Protokoll Fuzzing - Übung

In dieser Übung wollen wird den grey-box Fuzzer AFL++ verwenden, welcher hauptsächlich auf Mutationen basiert, mit etwas Skripting jedoch auf für protokollbasiertes Fuzzing eingesetzt werden kann.

Installation:

Da für diese Übung ein Docker Container verwendet wird kann sie auf jedem beliebigen 64-Bit Linuxsystem ausgeführt werden. Da viele der aktuellen VMs noch auf 32-Bit laufen empfehle ich die Kali VM zu verwenden.
Vorausgesetzt wird folgendes:

- Python3 (mindestens Version 3.10)
- Docker
- ein Editor der Wahl (z.B. VC-Code, vim, ...)

Zunächst muss Docker installiert werden.

```
# Docker installieren
sudo apt install docker.io

# Benutzer der Gruppe Docker hinzufügen
sudo usermod –aG docker $(whoami)

# Client neu starten
sudo reboot now
```

Danach kann der benötigte Docker Container heruntergeladen werden.

```
docker pull aflplusplus/aflplusplus
```

Vorbereitung

Starten Sie den Docker-Container und mounten Sie den Ordner „Fuzz“, welcher die Übungen enthält in ihm. Navigiere zum Ordner in dem sich die Übungen befinden und führen Sie dort folgenden Befehl aus:

```
1 docker run -ti -v $PWD:/fuzz aflplusplus/aflplusplus
```

Der Ordner sollte im Container nun unter dem Pfad „/fuzz/“ zu finden sein. Bearbeitungen der Dateien auf dem Host werden im Container ersichtlich sein also können während dieser Übung Dateien mit dem Editor der Wahl bequem bearbeitet werden.

Der Container stellt die Werkzeuge von AFL++ bereit, ohne dass ein compilieren stattfinden muss. Falls Sie in dieser Übung an eine Grenze stoßen sollten oder sich über ein Tool informieren möchten empfehle ich die Dokumentation zum Projekt welche hier zu finden ist: https://github.com/AFLplusplus/AFLplusplus. Weitere Informationen insbesondere über die Befehle die in dieser Übung benötigt werden finden sie in der „README.md“ und der „example.md“, welche den Übungen beiliegen.

Viel Erfolg!

Übungen

Aufgabe 1:

Ziel dieser Aufgabe ist es sich mit AFL++ vertraut zu machen und den Fuzzer das erste mal laufen zu lassen. Damit der Fuzzer überhaupt etwas in annehmbarer Zeit finden kann wird ein C-Programm zur Verfügung gestellt, welches eine Reihe an Implementierungsfehlern enthält, welche von Fuzzer gefunden werden können. Sie finden die Datei in dieser Übung unter „target/damn_vulnerable.c“.

Öffnen Sie die Datei und verschaffen Sie sich einen Überblick über das Programm. Wie viele eingebaute Programmierfehler enthält es? Zunächst muss das Programm instrumentalisiert/compiliert werden, hier wird der Befehl „afl-cc“ verwendet.

```
1 afl-gcc -g -fsanitize=address damn_vulnerable.c -o
    damn_vulnerable.out
```

Warum muss ein Sanitizer eingesetzt werden?

Als nächstes wird der Fuzzer mit der entsprechenden Binary ausgeführt, den Fuzzer kann man mittels des Befehls „afl-fuzz“ angesprochen werden. Verschaffen Sie sich eine Überblick über die möglichen Flags und Umgebungsvariablen mittels der „-hh“ Flag.

Um den Fuzzer mit der C-Binary auszuführen verwende:

```
1 afl-fuzz -i /fuzz/corpus/damn_vulnerable -o /fuzz/out -m none -
    S damn_vulnerable -- /fuzz/target/damn_vulnerable.out @@
```

In der Standardeinstellung startet der Fuzzer automatisch die UI auf der zahlreiche Informationen angezeigt werden. Um mehr über die einzelnen Felder zu erfahren siehe: https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md#understanding-the-status-screen.

Während des Fuzzing Vorganges kann es sinnvoll sein die Eingaben des Fuzzers im Blick zu behalten um festzustellen ob sie noch das richtige Format besitzen, dies kann nicht in der UI geschehen sonder muss in einem anderen Terminal erfolgen. Öffne ein neues Terminal (den Prozess nicht beenden) und führe folgende Befehle aus.

```
# die ID des Containers herausfinden (erster Wert in hex)
docker container ls

# eine neue Shell im Container starten
docker exec -it <container_id> /bin/bash

# die Eingabe Datei zyklisch auslesen
watch -n 0.1 cat /fuzz/finding/damn_vulnerable/.cur_input
```

Das Crash Zähler solle relativ schnell ansteigen, sobald einige Crashes erkannt wurde kann der Fuzzer beendet werden und wir können uns die Crashes ansehen. AFL++ speichert alle Information zu einer Kampagne im angegeben Order hinter dem Parameter „-o“. In unserem Fall ist dies „/fuzz/finding/damn_vulnerable“. Schaue im „crashes/“ Ordner in ein paar Dateien, kannst du herausfinden welche Crashes erkannt wurden und wie die Eingabe hierfür war?

Das Projekt aus dem diese Datei stammt finden Sie hier: https://github.com/hardik05/Damn_Vulnerable_C_Program.

