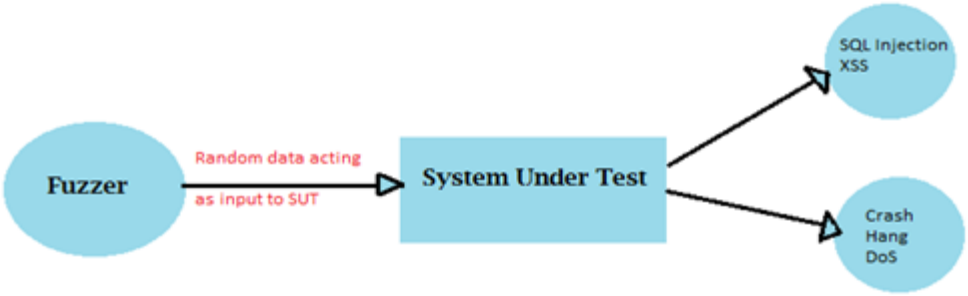
Basic concepts to start with:

이 글을 읽는 사람들 대부분은 "Fuzzing" 이라는 단어가 무엇인지 궁금해 할 것이라 생각한다. 그러나 가끔 특정 프로그램의 특정 파라미터, 인풋 값들을 변조하여 개발자가 예상하는 일반 입력값과 다르게 만들었다면 직접 스스로 퍼징을 해봤다고 할 수 있다.

IEEE의 Standard Glossary of Software Engineering 문서에 따르면 퍼징의 의미는 다음과 같다.

"시스템이나 컴포넌트가 입력값이나 과부화(stressful) 환경에서 정상적으로 동작할 수 있는 정도"

퍼징(혹은 fuzz testing)은 프로그램의 여러가지 문제점들을 알아내기 위해 사용하는 소프트웨어 테스팅 기법에 불과한 것이다. 이런 문제점들에는 "코딩 에러, XSS/BoF/DoS와 같은 보안 취약점" 등이 포함된다. 이를 위해 사용하는 예상하지 못한, 변조된, 랜덤 데이터 등을 프로그램의 입력으로 사용하여 프로그램 크래시를 유발하거나 예상하지 못한 형식으로 동작되게 만드는 것이다.



Fuzzing History:

퍼징의 기원은 1988년 위스콘신 대학교라고 알려져 있다. Barton Miller 교수는 "운영체제 유틸리티 프로그램 안정성 – 퍼즈 생성기" 라는 프로젝트를 과제로 학생들에게 주었다. 이는 처음이자 가장 간단한 형태의 퍼징이었으며 CLI 기반의 퍼저를 이용해 유닉스 프로그램을 대상으로 랜덤 비트 스트링을 전송하는 형태였다.

Miller 교수는 퍼즈(fuzz)의 초기 의미를 묻는 이메일에 대해 다음과 같은 답변을 보내왔다.

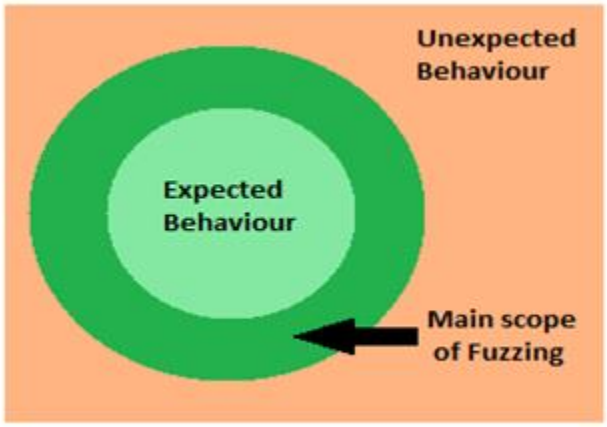
"폭풍이 내리는 동안 회선 잡음이 섞기면 모뎀에 로그인 할 수 있는 사실에 영감을 받아 시작한 것이 퍼징의 시작이다. 회선 잡음은 프로그램 크래시를 유발하는 듯한 문자들을 생성했다. 이 때 해당 잡음들을 퍼즈(fuzz) 라고 부르자고 제안하였다."

