ファジング実践資料（AFL 編）

「ファジング活用の手引き」別冊
ファジングツール「American Fuzzy Lop」の使い方

IPA 独立行政法人情報処理推進機構
技術本部 セキュリティセンター

2020 年 3 月

本書は、以下の URL からダウンロードできます。

「ファジング活用の手引き」別冊

https://www.ipa.go.jp/security/vuln/fuzzing.html

目次

はじめに

本書は、IPA の「ファジング活用の手引き」の別冊資料です。本書には、ファジングを実践するための「ファジングツール『AFL』の使い方」を収録しています。

「ファジングツール『AFL』の使い方」では、広く利用されているファジングツール「American Fuzzy Lop」（以下、AFL）の基本的な使い方を紹介します。

本書の使い方

本書は次の使い方を想定しています。

(1).ファジングツール「AFL」を試してみたい。

本書で紹介する手順にしたがって、基本的なファジングを実施することができます。より詳細な使用方法は各ファジングツールの開発元ウェブサイトが提供しているソフトウェアのドキュメント等を参照ください。

(2).ファジングで使われるテストパターンがどんなものか知りたい。

実際にファジングツールで使われているデータを確認できます。

本書の動作環境

本書で収録している内容は、Windows10 Pro 上で動作する VirtualBox で構築された Ubuntu18.04 および Ubuntu16.04 の仮想環境上で確認しています。

本書を読む上での注意事項

- 本書で収録している URL および手順は、2020 年 2 月末日時点で確認しています。
  読者の動作環境などによっては本書の手順では正しく動作しない可能性があります。

1. ファジングツール「AFL」の使い方

本章ではファジングツール「AFL」の概要と、「AFL」によるファジング手順を説明します。

1.1. 概要

AFL は、Google 社のセキュリティエンジニア（開発当時）である Michał Zalewski 氏を中心に開発されたオープンソースのファジングツールであり、主に Linux の実行可能ファイルのバグや脆弱性といった問題を検出できます。

AFL は現在最も利用されているファジングツールのひとつで、数年にわたって数百のアプリケーションで問題を検出した実績があります。代表的なものとしては、ウェブブラウザ（Safari、Firefox、Internet Explorer）、ウェブサーバ構築ソフトウェア（Apache や Nginx）、データベース構築ソフトウェア（SQLite、MySQL）などの様々なソフトウェアにおける実績があげられます。

AFL では、テスト対象のプログラムにおいて、テストデータがどの実行経路を通るかを計測し、実行経路毎に適したテストデータを生成することで、より効率の良いテストを実施できます。このようなファジング手法はグレーボックスファジング（Coverage-based Fuzzing や Coverage Guided Fuzzing などとも）と呼ばれており、AFL はその手法を用いる代表的なファジングツールのひとつです。

AFL は基本的にプログラミング言語 C/C++ または Objective-C で書かれたプログラムをサポートとしていますが、AFL をベースとして派生したファジングツールや拡張機能が多く開発されており、サポート対象の拡大や機能の効率化、高速化等が行われています。

表 1 AFL の概要

項目	内容
IPA 使用バージョン	2.52b
ライセンス	Apache License 2.0
動作対象 OS	32bit および 64bit の Linux、OpenBSD、FreeBSD、および NetBSD 一部機能に制限有り：MacOS X, Solaris
サポートするテスト対象	C/C++または Objective-C で書かれたプログラム
発見可能なバグや脆弱性	メモリエラーの問題、不正な例外処理、無限ループ　等
URL	開発者のウェブサイト https://lcamtuf.coredump.cx/afl/ Google 社の github リポジトリ https://github.com/google/AFL
ダウンロード先	https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

1.2. AFL のファジングの仕組み

AFL では通常、ファジングを開始する前にテスト対象のプログラムのソースコードを専用のコンパイラを使ってコンパイルします。この際に、テスト対象の処理を追跡するための計測用コード（instrumentation）を挿入します。厳密には、アセンブリレベルでのジャンプ命令の前後に計測用コードを挿入します。

