OST
Ostschweizer
Fachhochschule

Fuzzing .NET

Studienarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2020/2021

Autoren:	Kevin Moro, Danusan Premananthan, Aynkaran Sundralingam
Betreuer:	Prof. Dr. Markus Stolze

Abstract

Fuzzing ist eine Form von Software-Testing bei dem ein Programm mit zufällig generierten Daten gefüttert wird. Das Ziel dieser Tests ist es ein abnormales Verhalten des Programmes zu finden das man mit anderen Testmethoden nicht finden würde. Diese Technik ist für das Testen eines XML-Parsers sehr hilfreich, da man so auf Testszenarien/Programminputs kommt, die man sonst nicht in Betracht ziehen würde.
Das Ziel dieser Arbeit ist es, verschiedene Fuzzing Tools mit den dazugehörigen Tool-Chains für die .NET Plattform zu recherchieren und danach auch zu evaluieren, mit besonderer Aufmerksamkeit auf die Eignung der Tools für das Testen von einem XML-Parser, welchen wir von nxt Engineering zur Verfügung gestellt bekommen haben. Ebenfalls ein Teil der Arbeit ist die Evaluation des neuen, von Microsoft entwickelten, Tool OneFuzz.
Um dieses Ziel zu erreichen, wurden zuerst die theoretischen Eigenschaften verschiedener Fuzzing-Tools analysiert. In einem zweiten Schritt wurden die vielversprechendsten Tools in Betrieb genommen und praktisch an einem XML-Parser getestet. Ein Resultat dieser Arbeit ist die Tool-Chain die sich besonders gut eignet, um XML-Parser auf .NET Umgebung zu testen und auf was man bei der Einrichtung sowie dem Testen besonders achten muss. Des Weiteren wurden während dem Testen des XML-Parsers von nxt Engineering nur wenige kleine Programmierfehler gefunden.

Inhaltsverzeichnis

1 Allgemeines

1.1 Änderungsgeschichte

Datum	Version	Änderung	Autor
16.11.2020	1.0	Initialer Entwurf	Kevin Moro Danusan Premananthan Aynkaran Sundralingam
18.12.2020	4.3	Überarbeitet	Kevin Moro Danusan Premananthan Aynkaran Sundralingam

1.2 Konventionen

► Hinweise

Commands, Code Snippets, Scripts und Error messages / Exceptions

Querverweise innerhalb des Dokuments sind kursiv dargestellt

Links auf externe Ressourcen sind blau unterstrichen dargestellt

Teil I

Einleitung

2 Management Summary

Fuzzing .NET - XML-Parser Fuzzing

Diplomanden	Kevin Moro, Danusan Premananthan, Aynkaran Sundralingam
Examinator	Prof. Dr. Markus Stolze
Themengebiet	Sicherheit

2.1 Ausgangslage

Diese Studienarbeit befasst sich mit der Aufgabe, Fuzzing-Möglichkeiten in der .NET-Umgebung (C#) zu untersuchen. Fuzzing ist eine Test-Methode für Anwendungen, bei der einem Programm zufällig generierte Daten übergeben werden, um damit ein eventuelles Fehlverhalten auszulösen. Die Untersuchung soll an einem in C# geschriebenen XML-Parsers, welcher von "nxt Engineering" zur Verfügung gestellt wird, getestet und dokumentiert werden. Für das Fuzzing soll dabei die grammatikbasierte Greybox-Fuzzing Methode verwendet werden. Ziel dieser Arbeit ist es, mögliche Tool-Chains für das Fuzzing von C#-Anwendungen zu untersuchen und diese durch die Anwendung auf den Parser zu evaluieren.
Zudem soll ein Erfahrungsbericht über die Webseite "The Fuzzing Book" geschrieben werden. Da wir noch kein Wissen und keine Erfahrung mit Fuzzing haben, sollten wir diese Website vor allem nutzen, um uns das notwendige Wissen anzueignen. Ebenfalls wird die Eignung dieser Website für die eigenständige Erarbeitung des Themas "Fuzzing" bewertet.

2.2 Vorgehen und Technologien

Für das Fuzzing muss die Zielsoftware zuerst analysiert werden. Dadurch konnten wir feststellen, welche Klasse für unsere Arbeit die wichtigste ist. Um diesen Teil der Software möglichst gut fuzzen zu können, braucht es ein Fuzzing-Tool, welches Grammatik- und gleichzeitig Coverage-basiert arbeitet. Um die Software vollständig testen zu können, muss es möglich sein, die Daten nach Codeabdeckung zu generieren. Nach der Tool-Chain Recherche und der Vorgabe unseres Betreuers, wurden einige interessante Fuzzing-Tools festgelegt, welche möglicherweise unser Vorgehen unterstützen.

2.3 Ergebnisse

Das Ergebnis dieser Arbeit sind die gewonnenen Erkenntnisse über die Fuzzing-Möglichkeiten auf der .NET Umgebung, am Beispiel eines XML-Parsers.
Weitere Ergebnisse sind gefundene Fehler, die durch das Fuzzing entdeckt wurden. Zu diesen wurden mögliche Massnahmen für die Behebung dokumentiert.
Um der Firma nxt Engineering ein umfassenderes Ergebnis anzubieten, wurden neben dem Fuzzing weitere Tests durchgeführt und dokumentiert.

3 Aufgabenstellung

Studiengang Informatik
Studienarbeit HS 2020 «Fuzzing .NET»

Aufgabenstellung Studienarbeit «Fuzzing .NET»

1. Betreuer

Prof. Dr. Markus Stolze, OST, IFS
markus.stolze@ost.ch

2. Studierende

Diese Arbeit wird durchgeführt von

- Kevin Moro
- Danusan Premananthan
- Aynkaran Sundralingam

3. Ziele der Arbeit

Die folgenden Teilaufgaben sind zu bearbeiten

- Evaluation der Eignung der im Web-Site «Fuzzing Book» (https://www.fuzzingbook.org/) bereitgestellten Informationen für eigenständige Erarbeitung des Themas «Fuzzing» durch Studierende:
  - Dokumentation Lernaufwand und «schwierige» Passagen/Elemente.
  - Feedback an die Autoren zu Händen Prof. Stolze.
- Recherche und Evaluation von Fuzzing Tools (bzw. Tool-Chains) mit Fokus auf C#/.NET
  - OneFuzz https://www.microsoft.com/en-us/research/project/project-onefuzz/)
  - Weitere von den Studierenden zu bestimmenden Tools
- Lösung eines Praxis-Problems: Fuzzing eines .NET XML-Parsers mittels grammar-driven Greybox-Fuzzing mit mindestens einem der evaluierten Tools (bzw. Toolchain)
  - Angemessene Dokumentation der bei der Anwendung gefundenen Herausforderungen, Problempunkte, Stolpersteine und Einschränkungen der getesteten Tools und Toolchains.
  - Angemessene Dokumentation der mittels Fuzzing gefundenen Probleme im XML-Parser und Vorschlag zur Behebung dieser Probleme
  - Dokumentation wie allgemein beim Fuzzing von XML-Parsern (XSD vorhanden) im .NET Umfeld vorgegangen werden sollte:
    - Vorschlag einer verallgemeinerten Fuzzing Toolchain für ähnliche Probleme
    - Beschreibung der Anwendbarkeit der Toolchain:
      Erklärung was «ähnliche» Probleme sind
    - Sinnvoll dokumentierte Anleitung zur Fuzzing-Vorbereitung (notwendige Lizenzen etc., Tools-Installation) und Fuzzing-Durchführung.
    - Bereitstellung eines Beispielprojekts anhand dessen XML Fuzzing im Informatik Unterricht an der OST demonstriert werden kann.

  - Analyse und Dokumentation der weiteren Verallgemeinerbarkeit der Toolchain(s).
    - Welche andere Programmtypen kann man fuzzen nach einfachen Anpassungen der Toolchain z.B. .Net XML-Parser ohne XSD: was kann man machen; XML-Parser mit XSD in Java, ...
- Optional:
  - Recherche Tools die sich zum Fuzzing von Node/Express Anwendungen eignen.
  - Automatische Extraktion der Express-Routes im Sinne eines grammar-driven Fuzzing.
  - Demonstration von Greybox Fuzzing für ein oder mehr Express-Beispiel-Anwendungen z.B. WED2 Testate.

4. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien des Studiengangs Informatik zu verfassen. Die zu erstellenden Dokumente bzw. Berichtsteile sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.

5. Termine

Siehe Terminplan auf dem Skripteserver (Fachbereich/Bachelor-Arbeit_Informatik/BAI/)

6. Beurteilung

Eine erfolgreiche Studienarbeit zählt 8 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von 30 Stunden.

Für die Beurteilung ist der verantwortliche Dozent zuständig.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/5
2. Berichte (Abstract, Mgmt. Summary, technischer u. persönliche Berichte) sowie Gliederung, Darstellung, Sprache der gesamten Dokumentation	1/5
3. Inhalt*)	3/5

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit präzisiert (siehe Bewertungsmatrix)

Im Übrigen gelten die Bestimmungen des Studiengangs Informatik für Studienarbeiten.

Rapperswil, 4 November 2020

Prof. Dr. Markus Stolze

Teil II

Analyse

4 Fuzz Testing ("Fuzzing")

Fuzzing ist eine automatisierte Testing-Methode, mit der Schwachstellen in Applikationen, Betriebssystemen oder auch Netzwerken entdeckt werden können. Ziel des Fuzzings ist es, festzustellen, ob für alle möglichen Eingaben die notwendigen Reaktionen im Programm hinterlegt sind. Die Applikation wird dabei mit ungültigen bzw. zufällig generierten Daten, welche "Fuzz" genannt werden, überflutet, um sie zum Absturz zu bringen. Eine Schwachstelle (Sicherheitslücke oder Programmierfehler) wird gefunden, wenn ein Absturz vorliegt. In diesem Fall kommt ein spezielles Tool namens "Fuzzer", oder auch Fuzz-Tester genannt, zum Einsatz, der die Ursachen für den Absturz herausfinden soll.

Zusammengefasst von: (computerweekly 2016; IONOS 2020)

4.1 Geschichte

Der Begriff "Fuzzing" wurde erstmals 1989 von Barton Miller verwendet, der einen primitiven Fuzzer entwickelte, um die Robustheit von UNIX-Dienstprogrammen zu testen. Die Arbeit wurde durch einen Vorfall inspiriert, welcher einem der Autoren des Artikels auffiel, als er während eines Sturms über ein Wählmodem verbunden war, wobei Regen viel Leitungsrauschen verursachte. Das Rauschen führte zu zufälligen Zeichen, die zu Programmabstürzen führten. Dieses beobachtete Verhalten wurde mit dem Begriff "Fuzz" bezeichnet.

Heutzutage wird Fuzzing von Sicherheitsanalysten verwendet, um Fehler aufzudecken und zu melden, von Qualitätssicherungs-Mitarbeitern, um die Qualität ihrer Software zu verbessern, und von Hackern, um Software-Exploits zu entdecken und heimlich auszunutzen. Etablierte Softwarefirmen nehmen Fuzz-Tests in ihren Softwareentwicklungs-Lebenszyklus auf, um bessere und robustere Softwareprodukte herzustellen.

Zusammengefasst von: (computerweekly 2016)

4.2 Arten von Fuzzer

Ein Fuzzer kann in unterschiedliche Arten kategorisiert werden.

1. Nach der Methode der Fuzz-Erzeugung
2. Nach Kenntnis der Struktur des Inputs
3. Nach Kenntnis der Programmstruktur (Source Code)

(Wikipedia 2020)

4.2.1 Fuzz-Erzeugung

Zusammengefasst von: (Crain and Pe 2020)

4.2.1.1 Mutation-Based

Bei der mutation-based Methode werden die gültigen Input Daten zufällig verändert, um fehlerhafte Inputs zu erstellen.

4.2.1.2 Generation-Based

Bei der generation-based Methode werden die Input Daten grundlegend neu hergestellt. Im Gegensatz zur mutation-based Methode werden diese Inputs ohne Beispieldaten generiert.

4.2.2 Kenntnis der Eingabe-Struktur

Zusammengefasst von: (Hillman and F-Secure 2020)

4.2.2.1 Dumb Fuzzer

Hier werden völlig zufällige Eingaben generiert ohne Rücksicht auf die Daten, welche vom System akzeptiert werden. Es kann ohne grossen Aufwand gestartet werden.

4.2.2.2 Smart Fuzzer (a.k.a. Grammar-Based Fuzzer)

Smart-Fuzzer können mit Kenntnis der Eingabe-Struktur, des Formats oder mittels Regeln für ein Dateiformat intelligenter gefuzzt werden. So können meist gültige Eingaben konstruiert werden die dann tiefer in das Programm oder System fuzzen. Es ist aufwändiger als der Dumb Fuzzer und es muss ein Gleichgewicht zwischen diesen zwei Arten gefunden werden.

4.2.3 Kenntnis Programmstruktur

Zusammengefasst von: (Crain and Pe 2020)

4.2.3.1 Blackbox Fuzzing

Mit Hilfe von Blackbox Fuzzing ist man auf die Generierung von Inputdaten und die Untersuchung des Verhaltens des Systems eingeschränkt. Bei dieser Methode ist der Quellcode nicht bekannt und das Programm wird als Blackbox betrachtet.

4.2.3.2 Whitebox Fuzzing

Bei der Whitebox Fuzzing Methode wird der Quellcode des Programms getestet. Es können Schwachstellen und Fehler entdeckt werden. Mit Whitebox Fuzzing können Bugs, welche sich sehr tief im Programm befinden, aufgedeckt werden. Hier kann man zusätzlich die Code Coverage herausfinden.

4.2.3.3 Greybox Fuzzing

Mit Greybox Fuzzing können Whitebox- und Blackbox-Fuzzing kombiniert werden. Man wird beim Fuzzing über die Code Coverage benachrichtigt.

4.3 Wie werden Fuzz-Tests durchgeführt?

Der Fuzzing-Prozess umfasst vier Hauptphasen:

1. **Start**:
    1. Identifikation des Zielsystems
    2. Identifikation von Inputs
2. **Testcase Generation**: Generierung von Fuzz-Daten
3. **Programm Execution**: Ausführung des Testes mit Fuzz-Daten
    1. Analyse des Systemverhalten
4. **Bugs**: Protokollierung von Fehler

(Guru99 2020; Li, Zhao, and Zhang 2018)

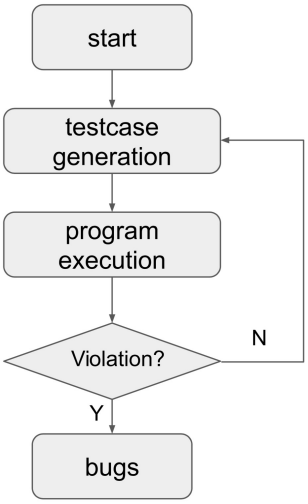


Abbildung 1:Fuzzing Cycle (Li, Zhao, and Zhang 2018)

4.4 AFL (American Fuzzy Lop)

Zusammengefasst von: (AFL @ github.com 2020; Dechand 2020)

AFL ist einer der bekanntesten Fuzzer, der generische Algorithmen und Kompilierzeit Instrumentierung, die für jede einzelne Eingabe detaillierte Informationen über den erreichten Codeabschnitt gibt, einsetzt. Diese Informationen werden von der Fuzzing Engine benötigt, um die erreichte Coverage zu bestimmen und um zu wissen was in der Anwendung, während dem Fuzzing, geschieht. Mit dem AFL können automatisch saubere und interessante Testfälle entdeckt und neue interne Zustände der betroffenen Binärdatei gefunden werden. Dadurch wird die funktionale Abdeckung für den Fuzzy-Code erheblich verbessert. AFL-Fuzzing hat einen geringen Performance-Overhead da es eine Reihe von Fuzzing-Techniken zur Aufwandsminimierung verwendet. AFL erfordert im Wesentlichen keine zusätzlichen Konfigurationen.

Standardmässig ist die AFL-Mutationsengine für kompakte Datenformate, wie zum Beispiel Bilder, Multimedia, komprimierte Daten, Syntax regulärer Ausdrücke oder Shell-Skripte, optimiert. Das heisst für Daten, welche meistens auch ohne eine spezifische Grammatik gültig sind. Etwas weniger geeignet ist es für Sprachen mit komplexerer Grammatik, wie JavaScript, SQL oder XML. Um das zu verbessern, bietet AFL die Möglichkeit, den Fuzzing-Prozess mit einem optionalen Dictionary mit Sprachschlüsselwörtern zu erweitern. Dieses wird dann verwendet, um die nötige Grammatik zu beachten oder zu rekonstruieren.

4.4.1 AFL Status Screen

Folgende Abbildung wird angezeigt, wenn der AFL Fuzzer in Betrieb ist.

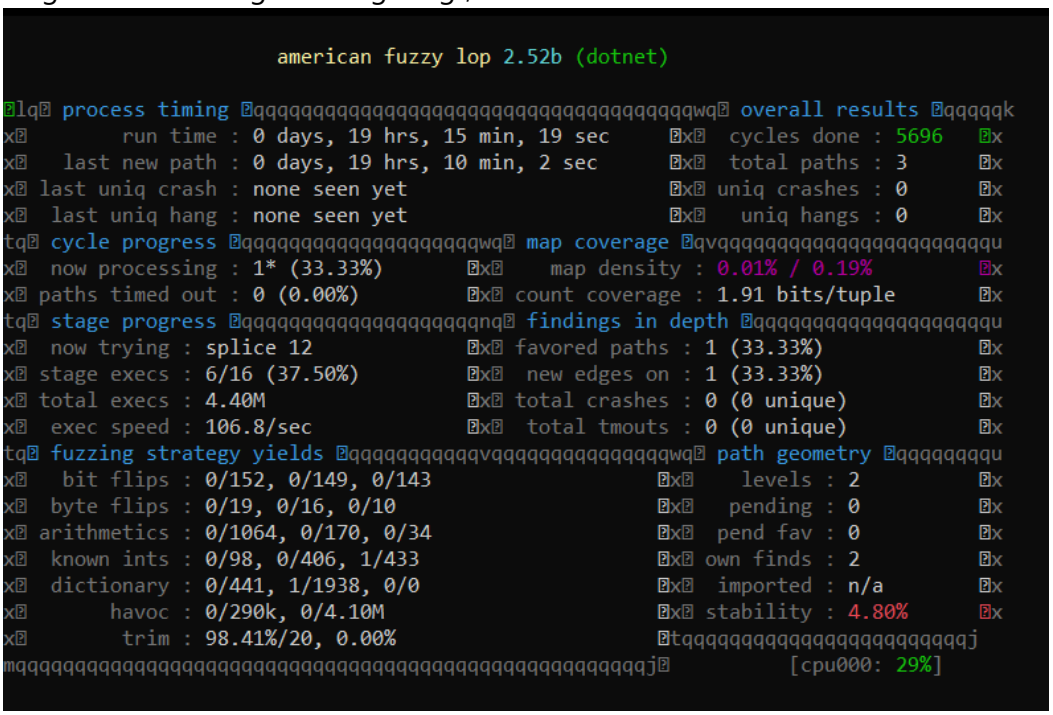


Abbildung 2: AFL Screen

Damit der Status Screen später bei der Anwendung der Fuzzing-Tools interpretiert werden kann, werden die einzelnen Bereiche grob beschrieben:

- **process timing**: Hier wird angezeigt, wie lange der Fuzzer schon läuft und wie viel Zeit seit seinem letzten Fund vergangen ist. Die Zeit des letzten Fundes wird in drei verschiedenen Kategorien aufgeteilt: Path (Pfad), Crash (Absturz) und Hang (Aufhängen).
- **overall results**: Hier wird die Gesamtstatistik angezeigt. Das erste Feld «cycles done» gibt die Anzahl durchgelaufener Testfälle an. Die restlichen Felder zeigen die Anzahl gefundene Pfade, aufgetretene Abstürze und Aufhängen.
- **cycle progress**: Dieser Bereich zeigt den Fortschritt des aktuellen Queue Zyklus an. Zuerst wird die ID des Testfalls angezeigt, an dem gerade gearbeitet wird. Danach wird die Anzahl der aufgrund eines Timeouts verworfenen Testfälle angezeigt.
- **map coverage**: In diesem Abschnitt werden Informationen über die Abdeckung geliefert, welche sich in der Zielbinärdatei befindet. Das erste Feld gibt das Verhältnis zwischen dem Verzweigungs-Tupel und Bitmap an. Die andere Zeile befasst sich mit der Variabilität der Tupel-Trefferzahlen in der Binärdatei an.
- **stage progress**: Dieser Teil gibt eine Übersicht, was er gerade tut. Das erste Feld informiert über die aktuell angewendeten Fuzzing-Strategien, welche eine der folgenden Arten sein können: calibration, trim, bitflip, arith, interest, extras, havoc, splice, sync. Die restlichen Felder geben die Anzahl der Ausführungen und Fortschritte der aktuellen Phase an.
- **finding in depth**: In diesem Teil werden mehrere Metriken angezeigt. Als erstes wird die Anzahl Pfade angezeigt, die der Fuzzer favorisiert und welches die Kanten-Coverage erhöht hat. Der Total Crash differenziert sich vom Total Timeout, weil dieser auch die Timeout Pfade beinhaltet.
- **fuzzing strategy yields**: Dieser Abschnitt gibt die Anzahl Pfade an, die im Verhältnis zu den versuchten Ausführungen pro Fuzzing-Strategie stehen, welche im stage process Bereich erwähnt wurde.
- **path geometry**: Das erste Feld dieses Bereichs verfolgt die Path-Tiefe, welche durch den Fuzzing-Prozess entdeckt wurde. Als Level 1 wird der initiale Testfall angegeben. Die aus dem initialen Testfall abgeleiteten Testfälle werden als Level 2 bezeichnet und so weiter. Das nächste Feld zeigt die Anzahl der bereits generierten, aber noch nicht durchlaufenen Testfälle an. Dasselbe gilt für die favorisierten Testfälle. Das Feld «own findes» gibt die Anzahl Pfade an, zu welchen noch keine Testfälle existieren. Die Stabilität gibt die Wahrscheinlichkeit der Reproduzierbarkeit des Systemverhaltens bei gleichem Testfall an.

