hochschule mannheim

Fakultät für Informatik
Entwicklungsprojekt CEP des Studiengang CSB

Technische Dokumentation

Abgabetermin: 29.06.2021

Vorgelegt von:

Philip Steinlein - 1921236
Mike Kittelberger - 1921224
Nicolas Biundo - 1924333
Gabriel Kremensas - 1925874
Bianca Knittel - 1921825
Simon Höfer - 1821059

Inhaltsverzeichnis

Softwarebezeichnung / Softwarename

Audit Tool - EZAudit

1. Allgemeine Beschreibung der Software

Das EZ-Audit Framework ist ein Audit Tool, um die gängigsten Betriebssysteme nach, zuvor definierten, Audit Schritten selbstständig zu analysieren und diese Ergebnisse/Artefakte in einer Datei für den Kunden zusammenzufassen.
Dabei fungiert die gelieferte Software “EZ-Audit” als Interpreter, welche die im Vorfeld von dem Kunden erstellte Konfigurationsdatei einliest und deren beinhaltenden Audit Schritte durchführt.

2. Funktionsweise der Software

Wie aus den Anforderungen des Kunden aus dem daraus formulierten Pflichtenheft zu entnehmen ist, muss die Software mobil, einfach zwischen Systemen übertragbar und ausführbar sein, als auch vor allem Betriebssystem unabhängig funktionieren. Bis auf die Konfigurationsdatei, welche von dem Kunden selbst auf den Einsatzzweck angepasst wird, ändert sich das von uns gelieferte Software Framework nicht.

Die Funktionsweise besteht darin, dass der Kunde das von uns gelieferte Framework zusammen mit der von Ihnen erstellten Konfigurationsdatei an den Auditnehmer übergeben werden kann und dieser einzig die Software ausführen muss, um die definierten Audit Schritte durchzuführen. Das Ergebnis soll vorrangig dem Kunden Auskunft über das zu überprüfende System liefern.

Dabei fungiert das von gelieferte Framework wie in der allgemeinen Beschreibung genannt, lediglich als Interpreter und beinhaltet eigenständig keine Audit Schritte, mit der Ausnahme eines internen Sanity Checks um die Umgebung zu identifizieren und mit der zugehörige Konfigurationsdatei abzugleichen.