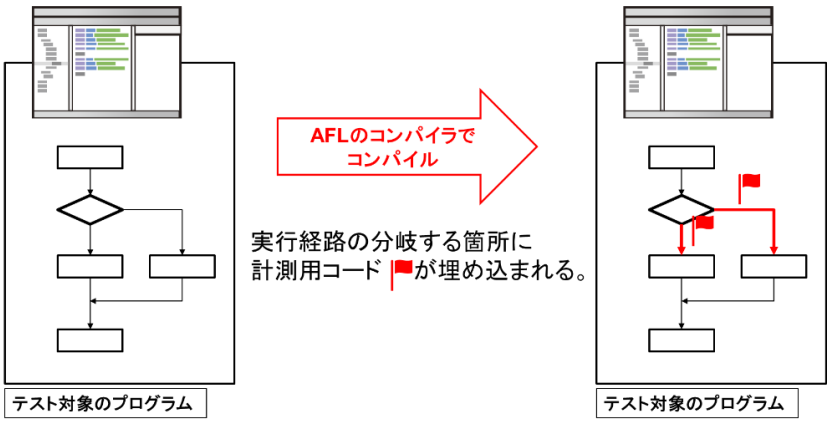

図 1 AFL による計測用コードの挿入

この計測用コードを使ってテスト対象の実行経路の分岐を観測し、新たな分岐を検知した際は、その分岐を生じさせたテストデータを保持し、別のテストデータ生成時の初期値として再利用します。分岐を生じさせたテストデータを初期値として生成されたテストデータは、同じ実行経路で分岐をする可能性が高いため、分岐先のコードを効率的にファジングすることができます。この仕組みが、実行経路毎に適したテストデータを生成することを実現しており、計測用コードを用いない単純なファジングよりもコードカバレッジ（コード網羅率）の高いテストを可能としています。

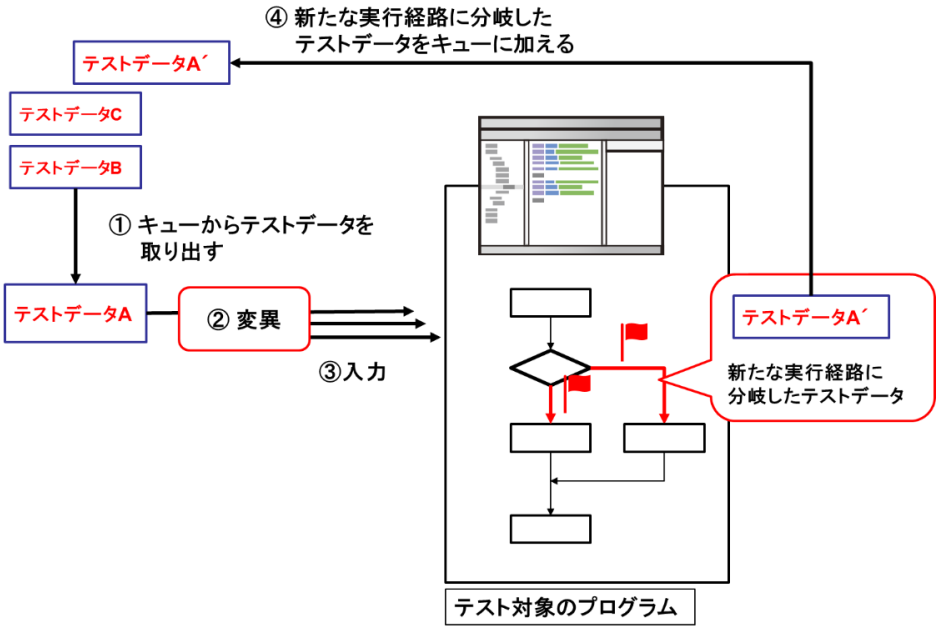

図 2　AFL がコードカバレッジの高いテストを実行する仕組み

1.3. AFL のセットアップ

AFL を使用するためのインストール手順と、オプションで実施する各種セットアップ手順を説明します。

（1） インストール

AFL の開発者のウェブサイト上（https://lcamtuf.coredump.cx/afl/）で、AFL のソースコードが配布されており、解凍後に make コマンドを使ってコンパイルすることですぐに使用できます。

下記の手順でダウンロードおよびインストールができます。本書では紹介を省きますが、apt 等のパッケージ管理システムでもインストールが可能です。

```
$ wget https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz
$ tar zxvf afl-latest.tgz
$ cd afl-2.52b/
$ make && sudo make install
```

（2） ファジング用 RAM 領域の作成

ファジングでは、実行時にデータの読み書きが大量に発生するため、SSD や HDD といった記憶装置に対する負荷が大きく、記憶装置の寿命を縮めてしまう恐れがあります。

AFL の開発者によると、仮想環境内で RAM ディスクを作成し、そのディスク内で AFL を実行することで、この問題に対処できるとのことです。

```
$ mkdir /tmp/afl-ramdisk && chmod 777 /tmp/afl-ramdisk
$ mount -t tmpfs -o size=512M tmpfs /tmp/afl-ramdisk
$ cd /tmp/afl-ramdisk/
```

1.4. AFL の主な機能

本節では AFL の主な機能およびディレクトリの紹介を行います。より詳細な内容については AFL フォルダ内の「docs/」配下に配置された各種ファイルを参照ください。