Fuzz 단어의 의미가 Barton Miller에 의해 제안되었다 해도 초기의 퍼징들은 "Moneky("라는 것이 포함되었다. 이는 작은 사이즈의 책상 액세서리였는데 갈고리를 이용해 대상 어플리케이션에 랜덤 이벤트를 발생할 때 사용하는 것이었다. 결과적으로 원숭이가 마우스와 키보드를 이용해 클릭 이벤트를 발생하고 랜덤한 위치에 드래그를 빠른 속도로 수행하기 때문에, 운영체제(매킨토시) 자체도 꽤 빠른 속도를 동작하는 것처럼 보여졌다.

1998년 오울루 대학교에서 "PROTOS" 프로젝트가 제안되었는데, 이는 소프트웨어 회사들이 모델 기반의 테스트 자동화 기법과 또 다른 차세대 퍼징 기법을 이용하여 소프트웨어의 심각한 보안 취약점을 찾을 수 있도록 돕기 위한 프로젝트이다. 이후(2001년) PROTOS 프로젝트를 기반으로 한 Codenomicon(네트워크 프로토콜 퍼즈 테스팅 솔루션)가 개발되었고 이와 더불어 퍼져의 기법과 방법론들은 지속적으로 발전되어 SPIKE, PeachFuzzer 등과 같이 차세대 퍼징 프레임워크가 등장 하게 되었다.

Why Fuzzing:

어떠한 제품에 대해 테스트 케이스가 정의되었을 때, 테스트 케이스들은 어떻게 동작되도록 개발되었는지(또한 어떻게 동작하면 안되는지)를 고려하면서 정의된다. 그러나 이런 일반적인 경우외에도 항상 제품 테스트 개발자 생각을 너머서는 정의되지 않은 영역들이 존재하게 된다. 정의되지 않은 해당 영역들을 분석하는 문제에 대하여 퍼징을 이용할 수 있다. 퍼징의 주된 목적은 제품이나 프로그램이 제대로 동작하는지 테스트 하는 것이 아니라 정의되지 않은 영역을 분석하고 테스트 하는 것이다.

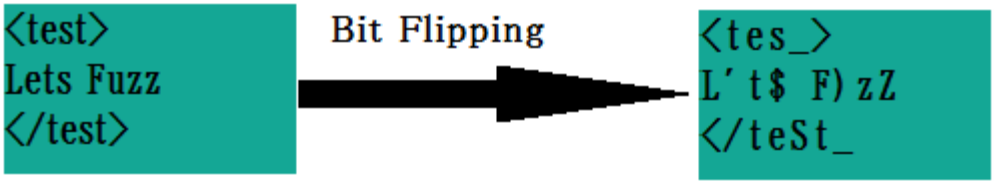


Mutation vs Generation (Dumb vs Intelligent)

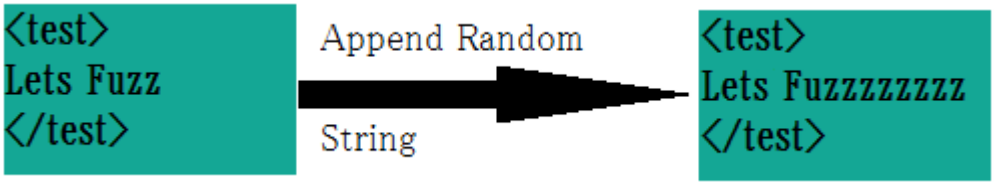
Mutation (Dumb Fuzzer):

덤프 퍼져는 이미 생성된 입력 값에 대해 특정한 조작을 가해 값을 변조하는 것이다. 그렇기 때문에 이런 방식은 입력으로 사용되는 데이터의 포맷이나 구조에 대한 지식이 없기 때문에 dump(무식한) 퍼져라고 한다. 예를 하나 들어보면 데이터의 랜덤 섹션 하나를 다른 것으로 바꾸거나 추가시키는 것이다.

