **new shell** run Infer - you should see the following

   ```
   # infer
   Nothing to compile. Have you run ‘infer capture‘? Try cleaning the build first.
   There was nothing to analyze.

   No issues found
   ```

5. Ensure that the Infer man pages are accessible by running the following command:
   `# man infer`
   The man pages are very comprehensive but also somewhat confusing. Another source of information is the Infer documentation web page - `https://fbinfer.com/docs/getting-started.html`. There are a number of important pages that cover more advanced operations.

That completes the Infer installation.

## B Building the Audit Targets Natively

We will use the following three Linux games to explore Infer.

- **Angband** - `https://rephial.org/`
- **Skynet** - `https://github.com/cloudwu/skynet`
- **BSDGames** - `https://github.com/vattam/BSDGames`

The targets were chosen because together they exhibit a good diversity of bugs and have modest build and analysis requirements. Some targets because of size and complexity issues require large amounts of memory (i.e. > 16 GB) and compute time (i.e. > 10 minutes) which is not appropriate for the typical resources available for this lab.

For the purposes of the lab we provide the source code tarballs required. The reason for this is to make sure we are auditing the same fixed version of code with known bugs.

### B.0.1 Angband

We are going to download, extract and build the game from source code.

**Complete the Following: Build Angband - Part One**

1. Make an audit directory
   `# mkdir ~/audit`
2. Download and move the game into the audit directory- TODO
3. Extract it
   `# tar xvf angband-4.2.0.tar.gz`
4. Change directory to the root of the game repo
   `# cd angband-4.2.0`

At this point we need to investigate how to build the game. Run the following command to examine what build options are available.

**NOTE:** In principle you would want to enable as many of the options that make sense in order to be able to analyse as much of the code base as possible.

Examine the options available in building the code:

```
# ./configure --help

...

Optional Features:
--disable-option-checking  ignore unrecognized --enable/--with options
--disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
--enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
--enable-curses         Enables Curses frontend (default: enabled)
--enable-x11            Enables X11 frontend (default: enabled)
--enable-sdl2           Enables SDL2 frontend (default: disabled)
--enable-sdl            Enables SDL frontend (default: disabled)
--enable-win            Enables Windows frontend (default: disabled)
--enable-test           Enables test frontend (default: disabled)
--enable-stats          Enables stats frontend (default: disabled)
--enable-sdl2-mixer     Enables SDL2 mixer sound support (default: disabled
unless SDL2 enabled)
```

```
--enable-sdl-mixer      Enables SDL mixer sound support (default: disabled
unless SDL enabled)
--disable-ncursestest       Do not try to compile and run a test ncurses program
--disable-sdl2test       Do not try to compile and run a test SDL2 program
--disable-sdltest       Do not try to compile and run a test SDL program

Optional Packages:
--with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
--without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
--with-setgid=NAME      install angband as group NAME
--with-private-dirs     use private scorefiles/savefiles
--with-no-install       don't install, just run in-place
--with-ncurses-prefix=PFX   Prefix where ncurses is installed (optional)
--with-ncurses-exec-prefix=PFX Exec prefix where ncurses is installed (optional)
--with-x                use the X Window System
--with-sdl2-prefix=PFX   Prefix where SDL2 is installed (optional)
--with-sdl2-exec-prefix=PFX Exec prefix where SDL2 is installed (optional)
--with-sdl-prefix=PFX   Prefix where SDL is installed (optional)
--with-sdl-exec-prefix=PFX Exec prefix where SDL is installed (optional)
```

**Complete the Following: Build Angband - Part Two**

Now we need to test that we can properly build the game such that Infer can analyse it:

1. **TODO: check - we need to make sure Clang and all the libraries are installed**
2. **Build Configuration**: In this case we will configure the build systems as follows:
   `# CC=clang CXX=clang++ ./configure --enable-sdl2`
3. **Build**: Make sure that Clang is being used to compile the code.
   `# make`
   There should be no errors.

### B.0.2 Skynet

We are going to download, extract and build the game from source code.

**Complete the Following: Build Skynet**

1. Download and move the game into the audit directory- TODO
2. Extract it
   `# tar xvf skynet.tar.gz`

3. **TODO: Install dependencies (detail ...)**
4. Change directory to the root of the game repo
   `# cd skynet`
5. Build the code:
   `# CC=clang CXX=clang++ make linux`

Verify that the code was built without errors and that the compilation used Clang.

### B.0.3 BSDGames

We are going to download, extract and build the game from source code.

**Complete the Following: Build BSDGames**

1. Download and move the game into the audit directory- TODO
2. Extract it
   `# tar xvf BSDgames.tar.gz`
3. **TODO: Install the library dependencies: ncurses, Lex, Yacc, libcrypto**
4. Change directory to the root of the game repo
   `# cd BSDGames`
5. Run the build configuration tool:
   `# ./configure`
6. Edit the file `./Makeconfig`
   Change the compilers from `gcc/g++` to `clang/clang++`
   Note: the usual method for specifying the compilers as environment variables does not seem effective.
7. Build the code:
   `# make`
   You **will** see compilation problems but this seems to be expected with this repo.

Verify that the code was mostly built without errors and that the compilation used Clang.

At this point we have prepared a set of targets ready for analysis with Infer. In practice you should not underestimate the effort required to properly prepare the target build system for analysis. You would normally try to enable as many build options as possible to maximise the amount of code that can be analysed. Depending on your objectives you may also want to build and analyse the library dependencies as well.