（1） afl-gcc

afl-gcc は、AFL が用いるコンパイラであり、対象プログラムをコンパイルする際に処理をトレースするための計測用コードを埋め込みます。gcc のラッパーであるため、基本的な機能は gcc と同様に使用することができます。

（2） afl-fuzz

ファジングの処理を実施する機能です。オプションの一覧は下記の通りです。

```
afl-fuzz 2.52b by <lcamtuf@google.com>

afl-fuzz [ options ] -- /path/to/fuzzed_app [ ... ]

Required parameters:

  -i dir        - input directory with test cases
  -o dir        - output directory for fuzzer findings

Execution control settings:

  -f file       - location read by the fuzzed program (stdin)
  -t msec       - timeout for each run (auto-scaled, 50-1000 ms)
  -m megs       - memory limit for child process (25 MB)
  -Q            - use binary-only instrumentation (QEMU mode)

Fuzzing behavior settings:

  -d            - quick & dirty mode (skips deterministic steps)
  -n            - fuzz without instrumentation (dumb mode)
  -x dir        - optional fuzzer dictionary (see README)

Other stuff:

  -T text       - text banner to show on the screen
  -M / -S id    - distributed mode (see parallel_fuzzing.txt)
  -C            - crash exploration mode (the peruvian rabbit thing)

For additional tips, please consult /usr/local/share/doc/afl/README.
```

図 3 afl-fuzz の実行オプション

-i オプション と -o オプションが必須であり、-i オプションに入力ファイルが格納されたディレクトリを、-o オプションにファジングの結果を出力するディレクトリを指定します。

実行形式としては、以下のようになります。橙色の箇所は個別の実行対象に合わせて変更が必要となります。実践的な内容は 1.5 章にて後述します。

```
$ ./afl-fuzz -i input_directory -o output_directory ./target
```

-i オプションによる入力ディレクトリの指定

ディレクトリを指定すれば、そのディレクトリ内のファイルを読み込んでテストデータを生成します。ディレクトリ内に格納するファイルは複数でも問題ありません。誤ってディレクトリではなくファイルを直接指定するとエラーとなります。

afl-fuzz は、-i オプションで指定されたディレクトリ内に格納されたファイルを読み込み、整形したものをテストデータの生成時に用いる初期値とします。

もしファイルを自前で用意することが難しい場合は、afl-2.52b/testcases/ ディレクトリ内に用意されているサンプルファイルも利用できます。サンプルファイルは様々なファイルフォーマットが用意されており、archives/, images/, multimedia/, others/ の 4 つのサブディレクトリに分類されて格納されています。

-o オプションによる出力ディレクトリの指定

ファジングの実行結果は -o オプションで指定したディレクトリにリアルタイムで出力されます。指定したディレクトリが存在しない場合は、ファジング実行時に自動で生成されるため、あらかじめディレクトリを作成する必要はありません。

ファジング結果の出力としては、ファジングの実行ステータスやログなどを記録したテキストファイルの他に、以下のサブディレクトリが生成されます。

- **crashes/ ディレクトリ**

  AFL が生成したテストデータにより、テスト対象がクラッシュ（メモリエラー等による異常終了）した際に、そのテストデータが記録されます。

- **hangs/ ディレクトリ**

  AFL が生成したテストデータにより、テスト対象でハング（処理の無限ループ等に起因するタイムアウト）が発生した際に、そのテストデータが記録されます。

- **queue/ ディレクトリ**

  AFL が生成したテストデータにより、テスト対象の処理が新たな実行経路に分岐した場合、そのテストデータが queue ディレクトリ内に保存されます。queue ディレクトリ内のテストデータは、ファジングの実行中に afl-fuzz によって再び読み込まれ、更なるテストデータを生成するための初期値として使用されます。

1.5. AFL によるファジングの流れ

AFL を実際に動作させつつ、詳細な使い方を説明していきます。

1.5.1. テスト対象のソースコードをコンパイル

AFL で対象プログラムをファジングする場合、基本的には対象プログラムを afl-gcc を用いてコンパイルします。ここでは、説明のための例として下記の小さなサンプルコードを「example.c」として、今回のファジング対象とします。

このサンプルコードには書式文字列攻撃（Format string attack）が可能となる脆弱性が含まれています。この脆弱性を悪用することで、メモリの内容を抜き出したり、任意のデータを書き込んだりすることができます。結果として、当該プログラムをクラッシュさせたり、不正なコードを実行させたりできる可能性があります。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char** argv)
{
   char buf[8];
   if (read(0, buf, 8) < 1)
   {
      exit(1);
   }
   printf(buf);
   exit(0);
}
```

afl-gcc で対象プログラムをコンパイルするは、afl-gcc の引数に対象プログラムのソースコードを指定して実行します。橙色の箇所は個別の実行対象に合わせて変更が必要となります。