In den folgenden Aufgaben soll das Vorgehen beim Fuzzen von Protokollen erläutert werden. Hierzu werden eine Reihe von „Servern“ als Binaries bereitgestellt auf welchen ein einfaches Beispielprotokoll implementiert wurde. Um bei den Eingabe auf einen komplexen Harness zu verzichten nehmen die Server Eingaben über eine Datei entgegen, der einzige Übergabeparameter der Binary ist der Pfad zu dieser Eingabedatei. Neben den normalen Mutationsgeneratoren unterstüzt AFL++ auch die Verwendung eigen implementierter, spezifischer Mutatoren welche durch eine API in C oder Python implementiert werden können. Diese Funktionalität wird verwendet werden um den Fehler in der Implementierung zu finden bzw. das Protokoll überhaupt zu unterstützen.

Das Protokoll ist nach einem einfachen Schema aufgebaut: „<COMMAND> <payload>“. Siehe die Dateien im „corpus/“ Verzeichnis für einige Beispiele.

In der ZIP-Datei finden Sie die vordefinierte Struktur für die Bearbeitung der Aufgaben. Die wichtigsten Verzeichnisse sind:

- **corpus**
  Hier sind sind Beispieleingaben für die verschiedenen Versionen zu finden. Analysieren Sie diese Datei zuerst und probieren Sie im angegebenen Schema einige Eingaben an die Binary auf um ein Gefühl dafür zu erhalten wie sie funktioniert.
- **target**
  Die Binaries der Server sind hier zu finden, diese wurden bereits instrumentiert also mit dem AFL++ Compiler compiliert.

- **finding**
  Dieser Ordner ist aktuell noch leer, hier werden während und nach der Ausführung Information zu gefundenen Crashes, Eingaben und der Coverage zu finden sein. Wenn die „-S“ Flag richtig verwendet wird werden die Unterordner genau wie der aktuelle Server heißen.
- **mutator**
  Die Implementierung der Eingabegeneratoren soll hier erfolgen, die Dateien sowie die Funktionen in ihnen wurden bereits angelegt und können einfach bearbeitet werden. Es finden sich hier außerdem zwei Skripte, welche hilfreich beim Debugging der Generatoren sein können. Das „test_mutator.py“ Skript kann verwendet werden um 10 Ausgaben des aktuellen Generators zu erhalten. Öffen Sie es und anlysieren Sie die Implementierung um mehr über die Verwendung zu erfahren.

Jede Binary enthält ein Flag, welche bei Ausnutzung des Implementierungsfehlers sichtbar wird. Ziel ist es alles Flags zu sammeln!

Aufgabe 2:

Die Binary lautet: **server_v1.out**

Finden Sie mithilfe des Fuzzers den Fehler! Reproduzieren Sie den Fehler durch eine manuelle Eingab, wie lautet die Flag? Unter welchen Bedingungen tritt der Fehler auf, definieren Sie die genaue Ober- & Untergrenze.

Hinweis: Einen Elefanten bekommt man nicht in einen Schuhkarton.

Aufgabe 3:

Die Binary lautet: **server_v2.out**

Finden Sie mithilfe des Fuzzers den Fehler! Reproduzieren Sie den Fehler durch eine manuelle Eingab, wie lautet die Flag? Unter welchen Bedingungen tritt der Fehler auf, bestimmen Sie die genaue Position.

Hinweis: Bei dieser „military grade“ Autorisierung geht es darum ein Bit an einer bestimmten Stelle zu setzen. An welcher nur?

Aufgabe 4:

Die Binary lautet: **server_v3.out**

Dieses Übung soll zeigen warum eigen erstelle Generatoren gerade bei Prüfsummen sehr wichtig sind. Die letzten zwei Stellen des Payloads werden von diesem Server als Prüfsumme interpretiert, wie die Prüfsumme kalkuliert wird kann dir die Binary „calculate_checksum.out“ verraten. Übergeben ihr einen String und versuche die Berechnung der Prüfsumme nachzubauen.

Hinweis: Als letzten Schritt der Berechnung wird die Prüfsumme modulo 100 gerechnet um sie auf zwei Ziffern zu begrenzen.

Finden Sie mithilfe des Fuzzers den Fehler! Reproduzieren Sie den Fehler durch eine manuelle Eingab, wie lautet die Flag? Unter welchen Bedingungen tritt der Fehler auf?

Hinweis: Zahlen sind nur etwas für Prüfsummen.

Aufgabe 5:

Die Binary lautet: **server_v4.out**

Selbst der Command ist in dieser Übung unbekannt?! Finden Sie den richtigen Command!

Hinweis: Man munkelt er sei vier Zeichen lang und beginnt mit einem „H“.

Finden Sie mithilfe des Fuzzers den Fehler! Reproduzieren Sie den Fehler durch eine manuelle Eingab, wie lautet die Flag? Unter welchen Bedingungen tritt der Fehler auf, bestimme die genaue Position und Zeichenfolge.

Hinweis: Es wird eine Kombination von Zeichen im Payload gesucht. Der Payload besitzt eine Länge von 20 Zeichen.

Achtung: Diese Aufgabe benötigt Zeit, es kann einige Minuten dauern bis der Fuzzer ein Ergebnis liefert.