1 Fuzzing

Key definitions

Poet The writing and structure of the fuzz testing

Courier The method used to deliver the test inputs

Oracle The all-knowing system of what good data looks like

Using Microsoft Visual Studio as an IDE (**I**ntegrated **D**evelopment **E**nvironment) it is simple to just use the arguments `/fsanitize=fuzzer /fsanitize=address`. This utilises `LibFuzzer` (`https://llvm.org/docs/LibFuzzer.html`), which can be used with Clang. For `gcc` the best option is the `AFL++` plugin which can be found at `https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.gcc_plugin.md`.

In this example a simple test case has been created for the script `fp.c` using `gdb` to better understand the mechanisms and reasoning behind fuzzing. This program doesn't do much except to demonstrate the imprecision in how decimal fractions are stored in binary, and crash in a clear, obvious way.

First, compiling the program with an exhaustive set of arguments means we catch many coding mistakes. This does not test with data though, and there are a huge variety of possible inputs that can be sent to the program.

Figure 1:

The `gcc` flags used are:

```
-Wall -pedantic -pedantic-errors -Wextra -Werror -Wcast-align -Wcast-qual
-Wchar-subscripts -Wcomment -Wconversion -Wdisabled-optimization -Wfloat-equal
-Wformat -Wformat=2 -Wformat-nonliteral -Wformat-security -Wformat-y2k
```

```
-Wimport -Winit-self -Winline -Winvalid-pch -Wlong-long -Wmissing-braces
-Wmissing-field-initializers -Wmissing-format-attribute -Wmissing-include-dirs
-Wmissing-noreturn -Wpacked -Wparentheses -Wpointer-arith -Wredundant-decls
-Wreturn-type -Wsequence-point -Wshadow -Wsign-compare -Wstack-protector
-Wstrict-aliasing -Wstrict-aliasing=2 -Wswitch -Wswitch-default -Wswitch-enum
-Wtrigraphs -Wuninitialized -Wunknown-pragmas -Wunreachable-code -Wunused
-Wunused-function -Wunused-label -Wunused-parameter -Wunused-value
-Wunused-variable -Wvariadic-macros -Wvolatile-register-var -Wwrite-strings
-Wno-long-long
```

Running the same in `gdb` has the same program output.

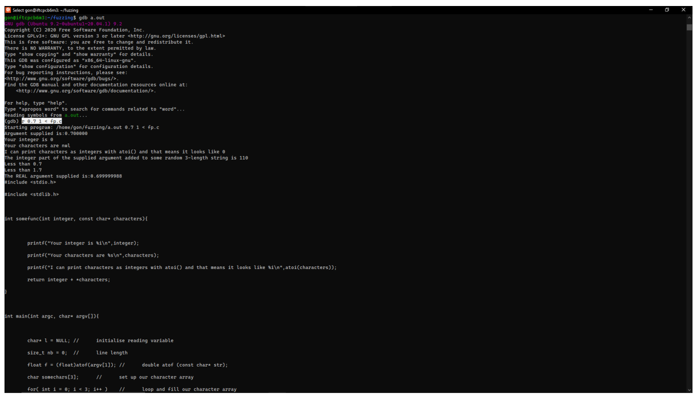

Figure 2:

What is interesting is the additional information that can be gleaned from this debug program. `info functions` shows all the functions in the program. Setting a breakpoint at these functions, using for example `b main`, then using the command `info locals` shows all the variables initialised in the main function during the running of the program.

This is then a very good start point for fuzzing as we have a list of variables in the program! How to proceed next will vary based on the complexity of the program, failure points, and sub-routines. Typically only new code is of interest and so the logic presented here can be applied to specific functions rather than the whole codebase.