예를 들어, Bit Flipping(비트 반전)은 뮤테이션 방식에서 사용하는 기법 중 하나인데, 특정 시퀀스나 랜덤하게 비트를 선택하여 반전시킨다.



유사한 방식으로 이미 존재하는 입력 데이터의 끝부분에 어떤 문자열을 추가할 수도 있다.

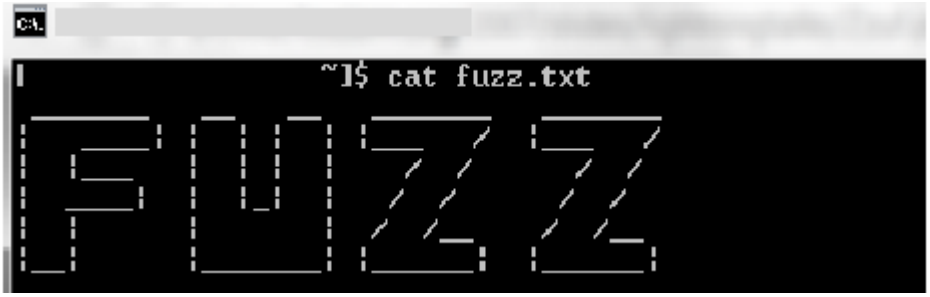


Bit Flipping with ZZUF (Mutation):

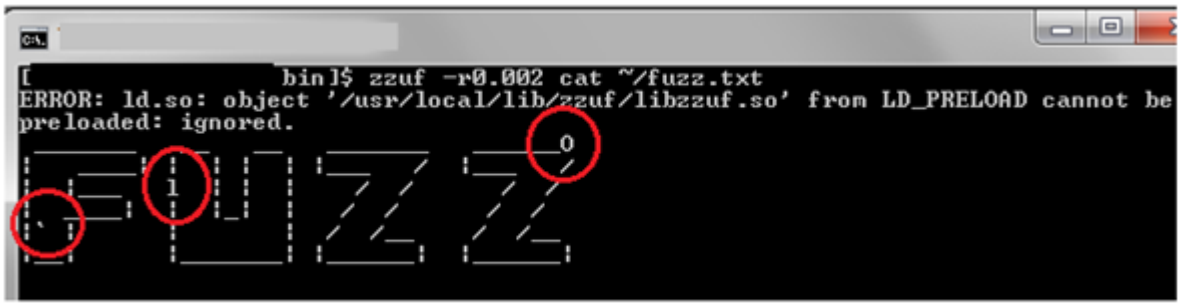
비트 반전 기법을 이용하는 대표적인 퍼져로 ZZUF가 있다. ZZUF는 사용하기 쉬운 퍼져인데, 사용자에 의해 작성되는 데이터들을 변조하고, 입력 값의 특정 비트를 랜덤하게 변경하여 어플리케이션의 버그를 찾는 것이 목적인 퍼져이다.