3. Softwarearchitektur

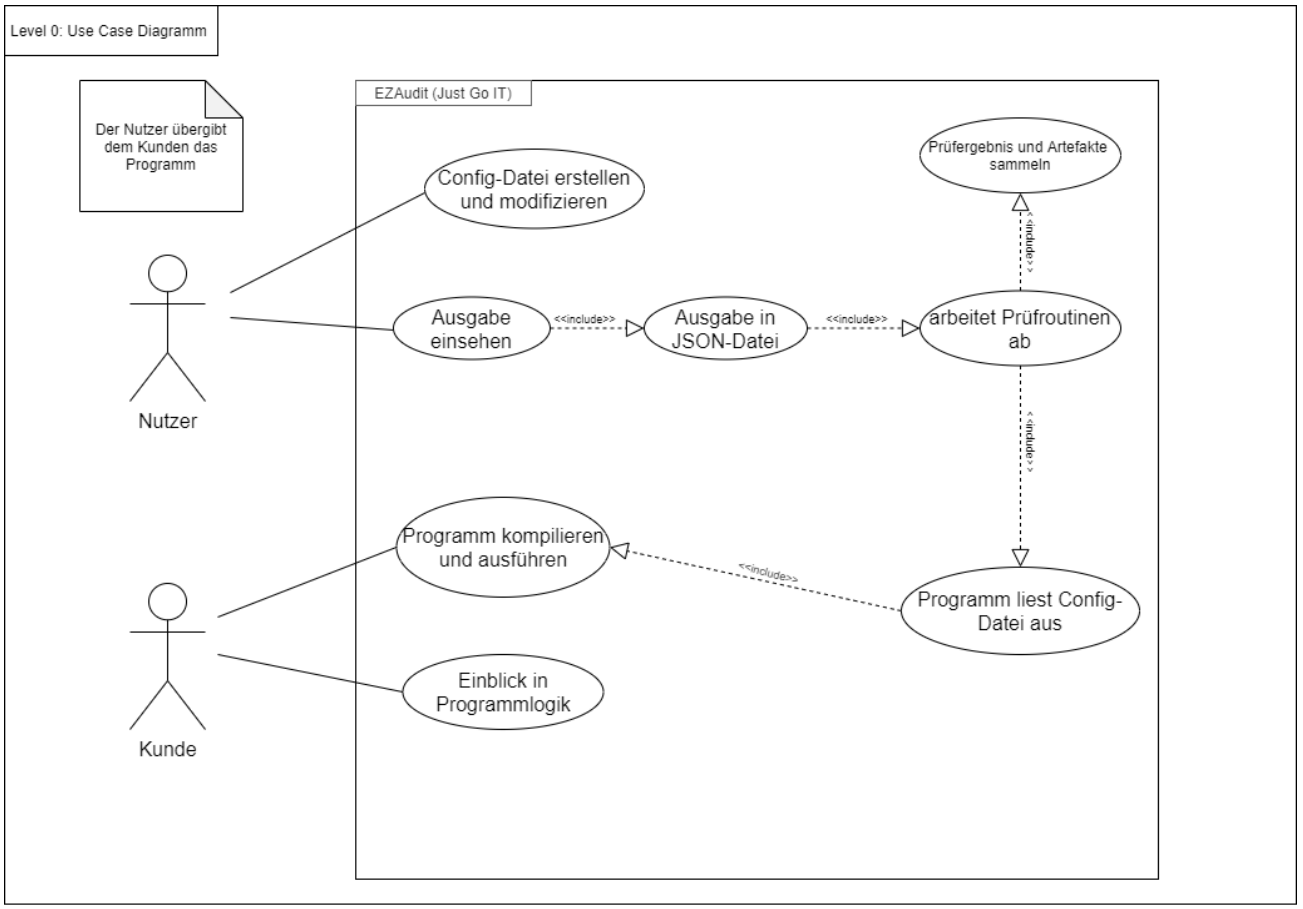
In diesem Kapitel soll ein grundlegenden Einblick in die Software Architektur hinter “EZ-Audit” ermöglicht werden.

Das untenstehende Use Case Diagramm/ Level 0 Bausteinsicht, spiegelt die in dem Kapitel “Funktionsweise der Software” beschriebenen Funktionalitäten wieder.

Der Kunde soll es möglich sein die Konfigurationsdatei (im Diagramm als Config-Datei abgekürzt) individuell auf das zu auditierende System anzupassen und an deren Kunden weiterzugeben.
Diese haben die Möglichkeit, Einblick in das Framework zu nehmen um es auf Schadsoftware zu überprüfen, danach müssen Sie lediglich EZ-Audit starten.

EZ-Audit arbeitet daraufhin eigenständig die zuvor in der Konfigurationsdatei festgelegten Audit Schritte (Prüfroutinen) ab und stellt die Ergebnisse, als auch die Artefakte zusammen und speichert diese in seperat getrennten Dateien in einem Zip-Ordner ab.

3.1 Use Case Diagramm



Der Mitarbeiter erhält das Programm und schneidet es auf das Zielsystem des Kunden zurecht. Der Kunde kann anschließend das Programm kompilieren und ausführen. Dieses arbeitet dann anhand der Config Datei die Prüfroutine ab und gibt eine Ausgabe aus. Diese Ausgabe kann dann von dem Mitarbeiter verarbeitet und ausgewertet werden.