In this example a simple way to overwrite variables is after assignment, meaning three breakpoints can be set. Ultimately this should be scripted, so variables should be assigned here too. A skeleton set of commands is then:

```
b fp.c:5
commands 1
set variable integer="STARTING"
```

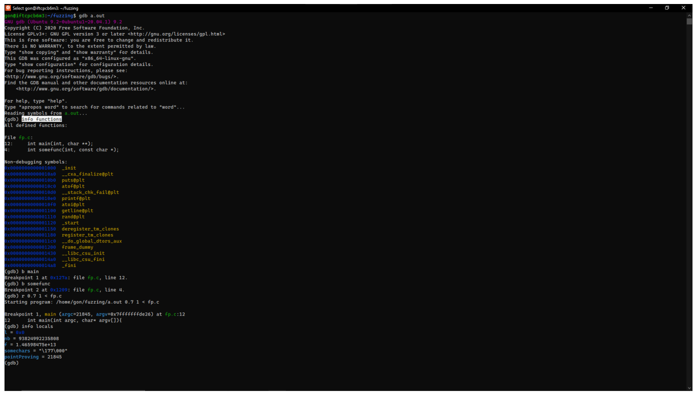

Figure 3:

```
set variable characters="STARTING"
continue
end
b fp.c:20
commands 2
set variable l="STARTING"
set variable nb="STARTING"
set variable f="STARTING"
set variable somechars="STARTING"
continue
end
b fp.c:43
commands 3
set variable pointProving="STARTING"
continue
end
r 0.7 1 < fp.c
if !$_isvoid($_exitsignal)
quit
else
shell echo ERRORERRORERROR
bt
end
```

There are various combinations in the program variables here. Each variable could be altered individually or in combination thus in one of two binary states; ‘good’ or ‘bad’. In this example there are seven variables and so 128 (2^7) combinations. Each variable can also have several different inputs, with 201 shown in the final section of this document.

To reduce the space complexity from 10^{16} this example takes each of the 201 inputs and applies them to all variables at once via a `bash` script. `gdb` allows for scripting with Python too which would result in a much cleaner solution! The below command allows a single line to run a series of tests:

```
theoldline="STARTING";while IFS= read -r theline ; do sed -i "s/=$theoldline/=$theline/g"
gdbscript.gdb; echo $theline replaced $theoldline; gdb -silent -return-child-result
a.out < gdbscript.gdb; theoldline=$theline; done < inputs.list
```

The halting behaviour of the program can be altered on a user basis. In the current setup shown here, it should be clear from the fuzzing that there is a bug on line 21 causing a cast to fail, and so the function should be altered to check the type of `f` first before exiting cleanly.

This example is 'dumb' fuzzing; a brute force technique that is trying all possible inputs in all possible places. The `somechars` array for example will always only be letters, and is passed directly to `somefunc` so in fact only altering 'somechars' should be enough. It is left as an exercise to the reader to modify the above to try different files. Some suggested paths are:

```
/dev/random (this should be infinite garbage)
/etc/passwd (this should be protected)
$(find ~ -printf '%s\t%p\n'| sort -nr | head -1 | awk -F '\t' '{print $2}')
(this should try the biggest file you have under your home directory)
/dev/null
```

2 Inputs

Strings should be encapsulated as per your scripting language; it is important to check the courier here is sending exactly what you want!

```
-1
0
0.0
1
10
-10
3.14159265358979323846264338327950288419 71
2147483647
2147483648
4294967295
4294967296
-2147483647
-2147483648
```