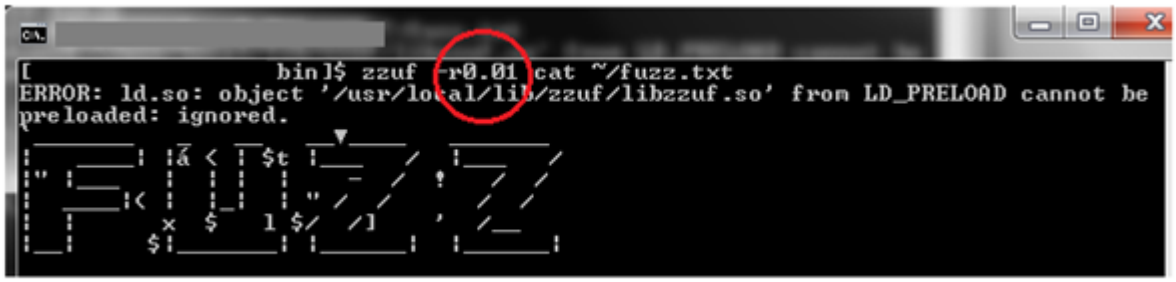
ZZUF 아키테쳐는 2개의 파트로 구성되어 있다. 첫 번째는 zzuf 바이너리이고, 두 번째는 Libzzuf 공용 라이브러리 파일이다. Zzuf 바이너리가 실행되면, 커맨드라인 퍼징 옵션을 읽어와 내부적인 환경(퍼져 자체 환경)에 저장한다. Libzzuf는 프로세스에 미리 로딩되어 실행된 다음 환경에 저장된 퍼징 옵션을 읽어온다. 파일과 네트워크 동작 등을 중간에 가로채기 위해 시그널 핸들러와 메모리 할당이 Libzzuf에 의해 동작의 방향을 바꾸고 관련 함수들을 다시 실행하는데, 이를 위해 비공개 C 라이브러리 심볼을 이용할 수 있다. 따라서 대상 어플리케이션의 동작들에 대해 방향을 변조하는 호출들은 Libzzuf를 향하게 된다.

zzuf를 이용하여 리눅스 cat 명령어의 입력 파일을 퍼징하는 방법을 알아보자.



위 사진은 파일을 zzuf 유틸리티의 입력으로 사용할 "fuzz.txt" 파일을 보여준다.





위 사진은 zzuf를 이용해, -r 옵션을 통해 우리가 퍼즈하고자 하는 비트의 개수를 지정하는 명령어와 결과를 보여준다. 보는 것처럼 0.002는 0.2% 비트를 퍼즈하라는 뜻이고, 0.01은 비트의 1%를 의미한다.

Mutation vs Generation (Dumb vs Intelligent/Smart)

Generation (Intelligent/Smart Fuzzer):

덤프 퍼져와의 비교를 위해 파일 포맷과 프로토콜을 이해하는 것이 스마트 퍼져에 있어서 매우 중요하다. 입력 파일 포맷이나 스펙를 기반으로 무작위 값을 만들어 입력값을 생성하는 것이 스마트 퍼져이다.

Intelligent Fuzzing with Peach Fuzzer:

피지 퍼져는 생성(Generation)과 변조(Mutation) 기능을 둘 다 지원하는 스마트 퍼져이다. PeatchPit 파일이라는 것을 생성함으로써 동작하는데, 해당 파일은 데이터 구조, 파일 종류, 데이터 간의 관계 등을 포함하는 XML 파일이다.

피지는 다양한 컴포넌트 위에서 동작하는데, 그 중에는 데이터 모델링, 상태 모델링, 공급자, 에이전트, 모니터, 로그 기록 등등이 있다. 따라서 이번 섹션에서는 모델링 파트를 중점으로 다를 것인데, 스마트 퍼져의 가장 핵심 부분이 모델링이기 때문이다. 이후에 퍼징의 두번째 파트에서는 피지 퍼저의 다른 컴포넌트들로 다룰 것이다.

예를 들어 HTTP 프로토콜을 대상으로 하는 샘플 PeachPit 파일을 생성하는 방법에 대해 얘기할 것이다.

피치는 데이터 모델링과 상태 모델링 총 2곳에 집중이 되어 있다. 퍼징의 디테일 수준은 해당 모델 내부로 전송되어 덤(dump) 퍼져와 스마트 퍼져의 차이를 만든다.

Data Modelling:

PeathPit 파일은 최소한 한 개 이상의 데이터 모델을 포함한다. 데이터 모델은 숫자나 문자열 등의 자식 요소 추가를 지정하여 데이터 블록의 구조를 정의한다.