```
$ afl-gcc -o example example.c
```

上記の通りコンパイルを実行すると、カレントディレクトリ内に「example」という実行可能ファイルが生成されます。この実行可能ファイルには afl-gcc により、計測用のコードが埋め込まれている状態です。

1.5.2. ファジングの実行

afl-gcc によってコンパイルされた実行可能ファイル（ここでは example）を afl-fuzz の引数に指定して実行します。ここでは、-i オプションを用いて、サンプルとして用意されているファイルが格納されているディレクトリ（testcases/others/text/）を指定しています。

```
$ ./afl-fuzz -i testcases/others/text/ -o out/ ./example
```

afl-fuzz がエラーを起こし、うまく実行できない場合、いくつかの原因が考えられます。

環境によってはコアダンプの出力先を設定する必要がある旨のエラー（下記の図を参照）が出力されることがあります。

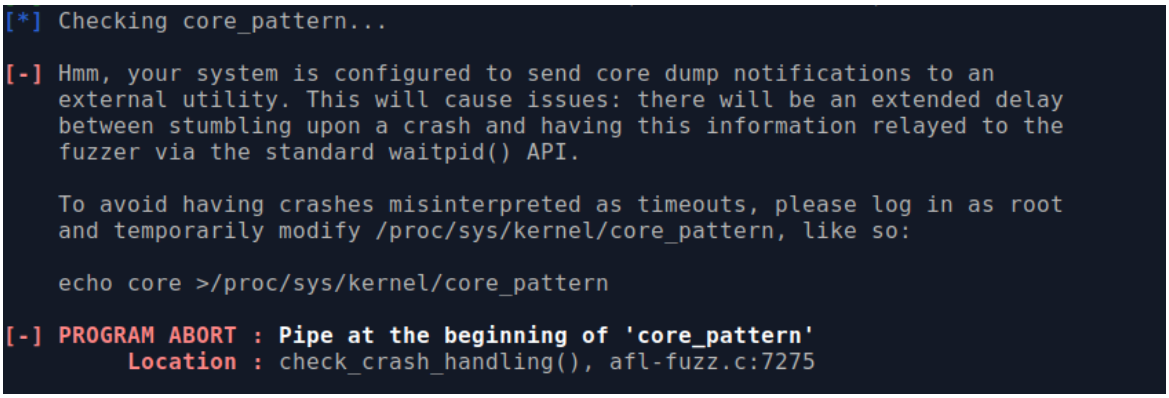

図 4 afl-fuzz の実行エラーの一例 1

その場合は、エラーの内容に従って core ファイルの保存先を指定します。

