How-To: Using the SoftwAre Fault Emulator tool

Critiware s.r.l.

1 Introduction

The SAFE fault injection tool has been developed for automating the injection of **realistic software faults** in C and C++ programs. In order to emulate software faults in fault injection experiments, several studies analyzed *field data* (i.e., data collected from deployed software) about failures and faults that caused them, that were adopted to characterize faults that can realistically occur in complex software.

The fault model adopted in SAFE is based on the *Orthogonal Defect Classification* (ODC). ODC adopts the notion of *defect type*, which reflects the *code fix* for correcting a fault. A fault can belong to exactly one ODC defect type (i.e., types are orthogonal) among Function, Checking, Assignment, Algorithm and Interface. ODC was aimed at providing feedback during development, based on the distribution of types across development phases. In recent studies, ODC was extended using a classification scheme that is detailed enough for fault injection purposes, by specifying the kind of programming construct involved in each type of fault (e.g., which kind of assignment should be targeted for emulating Assignment faults, such as initializations, assignments with constants or with expressions), and other rules to be followed for realistically emulate faults found in the field (e.g., the target construct must not be the only statement in its block, the assignment must not be part of a FOR construct, etc.). According to field failure data, the majority of software faults belong to the relatively small set of fault types (Table 1), and the distribution of faults tend to be generic across systems.

The high-level architecture of the tool is depicted in Figure 1. It consists of:

Table 1: Most frequent fault types occurring in deployed software.

Fault Type	*Description*
MFC	Missing function call
MVIV	Missing variable initialization using a value
MVAV	Missing variable assignment using a value
MVAE	Missing variable assignment with an expression
MIA	Missing IF construct around statements
MIFS	Missing IF construct + statements
MIEB	Missing IF construct + statements + ELSE construct
MLAC	Missing AND in expression used as branch condition
MLOC	Missing OR in expression used as branch condition
MLPA	Missing small and localized part of the algorithm
WVAV	Wrong value assigned to variable
WPFV	Wrong variable used in parameter of function call
WAEP	Wrong arithmetic expression in function call parameter

- A C macro pre-processor, that translates all macros (e.g., #define and #include directives) in a source code file in plain C language, in order to produce a self-contained compilation unit, that will be further processed by the tool.
- A C/C++ front-end, that analyzes the file and builds an Abstract Syntax Tree representation of the code. This representation guides the identification of locations where a fault type can be introduced in a syntactically correct manner, and that comply to fault types in Table 1.
- A Fault Injector that explores the Abstract Syntax Tree, and produces a set of faulty "patch" files, each containing a different software fault.

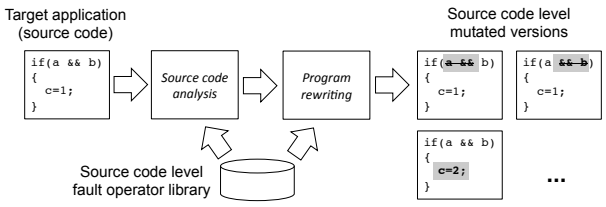

Figure 1: Overview of the SAFE tool

Actually, the tool consists of an executable, namely "safe", that embeds both the C/C++ front-end and the Fault Injector. The pre-processing of

C macros has to be performed using an external C pre-processor, namely MCPP. MCPP is an open-source project (BSD-licensed) that aims at providing a high-quality pre-processor to be used with existing compiler suites such as GCC, by replacing the standard pre-processor included with them. MCPP it is freely available at `http://mcpp.sourceforge.net/`.

The SAFE tool currently supports the Fedora Linux and Microsoft Windows operating systems, and it is intended to be used along with MCPP, and with the "patch" UNIX utility.

2 Set-up and usage of SAFE

We describe the use of the SAFE tool in a complex real-world system, the Apache HTTPD web server (http://httpd.apache.org). It is a widely-adopted web server written in the C language. Version 2.2.11 will be considered, which consists of approximately 242 thousands of lines of C code.

We first need to install the MCPP pre-processor. In a Linux environment, the following commands can be used to install MCPP:

```
[~] $ tar zxf <path_to_archives>/mcpp-2.7.2.tar.gz
[~] $ cd mcpp-2.7.2
[~/mcpp-2.7.2] $ ./configure --enable-replace-cpp
[~/mcpp-2.7.2] $ make
[~/mcpp-2.7.2] $ sudo make install
```

These commands replace the standard pre-processor of GCC with MCPP. Please note that this guide assumes that the user can issue privileged commands (e.g., installing new programs in directories only accessible by the administrator) using the "sudo" command. The "sudo" command is provided and already configured in most Linux distributions. To restore the standard GCC pre-processor, you can execute the following command in the directory in which MCPP was built:

```
[~/mcpp-2.7.2] $ sudo make uninstall
```

To use the tool on Apache HTTPD, we copy the source code of the web server and the fault injection tool in the same directory (e.g., "~/Desktop/demo"):

```
[~] $ mkdir ~/Desktop/demo
[~] $ cp <path>/safe ~/Desktop/demo
[~] $ cd ~/Desktop/demo
[~/Desktop/demo] $ wget http://archive.apache.org/dist/httpd/httpd-2.2.11.tar.bz2
[~/Desktop/demo] $ tar jxf httpd-2.2.11.tar.bz2
[~/Desktop/demo] $ cd httpd-2.2.11
```

