ファズ・テスト

他人が攻撃する前に自分でプログラムを攻撃する

Elliotte Rusty Harold
Adjunct Professor
Polytechnic University

2006年 9月 26日

ファズ・テストは単純な手法ですが、コード品質に与えるその影響は計り知れません。この記事では Elliotte Rusty Harold が、障害部分を調べるためにランダムな不良データを故意にアプリケーションに注入すると、どのような結果になるかを説明します。また、チェックサム、XML データ・ストレージ、そしてコード検証など、ランダム・データに対するプログラムの脆弱性を解消するための防衛的コーディング手法についても説明しています。そして最後に、コードを守るための決定的手法として、コード・クラッカーとしての思考を持つ訓練で記事を締めくくっています。

長年の間、私は Microsoft Word をクラッシュさせる可能性のあるエラー・ファイルの多さに愕然としていました。わずかな数のバイトが所定の場所にないだけで、アプリケーション全体がダウンし、メモリーが保護されていないかつてのオペレーティング・システムでは、コンピューター全体が故障するのが常でした。何故、Word では不良データを受け取ったときにそれを認識して、エラー・メッセージを表示することができないのでしょうか。少数のビットがいじられたからといって、Word 自体のスタックとヒープが壊れてしまうのは何故でしょう。不正な形式のファイルに直面してそんな非情な動作をするのは、もちろん Word に限ったことではありません。

この記事では、このような最悪の事態を回避するための手法、ファズ・テストを紹介します。ファズ・テストでは、ランダムな不良データ (別名、ファズ) でプログラムを攻撃し、何が壊れるのかを調べます。ファズ・テストは、論理的でないというところに秘訣があります。クラッシュを引き起こす可能性のあるデータを予測しようと試みるのではなく (これは、人間のテスターがとる方法です)、自動化されたファズ・テストでは単純に、できるだけ多くのでたらめなランダム・データをプログラムにスローします。このようなテストは、プログラマーにとって衝撃的な結果となります。ファズ・テストでは、論理的な人間には思いもよらない故障モードが識別されるからです。

ファズ・テストは単純な手法とはいえ、プログラムに潜む重大なバグを明らかにすることが可能です。このテストは実世界の故障モードを識別し、ソフトウェアをリリースする前に封じ込めておかなければならない攻撃の可能性を警告します。

ファズ・テストの仕組み

ファズ・テストは、以下のように極めて簡単な手順でインプリメントできます。

1. プログラムに入力する正しいファイルを準備する。
2. ファイルの一部をランダム・データに置き換える。
3. このファイルをプログラムで開く。
4. 何が故障したかを調べる。

ランダム・データは何通りにでも変えることができます。例えば、ファイルの一部をランダム・データに置き換えるだけでなく、ファイル全体をランダム化したり、ファイルを ASCII テキストまたはゼロ以外のバイトに制限することもできます。どのように変えるにしても、重要なのはアプリケーションに多くのランダム・データをスローして、どこに障害が発生するかを調べることです。

C ベース・アプリケーションのテスト

C 言語で作成されたプログラムの多くは、ストリングに余分なゼロが含まれている場合に問題が発生します。余分なゼロによる問題はあまりにも多いため、コードのその他の問題が見逃されてしまうことがあります。プログラムの「ゼロ・バイト」による問題を見つけたら、その問題を取り除き、その他の問題が表面化されるようにしてください。

テストは手動でも開始できますが、ファズ注入を自動化してその効果を最大限にすることをお勧めします。自動化する場合はまず、アプリケーションがエラー入力に遭遇したときに行う適切なエラー動作を定義する必要があります (プログラムに入力データが壊れている場合の動作が定義されていないことがわかったら、それが最初のバグです)。次に、適切なエラー・ダイアログ、メッセージ、例外などをトリガーしないファイルが見つかるまで、ランダム・データをプログラムに渡し続けます。該当するファイルが見つかったら、それを保管してログに記録し、問題を後で再現できるようにします。これを繰り返します。

ファズ・テストには通常、手動でのコーディングがいくらか必要となりますが、そのための支援ツールがあります。例えば、リスト 1 に示す単純な Java™ クラスは、ファイルの特定の長さをランダムに変更します。私が通常ファズ注入を開始する場所は、最初のいくつかのバイトの後です。これは、プログラムにとっては、最初にあるエラーよりも後のほうにあるエラーのほうが検出しにくいためです (プログラムがチェックするエラーではなく、プログラムがチェックしないエラーを見つける必要があります)。

