Smatch

Smatch - Static Code Analyzer--Bharath P (talk) 07:00, 11 April 2017 (UTC)

Introduction

Smatch is an open-source static analysis tool based on sparse, the checker used by the Linux kernel. Smatch tool is used for finding bugs at source code.

Smatch Tool

Smatch Tool is a Static analysis one based on spare, **it is used for finding bugs at source code without actually running it.** It is great way to find bugs for beginner to work on staging driver this tool is really help full. you can get the smatch tool in git hub.

Prerequisites

1. Install the dependency

```
$ sudo apt-get install libsqlite3-dev
```

2. Download the smatch source

```
$ git clone http://repo.or.cz/smatch.git
```

3. Install the smatch

```
$ cd smatch
$ make
$ sudo make install PREFIX=/usr
```

Usage

Compile the source code with smatch as below

```
$ make C=1 CHECK="smatch -p=kernel" | tee warns.txt
```

The output will be redirected into a file called warns.txt in the current directory. Bugs will be reported in warns.txt if found.

Smatch!!!

Smatch -- The Source Matcher, brought to you by the good folks at KernelJanitors.

Smatch has been completely rewritten in C and now uses sparse as a C parser instead of gcc as a C parser. The new URL is http://repo.or.cz/w/smatch.git.

Here are the new instructions:

```
git clone git://repo.or.cz/smatch.git
cd smatch
make
cd ~/your/code/
make clean
make CHECK="~/path/to/smatch/smatch --full-path" \
        CC=~/path/to/smatch/cgcc | tee warns.txt
```

Except if you are using smatch against the kernel the command is:

```
make CHECK="~/path/to/smatch/smatch -p=kernel" C=1 \
        bzImage modules | tee warns.txt
```

`#tee command is used to store and view (both at the same time) the output of any other command.`

2.3.37 and after: Please set "CONFIG_DYNAMIC_DEBUG=n". That feature uses declared label things that mess up Smatch's flow analysis.

If you are using smatch on a different project then the most important thing is to build the list of functions which don't return. Do the first build using the --info parameter and use smatch_scripts/gen_no_return_funcs.sh to create this list. Save the resulting file under smatch_data/(your project).no_return_funcs and use -p=(your project) for the next smatch run.

If you are using smatch to test wine then use "-p=wine" to turn on the wine specific checks.

---

https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

Tool	Language(s)	Avail.	CCR	Finds or Checks for	As of
Smatch	C	free		simple scripts look for problems in simplified representation of code. primarily for Linux kernel code	Apr 2006

http://clang.llvm.org/features.html#performance

Clang Static Analyzer	C, Objective-C	free		Resports dead stores, memory leaks, null pointer deref, and more. Uses source annotations like "nonnull".	Aug 2010

Low Level Virtual Machine(**LLVM**)

Clang

http://clang.llvm.org/get_started.html

- Features and Goals

Features are:

End-User Features:

- Fast compiles and low memory use
- Expressive diagnostics
- GCC compatibility

Utility and Applications:

- Library based architecture
- Support diverse clients
- Integration with IDEs
- Use the LLVM 'BSD' License

Internal Design and Implementation:

- A real-world, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++
- Conformance with C/C++/ObjC and their variants

Using CLang for Static Code Analysis

Mac OS X

Pre-built binaries of Clang Static Analyzer are available on Mac OS X (10.5 or later). Since Xcode 3.2, users have been able to run the Clang Static Analyzer directly within Xcode. But you are always recommended to check out the latest build.

1. Download the latest build checker-276.tar.bz2.

2. Unpack the package anywhere.

```
$  tar -jxvf checker-276.tar.bz2 -C /any/where/you/wish/
```

3. There are two tools, **scan-build** and **scan-view** at the top of the **checker-276** directory.

- **scan-build**: scan-build is the high-level command line utility for running the analyzer
- **scan-view**: scan-view is used to view analysis results generated by scan-build

4. To run scan-build, you are suggested to add the checker-276 directory to your path.

```
$ sudo nano /etc/paths
// add the checker-276 directory to the existing path
$ echo $PATH
```

Otherwise, you have to specify a complete path for scan-build in the command.