```
$ echo core >/proc/sys/kernel/core_pattern
```

また、afl-fuzz を実行する際、コマンドラインウィンドウのサイズが小さすぎる場合は実行画面を表示できない旨の警告（下記の図を参照）がでます。コマンドラインウィンドウを広げるかフォントサイズを小さくする等の調整をしてください。

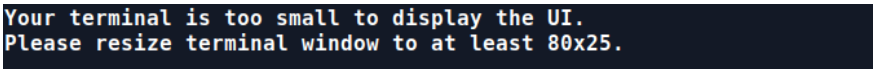

図 5 afl-fuzz の実行エラーの一例 2

ファジングが開始されると、下記のような画面が表示されます。AFL によるファジングは永続的に行われるため、適当なところで Ctrl+C キーを押してファジングを強制終了してください。

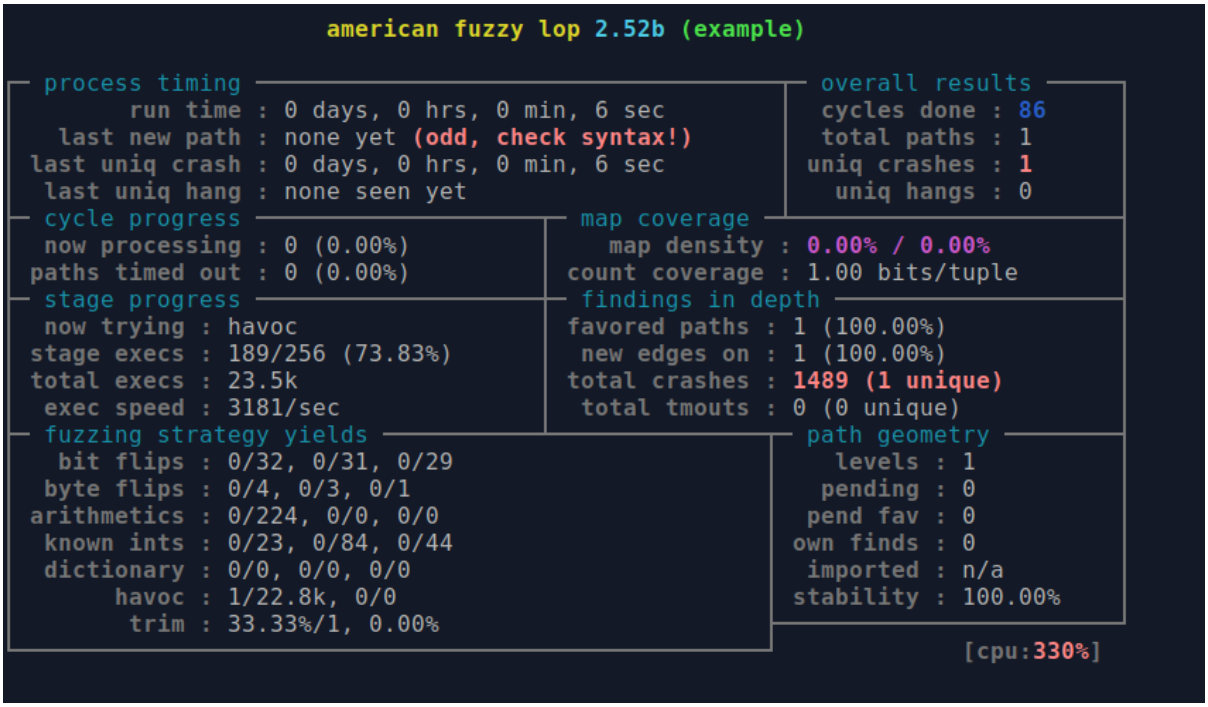

図 6 AFL によるファジングの実行画面

ファジング実行画面の表示項目のうち主要なものを概説します。より詳細な内容については、AFL のフォルダ内の「afl-2.52b/docs/status_screen.txt」を参照ください。

表 2 各表示項目の概説

process timing	
run time	ファジングを開始してから経過した時間を表示します。
last new path	新しい処理の分岐を最後に検出してから経過した時間を表示します。
last uniq crash	テスト対象で直近のクラッシュを検出後、経過した時間を表示します。
last uniq hang	テスト対象で直近のハングを検出してから経過した時間を表示します。
overall results	
cycles done	AFL が実行したサイクル（AFL がファジングを行う際の一連の処理群）の数を表示します。AFL が効果的なファジングを実施するためには、最低限 1 サイクル以上は継続して AFL を実行し続けることが推奨されています。
total paths	検出した実行経路の総数を表示します。
uniq crashes	検出したクラッシュの総数を表示します。
uniq hangs	検出したハングの総数を表示します。
stage progress	
now trying	AFL が現在実行している処理内容を表示します。各処理の詳しい説明は、後述の「表 3 Stage progress の now trying の概説」にて解説します。
stage execs	現在実行しているステージの進捗状況を表示します。
total execs	ファジングを開始してからテストケースの実行回数を表示します。
exec speed	1 テストケースあたりの実行速度を表示します。
cpu	AFL の大まかな CPU 使用率を表示します。

Stage progress の now trying で表示されるファジングステータスのうち、主だったものを説明します。こちらについても、より詳細な内容については、AFL のフォルダ内の「afl-2.52b/docs/status_screen.txt」を参照ください。

表 3 Stage progress の now trying の概説

calibration	ファジングを開始する前の事前調査を行います。具体的には、デフォルトのテストデータを使って、実行経路や実行速度を計測したり、異常が発生しないことを確認したりします。 この処理は、新たな実行経路が発見されるたびに実行されます。
trim	ファジングを開始する前のデータ整形を行います。
bitflip	排他的論理和を用いた変換を行います。
arith	数字加算/減算による変換を行います。
interest	テストデータに固定値を挿入する変換を行います。
havoc	ランダムな変換方式を繰り返します。変換方式の例としては、既述した bitflip、特定の整数値の上書き、データの削除、データの複製などがあげられます。

1.5.3. ファジングの結果を確認する

サンプルプログラムに対してファジングを行うと、おそらくすぐに overall results の uniq crashes がカウントされます。これはファジングの実行により、サンプルプログラムのクラッシュが検出されたことを意味します。

ここでクラッシュを引き起こしたテストデータの確認をしてみます。Ctrl+C キーを押下してファジングの実行を停止し、-o オプションで指定した出力先ディレクトリに移動します。出力先ディレクトリには、queue/、crashes/、そして hangs/ ディレクトリ等が生成されているはずです。