4.4.2 Output Interpretation

Vor dem Starten des AFL Fuzzings wird ein Ordner Findings mit den drei Unterverzeichnissen queue, crashes und hangs erstellt, welche dann in Echtzeit aktualisiert werden. Die drei Unterordner werden bei der späteren Analyse des Outputs genauer betrachtet:

- **queue/:** Alle Testfälle, welche vom Fuzzer erstellt worden sind und plus alle vom Benutzer angegebenen Testfälle, werden hier abgelegt.
- **crashes/:** Alle Testfälle, welche vom Fuzzer erstellt worden sind und das Programm zum Absturz gebracht haben, werden hier abgelegt.
- **hangs/:** Alle Testfälle, welche vom Fuzzer erstellt worden sind und das Programm zu Timeout gebracht haben, werden hier abgelegt.

5 XML

XML ist eine Auszeichnungssprache und wurde entwickelt, um mit klaren Regeln Daten abzugrenzen, Verbindung von eindeutigen und einzelnen Teilen zu erstellen und zum Aufbau einer Hierarchie. Früher war es nicht immer einfach, roher Text in einzelne Teile abzugrenzen. Wegen dieses schlechten Zustands entstanden viele Datenspeicherungsformate darunter auch XML.

XML ist in diversen Bereichen einsetzbar. Es dient zur Beschreibung, Speicherung und zum Austausch von Daten. Sowohl für Mensch als auch für Maschine ist es leicht zu verstehen. Das XML-Format wird etwas langsamer verarbeitet als Binärformate, da eine XML-Struktur mehr Speicherplatz benötigt als es eigentlich nötig ist.

Zusammengefasst von: (Hery-Moßmann 2018; Ray, McIntoshGmbH, and Data2type GmbH 2020)

5.1 Basics

- XML ist ein Standard für den Austausch von strukturierten Daten im Textformat.
- Das Format bzw. die Datenstrukturdefinition eines XML-Files wird entweder durch DTD (Document Type Definition) oder XSD (XML-Schema) definiert.
- XML File ist:
    - "Well-formed", wenn es der XML-Syntax entspricht.
    - "Valid", wenn das Format entweder der DTD oder XSD entspricht (die XML-Datei enthält einen Verweis auf die entsprechende DTD/XSD-Datei).
- Anfragen (XML-File) werden mittels einem XML-Parser geparst & müssen anhand DTD-/XSD-File validiert werden.
- Diese DTD-/XSD-Files können lokal wie auch extern abgelegt sein.
- Im Request-XML-File kann die URL des DTD-/XSD-File angegeben werden.

Zusammengefasst von: (Bütler and Compass Security n.d.; Herzog 2010)

5.2 Parsing von XML (XML-Parser)

XML-Parser ist ein Programm, welches die XML-Dokumente liest und diese danach zur weiteren Verarbeitung zur Verfügung stellt. Das sind die folgenden Aufgaben eines XML Parsers:

- Mit einem XML-Parser können Zeichenströme eingelesen werden. Es wird zwischen Markup und Textdaten unterschieden.
- Entities können durch ihre Werte ersetzt werden.
- Aus unterschiedlichen Quellen kann ein vollständiges und logisches Dokument entwickelt werden.
- Strukturierte Informationen und Daten werden an das Programm übermittelt.

Zusammengefasst von:(Ray, McIntoshGmbH, and Data2type GmbH 2020)

5.3 Security Standards

Es wurden diverse Sicherheitsstandards für den Schutz von XML-Dokumenten entwickelt.
Die grundlegenden XML-Sicherheitsstandards sind:

- XML Signature: definiert eine XML-Syntax für digitale Signaturen
- XML Encryption: wird für die Ver- und Entschlüsselung von XML-Dokumenten verwendet
- XML Key Management (XKMS): wird verwendet um die Kommunikation zwischen Anwendungen mit Unterstützung der Public Key Infrastructure (PKI) zu sichern
- Security Assertion Markup Language (SAML): wird zum Austausch von Authentifizierungs- und Autorisierungsinformationen verwendet
- XML Access Control Markup Language (XACML): standardisiert die Darstellung und Verarbeitung von Autorisierungs-Policies

Diese XML-Sicherheitsstandards allein machen eine Anwendung aber noch nicht sicher.

Zusammengefasst von: (Bütler and Compass Security n.d.; Herzog 2010)

5.4 Angriffsvektoren

1. Netzwerk Service
2. XML-Generator
3. XML-Parser
4. Applikationscode

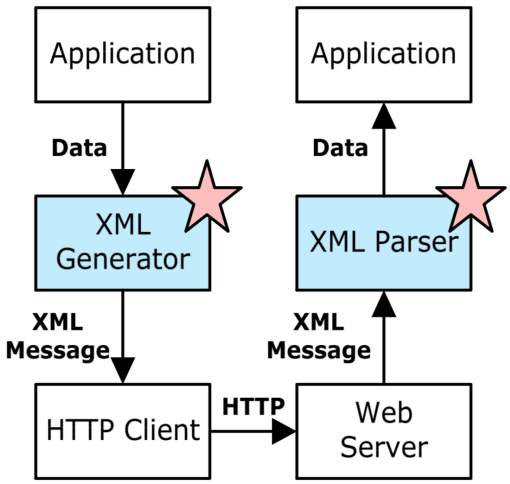


Abbildung 3: XML-Angriffsvektoren (OWASP, 2010)

Die im vorherigen Abschnitt "Security" genannten XML-Sicherheitsstandards bieten gegen XML-Generator und -Parser Angriffe nur einen geringen Schutz. Daher ist eine gezielte Untersuchung auf diese beiden Angriffsvektoren inklusive der Implementierung von Schutzmassnahmen notwendig.

Zusammengefasst von: (Bütler and Compass Security n.d.; Herzog 2010)

5.5 XML-Generator Attack (Fragment Injection)

Für die Integration des Backend wird häufig XML verwendet. XML-Generatoren sind für die Generierung der XML-Dokumenten zuständig, welche dann vom Backend System oder durch einen XML-Parser verarbeitet werden. Je nach Generator können Injektionen von XML-Fragmenten möglich sein. So hat der Angreifer die Möglichkeit, ein XML-Fragment mit gefälschten Daten anstelle eines Fragments, welches die originalen Daten des Kunden beinhaltet, zu injizieren.

Zusammengefasst von: (Bütler and Compass Security n.d.; Herzog 2010)

5.6 XML-Parser Attacks (XXE)

Die Grundlage für einen XML-Parser-basierten Angriff ergibt sich aus der Funktionalität eines XML-Parsers, die sich aus der Auslagerung des Marshalling-Problems durch die XML-Technologie ergibt.

Ein XML External Entity-Angriff ist eine Art von Angriff gegen eine Anwendung, die XML-Eingaben verarbeitet. XXE Attacken nutzen die Unterstützung von externen XML-Entitäten in XML-Eingaben aus, welche von einem unsicher konfigurierten XML-Parser verarbeitet werden. Dieser Angriff kann zur Offenlegung sensibler Daten, zum Durchqueren von Pfaden, zum Port-Scanning, zur Einbindung von Dateien, zur DNS-Auflösung oder zu vielen anderen Angriffen führen, wie zum Beispiel Denial-of-Service, SSRF oder sogar Remote-Codeausführung.

Das heisst, XXE Angriffe werden nicht auf die XML-Dateien durchgeführt, sondern sie greifen den Parser an, der letztlich die XML-Dateien verarbeitet. So können die Funktionen von XML für andere Zwecke ausgenützt werden.

Zusammengefasst von: (De.qaz.wiki 2020; OWASP 2020)

5.6.1 Wie XXE-Angriffe funktionieren

Die Struktur eines XML-Dokuments wird durch den XML 1.0-Standard festgelegt, der ein Konzept definiert, das Entität genannt wird. Eine Entität bezeichnet eine Speichereinheit irgendeiner Art. In XML-Dateien können solche Entitäten in der DOCTYPE-Deklaration definiert werden. Über einen angegebenen Systembezeichner "SYSTEM" können Entitäten auf lokale sowie auch auf externe Quellen zugreifen, was für XXE-Angriffe eine entscheidende Rolle spielt. Der XML-Prozessor, der für das Einlesen und Verarbeiten von XML-Dateien zuständig ist, nimmt den Systembezeichner für die Dereferenzierung der Entitäten als URI. Der XML-Prozessor ersetzt dann das Vorkommen der deklarierten Entitäten durch ihren Entitätswerte, welche er aber nicht validiert. Wenn der Systembezeichner nun verfälschte Daten referenziert kann der XML-Prozessor vertrauliche Informationen offenlegen, auf die die Anwendung normalerweise nicht zugreifen kann. Ähnliche Angriffsvektoren ergeben sich auch bei der Verwendung von externen DTD-Dateien, externe Stylesheets, usw.

Zusammengefasst von: (De.qaz.wiki 2020; OWASP 2020)

5.6.2 Angriffsarten

Zusammengefasst von: (Banach 2020; Späth et al. 2016)

Im Folgenden werden nur die Arten von Angriffen, die auf einen XML-Parser möglich sind, kurz beschrieben. Auf die einzelnen Angriffe wird nicht näher eingegangen, da diese Betrachtung im Rahmen der Arbeit nicht relevant ist, sondern nur auf die Schwachstellen, die ausgenutzt werden können. Für jede Angriffsart werden aber einige bekannte Angriffsvarianten aufgeführt.

'Alle XML-basierten Angriffe bauen auf dem Hinzufügen eines DOCTYPE in einem XML-Dokument auf. Um XML-Parser-Angriffe zu verhindern, ist die hilfreichste Methode, die Funktion für die Verwendung externer Entitäten zu deaktivieren.' (Securai.de 2020)

5.6.2.1 Resource Exhaustion Attacks

Diese Art von Angriffen stellt einen auf XML basierenden Denial-of-Service Angriff dar, welcher auf der Verwendung von XML-Entitäten beruht. Beim Verarbeiten dieser XML-Entitäten, können diese Attacken schnell die Server-Ressourcen erschöpfen, was schliesslich zu einem Serverabsturz führen kann.

Varianten: Recursive Entities, Billion Laugh, Quadratic Blowup, External Entities Expansion

5.6.2.2 File Inclusion Attacks

Bei diesen Angriffen versucht der Angreifer, eine Datei (z.B. DTD/XSD, SQL, etc.) aus der Ferne auf den Server zu laden und diese auszuführen. Diese Remote-Dateien werden in der Regel in Form einer HTTP- oder FTP-URI in einem Parameter an die Anwendung übergeben. Auch lokale Dateien können zur Ausführung einbezogen werden.

Varianten: DTD Retrieval, XSD Retrieval, XInclude,

5.6.2.3 Data Extraction Attacks

Ziel von Data Extraction Angriffen ist es, sensible Inhalte aus lokalen Dateien oder Netzwerkressourcen abzurufen. Dabei versucht der Angreifer Verweise auf bekannte, oder wahrscheinliche, Dateien im lokalen System herzustellen um diese dann zu exfiltrieren.

Varianten: External Entities Expansion, Parameter-Entities Expansion, Blind XXE, Local- & Global Connect Scan, DNS-Resolution

5.6.2.4 SSRF-Attacks (Server-Side Request Forgery)

SSRF-Angriffe, im Zusammenhang mit XML, werden dafür verwendet, HTTP Anfragen im Namen des XML-Parsers an andere Endpunkte innerhalb sowie ausserhalb des Netzwerks zu senden. Somit hat der Angreifer die Möglichkeit, indirekt einen Zugang zum internen Netzwerk aufzubauen, wo von er dann weitere Angriffe starten kann.

Varianten: External Entities Expansion, Parameter-Entities Expansion, XInclude

5.6.3 XXE Verhinderung

Zusammengefasst von: (Banach 2020; Bütler and Compass Security n.d.; Herzog 2010)

Da das XML-Dokument möglicherweise von einem nicht vertrauenswürdigen Client übermittelt wird, ist es in der Regel nicht möglich, manipulierte Daten innerhalb des Systemidentifikators in der DTD zu validieren oder zu umgehen. Angriffe auf externe XML-Entitäten beruhen auf der veralteten Unterstützung für Document Type Definitions. Das bedeutet, dass die Deaktivierung der DTD-Unterstützung der beste Weg zur Beseitigung von XXE-Schwachstellen ist. Wenn das nicht möglich ist, sollte man die Unterstützung für externe Entitäten deaktivieren und den XML-Prozessor so konfigurieren, dass er eine lokale statische DTD verwendet und alle deklarierten DTDs, die im XML-Dokument enthalten sind, nicht erlaubt werden.

Netzwerkseitig:

- System in der DMZ sollte keine DNS-Auflösungen machen können.
- System in der DMZ darf nicht nach Aussen kommunizieren können.

Softwareseitig:

- XML-Parser Hardening: XML-Parser soll die verletzbaren Funktionen nicht ausführen können.
    - Auflösung von externen Entitäten verbieten.
    - Schemafunktionen validieren & Eingabevalidierung
    - Auflösung von externen XML-Schema-Speicherorten vermeiden.
    - XML gegen lokale serverseitige XSDs und DTDs prüfen.
    - Security Manager verwenden, um Knoten und Entity-Erweiterungen zu begrenzen.
- XML Library aktualisieren bzw. auf dem neusten Stand halten.
- Second Line of Defense mittels Filtering via WAF
- XML-Decoder Library so benutzen, dass keine vom Client zur Verfügung gestellten Typen möglich sind.

6 Übersicht des Beispielprojekts ("Rodix")

Der folgende Abschnitt beschreibt den XML-Parser, den wir von der Firma nxt Engineering erhalten haben. Dieser XML-Parser wird als Beispielanwendung zur Untersuchung der Fuzzing-Möglichkeiten auf der .NET-Umgebung verwendet.

6.1 Solution 'Rodix'

Die Aufgabe dieser Solution ist es via Command Line ein XML-File mit zusätzlichem Parameter entgegen zu nehmen, validieren und schliesslich die Daten erfolgreich in der Datenbank zu speichern.

Die Solution Rodix besteht aus fünf Teilprojekten, davon sind zwei Testprojekte. (siehe Bild)

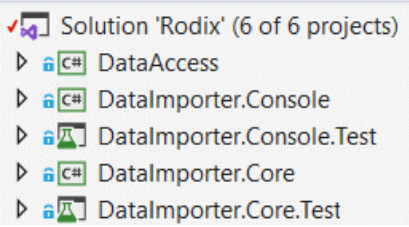


Abbildung 4: Übersicht Solution "Rodix"

- DataAcess: Das Projekt enthält (Entity Framework (EF) Core) Code für die tacslive Datenbank. Es enthält 104 Enities.
- DataImporter.Console: Dieses Projekt enthält Code zum Ausführen des DataImporter.Core Projektes via Command Line, welches die eigentliche XML-Parser Verarbeitung macht. Dieses Projekt enthält auch ein Docker File, welches vor dem Import des XMLs gestartet werden muss.
- DataImporter.Core: Das ist das Kernprojekt für die Solution 'Rodix' und ist hauptsächlich für die Verarbeitung und Import des XML-Files in die tacslive-Datenbank verantwortlich.
- Testprojekte: DataImporter.Console.Test und DataImporter.Core.Test sind Unit Test Projekte, welche Console und Core Projekte testet.

► Für die Einrichtung des XML-Parsers: siehe Kapitel "Installation & Konfiguration"

6.2 xUnit Test & Code Coverage

Die Solution enthält bereits zwei xUnit-Projekte, wobei diese eher als Systemtests gesehen werden können, da auch die externe Datenbank involviert wird. Diese zwei xUnit-Projekte haben den Source-Code mit über 90% Code Coverage abdeckt. Wie bereits erwähnt, sind in den DataImporter.Console.Test und DataImporter.Core.Test Projekten xUnit Test geschrieben. Mit dem Plugin dotCover, welches für Visual Studio und Jetbrains Rider verfügbar ist, konnten wir die Code Coverage analysieren und darstellen.

Für die Ausführung von xUnit Tests muss die DB erstellt und vorhanden sein und Docker muss aktiv sein. (siehe Allgemeine Einrichtung).
Wie man im folgenden Bild sehen kann, sind 173 xUnit Tests geschrieben worden und liefen erfolgreich durch.

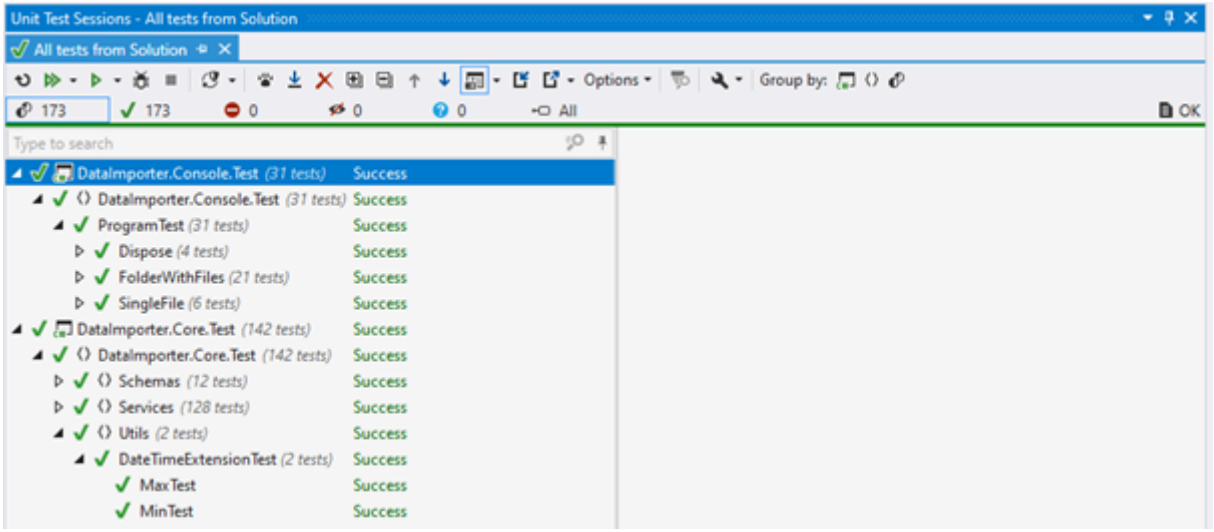


Abbildung 5: Anzahl Unit-Tests

In folgender Ansicht sieht man die Code Coverage von xUnit Tests, welche im Projekt bereits vorhanden waren. Die Code Coverage für die Solution 'Rodix' beträgt 92%.

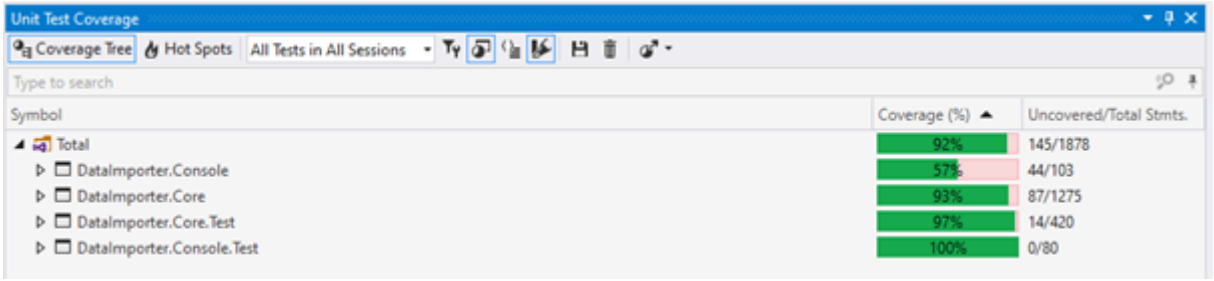


Abbildung 6: Unit-Test Coverage

7 Fuzzing-Tool Recherche & Evaluation

Die Fuzzing-Tools wurde aufgrund der folgenden Kriterien für die genauere Recherche ausgewählt:

- Fuzzing mit C# (.NET Umgebung)
- Unterstützung von XML Fuzzing
- Mutation-basiertes Fuzzing
- Coverage-basiertes Fuzzing
- Grammar-basiertes Fuzzing

Für die ausgewählten Tools wird eine kurze Beschreibung inklusive einer Auflistung der besonderen Eigenschaften gegeben. Darüber hinaus wird jedes Fuzzing-Tool für die spätere Verwendung in Bezug auf XML-Parser bewertet.