Ubuntu

Clang Static Analyzer can also be easily installed using Ubuntu Package Manager.

```
$ sudo add-apt-repository ppa:kxstudio-team/builds
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install llvm-3.2 clang-3.2
```

If you install with apt-get, the path to **scan-build** has already been included in **$PATH** and you can directly run it without specifying its path in command line.

Clang Analyzer documentation asks linux users to manually build clang and llvm. You are not recommended to do that since it's super hard to build them.

Prerequisites

Install LLVM and Clang packages

```
$ sudo apt-get install llvm clang
```

Usage

The CLang compiler contains a static analyzer that can be invoked to analyze code that is either being compiled with CLang or gcc.

The easiest way to use this tool is to invoke the scan-build command while building the project.

```
$ scan-build make
```

By default **scan-build** picks builtin **clang** compiler. It is possible to invoke **scan-build** to use specific compiler with "--use-cc[=compiler path]" option.

```
$ scan-build --use-cc[=compiler path] make
```

Or for a single program

```
$ scan-build gcc -O -Wall -c t1.c t2.c
```

This not only compiles the code but also runs the analyzer in parallel.

The output results are not only displayed on the console but a report is also generated in HTML format that can be viewed to get a summary of the analysis as well as information on each of the identified issues. By default the report is generated in the "/tmp" directory but the location can be set using options to the scan-build tool.

Below are a list of some of the options to scan-build:

Option	Description

--use-cc[=compiler path]	Guess the default compiler for your C and Objective-C code. Use this option to specify an alternate compiler.
-o	Target directory for HTML report files. Subdirectories will be created as needed to represent separate "runs" of the analyzer. If this option is not specified, a directory is created in /tmp to store the reports.
-h (or no arguments)	Display all scan-build options.
-k or --keep-going	Add a "keep on going" option to the specified build command. This option currently supports make and xcodebuild. This is a convenience option; one can specify this behavior directly using build options.
-v	Verbose output from scan-build and the analyzer. A second and third "-v" increases verbosity, and is useful for filing bug reports against the analyzer.
-V	View analysis results in a web browser when the build command completes.
--use-analyzer Xcode OR --use-analyzer [path to clang]	scan-build uses the 'clang' executable relative to itself for static analysis. One can override this behavior with this option by using the 'clang' packaged with Xcode (on OS X) or from the PATH.

A complete list of options can be obtained by running scan-build with no arguments.

For more information refer to the scan-build page.

The Makefiles

Overview

The build system defines a set of conventions for the correct use of Makefiles in the kernel source directories. The correct use of Makefiles is driven by the concept of recursion.

In the recursion model, each Makefile within a directory includes the source code and any subdirectories to the build process. Each subdirectory follows the same principle. Developers can focus exclusively in their own work. They integrate their module with the build system by adding a very simple Makefile following the recursive model.

Makefile Conventions

The following conventions restrict how to add modules and Makefiles to the build system. These conventions guard the correct implementation of the recursive model.

- Each **source code directory must contain a single Makefile**. Directories ***without a Makefile are not considered source code directories***.
- The scope of every Makefile is restricted to the contents of that directory. A Makefile can only make a direct reference to its own files and subdirectories. Any file outside the directory has to be referenced in its home directory Makefile.
- Makefiles list the object files that are included in the link process. The build system finds the source file that generates the object file by matching the file name.
- Parent directories add their child directories into the recursion model.
- The root Makefile adds the directories in the kernel base directory into the recursion model.

Adding Source Files

A source file is added to the build system through its home directory Makefile. ***The Makefile must refer the source build indirectly, specifying the object file that results from the source file using the `obj-y` variable.*** For example, if the file that we want to add is a C file named `<file>.c` the following line should be added in the Makefile:

```
obj-y += <file>.o
```

Note

The same method applies for assembly files with .s extension.

Source files can be added conditionally using configuration options. For example, if the option `CONFIG_VAR` is set and it implies that a source file must be added in the compilation process, then the following line adds the source code conditionally:

```
obj-$(CONFIG_VAR) += <file>.o
```

Adding Directories

Add a subdirectory to the build system by editing the Makefile in its directory. The subdirectory is added using the `obj-y` variable. The correct syntax to add a subdirectory into the recursion is:

```
obj-y += <directory_name>/
```

The backslash at the end of the directory name signals the build system that a directory, and not a file, is being added to the recursion.