3.2 Komponentendiagramme

Zur Veranschaulichung der grundlegenden Architektur dient das folgende Komponentendiagramm:

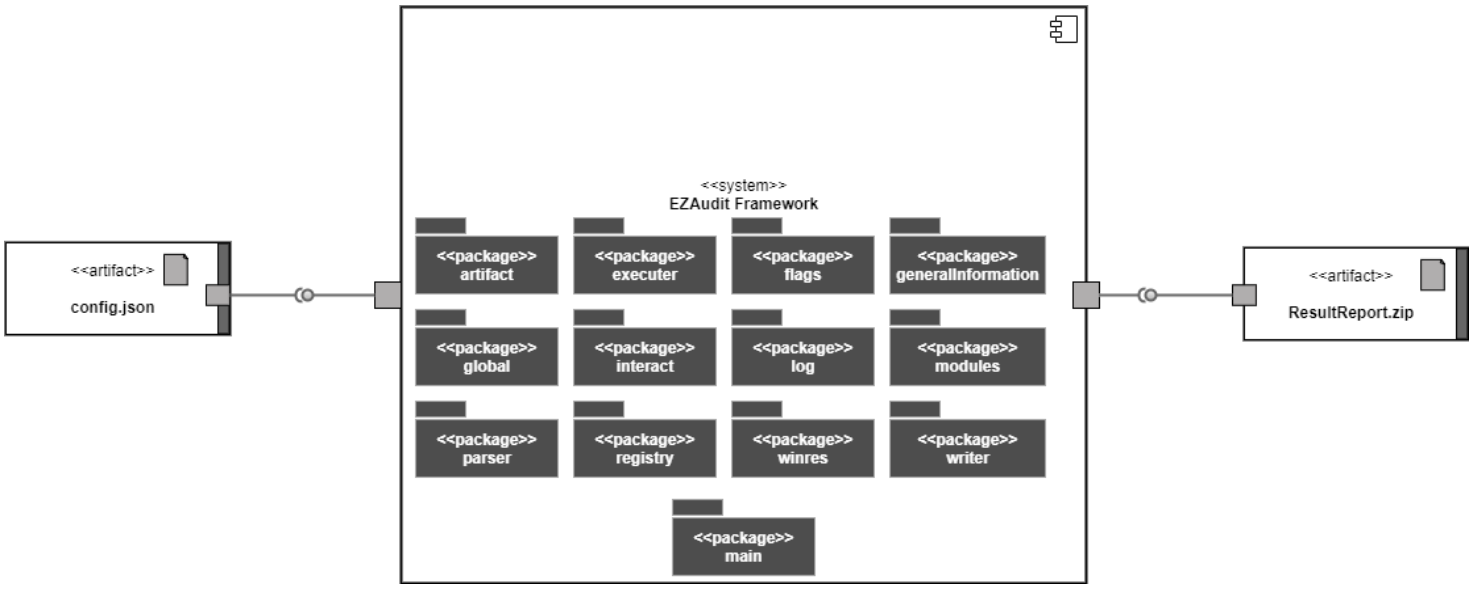
Die Software “EZ-Audit” besteht aus drei wesentlichen Bestandteilen:

1. Der Konfigurationsdatei welche die Audit Schritte beinhaltet
2. Dem “EZ-Audit” Framework welches die Rolle des Interpreters übernimmt
3. Dem Ergebnisreport welcher die Ergebnisse der Audit Schritte beinhaltet

Wie aus dem Komponentendiagramm zu entnehmen ist, steht das eigentlich entwickelte Framework in Abhängigkeit zu der Konfigurationsdatei.
Ohne diese ist keine ordnungsgemäße Funktionsweise möglich, da dem Framework jegliche Informationen der Audit Schritte und des Zielsystems zum Abgleich fehlen.
In einem solchen Fall wird der Nutzer über das Fehlen der Konfigurationsdatei hingewiesen.

Der Ergebnisbericht wird wiederum erst nach dem Durchlaufen des Frameworks erstellt, bei nicht Ausführung des Programms wird kein Ergebnisbericht erstellt.

Die einzelnen Bestandteile des eigentlichen Frameworks sind, wie in folgendem Komponentendiagramm der Ebene eins dargestellt, gegliedert:



Die im Diagramm als Package gekennzeichneten Komponenten beinhalten in diesem Framework weitere einzelne Module, welche die eigentlichen Funktionalitäten zur Verfügung stellen.
Um eine bessere Übersicht als auch eine Erläuterung der Funktionalität der einzelnen Packages zu gewährleisten dient folgende Auflistung samt Beschreibung.