7.1 Peach

Zusammengefasst von: (Mozilla Wiki 2017; Peach.tech 2020)

7.1.1 Beschreibung

Peach ist ein Fuzzer, der sowohl Generation-basiertes als auch Mutation-basiertes Fuzzing ermöglicht. Peach kann zur Erzeugung von Fuzz-Dateien, zur Erzeugung von Fuzz-Netzwerkverkehr oder zum Fuzzing von Shared Libraries verwendet werden. Peach Fuzzer erlaubt es dem Benutzer, die Fuzzing-Einstellungen entsprechend der zu testenden Anwendung zu wählen, was zu effizienteren Tests und besseren Ergebnissen führt. Drei Komponenten machen das erfolgreiche Fuzzing von Peach Fuzzer aus: Modeling, Mutation und Monitoring. (Für mehr Details: https://www.peach.tech/wp-content/uploads/Peach-Fuzzer-Platform-Whitepaper.pdf)
Peach erfordert die Erstellung von benutzerdefinierten "Peach Pit"-Dateien, welche die Struktur, Typinformationen und Beziehungen in den zu fuzzenden Daten definieren. Peach fuzzt die in der Pit-Datei modellierten Elemente durch die Generierung von Testfällen. Die Fuzzing-Engine von Peach verwendet dabei eine umfangreiche Sammlung von Mutationsalgorithmen und Strategien für die Generierung dieser Testfälle.
Peach Fuzzer unterstützt die Verwendung von mehreren Monitoren zur gleichzeitigen Erkennung verschiedener fehlererzeugender Bedingungen.

7.1.2 Features

- Bietet eine erweiterte Bibliothek von Peach Pits
- Enthält eine grafische Benutzeroberfläche (GUI)
- Bietet erweiterte Überwachungsfunktionen (Monitoring)
- Ist plattformübergreifend (Windows, Linux und OSX)

7.1.3 Evaluation

Peach Fuzzer unterstützt sowohl das mutationsbasierte als auch das abdeckungsbasierte Fuzzing, was für unsere Arbeit von Interesse sein könnte. Aufgrund der benötigten Peach-Pit-Dateien kann man daraus folgern, dass Peach auch grammatikbasiertes Fuzzing ermöglicht. Gemäss Recherchen kann Peach auch XML fuzzen, aber es ist noch unklar, ob ein in C# entwickelter XML-Parser getestet werden kann. Auf der Website von Peach Tech steht, dass sie für Kompatibilitätsanfragen kontaktiert werden sollen. Darüber hinaus sind keine weiteren Details verfügbar. Da Peach jedoch einer der interessantesten Fuzzer für unsere Arbeit ist, wird ein Versuch unternommen, mit Peach Kontakt aufzunehmen.

► Im *Anhang:* befindet die Mailanfrage betreffend Kompatibilität mit unseren XML-Parser (gemäss Peach Tech Webseite)

7.2 Fuzzware

Zusammengefasst von: (Fuzzware n.d.)

7.2.1 Tool-Beschreibung

Fuzzware ist ein generisches Fuzzing-Framework. Es wurde für die Automatisierung der Erstellung und Verarbeitung von Fuzz-Daten, sprich Testfällen, für die zu testende Software entwickelt. Die Testfälle können Netzwerkpakete, Dateien oder auch Aufrufe an Schnittstellen sein.
Mit diesem Tool ist es daher möglich, XML Fuzz-Daten zu generieren. Fuzzware ist jedoch nur ein Container für Daten. Fuzzware verwendet eine XML-Schema-Definitionsdatei (XSD) für die Generierung der Testfälle. Mit einer XML-Datei, die Daten enthält, und einer XSD-Datei, welche die Typen und Strukturen dieser Daten beschreibt, ist Fuzzware in der Lage, typ- und strukturabhängige Fuzz-Daten für die zu verarbeitende Applikation zu erstellen.

7.2.2 Features

- Mit einer XSD-Datei kann die Struktur einer gültigen XML-Datei beibehalten werden.
- Fuzzware trennt die Daten (XML-Datei) von ihrer Definition (XSD-Datei).
- Unter .NET kann Fuzzware erweitert werden, um eine vollständig benutzerdefinierte Eingabe zu ermöglichen.
- Verwendet insgesamt 21 verschiedene Arten von Fuzzing-Techniken für Datentypen zur Fuzz-Generierung.
- Die Fuzzing-Techniken können für bestimmte Teile des Datenformats individualisiert werden.

7.2.3 Evaluation

Fuzzware ist ein einfaches Tool, das die Erstellung und Bearbeitung von XML-Fuzz-Dateien oder Testfällen ermöglicht. Da Fuzzware zur Erstellung der Testfälle eine XML-Datei mit Daten und eine XSD-Datei benötigt, ist es in der Lage, typ- und strukturabhängige Testfälle für die zu fuzzende Applikation zu erstellen, was ein grammatikbasiertes Fuzzing widerspiegelt. Da für das XML-Parser-Fuzzing sowohl eine gültige XML-Datei als auch eine XSD-Datei mitgeliefert wurde, ist das Tool für diese Arbeit durchaus interessant.

7.3 SharpFuzz

Zusammengefasst von: (Mijailovic 2019b, 2019a)

7.3.1 Beschreibung

SharpFuzz ist ein Tool, welches abdeckungsgesteuertes (Path-Coverage) Fuzzing ermöglicht. Es basiert und bringt die Leistung von AFL-Fuzzing (American Fuzzy Lop) auf die .NET-Plattform.
Sharpfuzz wurde für .NET Projekte entwickelt um «Unhandled Exception», «Uncatchable Exceptions» und Timeouts zu finden. Die Exceptions, welche geworfen und nicht nicht abgefangen werden, werden im AFL als Crash rapportiert. Mit diesem Tool können Libraries oder Programme in C# für Fuzzing genutzt werden.

7.3.2 Features

- Liefert einzigartige Fehlerfälle, die immer reproduzierbar sind
- Fuzzing für .NET Projekte mit integriertem AFL
- Berichtet über «Unhandled Exceptions», «Uncatchable Exceptions» und Timeouts

7.3.3 Evaluation

Sharpfuzz ist speziell für .NET-Plattform Anwender erstellt worden. Man kann es mit C# Code erweitern um das gezielte Programm oder System zu fuzzen. Deshalb ist dieses Tool sehr gut geeignet für den XML-Parser, welcher ebenfalls in C# Code geschrieben worden ist. Zusätzlich kann man ein AFL Dictionary File wie z.B. xml.dict mitgeben um gezielter zu testen. Wie im Kapitel AFL beschrieben, können auch XML-Files als Testcase mitgegeben werden, was auch für den XML-Parser Fuzzing spricht.

7.4 Fuzzlyn

Zusammengefasst von:(Jakob Botsch 2020)

7.4.1 Beschreibung

Fuzzlyn ist ein Fuzzer, welcher zur Entdeckung von Programmierfehlern und Sicherheitslücken in der .NET-Plattform entwickelt wurde. Fuzzlyn verwendet Roslyn, eine .NET Compiler Plattform von Microsoft, um zufällige Programme zu erzeugen. Die erzeugten Programme werden dann auf einem .NET-Kern ausgeführt, um zu überprüfen, ob sie die gleichen Resultate liefern, wenn sie im Debug- wie auch Release-Modus kompiliert werden. Das heisst, mit Fuzzlyn können .NET Compiler getestet werden. Fuzzlyn erzeugt standardmässig keine Ausgaben. Wenn jedoch ein Programm mit ungewöhnlichem Verhalten gefunden wird, fügt Fuzzlyn seinen Seed und Informationen über seine Ausführung an eine Datei an.

7.4.2 Features

- Fuzzing von .NET Compiler
- Generiert Programme mit einer Teilmenge von C#

7.4.3 Evaluation

Fuzzlyn ist ein Fuzzer, der für das Fuzzing in der .NET Umgebung entwickelt wurde. Das Tool wurde komplett in C# geschrieben. Jedoch kann Fuzzlyn nur Compiler auf Fehler überprüfen, was für das Fuzzing vom XML-Parser nicht relevant ist. Aus diesem Grund wird auf Fuzzlyn in dieser Arbeit nicht näher eingegangen.

7.5 OneFuzz

Zusammengefasst von: (Microsoft 2020)

7.5.1 Beschreibung

Das Projekt OneFuzz von Microsoft ist ein Open-Source-Software Projekt, welches kontinuierliches developer-driven Fuzzing von Software ermöglicht. Es handelt sich dabei um ein sogenanntes Fuzz Testing Framework für Azure. OneFuzz steht auf GitHub (https://github.com/microsoft/onefuzz) unter einer MIT-Lizenz (freizügige Open-Source-Lizenz) zur freien Nutzung zur Verfügung. Es wurde am 18.09.2020 veröffentlicht.

7.5.2 Features

- Zusammenstellbare Fuzzing-Workflows
- Eingebautes Ensemble-Fuzzing (verwendet mehrere unterschiedliche Fuzzer, die interessante Eingaben zwischen den Fuzzer-Technologien austauschen, um eine bessere Leistung zu erzielen)
- Liefert einzigartige Fehlerfälle, die immer reproduzierbar sind
- Live-Debugging von gefundenen Abstürzen auf Abruf
- Transparentes Design, welches eine Einsicht in jede Phase ermöglicht
- Fuzzing unter Windows und Linux
- Callback-Benachrichtigungen bei Absturzberichten

7.5.3 Templates

Es gibt bei OneFuzz verschiedene vorgefertigte Jobs, sogenannte Templates, die auf bekannten Fuzzern basieren. Darunter AFL, OSSFuzz, LibFuzzer und Radamsa.

OSSFuzz ist ein Fuzzing-Tool für fortlaufendes Fuzzing für Open Source Software und wurde von Google entwickelt.

Radamsa ist ein Test Fall Generator für Robustheit Tests a.k.a. ein Fuzzer. Er wird typischerweise benutzt, um zu testen, wie gut ein Programm ungültigen Inputs widerstehen kann.

Zusammengefasst von: (oss-fuzz @ github.com 2020; radamsa @ gitlab.com 2020)

7.5.4 Evaluation

OneFuzz hat einige Features, die für uns interessant sind. So kann OneFuzz Grammatik-basiertes Fuzzing durchführen, was ein Hauptkriterium ist für unsere Tool Auswahl. Um einen XML-Parser mit einer vorhandenen XSD zu fuzzen, ist das grammar-based Fuzzing sehr wichtig, da man sonst die XSD nicht verwenden kann. Ebenfalls unterstützt das Tool C# und ist deswegen auch gut geeignet für den XML-Parser, welcher in C# geschrieben ist.
Des Weiteren ist die Evaluation von OneFuzz Teil unserer Aufgabenstellung. Deshalb nehmen wir OneFuzz zu unseren Tools, die wir genauer testen wollen, und werden genau evaluieren, ob OneFuzz für unsere Zwecke geeignet ist.

7.6 LibFuzzer

Zusammengefasst von: (LibFuzzer @ github.com 2020; LLVM 2020)

7.6.1 Beschreibung

LibFuzzer ist ein Coverage-basiertes Fuzzing-Tool. Ähnlich wie andere Fuzzer wird auch LibFuzzer mit der zu fuzzenden Anwendung verknüpft und über einen definierten Fuzzing-Einstiegspunkt mit Fuzz-Daten versorgt. Der Fuzzer verfolgt dann, welche Bereiche des Codes erreicht werden und generiert entsprechend Mutationen auf die anfängliche Eingabe, um die Codeabdeckung zu maximieren.

7.6.2 Features

- Unterstützt benutzerdefinierte Dictionaries mit Schlüsselwörtern der Input-Sprache.
- Mutationen basierend auf zuvor erhaltenen Rückgabeinformationen. (Verlangsamt das Fuzzing, verbessert aber die Ergebnisse)
- Erstellt einen speziellen "Fuzzing-freundlichen" Build, bei dem bestimmte Funktionen, die das Fuzzing erschweren, deaktiviert sind.
- Kann zusammen mit AFL auf demselben Testobjekt verwendet werden.
- Erlaubt die Verwendung von benutzerdefinierten Mutatoren.
- Versucht bei der Ausführung jeder Mutation, Memory Leaks zu erkennen.

7.6.3 Evaluation

LibFuzzer kann zusammen mit einem AFL Fuzzer verwendet werden. Das bedeutet, dass dieser Fuzzer Greybox-Fuzzing ermöglicht, was für das Fuzzing vom XML-Parser interessant ist. LibFuzzer ist ein Fuzzer, der die Grammatik der angegebenen Seed-Datei kennt und die Fuzz-Daten entsprechend generiert. Darüber hinaus kann die Mutation durch ein benutzerdefiniertes Dictionary unterstützt werden, was das grammatikbasierte Fuzzing verbessern kann. LibFuzzer ist ein Tool für das Fuzzing von C und C++ Code. Da aber unsere Arbeit auf die .NET Umgebung bzw. C# basiert, wird dieses Tool nicht genauer betrachtet.

7.7 Übersicht der recherchierten Fuzzing-Tools

Die folgende Tabelle gibt einen Überblick über die recherchierten Fuzzing-Tools mit den für uns relevanten Fähigkeiten.

	Grammar-based	Coverage-based	Mutation-based	Kompatibel mit .NET (C#)	XML (-Parser) Fuzzing
Peach	x	x	x	?	x
Fuzzware	**x**	**-**	**x**	**-**	**x**
SharpFuzz	**(x)**	**x**	**x**	**x**	**x**
Fuzzlyn	-	-	-	x	-
OneFuzz	**(x)**	**x**	**x**	**X**	**x**
LibFuzzer	x	x	x	-	x

Tabelle 1: Fuzzing-Tool Überblick

- ► Die mit "x" gekennzeichneten Felder werden unterstützt.
- ► Die mit "(x)" gekennzeichneten Felder werden in einer limitierten Variante unterstützt. (Für genauere Informationen: siehe das entsprechende Kapitel zum Tool-Recherche)
- ► Die mit "-" gekennzeichneten Felder sind für das jeweilige Tool nicht verfügbar. (Für genauere Informationen: siehe das entsprechende Kapitel zum Tool-Recherche)
- ► Die mit "?" gekennzeichneten Felder nicht ermittelt werden. (Für genauere Informationen: siehe das entsprechende Kapitel zum Tool-Recherche)

In dieser Tabelle ist zu erkennen, dass nur drei der untersuchten Tools, SharpFuzz, Fuzzlyn und OneFuzz, für das Fuzzing mit .NET-Anwendungen entwickelt wurden, wobei Fuzzlyn keine Anwendungen, sondern nur Compiler testen kann. Peach ist einer der interessantesten Fuzzer für unsere Arbeit. Da wir jedoch noch keine Bestätigung von Peach erhalten haben (für mehr Detail, siehe Tool-Evaluation), ob dieses Tool in der Lage ist, eine C#-Anwendung zu testen, wird es nicht weiter betrachtet, bis wir eine Rückmeldung bekommen. Um ein drittes Tool für Fuzzing auf dem Beispielprojekt zu testen, wurde Fuzzware ausgewählt. Dieses Tool wurde nicht für eine bestimmte Programmiersprache entwickelt und ist daher "umgebungsunabhängig".

Teil III

Anwendung der Fuzzing-Tools

8 Fuzzing

8.1 Tool Entscheidung

Das folgende Kapitel beschreibt die Umsetzung des Fuzzings mit den von uns ausgewählten Fuzzing-Tools. Die Auswahl der Tools erfolgte auf Basis der Evaluationen, die sich aus der Recherche ergaben.

- ► Evaluationen der Tools (inkl. ein Gesamtüberblick der untersuchten Tools) sind im Kapitel "Analyse" > "Fuzzing-Tool Recherche" beschrieben.
- ► Setup (inkl. Vorgehensweise) für das Aufsetzen der Fuzzing-Tools ist im Kapitel "Installation & Konfiguration" beschrieben.

8.2 Übersicht der Fuzzing-Ergebnisse

Die folgende Tabelle gibt einen Überblick über die beim Fuzzing erzielten Ergebnisse.

Tool	Crash	Code Coverage	Path Coverage
Fuzzware	keine	42%	-
SharpFuzz	68	-	39
OneFuzz	?	?	?

Tabelle 2: Übersicht der Fuzzing-Ergebnisse

- ► Die mit "-" gekennzeichneten Felder sind für das jeweilige Tool nicht verfügbar.
- ► Die mit "?" gekennzeichneten Felder konnten nicht ermittelt werden. (Für genauere Informationen: siehe das entsprechende Kapitel zum Fuzzing)

8.3 Fuzzing mit Fuzzware

8.3.1 Problem

Ein Problem ist, dass man nicht alle Datentypen, die in einem XML-File vorkommen, fuzzen kann. Fuzzware ist nicht in der Lage, die Datentypen 'Date' sowie 'DateTime' zu fuzzen. Diesen Umstand konnten wir nicht umgehen, was bedeutet, dass diese Felder nicht mittels Fuzzware getestet werden können. Die restlichen Felder konnten ohne Probleme gefuzzt werden.

Code-Coverage: Da aber Fuzzware eine reine Mutations-Engine ist, werden die Fuzz-Daten nur anhand der gegebenen Grammatik (XSD) generiert. Um mittels Fuzzing eine hohe Coverage, sei es Code- oder Pfad-Coverage, zu erreichen, müssen die Fuzz-Daten nicht nur grammatikbasiert, sondern auch abdeckungsbasiert erzeugt werden. Die Mutations-Engine muss also in der Lage sein, nach jeder Mutation zu überprüfen, wie weit die Abdeckung einer Fuzz-Datei gekommen ist, um entsprechend die nächsten Fuzz-Daten zu generieren. Daher konnten die mit Fuzzware generierten Fuzz-Daten nur ohne Rücksicht auf die Coverage erstellen werden, was beim Fuzzen zu einer tiefen Code-Coverage führte.

8.3.2 Resultate

Fuzz-Generation:

Da Fuzzware nur ein Tool für die Generierung von Fuzz-Daten ist, musste das Fuzzing hier in zwei unabhängigen Schritten gemacht werden. In einem ersten Schritt wurde mittels Fuzzware, anhand Daten_Tacs.xml-File, ca. 93'000 XML-Files generiert. Des Weiteren wurden auch aus einem XML-File, welches sich in den Testprojekten befand, zusätzlich ca. 3'200 XML-Files generiert.

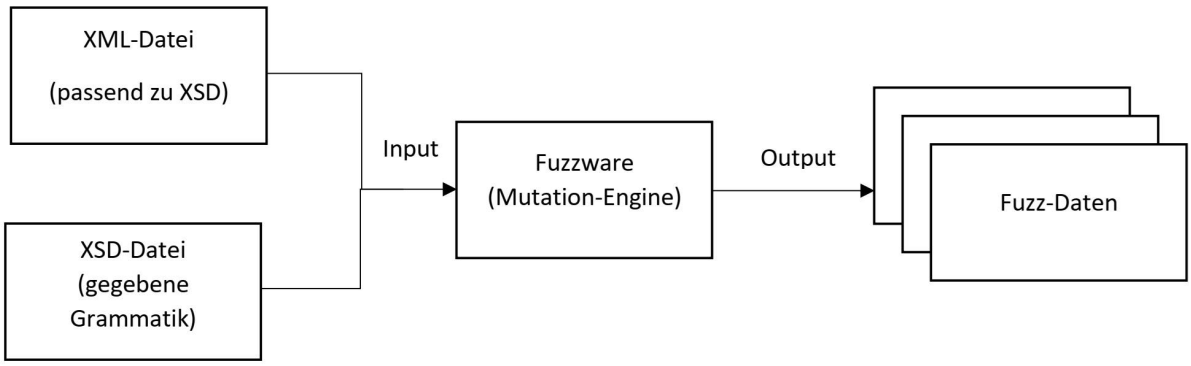


Abbildung 7: Fuzzware Pipeline

Manuelles Fuzzing:

Im zweiten Schritt wurden die generierten Files manuell am XML-Parser getestet. Um die generierten Files einfacher durch den XML-Parser laufen zu lassen und über Fehler informiert zu werden, wurde ein Python-Skript geschrieben, welches auch die Outputs der Parser-Verarbeitung in einem separaten Text-File speichert. So konnten die generierten XML-Files am XML-Parser getestet und die Resultate protokolliert werden.

Auswertung:
Abschliessend wurden die aus dem manuellen Fuzzing entnommenen Outputs ausgewertet. Dabei konnten nur einige Fehlermeldungen festgestellt werden, die vom Entity Framework geworfen wurden. Dies waren jedoch keine Fehler oder Abstürze, sondern behandelte Exceptions.

Code-Coverage Analyse:
Um Informationen über die mit dem manuellen Fuzzing erreichte Code Coverage zu erhalten, wurde zusätzlich ein Systemtest geschrieben. Dazu musste ein neuer Konstruktor für die Klasse "ImporterConsole" geschrieben werden, der die generierten Files im Rahmen eines Systemtests durchläuft, um die erreichte Coverage zu ermitteln. Die erreichte Code-Coverage konnte dann mit einem zusätzlichen Plug-In (dotCover) abgelesen werden.

Abbildung 8: Code Coverage durch manuelles Fuzzing

Wie es in der oberen Abbildung ersichtlich ist, wurde durch das manuelle Fuzzing ein Code-Coverage von 42% erreicht.