복잡한 프로토콜들은 대부분 여러 가지 파트로 나뉘는데, 각 파트들은 파트 자체의 재사용을 위해 고유의 데이터 모델을 포함한다. 데이터 모델은 세부적으로 블록으로 나뉜다. 또한 만약 레퍼런스(ref 속성)이 지원 된다면 레퍼런스 데이터들은 새로운 데이터 모델을 생성할 때 기본 값으로 복사되어 사용된다. 데이터 모델 내의 그 어떠한 자식 요소던지 동일한 이름으로 이미 존재하는 요소들은 오버라이드 되어 사용된다.

<!-- 본 필드와 “Body” 필드와의 관계를 나타냄 -->

따라서, 이는 문자열 정의와 나중에 다른 값으로 변형될 것들을 오버라이드 하게 된다.

<!-- 본 필드와 “Body” 필드와의 관계를 나타냄 -->

우리는 또한 각각 다른 데이터들의 관계를 모델하기 위해 사용하는 관계 속성들을 볼 수 있다. 따라서 이런 경우 다른 값으로 정의되는 Body 요소의 사이즈와 동일한 값을 가진다.

HTTP 프로토콜에 대한 샘플 데이터 모델은 다음과 같다.

```
<DataModel name="Header">
    <String name="Header" />
    <String value=": " />
    <String name="Value" />
    <String value="\r\n" />
</DataModel>

<DataModel name="HttpRequest">

    <!-- The HTTP request line: GET http://foo.com HTTP/1.0 -->
    <Block name="RequestLine">

        <String name="Method"/>
        <String value=" " type="char"/>
        <String name="RequestUri"/>
        <String value=" "/>
        <String name="HttpVersion"/>
        <String value="\r\n"/>
    </Block>

    <Block name="HeaderHost" ref="Header">
        <String name="Header" value="Host" isStatic="true"/>
    </Block>

    <Block name="HeaderContentLength" ref="Header">
        <String name="Header" value="Content-Length" isStatic="true"/>
        <String name="Value">
            <Relation type="size" of="Body"/>
        </String>
    </Block>

    <String value="\r\n"/>

    <Blob name="Body" minOccurs="0" maxOccurs="1"/>

</DataModel>
```

State Modelling:

상태 모델은 최소한 한 개 이상의 상태와 모델을 포함한다. 복수의 상태를 가지고 있는 경우, 초기 상태 속성이 모델의 첫 번째 상태를 결정한다.

다음 그림에서 보는 것처럼 "<Action>" 태그는 상태 모델에서 공급자에게 입력값 변조 결과를 전송하거나, 특정 데이터 모델에서 명시되어 있는 입력값을 공급자를 통해 읽는 등 다양한 역할을 수행한다. 이번 예제의 경우 딱 한 개의 상태만이 존재하고, HttpRequest 데이터모델에서 동작할 한 개의 액션만 존재한다. 이제 액션은 추가적인 자식 요소를 가지고 있는데 바로 "Data"이다. 해당 자식 요소는 기본 데이터 셋을 생성하고 불러온 다음 데이터 모델 내부에 저장한다.

위 과정을 통해 어떠한 데이터 모델이든, 해당하는 기본 데이터 셋을 생성할 수 있고 만약 기본 값이 이미 존재하는 경우 이를 오버라이드 할 수 있다. 데이터 모델 레퍼런스와 비슷하게, 데이터에도 레퍼런스가 있다.

HTTP 프로토콜에 대한 샘플 데이터 모델과 데이터는 다음과 같다.

```
    <Data name="HttpGet">
        <Field name="RequestLine.Method" value="GET" />
        <Field name="RequestLine.RequestUri" value="http://localhost" />
        <Field name="RequestLine.HttpVersion" value="HTTP/1.1" />
        <Field name="HeaderHost.Value" value="http://loclahost" />
        <Field name="Body" value="Test Fuzzzinggggg " />
    </Data>

    <Data name="HttpOptions" ref="HttpGet">
        <Field name="RequestLine.Method" value="OPTIONS" />
        <Field name="RequestLine.RequestUri" value="*" />
        <Field name="HeaderHost.Value" value="" />
    </Data>

    <StateModel name="State1" initialState="Initial">
        <State name="Initial">
            <Action type="output">
                <DataModel ref="HttpRequest" />
                <Data ref="HttpGet" />
            </Action>
        </State>
    </StateModel>

    <StateModel name="State2" initialState="Initial">
        <State name="Initial">
            <Action type="output">
                <DataModel ref="HttpRequest" />
                <Data ref="HttpOptions" />
            </Action>
        </State>
    </StateModel>
```