```
$ ls
crashes  fuzz_bitmap  fuzzer_stats  hangs  plot_data  queue
```

今回の例で確認されたのはクラッシュであるため、crashes/ ディレクトリを参照すると、クラッシュを引き起こしたテストデータ(id¥:～から始まるファイル名)が保存されているのが確認できます。ファイル名は個別の実行環境や状況によって異なることをご留意ください。

```
$ ls crashes/
id¥:000000¥,sig¥:06¥,src¥:000000¥,op¥:havoc¥,rep¥:4  README.txt
```

ファジングにより生成されたデータは通常のテキストエディタ等ではうまく表示できない文字列である場合が多いため、hexdump コマンドや xxd コマンド等といった、ファイルの内容を 16 進数等で表示するコマンドを使ってファイルを参照してみましょう。

```
$ hexdump -C id¥:000000¥,sig¥:06¥,src¥:000000¥,op¥:havoc¥,rep¥:4
00000000  67 65 25 6e                                       |ge%n|
```

テストデータの内容は、個別の実行環境や状況によって異なることをご留意ください。今回の例では、書式文字列攻撃が可能となる問題が含まれたコードに対してファジングを行ったため、書式指定子（%n 等）を含むテストデータで問題が検出されています。

1.5.4. ファジングの結果を分析する

一般的にファジングツールの仕事は、問題を引き起こすテストデータを検出することだけですが、プログラムの問題個所を修正するためには、その原因や箇所を調査することは必須と言えます。しかし、クラッシュを引き起こしたデータを見ただけでは問題の原因を特定することは困難です。問題の原因を特定するには、デバッグツールを使って、問題を起こしたテストデータがどのように処理され、どこで問題が発生するかを追跡し調査する必要があります。それは煩雑な作業である場合もありますが、AFL で問題を検出したテストデータのファイル名を参照することで問題の原因や箇所をある程度絞り込むことができます。

AFL で問題を検出したテストデータは、「id¥:」から始まる形式のファイル名で保存されます。

まず「id¥:」の後に続く 6 桁の数字はクラッシュを引き起こしたテストデータに順番に割り当てられる番号です。その後の「sig¥:」に続く 2 桁の数字は、クラッシュ時に通知されたシグナル（Unix 系 OS において発生したイベントを通知する機能）の番号を表します。ファジング時によく目にするものとしては、シグナル番号 6 番の SIGABRT（プロセスの中断）やシグナル番号 11 番 SIGSEGV（セグメンテーション違反）があげられます。例えば後者の場合、記憶領域への不正なアクセスによりクラッシュしていることがうかがえるため、バッファオーバーフロー等の問題が疑われます。その後に続いている、「src¥:」の後の 6 桁の数字は、queue/に格納された値の id とリンクしています。つまり、どの実行経路で問題が発生したかを調査するヒントになります。以降に続く「op:」「rep:」は、どのようにテストデータを生成したのかを示しています。テストデータの生成方法についての詳細は「afl-2.52b/docs/status_screen.txt」の「5) Stage progress」を参照ください。

1.6. ケーススタディ

afl-fuzz では、オプションによって様々な実行方式を指定できます。いくつか例を説明します。

1.6.1. 入力ファイルをコマンドライン引数として指定する場合

AFL はデフォルトでは、入力を標準入力で読み込む前提でファジングを実施しますが、入力ファイルをコマンドライン引数として指定して実行するような場合も考えられます。そういった場合、以下のようにコマンドライン引数が指定される箇所に「@@」を指定して afl-fuzz を実行します。

実行形式の例としては以下のようになります。この例では、example が引数でのファイル入力を受け付けていないため問題は検出されないことをご留意ください。

```
$ ./afl-fuzz -i testcases/other/text/ -o out/ ./example @@
```

1.6.2. ソースコードが手元にない実行可能ファイルをファジングする

テスト対象のソースコードが手元にない場合や、何らかの理由でテスト対象のプログラムの再コンパイルが禁止されている場合等は、テスト対象に計測用コードを埋め込むことができません。

そのような場合、AFL の QEMU モードを使えば、テスト対象のプログラムに計測用コードが埋め込まれていない状態でも、AFL によるファジングを実施できます。

QEMU とは、サードパーティのプロセッサエミュレータであり、本来は、あるアーキテクチャ向けに作成された OS やバイナリを別のアーキテクチャで動作させる際などに使用します。QEMU のユーザーモードエミュレーションと呼ばれる機能は、実行対象のバイナリを解析して、分岐命令毎にブロック化し、ブロック単位で命令を中間コードに変換したあと動的コンパイルを行うことで動作させる仕組みです。AFL の QEMU モードではこの仕組みを利用して、これらのブロックの間に計測用コードを挿入することで、テスト対象における実行経路の分岐を計測します。

QEMU モードを使用するには、別途セットアップが必要です。セットアップは afl-2.52b/QEMU_mode ディレクトリに移動して、関連パッケージ群（libtool-bin、automake、libbison-dev、libglib2.0-dev）をインストールした後、build_qemu_support.sh を実行します。実施するコマンドは下記の通りです。QEMU がサポートされていない環境ではインストールが上手くいかないためご注意ください。