- Im *Anhang*: befindet sich das Python Skript.
- Im *Anhang:* befindet sich der Systemtest und der angepasste ImporterConsole-Konstruktor.

8.3.3 Erfahrung

Fuzzware ist ein sehr gutes Tool für die Generierung von XML-Dateien aus einer Quelldatei (XML) und einer XSD. Es gibt viele verschiedene Möglichkeiten, die Ausgabe zu beeinflussen. So kann man z. B. für die Zahlen den Bereich angeben, in dem sie gefuzzt werden sollen. Für die Strings kann man unterschiedliche Längen für die gefuzzten Strings angeben. Ausserdem gibt es verschiedene Strategien, wie das Ersetzen, Einfügen oder Löschen eines Zeichens.
Das Tool ist einfach zu bedienen und intuitiv gestaltet. Es ist zwar schon etwas älter und basiert daher auf .NET Framework, funktioniert aber einwandfrei. Der einzige Nachteil ist, dass Fuzzware ein eigenständiges Tool ("Stand-Alone Tool") ist und nur grammatikbasierte, nicht aber abdeckungsbasierte Daten erzeugen kann, was für das Fuzzing von einem XML-Parser nicht ausreicht.

8.4 Fuzzing mit SharpFuzz

8.4.1 Problem

Während der Evaluation und Verwendung von Sharpfuzz traten folgende Probleme auf:

- AFL-Engine konnte nicht gestartet, weil der erste Durchlauf einen Absturz (Crash) meldete.
- Das Debuggen des Sharpfuzz-Codes konnte nach dem Instrumentieren der dll-Dateien nicht immer durchgeführt werden, da es eine System.NullReferenceException warf.

Grammar-based Fuzzing: Sharpfuzz unterstützt kein Grammar-Based Fuzzing. Um jedoch XML-Files zu generieren, mit welchen dann der Parser getestet werden kann, muss das XSD (XML-Schema) beachtet werden. Das bedeutet in diesem Fall, dass AFL dieses Schema kennen muss, um automatisch akzeptierte Test-Dateien generieren zu können. Aber es gibt keine Möglichkeit ein XSD File mitzugeben für ein optimales Grammar-Based Fuzzing. Es kann jedoch ein Dictionary File mitgegeben werden. So lässt sich Grammar-Based in limitierter Form emulieren.

Erweiterung mit LibFuzzer: Gemäss den Angaben auf dem GitHub-Repository von SharpFuzz ist es möglich, den LibFuzzer, der eigentlich für das Fuzzing von C und C++ Code bestimmt ist, mit SharpFuzz zu verwenden. Diese Erweiterung wird aber zurzeit nur unter Linux unterstützt. Daher wurde eine Linux VM (Ubuntu 18.04), eine der sicher bestätigten Linux-Distributionen für diese Funktionalität, mit den nötigen Installationen und Konfigurationen aufgesetzt. In einem ersten Schritt musste dann der Fuzzer-Code von SharpFuzz umgeschrieben und neu instrumentiert werden. In einem zweiten Schritt mussten die zu fuzzenden Projekte für den LibFuzzer zugänglich gemacht werden. Bevor aber die Ausführung dieser Erweiterung möglich ist, wird noch ein .NET Binary für LibFuzzer benötigt, welche heruntergeladen wurde. Abschliessend wurde versucht, das Fuzzing des XML-Parsers anhand der LibFuzzer-Erweiterung zu starten. Dies war aber nicht möglich, da das Binary im Zusammenhang mit unserem Beispielprojekt nicht erkannt wurde. (Mit einem anderen Tutorial-Beispiel hingegen war dies möglich.)

Die folgende Fehlermeldung wird ausgegeben, wenn versucht wird, den XML-Parser mit der LibFuzzer-Erweiterung zu fuzzen:

```
execlp() failed: No such file or directory
short read: expected 4 bytes, got 0 bytes
```

Abbildung 9: Error SharpFuzz with LibFuzzer

- ► Um die Annahmen, dass SharpFuzz kein wirklich grammatikbasiertes Fuzzing für unseren Fall (d.h. kein Dictionary) ermöglicht, wurde Kontakt mit dem Entwickler von SharpFuzz (Nemanja Mijailovic) aufgenommen, der dies bestätigte.
- ► Im *Anhang*: Mailverkehr mit Nemanja Mijailovic

8.4.2 Resultat

8.4.2.1 Vorgehen

Um gute Ergebnisse mit Sharpfuzz zu erhalten, wurde Fuzzing für den XML-Parser mit zwei verschiedenen Dictionary Files und einem gültigen XML-File durchgeführt. Zusätzlich wurde C# Code geschrieben, um den XML-Parser mit Sharpfuzz zu starten.
Im ersten Schritt wurde AFL mit einem Standard XML Dictionary, welches bei der AFL Installation mitgeliefert wird, gefuzzt. Dabei sind 62 Crashes aufgetreten. In einem zweiten Versuch wurde das XML Dictionary mit benutzerdefinierten XML-Elementen, die im mitgelieferten XML-File vorhanden sind, erweitert. Dadurch wurden 68 Crashes aufgetreten.

Die Variante mit dem benutzerdefinierten XML-Dictionary lieferte das bessere Ergebnis, da hier die meisten Crashes erkannt wurden.

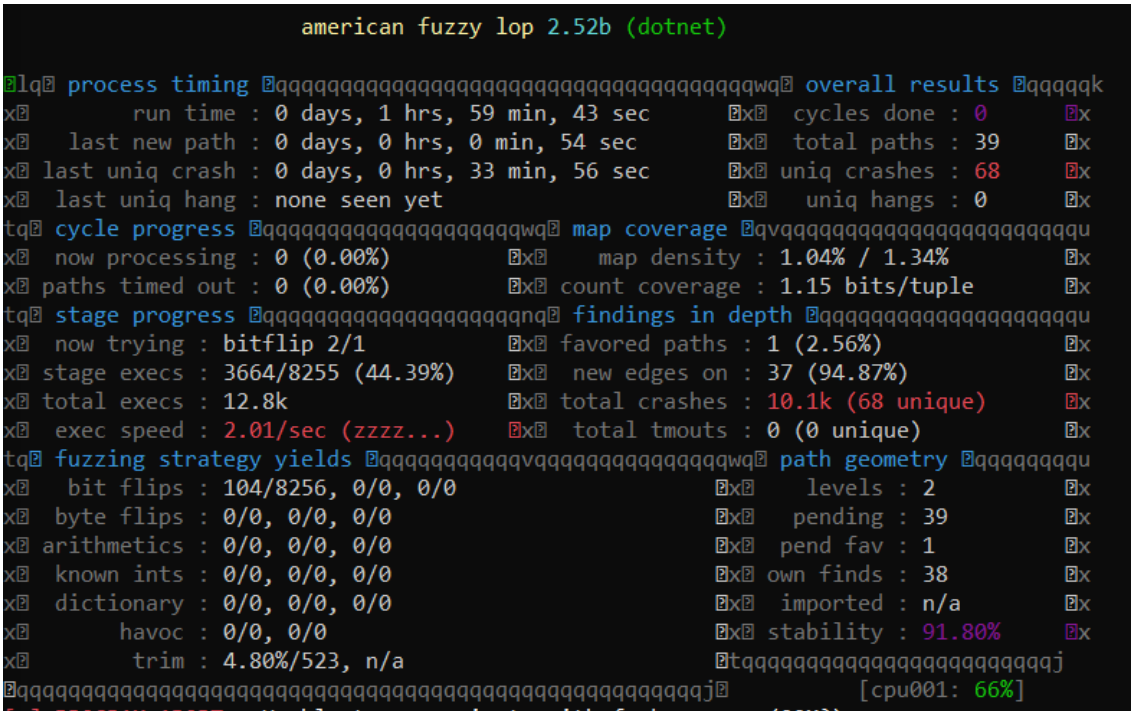


Abbildung 10: AFL Ergebnis

Wie man in dieser Abbildung sehen kann, wurden 68 Unique Crashes und 39 verschiedene Paths mit AFL gefunden.

► Weitere Fuzzing-Versuche können im *Anhang E: AFL Screens* entnommen werden

Bemerkung: DB-Mocking wurde nicht ausgeführt, da die Aufgabe des von nxt Engineering erhaltenen XML-Parsers neben der Verarbeitung der XML-Daten auch deren Import in die tacslive-Datenbank ist. Das bedeutet, dass ein wesentlicher Teil des Quellcodes aus Methoden zur Kommunikation mit der Datenbank besteht, die für die Funktionalität dieses Parsers wichtig sind. Dies zeigt sich auch in den von nxt Engineering bereitgestellten Systemtests.

8.4.2.2 Analyse Crashes

Alle mutierten XML-Files (68 Files), welche zu Crashes geführt haben, wurden automatisch von der AFL-Engine unter …/Findings/crashes/ abgespeichert. Für die Verifizierung der Files wurden alle diese mutierten Files mit DataImporter.Console Projekt über Command Line gestartet. (siehe wie Starten XML-Parser Installation)
Nach der Verifizierung wurde festgestellt, dass es überall die gleiche Exception mit leicht veränderter Beschreibung geworfen wurde. Eine dieser Exceptions sieht wie folgt aus:

Unhandled exception. System.InvalidOperationException: There is an error in XML document (16, 31).

---> System.Xml.XmlException: The 'geburtsdatum' start tag on line 16 position 6 does not match the end tag of 'Geburtsdatum'. Line 16, position 31.

at System.Xml.XmlTextReaderImpl.Throw(Exception e)

at System.Xml.XmlTextReaderImpl.Throw(String res, String[] args)

at System.Xml.XmlTextReaderImpl.ThrowTagMismatch(NodeData startTag)

at System.Xml.XmlTextReaderImpl.ParseEndElement()

at System.Xml.XmlTextReaderImpl.ParseElementContent()

at System.Xml.XmlTextReaderImpl.Read()

at System.Xml.XmlTextReaderImpl.Skip()

at System.Xml.XsdValidatingReader.Skip()

at System.Xml.Serialization.XmlSerializationReader.UnknownNode(Object o, String qnames)

at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationReaderExportTacs.Read13_KundenType (Boolean isNullable, Boolean checkType)

at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationReaderExportTacs.Read15_ExportTacs(Boolean isNullable, Boolean checkType)

at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationReaderExportTacs.Read16_ExportTacs()

--- End of inner exception stack trace ---

at System.Xml.Serialization.XmlSerializer.Deserialize(XmlReader xmlReader, String encodingStyle, XmlDeserializationEvents events)

at System.Xml.Serialization.XmlSerializer.Deserialize(XmlReader xmlReader)

at DataImporter.Services.DataProcessingService.GetExportTacs(Stream stream, ImportLogger importLogger) in C:\Users\Danusan\Documents\HSR-OST School\nxt_source_code\DataImporter.Core\Services\DataProcessingService.cs:line 111

```
   at DataImporter.Services.DataProcessingService.Process(ImportJob importJob, Stream contentStream) in C:\Users\Danusan\Documents\HSR-OST School\nxt_source_code\DataImporter.Core\Services\DataProcessingService.cs:line 57

   at DataImporter.Console.ImporterConsole.ProcessFile(FileInfo file) in C:\Users\Danusan\Documents\HSR-OST School\nxt_source_code\DataImporter.Console\ImporterConsole.cs:line 253

   at DataImporter.Console.ImporterConsole.Process() in C:\Users\Danusan\Documents\HSR-OST School\nxt_source_code\DataImporter.Console\ImporterConsole.cs:line 179

   at DataImporter.Console.ImporterConsole.<>c.<Main>b__10_0(Options options) in C:\Users\Danusan\Documents\HSR-OST School\nxt_source_code\DataImporter.Console\ImporterConsole.cs:line 152

   at CommandLine.ParserResultExtensions.WithParsed[T](ParserResult`1 result, Action`1 action)

   at DataImporter.Console.ImporterConsole.Main(String[] args) in C:\Users\Danusan\Documents\HSR-OST School\nxt_source_code\DataImporter.Console\ImporterConsole.cs:line 146
```

8.4.2.3 Mögliche Massnahmen

Wie im vorherigen Abschnitt beschrieben, wird die "System.InvalidOperationException" beim XML-Parser geworfen und nicht sauber abgefangen, was mit Sharpfuzz festgestellt werden konnte.
In der Klasse DataProcessingService, die sich im DataImporter.Core Projekt befindet, wird die Exception in der Methode GetExportTacs nicht abgefangen.
Eine mögliche Lösung des Problems wäre die Ergänzung von try und catch zur GetExportTacs-Methode um die «System.InvalidOperationException» abzufangen oder die Validierung der XML-Tags vor dem XML Parsing im DataImporter.Console Projekt.

8.4.3 Erfahrung

Die Integration des Sharpfuzz-Codes zum Starten des XML-Parsers konnte ohne große Probleme gelöst werden. Beim Starten der AFL-Engine trat in gewissen Fällen unerklärliches Verhalten auf, für das wir keine Ursache finden konnten. Dabei stürzte die Engine beim ersten Durchlauf mehrfach ab und konnte später mit den instrumentierten Dateien in der Entwicklungsumgebung nicht mehr debuggt werden, da sie immer die Exception «System.NullReferenceException» warf. Man findet auch im Internet bezüglich dieser Probleme wenig Unterstützung. Die einzige hilfreiche Informationsquelle ist die Github Repository von Sharpfuzz und AFL. Doch nach langem Ausprobieren und Versuche konnte der XML-Parser mit Sharpuzz gefuzzt werden und einige Resultate finden. Beim Start der AFL-Engine muss darauf geachtet werden, dass der erste Durchlauf erfolgreich ohne Exceptions durchläuft, ansonsten wird die AFL-Engine einen Crash melden und nicht fuzzen. Im Allgemeinen ist dieses Tool empfehlenswert um C# Programme zu fuzzen.

8.5 Fuzzing mit OneFuzz

8.5.1 Probleme

Es gab während der Installation/Konfiguration und auch während dem Evaluieren mehrere Probleme, die manchmal schneller gelöst werden konnten, aber es gab auch solche die einige Zeit in Anspruch genommen haben.
Nachfolgend sind einige Probleme aufgelistet, die während der Installation und dem Evaluieren des Tools aufgetreten sind, sowie die Lösung zu dem entsprechenden Problem. Falls die Probleme nicht gelöst wurden, ist aufgeführt, wieso das Problem nicht gelöst werden konnte.

- **Beispiel Programm fuzzen:** Um zu sehen, ob wir OneFuzz korrekt installiert und konfiguriert haben, haben wir das Beispiel auf dem GitHub Repository, welches im geting-started.md Dokument aufgeführt ist, durchgespielt. Bei dem LibFuzzer Beispiel von OneFuzz konnten wir, nachdem es uns gelungen war, die VMs korrekt zu starten, die Jobs ausführen lassen und danach konnten wir die generierten Reports analysieren.
- **Templates:** Bei OneFuzz gibt es verschiedene Templates, um Jobs zu generieren. Diese Templates basieren auf anderen bekannten Fuzzern, wie LibFuzzer, OssFuzz oder AFL (American Fuzzy Lop). Wir haben mit jedem Template, das zur Verfügung steht, versucht Jobs zu generieren und zu starten. Jedoch bekamen wir, als wir versucht haben Jobs zu generieren, bei den meisten Templates Fehlermeldungen. Bei LibFuzzer kam die Fehlermeldung, dass das angegebene EXE-File kein LibFuzzer-Binary ist. Bei OssFuzz muss man noch zusätzlich angeben welches OssFuzz Template genau man nehmen will. Es gab hier nur das LibFuzzer Template für OssFuzz und deswegen funktionierte auch OssFuzz nicht. AFL bekamen wir bei der Erstellung des Jobs eine «Invalid Container» Fehlermeldung. Da diese Templates nicht funktionierten, konnten wir nur mit dem Template Radamsa (ein Test Fall Generator a.k.a Fuzzer) arbeiten, mit welchem wir Jobs starten konnten.
- **Fehlende Reports:** Es wurden mehrere Jobs deployed mit dem XML-Parser als zu fuzzendes Programm. Um die Resultate dieser Jobs analysieren können, müssen die generierten Reports analysiert werden. Diese Reports wurden jedoch nicht generiert.
- **Nur ein Argument beim XML-Parser:** Standardmässig startet man den XML-Parser mit zwei Argumenten, der XML-Datei, die geparst werden soll, und einer TenantId. Da man bei OneFuzz jedoch nur einen Ordner mit Input-Daten mitgeben kann, musste der Quellcode des Parsers so abgeändert werden, dass man den Parser nur mit den XML-Files als Argument starten kann und die TendantId nicht mehr mitgegeben werden muss.

8.5.2 Resultate

- Nach einigem Ausprobieren konnten wir die VMs in den Pools erfolgreich starten und konnten somit einige Jobs mit dem OneFuzz Beispiel-Programm starten und anschliessend, als die Jobs beendet waren, die generierten Reports analysieren. In diesen Reports, die als JSON Dokument generiert werden, konnte man sehen, wieso das Programm bei gewissen Inputs nicht mehr ordnungsgemäss funktionierte.

8.5.3 Kosten

Derzeit benötigt man einen Azure-Account für die Benutzung von OneFuzz. Da OneFuzz auf Azure läuft und Ressourcen davon benötigt, fallen für die Benutzung von OneFuzz verschiedene Kosten an.
Während unserer Evaluation und Nutzung von OneFuzz beliefen sich die Kosten auf 783.19 CHF.

Nachfolgend sind die Kosten der einzelnen Ressourcen/Services aufgelistet, die während der Benutzung von OneFuzz angefallen sind.

Name	Kosten
virtual machines	449,58 CHF
azure app service	222,44 CHF
storage	61,96 CHF
signalr	49,21 CHF
bandwidth	0,00 CHF
event grid	0,00 CHF

Abbildung 11 Kostenaufstellung Azure

8.5.4 Erfahrung (inkl. Feedback für Microsoft)

Da das Tool noch neu ist, findet man sehr wenig Hilfe im Internet wenn irgendetwas nicht nach Plan läuft. Man muss sehr viel recherchieren bis man vielleicht findet was man sucht, um das Problem zu lösen. Eine gute Hilfestellung sind die Issues auf dem GitHub Repository von OneFuzz. Entweder wurde das Problem, welches man gerade hat, schon einmal dort diskutiert und gelöst oder man kann dort sein Problem reinschreiben und bekommt dann von Leuten, die am Tool mitarbeiten, Hilfe. Man bekommt in der Regel schnell eine Antwort auf die Probleme im GitHub Repository.
Das man auf Probleme stossen wird, ist wahrscheinlich, da die Anleitungen nicht sehr detailliert sind, z.B. wenn man sich mit Azure nicht auskennt, kommt man schnell an seine Grenzen und muss nach Hilfe suchen. Da das Tool wie erwähnt noch neu ist, gibt es auch noch einige Bugs im Programm, welche man dann irgendwie umgehen muss, da man sonst nicht weiterkommt. Oder man muss auf einen neuen Release warten, bei dem der Bug dann behoben wurde.

Das Tool ist eigentlich für kontinuierliches developer-driven Fuzzing entwickelt worden, und das ist nicht wirklich das, wofür wir OneFuzz benutzten wollten.

9 Weitere Tests

Um der Firma nxt Engineering, welche uns für diese Arbeit einen XML-Parser zum Testen zur Verfügung stellt, ein umfassenderes Ergebnis abliefern zu können, werden neben dem Fuzzing weitere Tests durchgeführt, die im folgenden Abschnitt dokumentiert werden.

9.1 XML-Verletzlichkeiten

Das Format bzw. die Datenstrukturdefinition von XML-Dokumenten wird von DTD-/XSD-Dateien definiert. Die XML-Dokumente haben daher die Möglichkeit die URL dieser Dateien innerhalb eines DOCTYPE-Deklaration anzugeben. Nun haben Angreifer die Möglichkeit, diese Funktionalität für ihre Zwecke auszunützen.
Aus diesem Grund wurden die in der Theorie beschriebenen Angriffsvarianten aufgrund ihrer Verletzlichkeit an ihnen getestet. Die verschiedenen XML-Angriffsversuche wurden immer auf eine separate Kopie der gültigen XML-Datei "Daten_Tacs.xml" angewendet, die von nxt Engineering mitgeliefert wurde.

Anwendungsbeispiele für die Angriffe sind unter den folgenden Links beschrieben:

- https://www.usenix.org/system/files/conference/woot16/woot16-paper-spath.pdf
- https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing
- https://www.netsparker.com/blog/web-security/xxe-xml-external-entity-attacks/

► Theorie zu den verschiedenen Angriffsarten ist unter dem Kapitel "Analyse" > "XML" zu finden.

9.1.1 XML-Generator Angriff (Fragment Injection)

Mit der Fragment Injection Methode wurde versucht, ein gültiges Fragment in Data_Tacs.xml mit einem ungültigen Tag auszukommentieren und durch manipulierte Daten zu ersetzen.