데이터 모델링과 상태 모델링 생성이 완료되면, 이제 우리의 목표는 (위 예제에서 HTTP를 샘플로 했기 때문에) 웹 서버를 대상으로 퍼져를 실행하는 것이다. 테스트 요소들은 공급자와 결합되는 상태 모델의 특정 퍼징 테스트 설정을 구성한다. (공급자에 대해서는 이후에 다룰 예정). 여기서 공급자는 localhost 웹 서버를 대상으로 우리의 요청(request)을 목표를 한다.

<!-- 대상 로컬 웹 서버는 80번 포트에서 동작 중 -->

유사하게 다른 테스트 역시 수행할 수 있다. 마지막으로 모든 요소를 한 그룹으로 만들기 위해 구성 요소 자체를 실행시킬 필요가 있다.

<!-- 실행하고자 하는 테스트 셋 -->

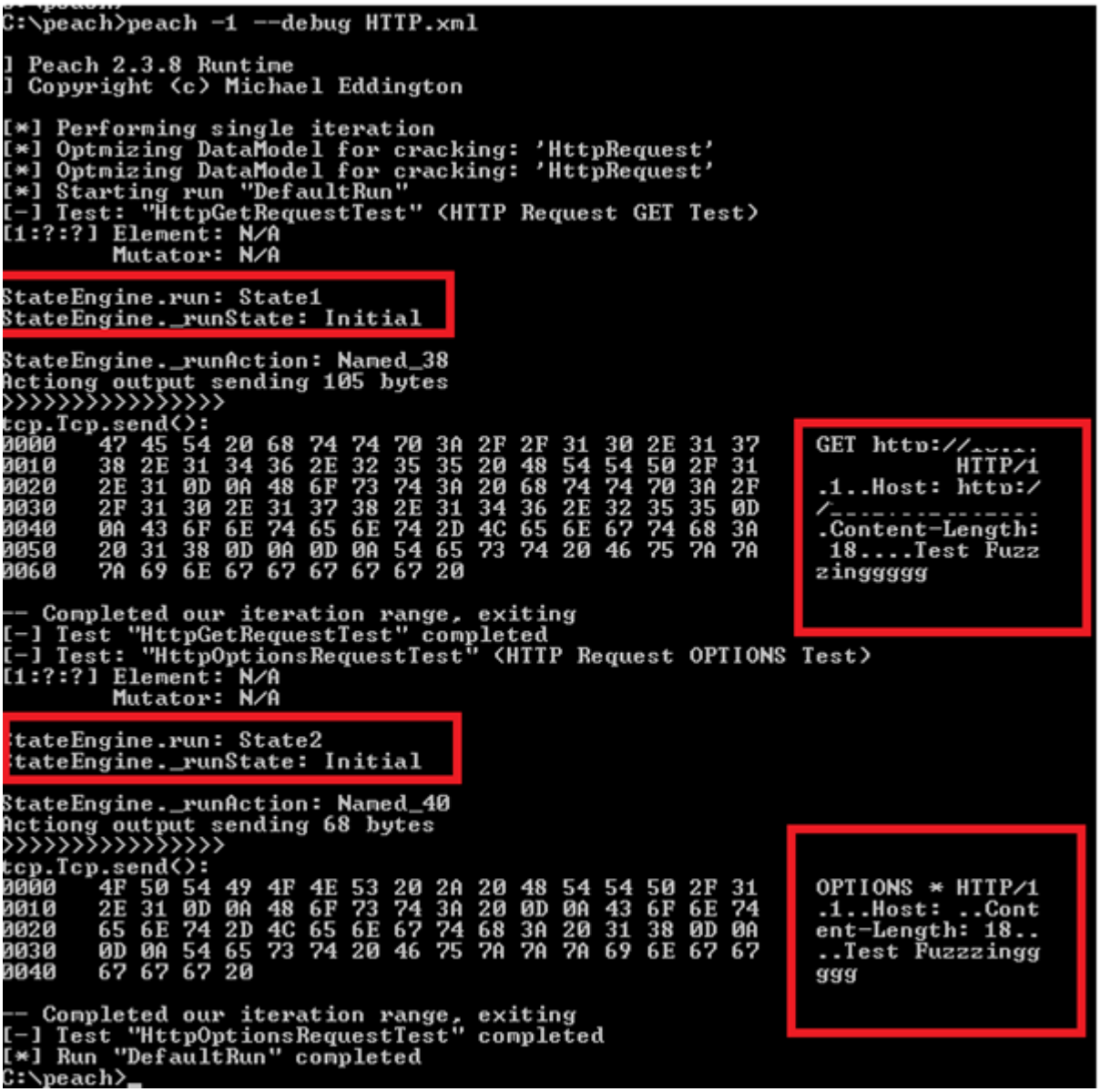
위와 같은 과정을 거치면 기본적인 PeachPit 파일 생성이 완료되며, 이를 통해 피지 퍼져를 실행시킬 수 있다.

```
C:\peach>peach -t HTTP.xml

] Peach 2.3.8 Runtime
] Copyright (c) Michael Eddington

[*] Optmizing DataModel for cracking: 'HttpRequest'
[*] Optmizing DataModel for cracking: 'HttpRequest'
File parsed without errors.
```