```
$ cd afl-2.52b/QEMU_mode/
$ apt install -y libtool-bin automake libbison-dev libglib2.0-dev
$ ./build_qemu_support.sh
```

QEMU モードでは、afl-gcc が用いる計測用コードの代わりに、この QEMU の機能を利用して処理の追跡を行います。ただし、ファジングの実行速度は大幅に低下するため、可能であれば対象プログラムをコンパイルする通常の手順でのファジングを推奨します。

QEMU モードを使用する場合は、-Q オプションを用います。試しに、先ほどのソースコードを afl-gcc ではなく gcc でコンパイルし、QEMU モードでファジングを実行してみます。先ほどの例で使用した実行可能ファイルと区別するため、ここではコンパイル後の実行可能ファイル名を「example_qemu」、出力先ディレクトリ名を「out_qemu」としています。

```
$ gcc -o example_qemu example.c
$ ./afl-fuzz -i testcases/others/text/ -o out_qemu/ -Q ./example_qemu
```

afl-gcc で計測用コードを埋め込んでいないのにもかかわらず、ファジングが実行できるかと思います。このように QEMU モードを用いることで、afl-gcc でコンパイルしていない実行可能ファイルもファジングできます。詳しくは AFL のフォルダ内の「afl-2.52b/qemu_mode/README.qemu」 を参照ください。

1.6.3. ファジングを並列で実行する

-M オプションと-S オプションを用いることで、afl-fuzz のプロセスを複数実行し、並列的にファジングを行うことができます。

並列のファジングを開始するには、まず-M オプションを使って、afl-fuzz のマスタープロセスを開始します。

```
$ ./afl-fuzz -i input_directory -o output_directory -M fuzzer01 ./target
```

さらに-S オプションを使って afl-fuzz を実行することで並列のファジングが開始されます。

```
$ ./afl-fuzz -i input_directory -o output_directory -S fuzzer02 ./target
$ ./afl-fuzz -i input_directory -o output_directory -S fuzzer03 ./target
```

ただし、CPU 利用率に十分な余力がないと並列実行の効果が出ない場合があります。単体のプロセスで afl-fuzz を実行した際に、実行ステータスに表示される CPU 利用率が緑色であれば効果が期待できます。

詳細は AFL のフォルダ内の「afl-2.52b/docs/parallel_fuzzing.txt」を参照ください。

1.6.4. 一旦停止したファジングを再開する

一旦停止したファジングを再開する場合、-i- オプションを用いることで、実質的にファジングを途中から再開することができます。-i- オプションでは、-o オプションで指定したディレクトリは以下の queue/ に保存されている内容もテストデータの生成元として使用します。既述した通り、queue/ には、対象プログラムへのファジングにより、新たな実行経路を発見した際のテストデータが格納されています。つまり、既に対象プログラムの実行経路がある程度探索されている状態からファジングを開始できることを意味します。

```
$ ./afl-fuzz -i- input_directory -o input_directory ./target
```

1.6.5. make ユーティリティを使用する実行可能ファイルに対するファジング

make ユーティリティを用いて、複数のソースコードファイルから生成される実行可能ファイルへのファジングの方法を説明します。

例として、GNU Binutils に含まれている readelf に対するファジングを紹介します。readelf は、引数に指定された実行可能ファイルの詳細情報を表示するコマンドとして使用されます。

まずはファジング対象である binutils のソースコードを用意します。

```
$ wget https://ftp.gnu.org/gnu/binutils/binutils-2.25.tar.gz
$ tar zxvf binutils-2.25.tar.gz
```

binutils のディレクトリに移動し、AFL のコンパイラを用いるようにコンパイラの設定を変更します。その後、make コマンドを使って binutils をビルドすることで、AFL の計測用コードが対象プログラムに挿入されます。

```
$ cd ~/binutils-2.25
$ CC=afl-gcc CXX=afl-g++ ./configure
$ make
```

binutils のディレクトリ内で、入力元ディレクトリと出力先ディレクトリを作成します。readelf の入力は、実行可能ファイルですので、テストデータとして elf ファイル等を用意することが望ましいです。サンプルとして用意されている「testcases/others/elf」を指定しても良いですが、ここでは、入力元ディレクトリには、/bin/ps の内容をコピーして用います。