9.1.1.1 Resultat

Die Applikation unterbindet jegliche Fragment Injection-Versuche und ist somit gegen XML-Generator Angriffe geschützt. Im folgenden Codeabschnitt wird die geworfene Fehlermeldung abgebildet.

```
fail: DataImporter.Services.ImportLogger[0]

    Error while processing data

System.InvalidOperationException: There is an error in XML document (32, 7).

 ---> System.Xml.XmlException: The 'comment' start tag on line 31 position 10 does not match the end tag of 'Leistungsdatensatz'. Line 32, position 7.
```

9.1.2 XML-Parser Angriff

Entity Attack
Die XML-Entity Schwachstelle wurde mit Hilfe der Billion Laugh-Attacke, welche auch unter dem Namen "XML Bomb" bekannt ist, überprüft. Diese Attacke beruht auf der Kombination mehrerer XML-Entitäten, die aufeinander verweisen. Beim Verarbeiten der XML-Datei, ruft der XML-Parser jede Entität nacheinander auf und erzeugt dabei exponentiell neue Instanzen der zuvor aufgerufenen Entität, was zur Erschöpfung der Server-Ressourcen führt.

External Entity Attack
Um den XML-Parser auf eine External Entity Schwachstelle zu überprüfen, wurde eine der Data Extraction Angriffsmethoden angewendet. Dabei wird versucht, mittels der Anwendung von External Entity auf die Datei "boot.ini" zuzugreifen, welche sich normalerweise im Root-Verzeichnis der Systempartition befindet. Dabei handelt es sich um eine Textdatei, die die Boot-Optionen für Computer mit BIOS-Firmware enthält.

Parameter Entity Attack
Die Parameter-Entity Schwachstelle wurde anhand einer SSRF-Attacke getestet. Ähnlich wie External Entities ermöglicht auch Parameter Entities es, Daten von einem anderen Ort zu referenzieren. Der Unterschied ist hier, dass der Parser diese Deklarationen liest und durch den Inhalt der referenzierten Entität ersetzt.

XInclude Attack
XInclude bietet eine weitere Möglichkeit, innerhalb von XML-Dokumenten auf andere XML- oder Textdokumente zu verweisen. Auch diese XInlcude-Referenzen werden, ähnlich wie bei Parameter-Entity, bei der Auflösung durch den Inhalt der referenzierten Dokumente ersetzt. Somit ergibt sich mit XInlcude eine weitere Variante, wie man schädliche Dokumente einbinden kann. Bei der Überprüfung dieser Sicherheitslücke wurde daher versucht, eine extern platzierte XML-Datei einzubinden.

9.1.2.1 Resultat

Die Anwendung verhindert jegliche Verwendung von Entitäten (Local Entity, External Entity & Parameter Entity). Daher ist das Referenzieren und das Einbetten von schädlichen Inhalten sowie das Extrahieren von sensiblen Daten nicht möglich.

```
fail: DataImporter.Services.ImportLogger[0]

    Error while processing data

System.InvalidOperationException: There is an error in XML document (0, 0).

 ---> System.Xml.XmlException: For security reasons DTD is prohibited in this XML document. To enable DTD processing set the DtdProcessing property on XmlReaderSettings to Parse and pass the settings into XmlReader.Create method.
```

Das Einbetten von externen Dokumenten mittels XInclude wird ebenfalls unterbunden.

```
fail: DataImporter.Services.ImportLogger[0]

      Error while processing data

System.InvalidOperationException: There is an error in XML document (2, 3).

 ---> System.InvalidOperationException: <data xmlns=''> was not expected.
```

9.1.3 Schlussfolgerung

Anhand der durchgeführten Überprüfungen auf XML-Schwachstellen und der daraus resultierenden Ergebnisse kann der Schluss gezogen werden, dass der XML-Parser nicht anfällig für die getesteten XML-Verletzlichkeiten ist.

10 XML-Parser Fuzzing auf .NET

Um einen XML-Parser korrekt fuzzen zu können, ist grammatikbasiertes Greybox-Fuzzing (Greybox-Fuzzing = mutations- & coverage-basiertes Fuzzing) erforderlich, da das Format einer gültigen XML-Datei durch eine DTD- oder XSD-Datei definiert wird. Um gültige Fuzz-Daten zu generieren, werden also eine gültige XML-Datei, die als Seed verwendet wird, und eine XSD/DTD-Datei, nach deren Grammatik bzw. Struktur Mutationen vorgenommen werden, benötigt.

Nach unseren Recherchen gibt es bislang noch keine ausreichende Unterstützung für grammatikbasiertes Greybox-Fuzzing eines XML-Parsers in der .NET-Umgebung. Daher wird die Beschreibung, wie man einen XML-Parser unter .NET fuzzt, mit dem Tool erfolgen, das wir für unsere Studie als am geeignetsten erachtet haben.

Durch die aus unserer Studie und der Anwendung der Fuzzing-Tools gewonnenen Erkenntnisse, bewerten wir SharpFuzz als das am besten geeignete Fuzzing-Tool für das XML-Parser-Fuzzing auf der .NET Umgebung (C#). SharpFuzz, welches die Fähigkeiten von AFL in C# implementiert, ermöglicht mutations- und coverage-basiertes Fuzzing, was eines der wichtigsten Kriterien für XML-Parser-Fuzzing ist. SharpFuzz unterstützt auch "grammatikbasiertes" Fuzzing, allerdings nur unter Verwendung von eines Dictionary, die Schlüsselwörter der entsprechenden Sprache enthält, die nicht mutiert werden. Das Dictionary kann mit benutzerdefinierten Schlüsselworten, in unserem Fall XML-Elementen, ergänzt werden.

Eine vollständige Struktur einer gültigen XML-Datei (z.B. Hierarchie der XML-Elemente), die durch eine XSD/DTD-Datei definiert ist, welche bei jeder Mutation erhalten bleibt, kann hingegen nicht angegeben werden, was das Fuzzing eines XML-Parsers erschwert. Dadurch, dass die Fuzz-Daten ohne vorgegebene Grammatik, also ohne Struktur, mutiert werden, kann es zu dauerhaft ungültigen Fuzz-Daten kommen, was der erreichten Coverage und dem Fuzzing-Prozess im Allgemeinen nicht förderlich ist.

Trotz dieser Einschränkung kann der XML-Parser mit SharpFuzz gefuzzt werden. Ein gezielteres Ergebnis wäre jedoch absehbar, wenn die Grammatik oder die Struktur (XSD) noch eingebunden werden könnte. Daher wird in der folgenden Abbildung gezeigt, wie ein optimales Fuzzing-Tool aussehen müsste, um einen XML-Parser vollumfänglich fuzzen zu können.

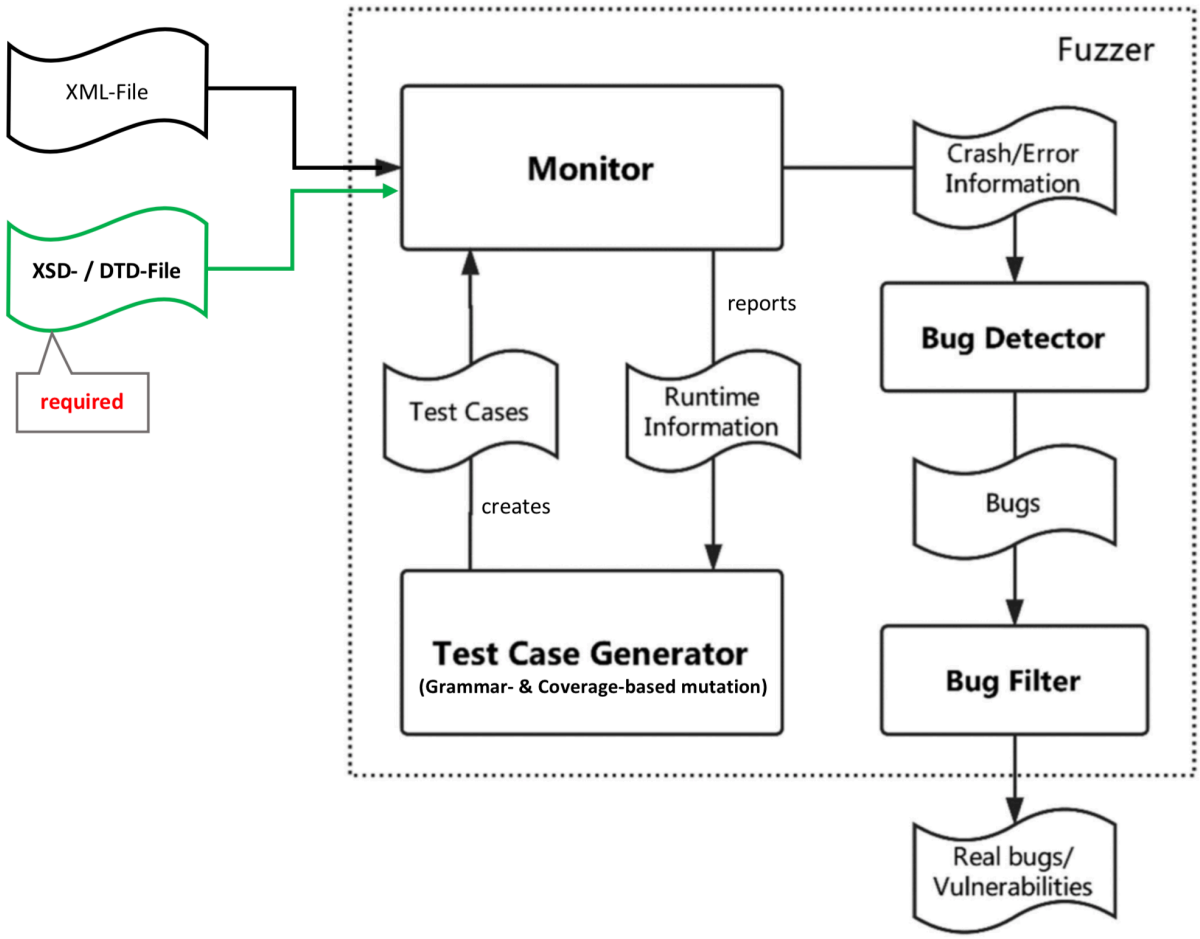


Abbildung 12: Optimaler Fuzzer [Grafik adaptiert von (IEEE, 2018)]

11 Verallgemeinerbarkeit der Fuzzing-Tools

Eine Aufgabe in der Aufgabenstellung war es, dass die Verallgemeinerbarkeit der Fuzzing-Werkzeuge dargestellt und beschrieben werden sollte. Während der Arbeit an diesem Projekt, vor allem gegen Ende hin, mussten wir jedoch gemeinsam mit Prof. Dr. Markus Stolze erkennen, dass das Wissen, das wir uns im Rahmen der Studienarbeit über die Recherche und Anwendung der Fuzzing-Tools angeeignet hatten, nicht ausreicht, um die behandelten Tools bzw. Tool-Chains weiter zu verallgemeinern. Eine Verallgemeinerbarkeit der Fuzzing-Tools können wir daher in dieser Arbeit nicht präsentieren.

Nach unserer Recherche können wir eine Tool-Chain aufzeigen, die es ermöglicht, das Fuzzing-Tool "SharpFuzz" so zu erweitern, dass die Grammatik der gegebenen Seed-Datei berücksichtigt wird und nicht nur die Generierung der Fuzz-Daten in Richtung der Grammatik mit Hilfe eines Dictionaries unterstützt.

► Jedoch hat das bei dem XML-Parser nicht funktioniert. (Für genauere Informationen siehe Kapitel Anwendung der Fuzzing Tools)

11.1 Verwendung von LibFuzzer mit SharpFuzz

SharpFuzz kann mit einer Erweiterung von LibFuzzer vwrwendet werden. Im Gegensatz zum AFL-Fuzzer unterstützt LibFuzzer viele fortgeschrittene Techniker, wie zum Beispiel das strukturierte bzw. grammatikbasierte Fuzzing.

LibFuzzer befindet sich aber noch in aktiver Entwicklung. Sobald alle LibFuzzer-Funktionen in SharpFuzz unterstützt werden, wird es vermutlich die empfehlenswerteste Fuzzing-Engine sein.

Installationsanleitung und weitere Informationen sind unter den folgenden Links zu finden:

- Using LibFuzzer with SharpFuzz
  https://github.com/Metalnem/sharpfuzz/blob/master/docs/LibFuzzer.md
- Structure-Aware Fuzzing with LibFuzzer
  https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md

Teil IV

Software Installation & Konfiguration

12 Einrichtung des Beispielprojekts

12.1.1 Docker

Docker muss auf Mac OS oder Windows installiert werden.

12.1.2 Database

Der XML-Importer von nxt Engineering GmbH verwendet Microsoft SQL-Server 2019 als Datenbank-Server. Das DB-Backend kann mit folgendem Docker-Befehl im DataImporter.Console Verzeichnis in der Command Line gestartet werden:

```
docker-compose up
```

Die Datenbank für die Solution Rodix muss noch erstellt werden.

12.1.3 Wie kann der Importer/XML-Parser gestartet werden?

Bevor der XML-Parser erfolgreich durchläuft, müssen einige DB Einträge existieren. Andernfalls wird der XML-Parser vor dem Parsen abbrechen und folgende Fehlermeldung ausgeben.

```
DataImporter.Services.DataProcessingException: MandantId = '12345.001'
```

12.1.4 DB Eintrag

Für den erfolgreichen Import eines XMLs müssen Kantone, Adresse, Anwenderbetrieb und Mandant Tabelle mit folgenden Query abgefüllt werden.

```
insert into tacslive.dbo.Kantone (Kuerzel, Bezeichnung, LandCode) values ('ZH', 'Zürich', 'CH');


insert into tacslive.dbo.Adresse (Name, KantonId) values ('Mustername', 1);


insert into Anwenderbetrieb (BetriebId, Bezeichnung, GueltigAb) values (100, 'MusterAnwenderbetrieb',
'2019-01-01');


insert into Mandant (MandantId, Bezeichnung, GueltigAb, AdressId, AnwenderbetriebsId)

values (12345.001, 'Musterspital', '2019-01-01', 1, 1);
```

12.1.4.1 XML-Parser Starten

Für das Starten des Importers oder XML-Parsers ist ein gültiges XML-File erforderlich, welches wir von nxt Engineering GmbH zur Verfügung gestellt bekommen haben. Zusätzlich muss noch die «tenantId» als Parameter mitgeben werden, welche vom BetriebId Tag im XML-File entnommen werden kann. Der BetriebId Tag Wert und das «MandantId» Feld von der Tabelle Mandant im DB müssen übereinstimmen, ansonsten wird eine Exception geworfen, wie vorher beschrieben. Das XML-File muss sich im DataImporter.Console Projekt befinden und kann dort über die Command Line mit folgendem Befehl gestartet werden:

```
dotnet run data.xml 12345.001
```

12.2 Fuzzware

12.2.1 Installation

Fuzzware kann nur auf Windows benutzt werden. Es kann unter folgendem Link heruntergeladen werden: https://www.fuzzware.net/download

12.2.2 Konfiguration

Für die Nutzung von Fuzzware müssen folgende Anweisungen befolgt werden.

1. Als erstes muss ein neues Projekt erstellt werden. **(Project => New)**
2. Anschliessend drückt man auf den Button «Fuzz an XML file» um die Files zu spezifizieren. **(Choose an input source)**
3. Danach kann man das XSD-File und XML-File angeben. **(Configure the input source)**
4. Optional: Für die Datentype wie Strings, Integers, Decimals und Bytes kann man Werte anpassen. Zusätzlich können auch noch Datenstrukturen angepasst werden. **(Options for fuzzing data types/ Options for fuzzing data structures)**
5. Als nächstes kann man die Output Destination angeben, wo die gefuzzten Files abgespeichert werden sollen. **(Configure the output destination)**
6. Test Mode muss auf “OFF” gesetzt werden, damit beim Start gefuzzt wird. Ansonsten wird das XML-File nicht gefuzzt. **(Configure and Run fuzzer)**
7. Man ist bereit zu fuzzen und drückt auf «Start» Button **(Configure and Run fuzzer)**

12.4 SharpFuzz

Zusammengefasst von: (Mijailovic 2019a)

AFL funktioniert unter Linux und MacOS. Um es auf Windows zu verwenden, kann jede Linux-Distribution verwendet werden, die unter dem Windows-Subsystem für Linux funktioniert. GNU make und ein funktionierender Compiler (gcc oder clang) werden benötigt, um AFL-Fuzz zu kompilieren. Ausserdem muss .NET Core 2.1 oder höher auf dem Rechner installiert sein, um .NET-Assemblies mit SharpFuzz instrumentieren zu können.

Um das aktuelle AFL-Fuzz Source-Package herunterzuladen und zu entpacken, wird `wget` von Homebrew oder ein ähnliches Kommandozeilenprogramm benötigt.

LINUX

Die meisten, wenn nicht sogar alle, Linux-Distributionen werden standardmässig mit `wget` ausgeliefert. Linux-Benutzer brauchen daher keine zusätzlichen Installationen, um AFL einzurichten bzw. um dieses Skript laufen zu lassen.

MAC OS

macOS-Systeme werden nicht mit `wget` ausgeliefert, aber es ist möglich, diesen mittels Homebrew auf der Kommandozeile zu installieren. Sobald Homebrew eingerichtet ist, kann `wget` einfach mit dem Befehl `brew install wget` im Terminal installiert werden.

WINDOWS

Unter Windows kann SharpFuzz nur mit Windows-Subsystem für Linux verwendet werden.

- Windows-Subsystem für Linux muss aktiviert und eingerichtet werden.
  - https://docs.microsoft.com/en-us/windows/wsl/install-win10
- Für die Distribution haben wir für Ubuntu 20.04 LTS entschieden, da wir damit bereits Erfahrungen hatten. Ubuntu 20.04 LTS kann unter Windows im Microsoft Store installiert werden.
- Nach der Einrichtung von Windows-Subsystem für Linux kann man wie bei Linux das Skript laufen lassen.

12.5 Installation

AFL-Fuzz und das globale .NET-Tool SharpFuzz.CommandLine können mit dem folgenden Skript installiert werden: https://github.com/Metalnem/sharpfuzz - Installation

12.6 Konfiguration

Damit Sharpfuzz mit der Solution ‘Rodix’ gestartet werden kann, müssen vorher einige Schritte erledigt werden.

1. In der Solution soll ein neues .NET Console Projekt für Sharpfuzz im Visual Studio oder Jetbrains Rider erstellt werden. In diesem Fall heisst das Projekt ParserFuzzer.
2. Folgende Abhängigkeiten müssen im Projekt Parserfuzzer hinzugefügt werden:
    1. Da der XML-Parser .Net Core 3.1 verwendet, muss diese hinzugefügt werden.
    2. DataImporter.Core.dll und DataAcess.dll sind die Files, welche zum XML-Parser gehören.
    3. Damit der XML-Parser gefuzzt werden kann, muss das Sharpfuzz Package hinzugefügt werden. (wird im Schritt 4 beschrieben)

```
<Project Sdk="Microsoft.NET.Sdk">

 <PropertyGroup>

  <OutputType>Exe</OutputType>

  <TargetFramework>netcoreapp3.1</TargetFramework>

 </PropertyGroup>

  <ItemGroup>

    <Reference Include="DataImporter.Core">

      <HintPath>DataImporter.Core.dll</HintPath>

    </Reference>

  </ItemGroup>

  <ItemGroup>

    <Reference Include="DataAccess">

      <HintPath>DataAccess.dll</HintPath>

    </Reference>

  </ItemGroup>

  <ItemGroup>

   <PackageReference Include="SharpFuzz" Version="1.6.2" />

  </ItemGroup>

</Project>
```

3. Die zwei dll Files(DataImporter.Core.dll und DataAcess.dll), welche im folgenden Verzeichnis ...\ParserFuzzer\bin\Debug\netcoreapp3.1 befinden, müssen mit folgenden Sharpfuzz Commands auf Mac/Linux instrumentiert werden:

```
cd ...\ParserFuzzer\bin\Debug\netcoreapp3.1

...\ParserFuzzer\bin\Debug\netcoreapp3.1> sharpfuzz DataAccess.dll

...\ParserFuzzer\bin\Debug\netcoreapp3.1> sharpfuzz DataImporter.Core.dll
```

Diese zwei Files müssen instrumentiert sein, sonst wird beim Start von AFL folgende Fehlermeldung ausgeben:

```
[-] PROGRAM ABORT : No instrumentation detected
```

4. Das Sharpfuzz Package muss hinzugefügt werden, damit der Fuzzer in C# Code geschrieben werden kann. Es kann mit folgendem Command hinzugefügt werden:

```
cd ...\ParserFuzzer\

...\ParserFuzzer\> dotnet add package SharpFuzz
```

5. Mit der Program Klasse in C# soll die Klasse DataProcessingService aufgerufen werden, welche im DataImporter.Core Projekt befindet. Mit dem Stream Parameter, welche die Fuzzer.Run Methode liefert, werden die Fuzzing Input Daten erstellt. Der restliche Code wurde anhand von der ImporterConsole Klasse, welche sich im DataImporter.Console Projekt befindet, erstellt. So werden die Mindestanforderungen der Parameter wie «tenantId» Parameter und XML-Stream erfüllt, um den XML-Parser zu starten und fuzzen zu können.

► Im *Anhang: Sharpfuzz Code* finden sie den Code, um den XML-Parser zu starten.

6. Der erste Durchlauf mit AFL Fuzzing muss mit einem gültigen XML geschehen, deshalb muss im Projekt ParserFuzzer ein Verzeichnis 'Testcases' erstellt und dort ein gültiges XML-File abgelegt werden. Dieses soll ein gültiges XML-File, welches von XML-Parser Code akzeptiert wird, enthalten. Dieses XML wird, wie im vorherigen Schritt 6 erklärt, als Stream gelesen und mit AFL gefuzzt.