リスト 1. ファイルの一部をランダム・データに置換するクラス

```
import java.io.*;
import java.security.SecureRandom;
import java.util.Random;

public class Fuzzer {

     private Random random = new SecureRandom();
     private int count = 1;

     public File fuzz(File in, int start, int length) throws IOException
{

          byte[] data = new byte[(int) in.length()];
          DataInputStream din = new DataInputStream(new FileInputStream(in));
```

```
            din.readFully(data);
            fuzz(data, start, length);
            String name = "fuzz_" + count + "_" + in.getName();
            File fout = new File(name);
            FileOutputStream  out = new FileOutputStream(fout);
            out.write(data);
            out.close();
            din.close();
            count++;
            return fout;
        }


        // Modifies byte array in place
        public void fuzz(byte[] in, int start, int length) {

            byte[] fuzz = new byte[length];
            random.nextBytes(fuzz);
            System.arraycopy(fuzz, 0, in, start, fuzz.length);

        }

}
```

コードについて

リスト 1 のコードを最適化する方法はいろいろとあります。例えば最適化の優良候補として、java.nio でメモリーをファイルにマップする方法が考えられます。一方、エラー・ハンドリングと構成しやすさを向上させることもできます。ですが、この記事の本筋に詳細を割り込ませたくはないので、コードはそのままとします。

ファイルにファズを注入するのは簡単です。これをアプリケーションに渡すのも通常は同じく簡単です。ファズ・テストのこの部分を作成するには、大抵の場合、AppleScript や Perl などの言語でスクリプトを作成するのが最善の選択です。GUI プログラムの場合、アプリケーションが正しい故障モードを示しているかどうかを認識するのが非常に困難な部分になります。場合によっては、プログラムの前に座って、それぞれのテストが合格したか、不合格であるかをマークするのが一番簡単な方法になります。ここで確実にしなければならないのは、生成されたすべてのランダム・テスト事例に名前を付けて保管し、この手順で検出した障害を再現できるようにすることです。

防衛的コーディング手法

堅実なコードは、一貫性と正常性が検証されていない外部データは絶対にプログラムに入力しないという原則を守ります。

ファイルから数字を読み込んで、これが正の値であることを期待する場合、その数字を処理する前に、正の値であることを確認してください。ストリングに ASCII 文字だけが含まれていることを期待する場合は、ASCII 文字しか含まれていないことを確実にしてください。ファイルに4 バイトの整数倍が含まれていると思う場合は、それを確認してください。絶対に、外部から供給されるデータの特性が期待どおりのものであると仮定してはなりません。

もっとも多い過ちは、プログラムのインスタンスから作成されたデータは、検証しなくてもプログラムに再読み込みできるものだと仮定することですが、これは危険です。データはディスク上で別のプログラムによって上書きされているかもしれません。故障したディスクや誤ったネットワーク転送によって壊されているかもしれません。バグを持った別のプログラムによって変更さ

れているかもしれません。あるいはプログラムのセキュリティーを破るために故意に変更されている可能性だってあります。仮定は禁物です。何事も検証してください。

もちろん、エラー・ハンドリングと検証は面倒でやっかいで面倒な作業であるため、世界中のプログラマーから倦厭されています。コンピューターの60 年にわたる歴史のなか、ファイルが正常に開いたかどうか、メモリーが正常に割り振られているかどうかなどの基本的なチェックはいまだに行われていません。ファイルを読み込むときに各バイトとすべてのインバリアントをテストするようにプログラマーに頼んでも望みはなさそうですが、これを怠ると、プログラムはファズの被害を受けやすいままになってしまいます。ただし、幸いなことに救いはあります。現在のツールと技術を正しく使用することによって、アプリケーションを強固にするための苦労が大幅に軽減されます。なかでも卓越しているのは、以下の 3 つの手法です。

- チェックサム
- XML などの文法に基づくフォーマット
- Java などの検証済みコード

チェックサムによるファズの防御

ファズから保護するためにもっとも簡単にできることは、チェックサムをデータに追加することです。例えば、ファイルに含まれるすべてのバイトを合計し、256 で割った残りの値を算出します。このチェックサムと一致するかどうかによって、入力データが信用できるものであるかどうかを調べることができます。この非常に単純な方法によって、不測の故障が検出されないままになるリスクは 256 分の 1 に減ります。

MD5 や SHA などの確固たるチェックサム・アルゴリズムでは、256 で割った余りの値を算出する以上のことを行います。Java 言語では、java.security.DigestInputStream および java.security.DigestOutputStream クラスが、チェックサムをデータに付加する便利な手段となります。これらのチェックサム・アルゴリズムのいずれかを使用すれば、不測の損壊が起こる可能性が 10 億分の 1 以下にまで減ります (故意の攻撃はまだ可能性として残されていますが、それについてはこの後、説明します)。