3.2.1 Artifact

Hier wird der Ordner für die Artifacts erstellt. Bei Artifact werden, außerdem die Artifacts nicht nur gesammelt, sondern auch falls nötig sanitisiert.

3.2.2 Executer

Executer ist verantwortlich für die Ausführung des Audits. Hier werden parallel die Commands gestartet (Concurrency).
Die Commands laufen zwar parallel, aber die Steps innerhalb der Commands werden iterativ abgearbeitet. Der Fortschritt wird durch eine Progressbar angezeigt. Am Ende wird gewartet, bis alle Commands fertig sind, bis das Programm weiter machen darf.

3.2.3 Flags

Jegliche Flags, welche genutzen werden, werden hier definiert, mehr zu den Flags und deren Benutzung können im Benutzerhandbuch nachgelesen werden.

3.2.4 GeneralInformation

In GeneralInformation werden Daten wie das genaue Datum von der Durchführung, Laufzeitdauer, Informationen zum Betriebssystem, Informationen zu den Benutzerrechten mit welchen das Programm ausgeführt wurde, zwischengespeichert. Hier wird zwischen Linux und Windows unterschieden.

3.2.5 Global

Global bezieht sich auf die Informationen, die Allgemein für das System gelten. Namenskonvention vom Parsen der Config.json, default Dateipfade und interne Objektstrukturen (Module, Commands und Steps).

3.2.6 Interact

Interact beinhaltet zwei Go-Files, welche den gleichen Namen haben, jedoch hat eine Files das Postfix _ linux und die andere _windows. Je nachdem für welches Betriebssystem das Programm kompiliert wird, wird die zu dem Betriebssystem passende File ausgewählt. Bei Interact_windows wird bei der Funktion Shell die Powershell ausgeführt bei Interact_linux wird stattdessen die Bash Shell ausgeführt.

3.2.7 Log

Im Log wird der debug.log aufgebaut. Der Log wird abhängig von dem zuvor gesetzten Verbosity Level geschrieben.

3.2.8 Modules

Modules beinhaltet alle Module, die bisher in dem Programm für das jeweilige Betriebssystem implementiert wurden.

3.2.9 Parser

Im Parser wird geprüft, ob eine Konfigurationsdatei vorhanden ist und falls nicht wird nach einem Pfad für eine andere Konfigurationsdatei gefragt. Außerdem werden die Konfigurationsdateien eingelesen, verarbeitet und in das interne Struct geschrieben.

3.2.10 Registry

Die Registry stellt die Modulverwaltung dar. Hier wird geregelt, welches Betriebssystem ein Modul unterstützt und ob dieses Adminrechte benötigt.

3.2.11 Winres

In diesem Ordner liegen die Informationen für die zu kompilierende Windows .exe-Datei, wie zum Beispiel das Bild oder die Versionsnummer.

3.2.12 Writer

Der Writer beinhaltet Funktionen, um die gesammelten Ergebnisse zu exportieren und darzustellen.

3.2.13 Main

Die Main beschreibt den zentralen Start des Programms. Von hier aus werden die zuvor beschriebenen Packages aufgerufen.

3.3 Sequenzdiagramm

3.3.1 Main Methode

Um die Abläufe im Framework besser nachvollziehen zu können, finden sich im Anhang vier Sequenzdiagramme. Das Erste Diagramm ist dabei als grober Ablaufplan der Aufrufhierarchie zu verstehen und visualisiert die Main Methode. Da dies einen allgemeinen Überblick der Aufrufhierarchie vermitteln soll, wurde bei diesem Diagramm bewusst auf übergebene Parameter innerhalb der UML Notation verzichtet.