   Wenn es im Projekt ParserFuzzer kein Verzeichnis 'Testcases' existiert, kann es beim Start des AFLs zur folgenden Fehlermeldung kommen.

```
[-]  SYSTEM ERROR : Unable to open 'Testcases'
```

7. **Optional:** Bei Formaten wie XML kann der Fuzzing-Prozess mit Dictionary Files verbessert werden. AFL wird mit einer Reihe von Dictionaries geliefert, die man nach der Installation in /usr/local/share/afl/dictionaries/ findet. Für unseres Fuzzing Projekt haben wir das xml.dict von AFL genommen und haben zusätzlich ein eigenes xml Dictionary File mit dem bestehenden xml.dict ersellt. Für die Erstellung von Dictionary Files findet man in /usr/local/share/afl/dictionaries/README.dictionaries weitere Informationen.

8. Nach all diesen Schritten sind wir bereit das Projekt mit AFL zu fuzzen.

```
afl-fuzz -i Testcases -o Findings -t 5000 -m 10000   -x /usr/local/share/afl/dictionaries/xml.dict dotnet bin/Debug/netcoreapp3.1/ParserFuzzer.dll
```

Wenn AFL nicht startet und folgende Fehlermeldung ausgibt, ist etwas im C# Code nicht richtig oder im XML-File etwas falsch:

```
[-] PROGRAM ABORT : Test case 'id:000000,orig:Daten_Tacs.xml' results in a crash
```

Solche Fälle müssen im Visual Studio oder Jetbrains Rider mit Debug Modus im Projekt ParserFuzzer gestartet und analysiert werden.

12.7 OneFuzz

12.7.1 Installation

Damit man OneFuzz benutzen kann, benötigt man einen Microsoft Azure-Account. Mit unseren OST-Adressen können wir uns bei Azure anmelden. OneFuzz läuft auf Windows, MacOS sowie allen gängigen Linux Distributionen. Lokal auf dem eigenen Rechner benötigt man Python 3.6 oder höher, sowie Azure CLI für die Kommunikation mit Azure. Auf dem GitHub Repository befindet sich eine Anleitung wie man OneFuzz einrichtet und ein Beispiel Programm fuzzt. Mithilfe dieser Anleitung haben wir OneFuzz auf unseren lokalen Rechnern installiert. Der Download und die Einrichtung von OneFuzz gehen ziemlich schnell. Das Deployment auf Azure mit dem deploy.py Skript geht dagegen ein paar Minuten.
Danach muss man noch zwei. whl Dateien installieren.

Zusammengefasst von: (sMicrosoft 2020)

12.7.2 Konfiguration

Damit man OneFuzz korrekt deployen kann, muss man eine gültige Subscription einer Premium V2 Instanz haben. Wenn man dann das deploy.py Skript ausführt, müssen vier Argumente mitgegeben werden. Das erste Argument ist die Regionsangabe und diese muss in einem bestimmten Format angegeben werden und es muss eine Region angegeben werden, die auf Azure verfügbar ist. Mit dem Befehl `az account list-locations -o table` können alle verfügbaren Regionen von Azure angezeigt werden. Für die Argumente zwei und drei können beliebige Namen für die Ressource-Gruppe und die Ressource-Instanz angegeben werden. Jedoch dürfen nur Namen angegeben werden, welche noch nicht benutzt worden sind, da eine Website erzeugt wird und die URL zu dieser Seite hat den Namen der Ressource-Gruppe im Namen. Der Rest der URL ist immer gleich, also ist der Name der Ressource-Gruppe entscheidend. Das vierte Argument ist die E-Mail-Adresse, mit der man bei Azure angemeldet ist.

Nachdem das deploy.py Skript fertig ausgeführt worden ist und OneFuzz auf Azure deployed ist, muss man einen Pool erstellen. In diesem Pool werden, dann die Jobs die später erstellt werden, ausgeführt. Danach erstellt man eine bestimmte Anzahl von general purpose Azure VM-Instanzen. Die Anzahl kann man selbst bestimmen. Diese VMs führen später die Jobs aus. Nachdem der Pool und die VMs aufgestartet sind, was ein paar Minuten dauert, kann man einen Job erstellen und ausführen lassen. Beim Erstellen eines Jobs kann man zwischen verschiedenen vorbereiteten Templates für den Job wählen. Unter anderem LibFuzzer, AFL, OssFuzz und Radamsa. Wir haben für unsere Jobs das Template Radamsa benutzt. Beim Konfigurieren des Jobs kann man neben der zu fuzzenden exe-Datei auch ein Ordner mit Inputs mitgeben. Wir haben jeweils XML-Files mit Fuzzware gefuzzt und diesen Ordner danach beim Erstellen des Jobs mitgegeben.

Teil V

Evaluation "Fuzzingbook"

13 Einführung

Für uns war Fuzzing ein Begriff, den wir bisher nur in der Theorie kannten und im Groben verstanden haben. Aber wir hatten uns bisher weder während dem Studium noch im Privaten oder im Berufsleben praktische Erfahrung aneignen können. Deshalb haben wir für das Projekt Fuzzing entschieden. Die Website «Fuzzing Book» (https://www.fuzzingbook.org/) half uns, die Theorie des Fuzzings, welche mit konkreten Code Beispielen mit Python programmiert sind, besser zu verstehen.

14 Eignung für eigenständige Erarbeitung

Das Fuzzing Book ist sehr hilfreich für Personen, welche noch keinerlei Erfahrungen mit Fuzzing gesammelt haben. Die Struktur war am Anfang ein bisschen schwierig zu verstehen, aber mit der Zeit versteht man die Zusammenhänge besser. Am Anfang von jedem Kapitel ist beschrieben, welche Themen oder Kapitel man gelesen haben sollte, damit das Verständnis und zusammenhängende Themen besser aufgenommen werden können. Ebenfalls sieht man auf der Sitemap wie das Buch aufgebaut ist und welche Kapitel auf welchen anderen Kapiteln aufbauen. Die einzelnen Kapitel wurden sehr detailliert erklärt mit sehr guten Code Beispielen in Python erklärt.

Da Fuzzing ein sehr grosses Thema ist, wurden im Fuzzing Book für die verschiedenen Interessenten diverse Touren erstellt, um den Einstieg zu erleichtern. Somit kann man sich auf einen bestimmten Rundgang konzentrieren, ob man Programmierer, Student oder Forscher ist. Folgende Touren werden von Fuzzing Book vorgeschlagen:

Zusammengefasst von: (Fuzzingbook.org 2020)

- **The Pragmatic Programmer Tour:** Es wird die Zielgruppe angesprochen, die so schnell wie möglich Tests generieren will.
- **The Page-by-Page Tours:** Diese Tour geht durch das Buch, wie das Fuzzing Book auch organisiert ist.
- **The Undergraduate Tour:** Es ist für Informatikstudenten geeignet, die die Grundlagen des Testens und verwandter Gebiete kennen lernen wollen. Es werden nicht nur Techniken angeschaut, sondern auch die Algorithmen und Implementierungen werden vertieft.
- **The Graduate Tour:** Zusätzlich zur «Undergraduate Tour» ist es für Absolventen geeignet, die sich tiefer mit den Techniken der Testgenerierung beschäftigen wollen.

- **The Researcher Tour:** Geeignet für Forscher, die neben der "Graduate" Tour nach Techniken suchen, welche irgendwo zwischen Laborstudium und weit verbreiteter Verwendung befinden.

- **The Author Tour**: Geeignet für Autoren, welche einen Beitrag zum Fuzzing Book leisten wollen. Es lohnt sich für die Personen, welche schon alles gelernt haben, und bereits praktische Erfahrungen im Bereich Fuzzing haben.

Das Fuzzing Book ist in folgende sechs Teile aufgeteilt. In den jeweiligen Teilen werden Links zu den jeweiligen Kapiteln mit einer kurzen Beschreibung zur Verfügung gestellt:

- **Whetting your Appetite:** Hier in diesem Kapitel werden zwei Kapitel vorgestellt/vorgeschlagen, und zwar die Übersicht über die verschiedenen Touren und auch noch das Kapitel zur Einführung in Software Testing. Dieses Einführungskapitel ist nicht spezifisch auf Fuzzing ausgelegt, sondern soll einen Überblick über das Testen von Software geben und erläutern, warum Testing so wichtig ist.

- **Lexical Fuzzing:** In diesem Teil wird man in die Test-Generation auf lexikalischer Ebene (zusammensetzen von Zeichen) eingeführt. Dafür wird als erstes das Kapitel zur Einführung ins Thema Fuzzing vorgeschlagen. In diesem Kapitel werden die Grundlagen von primitiven (true random) Fuzzing erläutert. Danach wird das Kapitel zum Thema Coverage aufgeführt. Darin wird erklärt, was Coverage ist, wieso es für Fuzzing wichtig ist und wieso es genutzt wird. Danach wird das Kapitel zur Einführung in das Mutation-Based Fuzzing aufgeführt. Danach wird Greybox Fuzzing genauer behandelt. Die letzten zwei Kapitel sind dann Search-Based Fuzzing und Mutation Analysis.

- **Syntactical Fuzzing:** Hier in diesem Teil wird Test-Generation auf syntaktischer Ebene (zusammensetzen von Inputs von Sprachstrukturen) vorgestellt. Dafür wird zuerst eine allgemeine Einführung in das Thema Grammatik gemacht, welches essenziell für diesen Teil des Buches ist. Die meisten Kapitel drehen sich um das Thema Grammatiken und wie man diese verwenden kann, um effizienter zu testen. Ebenfalls gibt es eine Einführung in das Thema Parsing und was das mit Testing zu tun hat.

- **Semantical Fuzzing:** In diesem Teil des Buches geht es um Techniken der Test-Generierung, welche die Semantik ebenfalls beachten, vor allem das Verhalten des Programms, welches den Input verarbeitet. Hier werden verschiedene Kapitel vorgestellt, die sich damit befassen, wie man zu einer Grammatik kommt, wenn man diese nicht schon hat, wie man den Input durch das Programm hindurch verfolgt und so zu Informationen kommt, die helfen das Programm noch besser zu analysieren. Auch wird in einem Kapitel beschrieben, wie man Pre- und Postconditions von Programmausführungen extrahiert.

- **Domain-Specific Fuzzing:** In diesem Teil geht es um Test-Generation für spezifische Domains. In diesem Teil werden vor allem Techniken beschrieben, mit denen man verschiedene Arten von Programmen testen kann, wie z.B. Web Applikationen, APIs oder Graphical User Interfaces. Für all diese Domänen wird in diesem Teil beschrieben, wie man Input generiert und wie man die Struktur des Inputs analysieren kann.

- **Managing Fuzzing:** Dieser Teil beschreibt, wie man Fuzzing im Grossen betreibt. In diesem Teil sind zwei Kapitel aufgeführt. Zum einen das Kapitel Fuzzing in the Large, welches beschreibt, wie man grosse Infrastruktur für Fuzzing erstellt, um Millionen von Tests auszuführen und deren Resultate zu verwalten. Das zweite Kapitel ist When to Stop Fuzzing. Das beschreibt, wie man abschätzen kann, wann Fuzzing genug ist und man damit aufhören sollte.

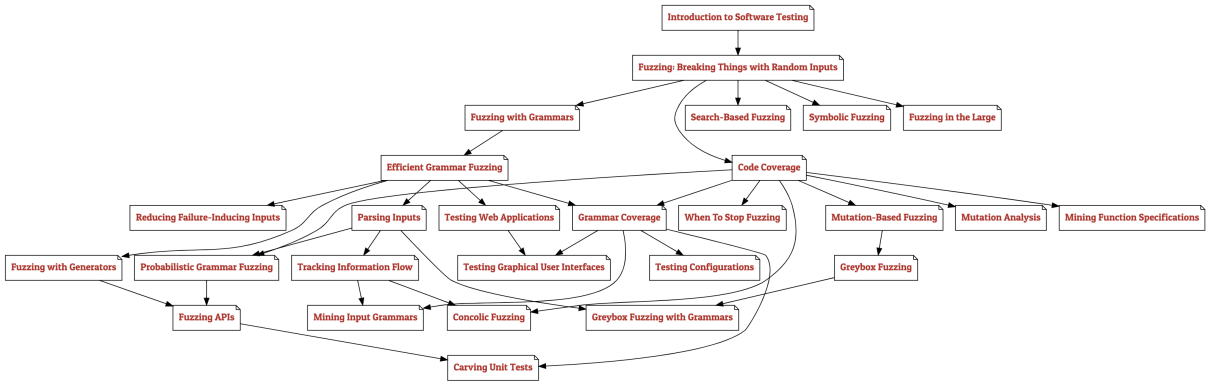


Abbildung 13: Sitemap (Fuzzingbook, 2020)

15 Unsere Bewertung & Feedback

15.1 Bewertung

Das Buch hat uns sehr geholfen uns ins Thema Fuzzing einzuarbeiten. Es ist einfach geschrieben und relativ ausführlich gehalten. Wir persönlich finden es ein sehr gutes Buch für jeden, der sich mit der Thematik Fuzzing beschäftigen möchte, aber noch kein oder sehr wenig Grundwissen hat. Das Buch ist in Englisch geschrieben, aber das sollte für einen Informatikstudenten kein Problem sein. Es ist sicher von Vorteil, wenn man bereits Erfahrungen mit dem Programmieren hat (nicht zwingend in Python) und etwas Grundwissen von Software Engineering allgemein hat, denn dann versteht man die Code-Beispiele viel besser. Falls man keine Erfahrungen hat, wird es etwas schwieriger den Inhalt des Buches richtig zu verstehen. Die Code-Beispiele, welche in Python geschrieben sind, helfen dem Leser sehr, das Gelesene auch wirklich zu verstehen. Daneben hat man auch gleich noch ein Beispiel, wie solcher Testing-Code in Programmcode aussieht. Die Kapitel im Buch sind zum Teil ziemlich lang, aber das ist nicht wirklich ein Problem, denn man ist auch mit diesem langen Kapiteln relativ schnell fertig, da es in einfacher Sprache geschrieben ist und auf komplizierte Erklärungen verzichtet. Es gibt auch ein paar Stellen, da geht es einfach nicht unkompliziert, aber auch diese Stellen sind gut verständlich. Und da die Kapitel zum Teil wirklich lang sind, sind gewisse Themen auch ziemlich genau und ausführlich beschrieben. Es gab aber auch Kapitel, mit denen wir mehr zu kämpfen hatten als mit anderen. Für diese Kapitel brauchten wir deutlich länger, bis wir alles wirklich verstanden hatten. Wir mussten einiges davon mehrmals lesen und die Beispiele genauer studieren als bei anderen Kapiteln. Zu diesen schwierigeren Kapiteln gehört das Kapitel "Parsing Inputs" in welchem das Thema Parsing vorgestellt wird und ebenfalls mit den Kapiteln in denen Fuzzing mithilfe von Grammatiken vorstellt wurden (zu welchen auch das Kapitel mit dem Parsing gehört), hatten wir mehr Mühe als bei den anderen. Trotz den zum Teil mühsamen Kapiteln ist das Buch sehr gut geeignet, um sich ein Grundwissen in verschiedenen Bereichen von Fuzzing anzueignen. Und einige Kapitel sind auch für Personen gedacht, die schon etwas Vorwissen haben und sich in eine bestimmte Richtung vertiefen möchten. Für Personen, die noch keine oder wenig Erfahrungen haben, ist das Buch sicher zu empfehlen, falls sie sich mit dem Thema Fuzzing auseinandersetzen wollen. Denn zu Beginn wird viel Grundsätzliches und Wichtiges erklärt, was für ein vertieftes Verständnis von Fuzzing unabdingbar ist. Unserer Meinung nach sind für Personen, die schon sehr viel Erfahrung mit Fuzzing haben, nur einige ausgewählte Kapitel interessant, da viele Kapitel eine Einführung sind und die Grundlagen von Fuzzing erklären und die erfahrenen Personen dieses Wissen bereits besitzen. Für erfahrene Personen können aber trotzdem ein paar Kapitel interessant sein. Nämlich diejenigen, die verschiedene Techniken beschreiben, welche diese Personen bis jetzt noch nicht kannten.

15.2 Feedback an die Autoren

Wie in der Bewertung beschrieben ist Fuzzing Book sehr empfehlenswert für Anfänger und Fortgeschrittene. Die Sitemap gibt eine sehr gute Übersicht aller Fuzzing-Arten auf einen Blick. Python Beispiele sind von Vorteil, da es eine einfach zu lernende und zu verstehende Programmiersprache ist.
Es wäre gut, wenn die Sitemap auf der Startseite angezeigt wird, damit diese nicht im Fuzzing Book gesucht werden muss. Einige Kapitel sind sehr lange und könnten eventuell zusammengefasst oder auf mehrere Seiten aufgeteilt. Zusätzlich wäre eine Unterstützung oder Hinweise für Fuzzing in anderen Programmiersprachen wie z.B. C#, Java etc. nützlich, für Personen, welche in anderen Technologien oder Sprachen Fuzzing betreiben wollen.

15.3 Schätzung Lernaufwand

Der Lernaufwand für die Aneignung und Verständnis des Wissens von Mutation-Based, Greybox-Based, Grammar-Based Fuzzing und Code Coverage Themen beträgt etwa 20-30 Stunden, für Personen, welche noch keinerlei Erfahrungen mit Fuzzing haben.

15.4 Schwierige Passagen/Elemente

Das schwierigste war das Kapitel «Parsing Inputs» in welchem das Thema Parsing vorgestellt wird und ebenfalls mit den Kapiteln in denen Fuzzing mithilfe von Grammatiken vorstellt wurden (zu welchen auch das Kapitel mit dem Parsing gehört), hatten wir mehr Mühe als bei den anderen.

16 Anhang A: Glossar

.NET Framework	Das .NET Framework ist eine Entwicklungsplattform von Microsoft.

A

AFL	American fuzzy lop ist ein freier Software-Fuzzer, die genetischen Algorithmen verwendet, um die Code-Abdeckung mit Testfällen effizient zu steigern.
AFL-Engine	Hier wird das eigentliche Fuzzing von AFL durchgeführt.
Azure	Eine auf Microsoft-Diensten basierende Betriebsumgebung (auch Cloud-Computing-Plattform genannt), die es Entwicklern ermöglicht, Dienste auf der Microsoft-Infrastruktur zu erstellen und zu hosten.

B

Backend	Teil eines IT-Systems, welches sich mit der Datenverarbeitung beschäftigt
BIOS-Firmware	BIOS ist ein Akronym für Basic Input/Output System. Ist eine Firmware, welche auf dem Motherboard eines Computers in einem nichtflüchtigen Speicher befindet.
Blackbox Fuzzing	Mit Hilfe von Blackbox Fuzzing wird man auf die Generierung von Inputdaten und die Untersuchung des Verhaltens des Resultates des Systems eingeschränkt. Das Programm wird als Blackbox betrachtet.
Build	Erstellungsprozess einer Software

C

C/C++	C und C++ sind Programmiersprachen.
Callback Benachrichtigungen	Callback ist, wenn eine Funktion als Parameter eine weitere Funktion gibt und diese dann eine Meldung zurückgibt.
Code Coverage	Code Coverage gibt die Anzahl Lines oder Prozent des Source Codes an, der durch einen Test abgedeckt ist.
Compiler	Ein Tool welches den Quellcode in Maschinesprache umwandelt/übersetzt.
Coverage-based	Basierung auf Anzahl Zeilen oder Prozent des Source Codes

D

Datenbank	System, um Daten zu beschreiben, speichern und abrufen
Debug Modus	Ein Modus, um Fehler zu beseitigen
Debugging	Ein Vorgang, um Fehler zu beseitigen
Denial-of-Service	Ein Angriff, um eine Dienstblockade darzustellen, damit der Server nicht mehr antworten kann.
Deployen	Halb- oder vollautomatisierter Prozess zur Verteilung und Installation von Software
Dictionary File	Eine Datei mit Schlüsselwörtern, welche nicht verändert werden sollte.
DMZ	Eine Demilitarisierte Zone ist ein Computernetz, welches durch eine oder mehrere Firewalls geschützt ist.
Docker	Docker ist eine Container Virtualisierung der isolierten freien Software.
Docker DB	Es ist eine isolierte Container Datenbank.
DTD	Eine Dokumenttypdefinition gibt die Regel für die XML an.

E

Ensemble Fuzzing	Es verwendet mehrere unterschiedliche Fuzzer, die interessanten Eingaben zwischen den Fuzzer-Technologien austauschen, um eine bessere Leistung zu erzielen.
Entity	Eine Entität ist eine Informationsmenge zu einem definierten Oberbegriff.
Entity Framework	Ein Framework von Microsoft, welche für objektrelationale Abbildung verwendet wird.
Exceptions	Es gibt den Ausnahmezustand in einem Programm aus.
Exploit	Es zeigt die Sicherheitslücken von Software auf und ermöglicht die Ausnutzung.