-4294967295
-4294967296
1e2
1.0e2
13e3
1.3e4
13.4e3
2.1e103̂08
9.5e103̂08
39.5e103̂08
-1.9e103̂08
-19.2e103̂08
0.00000000000001
-0.00000000000001
0.0000g000000001
-0.0000g000000001
g-1
g0
g0.0
g1
g10
g-10
g3.1415926535897932384626433832795028841971
g2147483647
g2147483648
g4294967295
g4294967296
g-2147483647
g-2147483648
g-4294967295
g-4294967296
-g2147483647
-g2147483648
-g4294967295
-g4294967296
g-1g
g0g
g0.0g
g1g
g10g
g-10g
g3.1415926535897932384626433832795028841971g
g2147483647g
g2147483648g
g4294967295g
g4294967296g

```
g-2147483647g
g-2147483648g
g-4294967295g
g-4294967296g
-g2147483647g
-g2147483648g
-g4294967295g
-g4294967296g
-1g
0g
0.0g
1g
10g
-10g
3.14159265358979323846264338327950288419 71g
2147483647g
2147483648g
4294967295g
4294967296g
-2147483647g
-2147483648g
-4294967295g
-4294967296g
-2147483647g
-2147483648g
-4294967295g
-4294967296g
0ghejkl
0.0ghejkl
1ghejkl
10ghejkl
-10ghejkl
3.14159265358979323846264338327950288419 71ghejkl
2147483647ghejkl
2147483648ghejkl
4294967295ghejkl
4294967296ghejkl
-2147483647ghejkl
-2147483648ghejkl
-4294967295ghejkl
-4294967296ghejkl
-2147483647ghejkl
-2147483648ghejkl
-4294967295ghejkl
-4294967296ghejkl
ghejkl-1
```

ghejkl0
ghejkl0.0
ghejkl1
ghejkl10
ghejkl-10
ghejkl3.1415926535897932384626433832795028841971
ghejkl2147483647
ghejkl2147483648
ghejkl4294967295
ghejkl4294967296
ghejkl-2147483647
ghejkl-2147483648
ghejkl-4294967295
ghejkl-4294967296
-ghejkl2147483647
-ghejkl2147483648
-ghejkl4294967295
-ghejkl4294967296
ghejkl-1ghejkl
ghejkl0ghejkl
ghejkl0.0ghejkl
ghejkl1ghejkl
ghejkl10ghejkl
ghejkl-10ghejkl
ghejkl3.1415926535897932384626433832795028841971ghejkl
ghejkl2147483647ghejkl
ghejkl2147483648ghejkl
ghejkl4294967295ghejkl
ghejkl4294967296ghejkl
ghejkl-2147483647ghejkl
ghejkl-2147483648ghejkl
ghejkl-4294967295ghejkl
ghejkl-4294967296ghejkl
-ghejkl2147483647ghejkl
-ghejkl2147483648ghejkl
-ghejkl4294967295ghejkl
-ghejkl4294967296ghejkl
1g2
1.0g2
13g3
1.3g4
13.4g3
2.1g103̂08
9.5g103̂08
39.5g103̂08
-1.9g103̂08

```
-19.2g103̂08
1ghejkl2
1.0ghejkl2
13ghejkl3
1.3ghejkl4
13.4ghejkl3
2.1ghejkl103̂08
9.5ghejkl103̂08
39.5ghejkl103̂08
-1.9ghejkl103̂08
-19.2ghejkl103̂08
-19.21̂02̂
13.11̂02̂
\0
\129
aslkdjasdoiuqw
a
\n
\r
\f
\r\f
@
!
$
(
'
"
)
=
+
/

\
_£
{
}
@!$'")=+/\\\_}
@! '")=+/\\\_}
@!$'"(=+/\\\_}
@! '"(=+/\\\_}
!$'")=+/\\\_}
! '")=+/\\\_}
!$'"(=+/\\\_}
! '"(=+/\\\_}
$'")=+/\\\_}
'")=+/\\\_}
```

$'"(=+/_}
'"(=+/_}
asdna@!$'")=+/\\\_}sdhasoidhqwdaskldjqwoidjqoiwjdalksdjiudghoidjsaoidjsoidj@!$'")=+/_}laks
asdkjwdjasdkjaslkdjasdasdkjwdjasdkjaslkdjasdasdkjwdjasdkjaslkdjasdasdkjwdjasdkjaslkdjasdasdk