XML ストレージと検証

XML でのデータ保管は、データ破壊による問題を回避する絶好の方法となります。XML は元々、Web ページ、本、詩、記事などの文書を対象としていましたが、財務データからシリアライズされたオブジェクトのベクトル・グラフィックに至るまで、ほとんどすべての分野で広範に活用されています。

非現実的な境界

XML パーサーをどうしても破壊させたいという場合、いくつかの方法を試してみることができます。例えば、ほとんどの XML パーサーでは特定の最大サイズを条件としています。SAX パーサーは、要素名が 22 億文字 (Java String の最大サイズ) より長くなると故障します。ただし実際には、これらの制限値はあまりにも大きいため、制限に達する前にメモリーが使い果たされることになります。

XML フォーマットでファズを阻止できるようにしている主要な特性は、XML パーサーは入力に関して何も仮定しないということです。これこそが、堅固なファイル・フォーマットに必要なものです。XML パーサーは、あらゆる入力 (整形式または有効であるなしに関わらず) を定義された方

法で処理するように設計されています。そして、XML パーサーではどんなバイト・ストリームでも処理できます。データが最初に XML パーサーを通過すると、このパーサーが提供可能なデータだけが出力されてきます。例えば、XML パーサーは決してヌルを渡さないため、データにヌル文字が含まれているどうかをチェックする必要はありません。XML パーサーが入力にヌル文字を見つけた場合は、例外をスローして処理を停止するからです。もちろんこの例外に対処する必要はありますが、考えられるすべてのエラーを検出するためのコードを作成するより、検出されたエラーを処理する catch ブロックを作成するほうがずっと簡単です。

セキュリティーを強化するには、DTD やスキーマで文書を検証します。この検証では、XML が整形式であるだけでなく、少なくとも期待に近いものであることがチェックされます。検証で文書について知りたいすべてのことを把握することは無理ですが、検証によって、多数の単純なチェックを作成しやすくなります。XML を使えば極めて簡単に、許容する文書を、処理方法がわかるフォーマットだけに限定することができます。

コードには、DTD やスキーマでは検証できない部分があります。例えば、請求書のある商品の価格が、在庫データベースでのその商品の価格と同じであるかどうかをテストすることはできません。価格が含まれる注文書をカスタマーから受け取ったら、そのフォーマットが XML であろうとなかろうと常に、カスタマーが価格を変更していないことを確実にしてから送信する必要があります。ただし、このような最終チェックはカスタム・コードでインプリメントできます。

文法に基づくフォーマット

XML の耐ファズ性を決定している特性の一つは、フォーマットが BNF (Backus-Naur Form) 文法で慎重かつ正式に定義されていることです。多くのパーサーは、JavaCC や Bison などのパーサー生成ツールを使用して、この文法から直接作成されています。そのようなツールは本質として、任意の入力ストリームを読み取り、それが文法を満足させるものであるかどうかを判断します。

XML がお使いのファイル・フォーマットに適切でないとしても、パーサーを基本としたソリューションの頑強性を備えることはできます。ただしそのためには、ファイル・フォーマットに独自の文法を作成し、その文法を読み取る独自のパーサーを開発する必要があります。既製の XML パーサーを使用する場合に比べ、独自のパーサーを開発するのは大変な作業ですが、データを文法に照らし合わせて検証せずに単にメモリーにロードするより、はるかに確実なソリューションになります。

Java コードの検証

ファズ・テストによるクラッシュの多くは、メモリー割り当てのエラーとバッファーのオーバーフローに直接起因しています。仮想マシンで実行するアプリケーションを Java や管理された C# 言語などのガーベッジ・コレクションが行われる安全な言語で作成すると、多くの潜在的問題が排除されることになります。C や C++ でコードを作成する場合でも、信頼性の高いガーベッジ・コレクション・ライブラリーを使用しなければなりません。2006 年の現在、独自のメモリーを管理しているデスクトップやサーバーのプログラマーがいてはなりません。

Java ランタイムでは、ランタイム自体のコードに保護レイヤーが追加されています。.class ファイルは仮想マシンにロードされる前に、バイト・コード・ベリファイヤー、そしてオプションで SecurityManager によって検証されます。Java は、.class ファイルを作成したコンパイラーにバグが出ないと仮定したり、コンパイラーが正しく動作するとは仮定しません。Java 言語は当初か

ら、信頼できない、場合によっては悪意のあるコードをセキュア・サンドボックスで実行できるように設計されています。Java 言語自体でコンパイルしたコードでさえも信用しません。場合によっては、誰かが16 進エディターを使って、バッファーのオーバーフローを引き起こそうと手動でバイト・コードを変更していることもあり得ます。プログラムへの入力に対しては、私たちの全員がこれくらいの被害妄想を持っていなければなりません。