F

Forkserver	Ein Server, der eingehenden Datenverkehr jeglicher Art verarbeitet, indem er Kind-Prozesse erzeugt.
Framework	Framework ist ein anderer Begriff für Rahmenwerk oder Grundstruktur für die Erstellung von Softwareprodukten
Fuzz	zufällig generierte Daten
Fuzz Testing Framework	Programmiergerüst für das Fuzzing
Fuzzer	Programm, welches zufällige Testdaten generiert und dem zu testenden Programm übergibt

Fuzzing	Testmethode, bei welcher das zu testende Programm mit zufälligen Daten getestet wird
Fuzzing Engine	Teil des Fuzzers, welcher die zufälligen Daten erzeugt
Fuzzing Workflows	Ablauf des Fuzzing-Prozesses
Fuzzlyn	Ein Fuzzer welcher Roslyn verwendet, um zufällige C# Programme zu erzeugen und dann auf .NET zu testen
Fuzzware	Tool, welches mithilfe von einer XML-Datei und einer XSD XML-Dateien generiert
Fuzzy Code	Der Code, welcher gefuzzt werden muss.

G

GCC	GNU Compiler Compilation, beinhaltet verschiedene Compiler für verschiedene Programmiersprachen
Generation-based	Daten werden ohne Seed generiert
Github	Versionsverwaltungstool
Go	Kompilierbare Programmiersprache
Grammar-based	Fuzzing Methode, welche eine Grammatik/Struktur entgegennimmt, und innerhalb der Gültigkeit dieser Grammatik die Inputs mutiert
Greybox Fuzzing	Kombination von Blackbox-Fuzzing und Whitebox-Fuzzing

H

Hardening	Sicherheit des Systems wird durch bestimmte Massnahmen erhöht
Homebrew	Paketverwaltungssoftware für MacOS und Linux

I

Instrumentierung	Prozess, bei dem eine zu überwachende oder zu testende Software so verändert wird, dass es einfacher ist sie zu überwachen oder zu testen

J

Java	Objektorientierte Programmiersprache
JSON	JavaScript Object Notation, kompaktes Datenformat in einer lesbaren Textform

K

Kontinuierliches Developer-Driven Fuzzing	Fuzzing ist in die Entwicklungs-Pipeline eingebunden und wird bei jeder Änderung der Software ausgelöst

L

LibFuzzer	Eine in-prozess, coverage-based Fuzzing Engine

M

Markup	Eine maschinenlesbare Sprache für Gliederung und Formatierung von Texten und anderen Dateien
Memory Leaks	Fehler in der Speicherverwaltung eines Programms, bei dem ein Teil des Arbeitsspeichers belegt ist, aber das Programm diesen nicht freigibt und nicht nutzt.
Microsoft SQL-Server 2019	Datenbankmanagementsystem von Microsoft
Mock DB	Einen Platzhalter in Softwaretests für die echte Datenbank
Monitoring	Überwachung verschiedener Parameter einer Software die in Betrieb ist
Mutation based	Daten werden mithilfe eines Seeds generiert und mutiert
Mutation Engine	Teil der Software, welche die Daten verändert/mutiert

O

OneFuzz	Open-Source Fuzzing Tool welches von Microsoft entwickelt wird.
OneFuzz Job	Sammlung von Fuzzing-Aufgaben.
OneFuzz Template	Vorkonfigurierter Job mit einer Reihe von Aufgaben, die die gängigsten Konfigurationen für einen bestimmten Fuzz-Job enthalten.
OSS-Fuzz	Ein von Google entwickeltes Fuzzing-Tool (Open Source Software - Fuzz)

P

Parser	Eine Applikation, die für die Zerlegung und Umwandlung einer Eingabe in ein für die Weiterverarbeitung geeignetes Format zuständig ist.
Path	Zeichenfolge, die eine Datei, ein Verzeichnis oder auch Code-Verzweigungen bezeichnet.
Path Coverage	Pfadabdeckung bezieht sich darauf, Testfälle so zu entwerfen, dass alle unabhängigen Pfade im Programm mindestens einmal ausgeführt werden.
Peach	Ein von der Firma Peach Tech entwickeltes Fuzzing-Tool.
Peach Pit	Gibt die Definition (Grammatik) für das Fuzzing mit Pech.
Policy	Richtlinien oder Verhaltensregeln für die Benutzung von Systemen.
Python	Programmiersprache.

R

Radamsa	Fuzzing-Tool.
Release Modus	Modus für den endgültigen Build der Assembly-Datei (.dll oder .exe) eines Projektes.
Repository	Verwaltetes Verzeichnis zur Speicherung und Beschreibung digitaler Objekte.

S

Seed	Bezeichnung für eine Vorlage, die als Eingabe in ein Programm gegeben wird. Von dieser Vorlage aus werden Mutationen vorgenommen.
Server-Ressourcen	Systemelemente eines Computers, die zur korrekten Ausführung benötigt werden. (Festplatte, RAM, CPU, etc.)
SharpFuzz	Fuzzing-Tool.
Solution	Ist in .NET eine Sammlung von zusammengehörenden Projekten.
SSRF	Angriff (Serverseitige Anforderungsfälschung), bei dem der Angreifer die Funktionalität eines Servers missbrauchen kann.
String	Eine endliche Folge von Zeichen aus einem definierten Zeichensatz,

T

Tenant Id	Global Eindeutiger Bezeichner (GUID), der sich vom Organisationsnamen oder Domäne unterscheidet.
Tool-Chain	Ist eine miteinander verknüpfte Sammlung von verschiedenen Tools, für die Zusammenarbeit an der gleichen Aufgabe.

U

Ubuntu	Eine Linux-Distribution, die auf Debian basiert.
Unique Crashes	Einzigartiger Absturz

V

VM	Virtuelle Maschine, eine software-technische Kapselung eines Rechnersystems innerhalb eines lauffähigen Rechnersystems.

W

WAF	Web Application Firewall ist Verfahren, das Webanwendungen vor Angriffen über http schützen soll.
Wget	Ein freies Kommandozeilenprogramm des GNU-Projekts zum Herunterladen von Dateien aus dem Internet.
Whitebox Fuzzing	Eine Fuzzing-Methode, bei dem der Fuzzer mit Wissen über das interne Funktionssystem des zu fuzzenden System entwickelt wurde.

X

XML	Extensible Markup Langugae (XML), ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten.
XML 1.0-Standard	Spezifiziert die XML-Syntax.
XSD	Schema zur Definition von Strukturen für XML-Dokumente.
XXE	Ist ein Angriff auf eine Anwendung, die XML-Eingaben, welche externe Entitäten enthalten, verarbeitet.

17 Anhang B: Literaturverzeichnis

“AFL @ Github.Com.” 2020. https://github.com/google/AFL/tree/master/docs.

Banach, Zbigniew. 2020. “XML External Entity (XXE) Attacks and How to Avoid Them.” https://www.netsparker.com/blog/web-security/xxe-xml-external-entity-attacks/.

Bütler, Ivan, and Compass Security. “XML Attacks - InfSi3 Lecture.” : 1–40.

computerweekly. 2016. “Fuzz-Testing (Fuzzing).” https://www.computerweekly.com/de/definition/Fuzz-Testing-Fuzzing#:~:text=Fuzz-Testing oder Fuzzing ist,sie zum Abstürzen zu bringen.

Crain, Adam, and Chris Sistrunk Pe. 2020. “What ’ s All the Fuzz about ? Project Robus Aegis ™ Platform.” https://www.datocms-assets.com/15028/1572885994-fuzzingwhitepaper.pdf.

De.qaz.wiki. 2020. “XML-Angriff Auf Externe Entitäten.” https://de.qaz.wiki/wiki/XML_external_entity_attack.

Dechand, Sergej. 2020. “Fuzzing – Angriff Ist Die Beste Verteidigung.” https://www.heise.de/hintergrund/Fuzzing-Angriff-ist-die-beste-Verteidigung-4659818.html?seite=all.

“Fuzzingbook.Org.” 2020. https://www.fuzzingbook.org/.

“Fuzzware.” https://www.fuzzware.net/about-fuzzware.

“Guru99.” 2020. https://www.guru99.com/fuzz-testing.html.

Hery-Moßmann, Nicole. 2018. “XML - Was Ist Das? Einfach Erklärt.” https://praxistipps.chip.de/xml-was-ist-das-einfach-erklaert_47836.

Herzog, Sascha. 2010. “XML External Entity Attacks (XXE) The OWASP Foundation Agenda Introduction.” *Exchange Organizational Behavior Teaching Journal*.

Hillman, Matt Hillman, and F-Secure. 2020. “15-Minute-Guide-to-Fuzzing @ Www.f-Secure.Com.” https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing.

IONOS. 2020. “Fuzzing: Das Steckt Hinter Der Automatisierten Test-Methode.” https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-fuzzing/.

Jakob Botsch, Nielsen. 2020. “Fuzzlyn.” https://github.com/jakobbotsch/Fuzzlyn.

Li, Jun, Bodong Zhao, and Chao Zhang. 2018. “Fuzzing: A Survey.” *Cybersecurity* 1(1): 1–13.

“LibFuzzer @ Github.Com.” 2020. https://github.com/google/fuzzing/blob/master/tutorial/LibFuzzerTutorial.md.

LLVM. 2020. “LibFuzzer.” http://llvm.org/docs/LibFuzzer.html.

Microsoft. 2020. “OneFuzz.” https://github.com/microsoft/onefuzz.

Mijailovic. 2019a. “Sharpfuzz - GitHub Repo.” https://github.com/Metalnem/sharpfuzz.

Mijailovic, Nemanja. 2019b. “SharpFuzz: Bringing the Power of Afl-Fuzz to .NET Platform.” https://mijailovic.net/2019/01/03/sharpfuzz/.

“Mozilla Wiki.” 2017. https://wiki.mozilla.org/Security/Fuzzing/Peach#Minimally_setting_up_Peach_on_Ubuntu_10.04.

“Oss-Fuzz @ Github.Com.” 2020. https://github.com/google/oss-fuzz.

OWASP. 2020. “XML External Entity (XXE) Processing.” https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing.

Peach.tech. 2020. “Peach Fuzzer.” https://www.peach.tech/products/peach-fuzzer/peach-pits/.

“Radamsa @ Gitlab.Com.” 2020. https://gitlab.com/akihe/radamsa.

Ray, Erik T., Jason McIntoshGmbH, and Data2type GmbH. 2020. “XML-Parser.” https://www.data2type.de/xml-xslt-xslfo/xml/uebersicht-zu-perl-und-xml/xml-grundlagen-lesen-und-schre/xml-parser/.

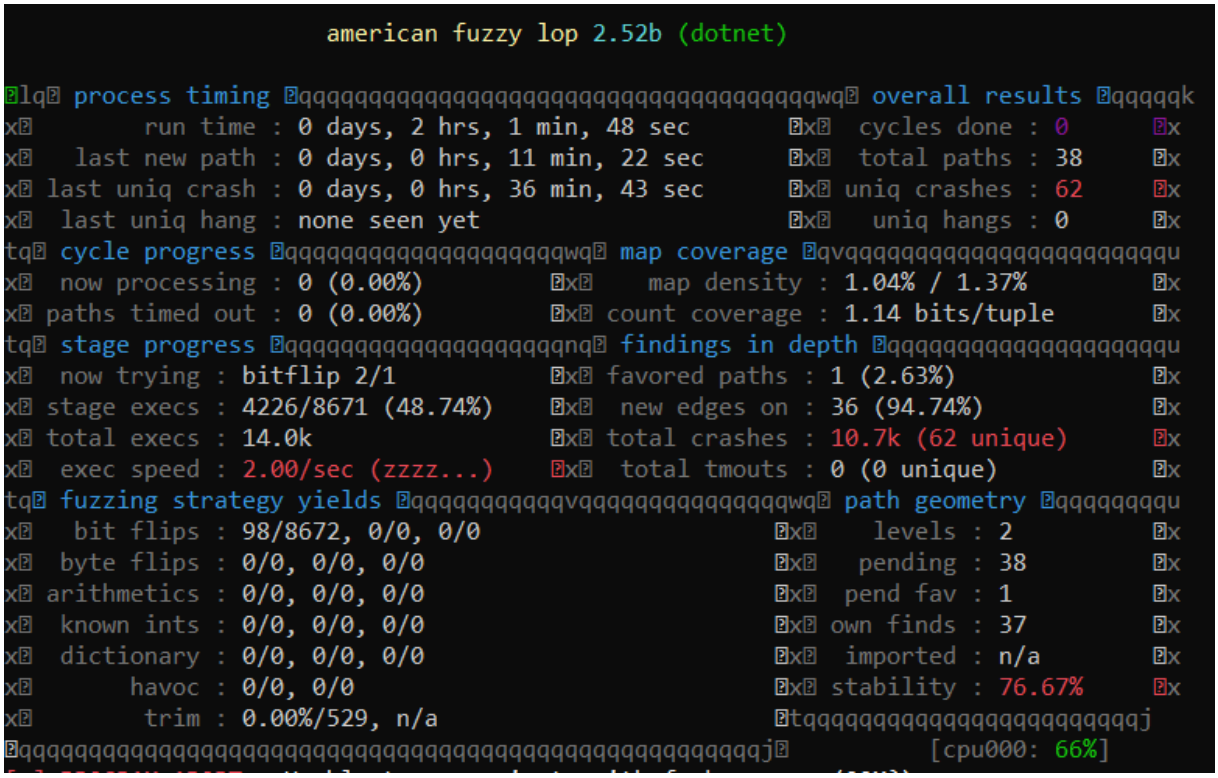
“Securai.De.” 2020. https://www.securai.de/veroeffentlichungen/blog/xxe-angriff-ueber-ein-

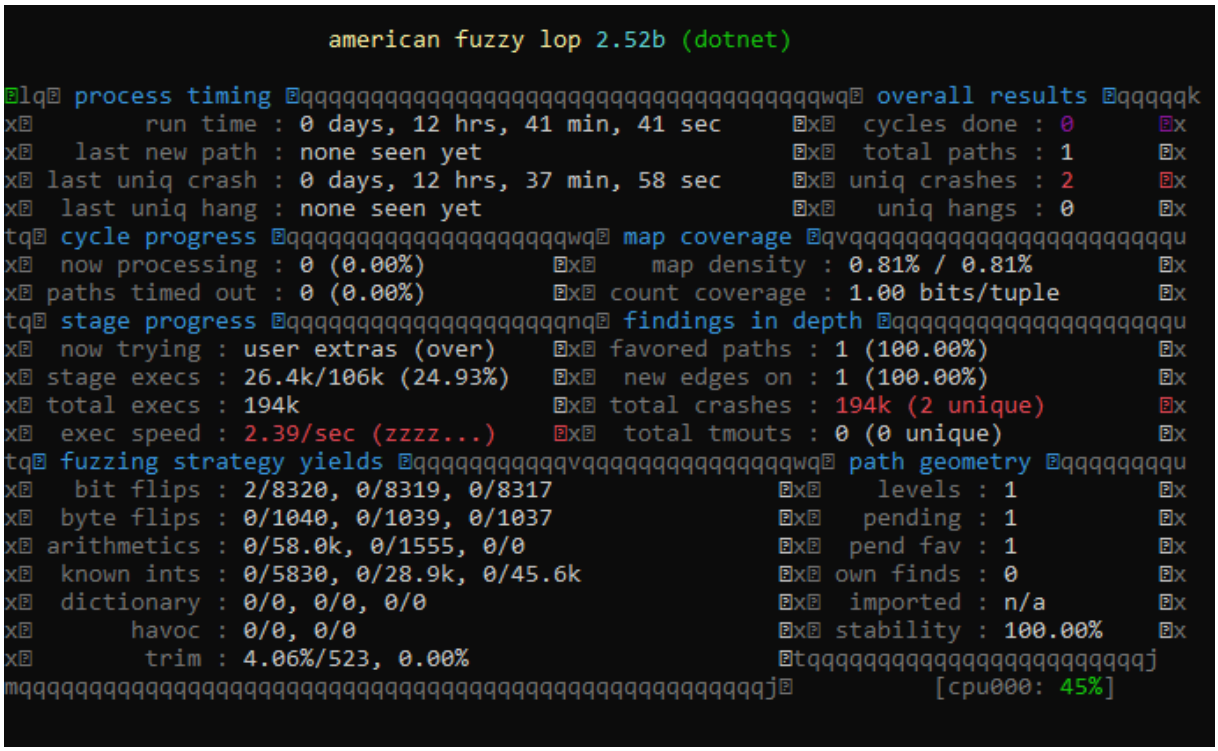
serialisierungsformat/.
Späth, Christopher, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. 2016. “Sok: XML Parser Vulnerabilities.” *10th USENIX Workshop on Offensive Technologies, WOOT 2016*. https://www.usenix.org/system/files/conference/woot16/woot16-paper-spath.pdf.
Wikipedia. 2020. “Fuzzing @ En.Wikipedia.Org.” https://en.wikipedia.org/wiki/Fuzzing.