```
$ mkdir afl_in afl_out
$ cp /bin/ps afl_in/
```

afl-fuzz を実行します。テスト対象の readelf は binutils ディレクトリ内に格納されており、入力はコマンドライン引数から得るため、末尾に「@@」を指定します。

```
$ ./afl-fuzz -i afl_in -o afl_out ./binutils/readelf -a @@
```

上記を実行すれば、binutils に対するファジングが実行されますが、上記はあくまで一般的なツールに対して AFL を実行するケースを例示したものであり、このケースでは問題は発見されません。

1.7. AFL の拡張機能と派生ツール

AFL には多くの派生プロジェクトが存在しており、AFL の機能を拡張・改善したツールの開発が行われています。以下で主だったものをいくつか紹介します。

1.7.1.AFL のユーティリティ

本節では AFL によるファジングを支援するユーティリティについて説明します。

■ afl-plot

afl-plot は、AFL によるファジングの進行状況を可視化するユーティリティです。AFL の結果出力先ディレクトリを指定することで動作します。生成される図は、秒間の実行速度を示す図、テストカバレッジを示す図、発見されたクラッシュとハングの数を示す図の 3 つです。

■ afl-crash-analyzer

ファジングでは、問題を引き起こしたテストデータが記録されていきますが、複数のテストデータが同一の問題を検出している場合があります。このような場合は、トリアージと呼ばれる手法で、重複を排除し、テストデータと問題とが一対一になるように整理します。afl-crash-analyzer は、AFL で検出されたクラッシュを分析し、そのトリアージを支援する Python スクリプトです。

1.7.2.AFL から派生したファジングツール

本節では AFL から派生したファジングツールについて説明します。より詳細な内容については AFL フォルダ内の「afl-2.52b/docs/sister_projects.txt」を参照ください。

■ Python AFL

プログラミング言語 Python で作成されたソフトウェアに対して、ファジングを行うことができる実験的なモジュールです。

利用には Python のバージョン 2.6 以降またはバージョン 3.2 以降、Cython のバージョン 0.19 以上と AFL のインストールが必要です。

Python AFL では、対象となる Python スクリプトに、専用のコードを追記することでファジングを実施することができます。カバレッジの計測は、Python の標準ライブラリである trace モジュールにより実行される仕組みです。Python AFL の詳細は、下記 URL から確認できます。

https://github.com/jwilk/python-afl

■ Go-fuzz

プログラミング言語 Go で作成されたパッケージに対してファジングを行います。Go-fuzz では、Fuzz( )関数というファジング用の関数を作成し、その関数の中でファジング対象の関数を呼び出します。go-fuzz はこの Fuzz( )関数を経由して、ファジング対象にテストデータを入力し、クラッシュ等の問題の発生を監視します。

https://github.com/dvyukov/go-fuzz

■ Win-AFL

AFL によるファジングを Windows の実行可能ファイルに対して実行できるように拡張したものです。WinAFL は、Google のセキュリティアナリストチームである Google Project Zero によって開発され、数多くの Windows 向けアプリケーションの脆弱性を発見した実績があります。WinAFL では、サードパーティの動的プログラム分析ツール（主に DynamoRIO や intelPIN）を併用することで、対象プログラムに対するテストのカバレッジを計測します。

https://github.com/googleprojectzero/winafl

■ 動的シンボリック実行を併用するファジングツール

AFL の分岐探索の手法に、動的シンボリック実行を併用することでコードカバレッジを向上させたファジングツールが複数存在します。動的シンボリック実行とは、プログラムの実行処理において、特定の実行経路とその経路を通る入力値を検出する手法の一つです。AFL による実行経路探索が限界に達した際に、この動的シンボリック実行によって新たな実行経路探索を行い、ファジングを継続することで、よりコードカバレッジの高いファジングを実施できます。この手法を利用するツールとしては、MAYHEM や Driller があげられます。

更新履歴

更新日	更新内容
2020 年 3 月 27 日	第 1 版　発行

著作・制作　独立行政法人情報処理推進機構（IPA）

編集責任　　渡辺 貴仁

執筆者　　小林 桂

協力者　　板橋 博之
田村 智和
熊谷 悠平
堀江 亘
木村 泰介
佐藤 輝夫

※独立行政法人情報処理推進機構の職員については所属組織名を省略しました。

「ファジング活用の手引き」別冊

ファジング実践資料(AFL 編)

― ファジングツール「American Fuzzy Lop」の使い方 ―

[発　　行] ２０２０年　３月　２７日 第１版

[著作・制作] 独立行政法人情報処理推進機構 セキュリティセンター