Before injecting software faults in the target program, it is necessary to preprocess source files in order to translate all preprocessor macros in these files to plain C language. This can be achieved using MCPP and GCC, by compiling the target software by passing the flags "-Wp,-K,-W0 -save-temps" to GCC (or to G++ in the case of C++ software). This operation will generate a set of ".i" (or ".ii" for C++) files, each corresponding to a ".c" (or ".cpp", respectively) file of the target program. They are temporary files created by GCC after preprocessing of source files. If the target program is based on the "make" build system, then it is typically possible to pass these flags using the CFLAGS and CXXFLAGS environment variables (for gcc and g++, respectively). The web server will be installed in the "∼/Desktop/demo/installdir" directory:

```
[~/Desktop/demo/httpd-2.2.11] $ ./configure --prefix=/home/user/Desktop/demo/installdir/
...
[~/Desktop/demo/httpd-2.2.11] $ make CFLAGS="-Wp,-K,-W0 -save-temps"
...
[~/Desktop/demo/httpd-2.2.11] $ mkdir /home/user/Desktop/demo/installdir/
[~/Desktop/demo/httpd-2.2.11] $ make install
...
```

Alternatively, MCPP pre-processing can be performed without involving GCC, by invoking the "mcpp" command on a source code file, using the following syntax:

```
[~] mcpp -K -W0 source.c source.i
```

In this howto, we focus on the "server" sub-directory of HTTPD, which includes the core components of the web server. The previous commands generate several ".i" in that directory. The following command applies the "safe" tool to each ".i" file, and generates a set of "patch" files (in total, 5027 patches), each representing a fault to be injected in the web server code.

```
[~/Desktop/demo/httpd-2.2.11] $ ~/Desktop/demo/safe server/*.i

Target file: server/config.i
Fault injection: OMFC
Fault injection: OMVIV
Fault injection: OMVAV
...
Target file: server/connection.i
Fault injection: OMFC
Fault injection: OMVIV
Fault injection: OMVAV
...
```

In the example that follows, we inject a WPFV fault (wrong variable used in parameter of function call) in the "request.c" source file. In particular, we consider the fault contained in the "request.i_OWPFV_17.patch" file, which injects a WPFV fault at line 708: the first parameter of the "memcpy()" function call (the pointer "buf") is replaced with another variable ("save_path_info", previously declared in the function) of the same type (a character array "char *"). The tool assures that the type of the variables is the same, and that the variable has the same scope of the replaced variable. Therefore, the replacement generates syntactically correct code. The patch can be injected using the "patch" command. The faulty web server is then compiled again and installed.

```
[~/Desktop/demo/httpd-2.2.11] $ cat server/request.i_OWPFV_17.patch

--- /home/user/Desktop/demo/httpd-2.2.11/server/request.c
+++ /home/user/Desktop/demo/httpd-2.2.11/server/request.c
@@ -708,1 +708,1 @@
-        memcpy(buf, r->filename, filename_len + 1);
+        memcpy(save_path_info, r->filename, filename_len + 1);

[~/Desktop/demo/httpd-2.2.11] $ patch -p0 < server/request.i_OWPFV_17.patch

patching file /home/user/Desktop/demo/httpd-2.2.11/server/request.c

[~/Desktop/demo/httpd-2.2.11] $ make
[~/Desktop/demo/httpd-2.2.11] $ make install
```

To perform a fault injection experiment, we run the web server and generate three HTTP requests to the web server using the "wget" tool, which is a command-line HTTP client. The wget tool reports that the faulty web server is unable to reply to the requests. By inspecting the log file generated by the web server ("installdir/logs/error_log"), we notice that each HTTP request causes a crash of a server process (denoted with the "Segmentation fault" message).

```
[~/Desktop/demo/httpd-2.2.11] $ sudo ~/Desktop/demo/installdir/bin/apachectl start
[~/Desktop/demo/httpd-2.2.11] $ wget --tries=3 localhost

--2012-01-16 15:18:00--  http://localhost/
Resolving localhost... 127.0.0.1
Connecting to localhost|127.0.0.1|:80... connected.
HTTP request sent, awaiting response... No data received.
Retrying.
...

[~/Desktop/demo/httpd-2.2.11] $ cat ~/Desktop/demo/installdir/logs/error_log

[Mon Jan 16 15:17:55 2012] [notice] Apache/2.2.11 (Unix) configured -- resuming normal operations
```

```
[Mon Jan 16 15:18:00 2012] [notice] child pid 5036 exit signal Segmentation fault (11)
[Mon Jan 16 15:18:01 2012] [notice] child pid 5040 exit signal Segmentation fault (11)
[Mon Jan 16 15:18:03 2012] [notice] child pid 5039 exit signal Segmentation fault (11)
```

To terminate the experiment and restore the original code:

```
[~/Desktop/demo/httpd-2.2.11] $ sudo ~/Desktop/demo/installdir/bin/apachectl stop
[~/Desktop/demo/httpd-2.2.11] $ patch -R -p0 < server/request.i_OWPFV_17.patch

patching file /home/user/Desktop/demo/httpd-2.2.11/server/request.c
```

The steps that have been described above can be repeated several times, each time injecting a different fault (i.e., using a different patch file), to perform a full fault injection campaign. At each iteration, log files (e.g., "error_log", wget messages) and other relevant data (e.g., a dump of the process memory, which is generated by the OS in the case of a crash) can be collected, in order to analyze the effects of faults of the system (e.g., which kind of failure was caused by the fault, or whether the fault has been perceived by the user).