The conventions require us to add only one directory per line and never to mix source code with directory recursion in a single `obj-y` line. This helps keep the readability of the Makefile by making it clear when an item adds an additional lever of recursion.

Directories can also be conditionally added:

```
oby-$(CONFIG_VAR) += <directory_name>/
```

The subdirectory must contain its own Makefile following the rules described in Makefile Conventions

our tutorial promotes bad practices, you should avoid it IMHO.

In your rule here:

```
$(ODIR)/%.o: %.c $(DEPS)
```

You're telling make to look in the current directory while your source files are in the `src` directory, thus this pattern is never used and you have no suitable one.

Make sure you organize your project directory like this :

```
+ include/
|-- all .h files here
+ lib/
|-- all thirdparty library files (.a files) here
+ obj/
|-- all .o files will be placed here
+ src/
|-- all .c files here
+ Makefile
```

Then let's take the process step by step, using good practices.

Firstly, don't define anything if you don't need to. Make has a lot of predefined variables and functions that you should use before trying to do it manually. In fact, he has so many that you can compile a simple project without even having a Makefile in the directory at all !

1. Name your final target, that is, your executable:

   ```
   EXE = hellomake
   ```

2. List your source and build output directories:

```
SRC_DIR = src
OBJ_DIR = obj
```

3. List your source files:

```
SRC = $(wildcard $(SRC_DIR)/*.c)
```

4. From the source files, list the object files:

```
OBJ = $(SRC:$(SRC_DIR)/%.c=$(OBJ_DIR)/%.o) # patsubst is far less readable
# You can also do it like that
OBJ = $(addprefix $(OBJ_DIR)/, $(notdir $(SRC)))
```

5. Got some preprocessor flags to pass? Go on.

```
CPPFLAGS += -Iinclude # -I is a preprocessor flag, not a compiler flag
```

6. Got some compiler flags to add? Here you go.

```
CFLAGS += -Wall # some warnings about bad code
```

7. Got some linker flags?

```
LDFLAGS += -Llib # -L is a linker flag
```

8. Got some thirdparty libraries to link against?

```
LDLIBS += -lm # Left empty if no libs are needed
```

Ok, time to roll some recipes now that our variables are correctly filled.

It is widely spread that the default target should be called `all`. For simplicity, it shall be the first target in your Makefile, and its prerequisites shall be the target you want to build when writing `make` on the command line:

```
all: $(EXE)
```

Now list the prerequisites for building your executable, and fill its recipe to tell make what to do with these:

```
$(EXE): $(OBJ)
    $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

Some quick notes:

- `$(CC)` is a built-in variable already containing what you need when compiling and linking in C,
- To avoid linker errors, you *should* put `$(LDFLAGS)` **before** your object files and `$(LDLIBS)` **after**.
- `$(CPPFLAGS)` and `$(CFLAGS)` are **useless** here, the compilation phase is already over, it is the linking phase here.

Next step, since your source and object files don't share the same prefix, you need to tell make exactly what to do since its built-in rules don't cover your specific case:

```
$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
```

```
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

Ok, now the executable should build nicely, we can clean the build directory if we want:

```
clean:
    $(RM) $(OBJ)
```

Last thing. You should indicate whenever a rule does not produce any target output with the `.PHONY` sepcial rule:

```
.PHONY: all clean
```

Final result:

```
EXE = hellomake

SRC_DIR = src
OBJ_DIR = obj

SRC = $(wildcard $(SRC_DIR)/*.c)
OBJ = $(SRC:$(SRC_DIR)/%.c=$(OBJ_DIR)/%.o)

CPPFLAGS += -Iinclude
CFLAGS += -Wall
LDFLAGS += -Llib
LDLIBS += -lm

.PHONY: all clean

all: $(EXE)

$(EXE): $(OBJ)
    $(CC) $(LDFLAGS) $^ $(LDLIBS) -o $@

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

clean:
    $(RM) $(OBJ)
```