敵と同じように考える

今まで説明したそれぞれの手法には、予期しない損害を防ぐまでには長い道のりがあります。すべてを組み合わせて正しくインプリメントすれば、未発見の偶発的な損害が起こる可能性が本質的にはゼロになります(完全にゼロにはならないにしても、浮遊する宇宙線によって CPU が 1 + 1 = 3 の結果を出す可能性と同じくらいの確率です)。ただし、すべてのデータ破壊が偶発的なものだとは限りません。誰かがプログラムのセキュリティーを破るために故意に不良データを導入したとしたらどうなるでしょう。クラッカーと同じように考えることが、コードを保護するための次のステップです。

アタッカーの考え方に頭を切り替えて、攻撃対象のアプリケーションが Java プログラミング言語で作成されている場合を考えてみましょう。このアプリケーションは非ネイティブ・コードを使用し、すべての外部データは XML で保管されるため、入力される前に完全に検証されます。それでも、攻撃は成功するでしょうか。答えはイエスです。ただし、ファイル内のバイトをランダムに変更するような単純な手段では歯が立ちません。必要なのは、プログラムに組み込まれたエラー検出メカニズムと、このメカニズムを中心とした経路を考慮に入れた、もっと高度な手法です。

ファズ対策を施したアプリケーションをテストするときには、純粋なブラックボックス・テストは使用できません。明らかな変更を加え、ただし基本概念はそのまま適用してテストしなければなりません。チェックサムを例に挙げれば、ファイル・フォーマットにチェックサムが含まれている場合、チェックサムがランダム・データと一致するように変更してから、ファイルをアプリケーションに渡します。

XML の場合は、文書に含まれる任意のバイト・セクションを選んでファズに置き換えるのではなく、個別要素のコンテンツと属性値にファズを注入するようにします。データはランダム・バイトではなく、正規の XML 文字に置き換えてください。これは、100 バイトのランダム・データであっても、ほとんど確実に不正な形式になるためです。要素名と属名を変えることもできます。ただし、変更した後も文書が正しい形式であることが確実な場合に限ります。XML 文書が非常に限定的なスキーマに照らし合わせてチェックされる場合は、スキーマがチェックしていない内容を調べて、効率的にファズを注入できる場所を判断する必要があります。

極めて限定的なスキーマと残りのデータのコード・レベルでの検証を組み合わせれば、データを操作する余地がなくなります。開発者にとっては、これが目指すところです。アプリケーションが当然無効だとして拒否しないようなバイト・ストリームを送信して、アプリケーションがこのストリームを有意義に処理できるようにしなければなりません。

まとめ

ファズ・テストでは、プログラム内のバグの存在を実証することができます。このテストは、バグが存在しないことを証明するものではありませんが、ファズ・テストに合格すれば、アプリ

ケーションが不測の入力に対して堅固でセキュアであるという大きな自信になります。プログラムに 24 時間ファズを投入して、それでもプログラムが持ちこたえているとしたら、今後同じような攻撃にセキュリティーが侵害される可能性はほとんどありません (不可能というわけではなく、可能性が低いということです)。ファズ・テストでプログラムのバグが明らかになった場合は、バグを修正する必要があります。ランダムに現れたバグにその都度を対処するより、チェックサム、XML、ガーベッジ・コレクション、そして文法に基づくファイル・フォーマットを賢明に使用して、ファイル・フォーマットを基礎から強固にするほうが生産的な場合もあります。

ファズ・テストはプログラム内の実際のエラーを識別するには不可欠のツールです。セキュリティーを意識して頑強性を目指すプログラマーにとっては、ツールボックスの中になくてはならない必需品です。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold はニューオーリンズ出身であり、時たま、おいしい gumbo (オクラ入りのスープ) を食べに帰っています。ただし現在はニューヨークのブルックリン近郊の Prospect Heights に、妻の Beth と猫の Charm (charmed quark からとりました) と Marjorie (義理の母の名前からとりました) と一緒に住んでいます。彼は Polytechnic University のコンピューター・サイエンスの非常勤教授として、Java 技術とオブジェクト指向プログラミングを教えています。彼の Cafe au Lait Web サイトは、インターネット上で最も人気のある独立系 Java サイトの一つです。また、そこから派生した Cafe con Leche は、最も人気のある XML サイトの一つです。彼の最近の著作には『Java I/O, 2nd edition』があります。現在は XML 処理用の XOM API や Jaxen XPath エンジン、Jester テスト・カバレッジ・ツールなどに取り組んでいます。