"-t" 옵션을 통해 피지 XML 파일을 파싱한다.



"-1" 옵션을 이용해 1번의 반복만을 수행하고 "--debug" 옵션을 통해 디버그 메시지에 대한 디버그를 진행한다.

로그 기록 데몬과 에이전트를 구성함으로써, 진행 상황과 잘못된 액션 등을 추적할 수 있다.

Comparison:

변조(mutation) 기반 방식이 생성(generation) 기반 방식보다 접근하기 쉽지만(대상 파일 포맷의 이해가 필요하지 않기 때문), 대부분의 경우 생성 방식이 다양한 입력 데이터를 포함하고 코드 커버리지와 실행 경로들을 더 많이 커버할 수 있기 때문에 훨씬 더 퍼징 결과과 좋게 나타난다. 또한 생성 방식이 시간이 좀 더 오래 걸리는데 이는 퍼징 프로세스에 대해 좀 더 많은 것을 고려하기 때문이다.

Advantages of Fuzzing

퍼징은 정의/접근 기반(defined testing/approach-base tesing) 테스팅 방법에서는 찾을 수 없는 버그를 찾을 수 있게 해주는 랜덤 테스팅 방식이다. 이는 실제 입력과 프로세스 시작 직전에 시작한 사항들에 대한 explotiable 이슈를 탐지한다. 퍼징/퍼져는 꽤 다루기 쉽고 초기 설정이 용이하고, 초기 설정만 제대로 된다면 재사용하기 간편하다.

Limitations of Fuzzing

퍼징은 간단한 버그는 꽤 효과적으로 탐지한다. 특히 블랙 박스 테스팅 방식에서 꽤 효과적이지만, 내부적으로 제약이 존재한다. 제약이라 하면 퍼징의 영향도를 체크할만한 정보가 적다는 것이다. 프로토콜, 파일 구조 스펙 문서를 작성하는 것은 꽤 지루하고 복잡한 일이다. 랜덤 접근 방식은 랜덤이라는 장점이 있지만, 반대로 말하면 경계값에서 발생하는 이슈는 찾기 힘들다는 것이다.

2번째 문서에서는 어플리케이션과 파일 퍼징에 대해 다룰 예정