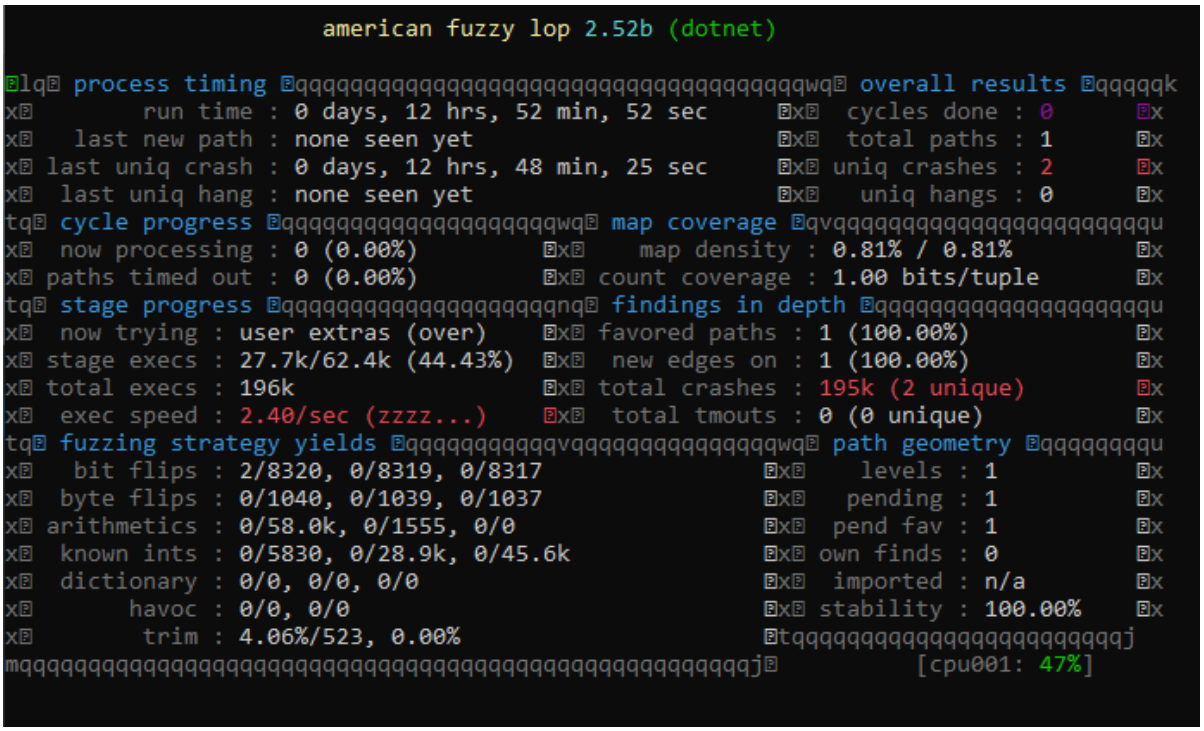
18 Anhang C: Abbildungsverzeichnis

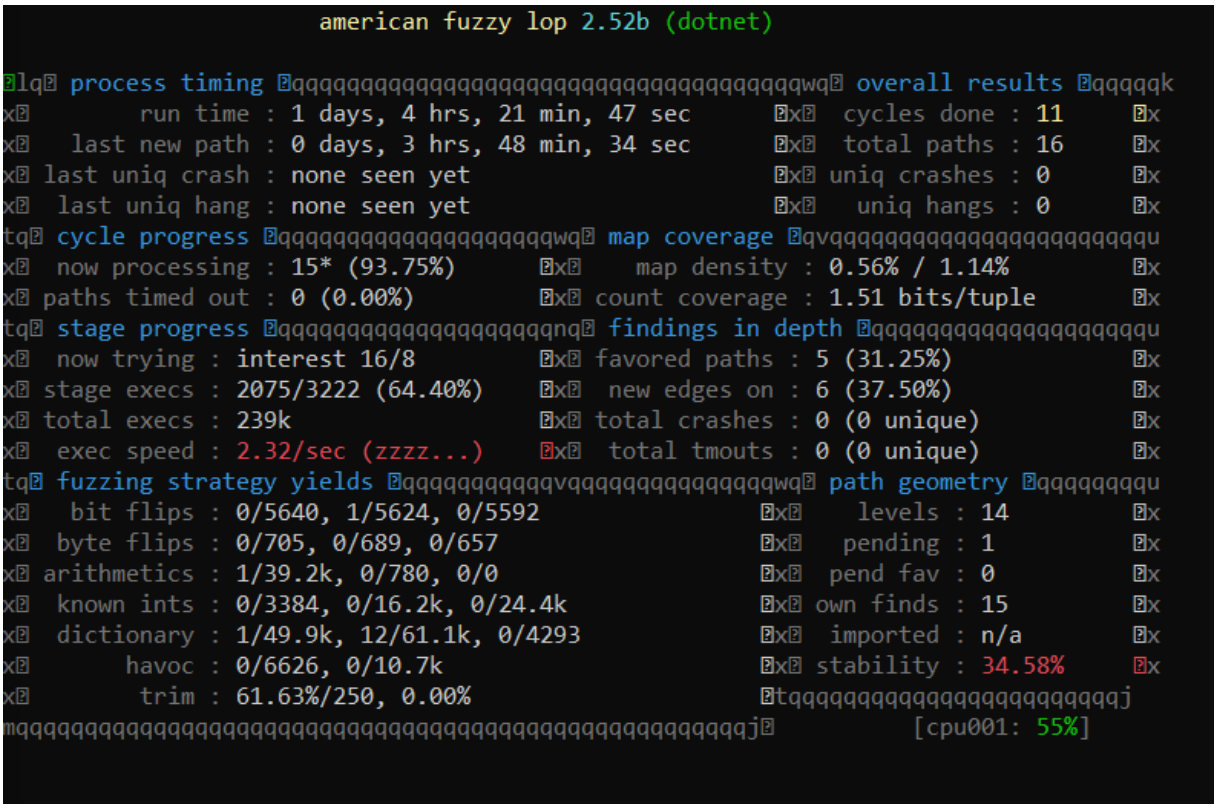
19 Anhang D: Tabellenverzeichnis

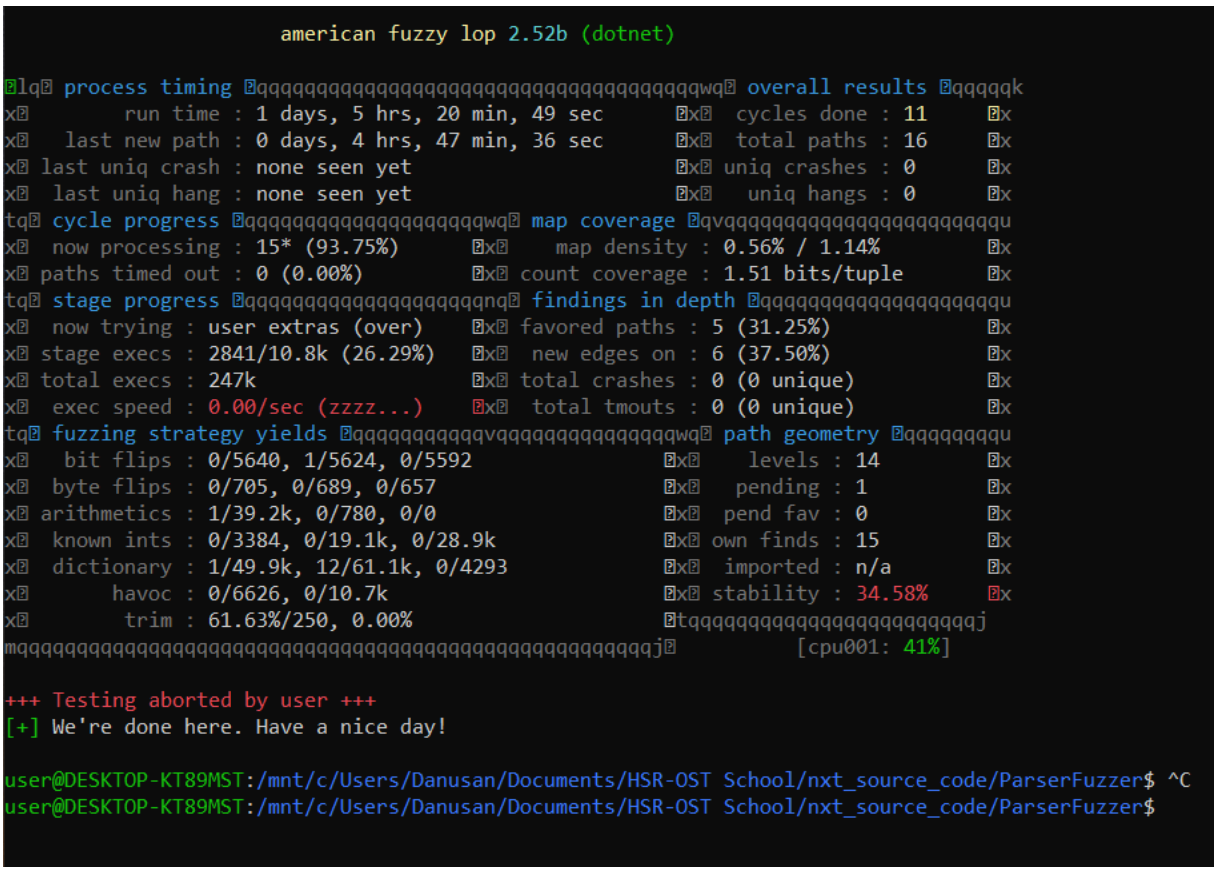
20 Anhang E: AFL Screens

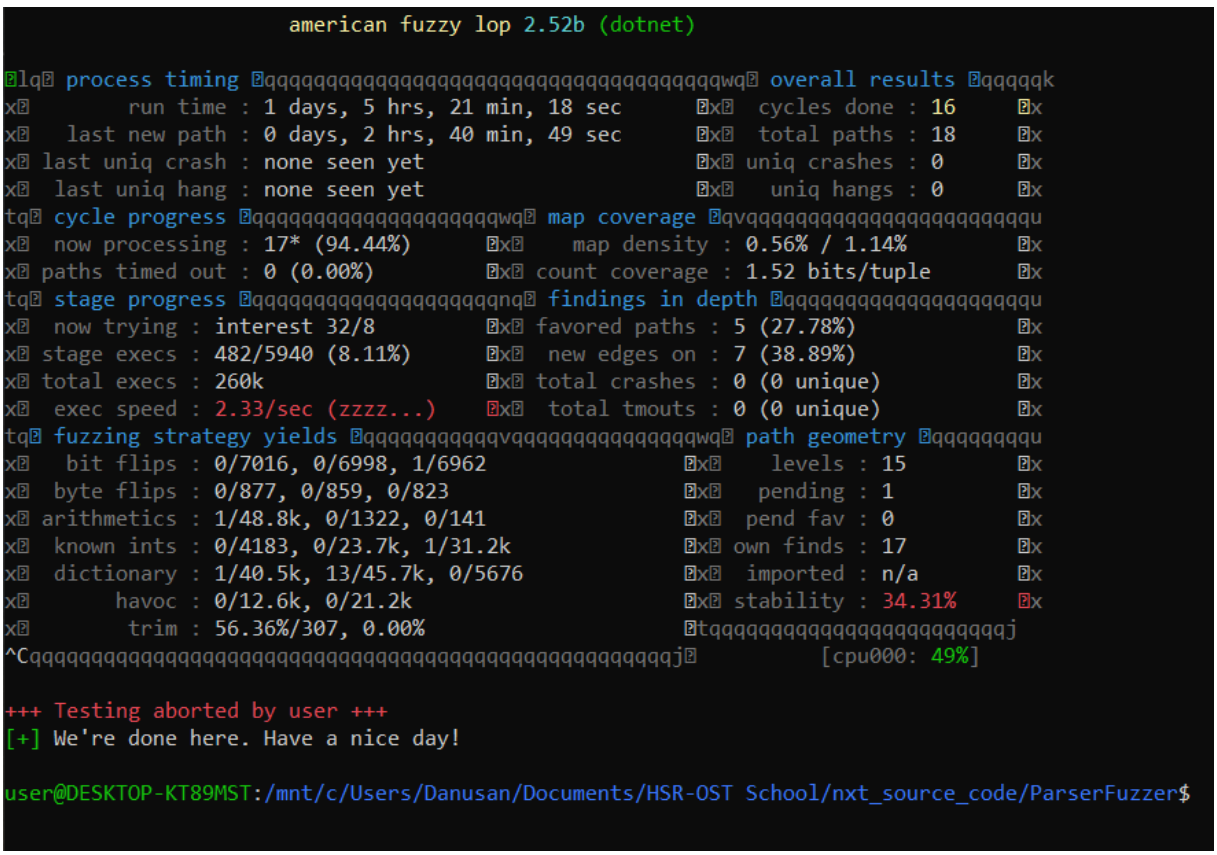





american fuzzy lop 2.52b (dotnet)
process timing
run time : 0 days, 12 hrs, 52 min, 52 sec
last new path : none seen yet
last uniq crash : 0 days, 12 hrs, 48 min, 25 sec
last uniq hang : none seen yet
overall results
cycles done : 0
total paths : 1
uniq crashes : 2
uniq hangs : 0
cycle progress
now processing : 0 (0.00%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.81% / 0.81%
count coverage : 1.00 bits/tuple
stage progress
now trying : user extras (over)
stage execs : 27.7k/62.4k (44.43%)
total execs : 196k
exec speed : 2.40/sec (zzzz...)
findings in depth
favored paths : 1 (100.00%)
new edges on : 1 (100.00%)
total crashes : 195k (2 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 2/8320, 0/8319, 0/8317
byte flips : 0/1040, 0/1039, 0/1037
arithmetics : 0/58.0k, 0/1555, 0/0
known ints : 0/5830, 0/28.9k, 0/45.6k
dictionary : 0/0, 0/0, 0/0
havoc : 0/0, 0/0
trim : 4.06%/523, 0.00%
path geometry
levels : 1
pending : 1
pend fav : 1
own finds : 0
imported : n/a
stability : 100.00%
[cpu001: 47%]


american fuzzy lop 2.52b (dotnet)
process timing
run time : 1 days, 4 hrs, 21 min, 47 sec
last new path : 0 days, 3 hrs, 48 min, 34 sec
last uniq crash : none seen yet
last uniq hang : none seen yet
overall results
cycles done : 11
total paths : 16
uniq crashes : 0
uniq hangs : 0
cycle progress
now processing : 15* (93.75%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.56% / 1.14%
count coverage : 1.51 bits/tuple
stage progress
now trying : interest 16/8
stage execs : 2075/3222 (64.40%)
total execs : 239k
exec speed : 2.32/sec (zzzz...)
findings in depth
favored paths : 5 (31.25%)
new edges on : 6 (37.50%)
total crashes : 0 (0 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 0/5640, 1/5624, 0/5592
byte flips : 0/705, 0/689, 0/657
arithmetics : 1/39.2k, 0/780, 0/0
known ints : 0/3384, 0/16.2k, 0/24.4k
dictionary : 1/49.9k, 12/61.1k, 0/4293
havoc : 0/6626, 0/10.7k
trim : 61.63%/250, 0.00%
path geometry
levels : 14
pending : 1
pend fav : 0
own finds : 15
imported : n/a
stability : 34.58%
[cpu001: 55%]


american fuzzy lop 2.52b (dotnet)
process timing
run time : 1 days, 5 hrs, 20 min, 49 sec
last new path : 0 days, 4 hrs, 47 min, 36 sec
last uniq crash : none seen yet
last uniq hang : none seen yet
overall results
cycles done : 11
total paths : 16
uniq crashes : 0
uniq hangs : 0
cycle progress
now processing : 15* (93.75%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.56% / 1.14%
count coverage : 1.51 bits/tuple
stage progress
now trying : user extras (over)
stage execs : 2841/10.8k (26.29%)
total execs : 247k
exec speed : 0.00/sec (zzzz...)
findings in depth
favored paths : 5 (31.25%)
new edges on : 6 (37.50%)
total crashes : 0 (0 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 0/5640, 1/5624, 0/5592
byte flips : 0/705, 0/689, 0/657
arithmetics : 1/39.2k, 0/780, 0/0
known ints : 0/3384, 0/19.1k, 0/28.9k
dictionary : 1/49.9k, 12/61.1k, 0/4293
havoc : 0/6626, 0/10.7k
trim : 61.63%/250, 0.00%
path geometry
levels : 14
pending : 1
pend fav : 0
own finds : 15
imported : n/a
stability : 34.58%
[cpu001: 41%]
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
user@DESKTOP-KT89MST:/mnt/c/Users/Danusan/Documents/HSR-OST School/nxt_source_code/ParserFuzzer$ ^C
user@DESKTOP-KT89MST:/mnt/c/Users/Danusan/Documents/HSR-OST School/nxt_source_code/ParserFuzzer$


american fuzzy lop 2.52b (dotnet)
process timing
run time : 1 days, 5 hrs, 21 min, 18 sec
last new path : 0 days, 2 hrs, 40 min, 49 sec
last uniq crash : none seen yet
last uniq hang : none seen yet
overall results
cycles done : 16
total paths : 18
uniq crashes : 0
uniq hangs : 0
cycle progress
now processing : 17* (94.44%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.56% / 1.14%
count coverage : 1.52 bits/tuple
stage progress
now trying : interest 32/8
stage execs : 482/5940 (8.11%)
total execs : 260k
exec speed : 2.33/sec (zzzz...)
findings in depth
favored paths : 5 (27.78%)
new edges on : 7 (38.89%)
total crashes : 0 (0 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 0/7016, 0/6998, 1/6962
byte flips : 0/877, 0/859, 0/823
arithmetics : 1/48.8k, 0/1322, 0/141
known ints : 0/4183, 0/23.7k, 1/31.2k
dictionary : 1/40.5k, 13/45.7k, 0/5676
havoc : 0/12.6k, 0/21.2k
trim : 56.36%/307, 0.00%
path geometry
levels : 15
pending : 1
pend fav : 0
own finds : 17
imported : n/a
stability : 34.31%
[cpu000: 49%]
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
user@DESKTOP-KT89MST:/mnt/c/Users/Danusan/Documents/HSR-OST School/nxt_source_code/ParserFuzzer$

21 Anhang F: Bericht für nxt Engineering

Bericht für
nxt Engineering GmbH

XML-Parser Fuzzing

Autoren: Kevin Moro,
Danusan Premananthan,
Aynkaran Sundralingam
Betreuer: Prof. Dr. Markus Stolze

1 Einleitung

Unsere Studienarbeit hat sich mit dem Auftrag befasst, einen XML-Parser, welchen wir von nxt Engineering bekommen haben, mittels der Testing-Methode "Fuzzing" auf Fehler zu prüfen. Fuzzing ist eine automatisierte Testing-Methode, mit der Schwachstellen in Applikationen, Betriebssystemen oder auch Netzwerken entdeckt werden können. Ziel des Fuzzings ist es, festzustellen, ob für alle möglichen Eingaben die notwendigen Reaktionen im Programm hinterlegt sind. Die Applikation wird dabei mit ungültigen bzw. zufällig generierten Daten, welche "Fuzz" genannt werden, überflutet, um sie zum Absturz zu bringen. Eine Schwachstelle (Sicherheitslücke oder Programmierfehler) ist gefunden, wenn ein Absturz vorliegt.

Nach einer Recherche und Evaluation der verschiedenen Fuzzing-Tools, wurden drei Tools definiert, welche am besten für das Fuzzing vom XML-Parser geeignet waren. Der XML-Parser wurde mit diesen Tools auf Fehlverhalten getestet. Die dabei erhaltenen Resultate werden in diesem Dokument zusammengefasst. Detailliertere Informationen zur Durchführung des Fuzzings, die erzielten Resultate oder auch zu den entstandene Probleme sind in unserem Hauptdokument "SA_Fuzzing_DotNet" zu finden.

2 Resultate

2.1 **Fuzzing**

Für das Fuzzing wurden zwei verschiedene Ansätze verwenden:

1. Grammatikbasiertes Greybox-Fuzzing
2. Manuelles Fuzzing

2.1.1 Grammatikbasiertes Greybox-Fuzzing

Der XML-Parser wurde hier mit einem bestehenden Fuzzing-Tool getestet. Dabei wurde ein Fuzzing-Tool verwendet, das die zu erzeugenden Testdaten aufgrund der Kenntnis, der zuvor durch die Ausführung erreichten Pfadabdeckung mutiert und versucht, diese Mutation mit einer vorgegebenen Grammatik zu unterstützen.

Um den XML-Parser mit diesem Fuzzing-Tool testen zu können, musste ein zusätzlicher Konstruktor für die Klasse "ImporterConsole" geschrieben werden, der eine gültige XML-Datei und die notwendigen Informationen (Tenant-ID) entsprechend der Anforderung des verwendeten Fuzzers übermittelt.

2.1.1.1 Resultat

Es konnten 68 Abstürze des XML-Parsers festgestellt werden, wobei diese bei genauerer Analyse des Absturzes alle auf die gleiche Ursache zurückführen.

Erhaltene Fehlermeldung:

```
Unhandled exception. System.InvalidOperationException: There is an error in
XML document.......
```

2.1.2 Massnahme

Wie in der Fehlermeldung zu sehen ist, wird der Fehlerfall bzw. die Exception "System.InvalidOperationException" vom XML-Parser geworfen, aber nicht abgefangen.

Ursache: In der Klasse "DataProcessingService", welche sich im Projekt "DataImporter.Core" befindet, wird in der Methode "GetExportTacs" dieser Fehlerfall nicht behandelt.

Mögliche Massnahme:

1. Methode "GetExportTacs" mit Try-Catch-Block ergänzen.
2. Vor dem XML-Parsing im Projekt "DataImporter.Console" die XML-Elemente validieren

2.1.3 Manuelles Fuzzing

Bei diesem Ansatz wurden die Testdateien separat durch eine Mutation-Engine, welche grammatikbasierte XML-Dateien generiert, erstellt. Dabei wurden insgesamt ca. 96'000 unterschiedliche Testdateien erzeugt. Diese wurden dann mit Hilfe eines Skripts am XML-Parser getestet und protokolliert.

Um Informationen über die mit dem manuellen Fuzzing erreichte Code Coverage zu erhalten, wurde zusätzlich ein Systemtest geschrieben. Dazu musste erneut ein zusätzlicher Konstruktor für die Klasse "ImporterConsole" geschrieben werden, der die generierten Testdateien im Rahmen eines Systemtests (xUnit) durchläuft, um die erreichte Coverage zu ermitteln.

2.1.3.1 Resultat

Es wurde kein Fehlverhalten oder Absturz des XML-Parsers festgestellt. Es wurden nur behandelte Exceptions des Entitiy Frameworks protokolliert. Diese Fehlermeldungen wurden durch zu lange Strings ausgelöst.

Abgesehen von den behandelten Exceptions ergaben sich bei diesem manuellen Fuzzing-Ansatz keine weiteren Fehler.

2.2 **XML-Parser Angriffe**

Um ein umfassenderes Ergebnis abliefern zu können, wurden neben dem Fuzzing weitere Tests durchgeführt, die im folgenden Abschnitt dokumentiert werden.

2.2.1 Theorie

Das Format bzw. die Datenstrukturdefinition von XML-Dokumenten werden von sogenannten DTD (Document Type Definition) - oder XSD (XML-Schema) -Dateien definiert. Die XML-Dokumente haben daher die Möglichkeit die URL dieser Dateien innerhalb einer DOCTYPE-Deklaration anzugeben. Nun haben Angreifer die Möglichkeit, diese Funktionalität für ihre Zwecke auszunützen. Aus diesem Grund wurde der XML-Parser auf verschiedene XML-Schwachstellen getestet.

2.2.2 Test

Auf folgende XML-Parser Angriffsarten wurde der XML-Parser getestet:

- Resource Exhaustion Attacks
- File Inclusion Attacks
- Data Extraction Attacks
- SSRF-Attacks (Server-Side Request Forgery)

Dabei wurden die Ausnutzungsmöglichkeiten von folgenden XML-Funktionalitäten getestet:

- Entity
- External Entity
- Parameter Entity
- XInclude

2.2.3 Resultat

Alle XML-basierten Angriffe bauen auf dem Hinzufügen eines DOCTYPE in einem XML-Dokument auf. Um XML-Parser-Angriffe zu verhindern, ist die hilfreichste Methode, die Funktion für die Verwendung externer Entitäten zu deaktivieren.

Die folgenden Ergebnisse wurden bei den Angriffsversuchen beobachtet:

Die Anwendung verhindert jegliche Verwendung von Entitäten (Local Entity, External Entity & Parameter Entity). Daher ist das Referenzieren und das Einbetten von schädlichen Inhalten sowie das Extrahieren von sensiblen Daten nicht möglich.

Erhaltene Fehlermeldung:

```
fail: DataImporter.Services.ImportLogger[0]

    Error while processing data

System.InvalidOperationException: There is an error in XML document (0, 0).

 ---> System.Xml.XmlException: For security reasons DTD is prohibited in this XML document. To enable DTD processing set the DtdProcessing property on XmlReaderSettings to Parse and pass the settings into XmlReader.Create method.
```

Das Einbetten von externen Dokumenten mittels XInclude ist ebenfalls unterbunden.

Erhaltene Fehlermeldung:

```
fail: DataImporter.Services.ImportLogger[0]

      Error while processing data

System.InvalidOperationException: There is an error in XML document (2, 3).

 ---> System.InvalidOperationException: <data xmlns=''> was not expected.
```

Anhand der durchgeführten Überprüfungen auf XML-Schwachstellen und der daraus resultierenden Ergebnisse kann der Schluss gezogen werden, dass der XML-Parser nicht anfällig für die getesteten XML-Verletzlichkeiten ist.