Die übrigen Sequenzdiagramme sind hingegen detailreicher und beinhalten auch aufgerufene Untermethoden und übergebene Parameter. Sie visualisieren in dem Framework ausgewählte Schlüsselereignisse, wie das Parsen der Konfigurationsdatei, dem Starten des Frameworks in dem Modus “DryRun”, sowie das Ausführen (“executen”) der einzelnen Module.
(s. 8.1 Sequenzdiagramm zum allgemeinen Ablauf)

3.3.2 Parser

In Diagramm zwei wird das Parsen der Module visualisiert, dabei wird zunächst geschaut, ob die Konfigurationsdatei überhaupt existiert oder gefunden werden kann. Solange dies nicht der Fall ist, wartet das Framework auf die Eingabe einer gültigen Konfigurationsdatei.
Ist diese gefunden, wird sie im Parser eingelesen, sowie mittels der in Golang eingebauten Json-Library unmarshalled und in das interne Konfiguration Struct übertragen.
Anhand dieses internen Structs, welches das Abbild der Konfigurationsdatei widerspiegelt, werden die darin enthaltenen Module versucht zu parsen. Treten dabei Fehler auf werden diese in einem Error Counter geloggt und zurückgegeben.
(s. 8.2 Sequenzdiagramm zum Parser)

3.3.3 Dry Run

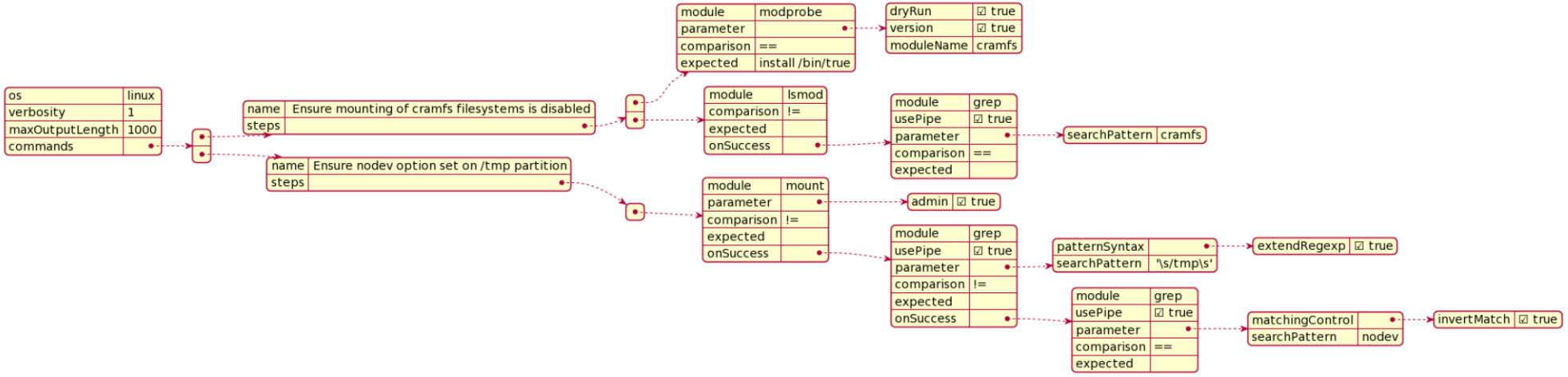
Der Modus “DryRun” welcher im dritten Diagramm gezeigt wird, setzt zunächst wie im normalen Betriebsmodus das Parsen der jeweiligen Module voraus. Ist jedoch die Flag “dryRun” gesetzt, so wird der zuvor vom Parsen zurückbekommene “Errorcounter” an die Funktion “DryRunResult” des Parsers übergeben und ausgewertet.
Ist die Auswertung abgeschlossen, so werden die spezifisch gefundenen Fehler detailliert im Log festgehalten, sowie eine kurze Zusammenfassung an den Nutzer ausgegeben. Anschließend beendet sich das Programm.
(s. 8.3 Sequenzdiagramm zum Dry Run)

3.3.4 Executer

Das Ausführen (executen) der einzelnen Module ist der wichtigste Schritt innerhalb des Frameworks. Wie in Diagramm Nummer vier dargestellt, beginnt dies damit, dass von der Main die Funktion “ExecuteAllCommands” innerhalb des Package “executer” aufgerufen wird. In dieser Funktion wird die Funktion “executeCommand” aufgerufen, welche wiederum auf die Funktion “executeStep” verweist.
Dabei ist wichtig zu erwähnen, dass innerhalb des Frameworks die Module ein Interface implementieren und so alle die selben Methoden beinhalten. In diesem Diagramm wird daher von einem unspezifischen Modul ausgegangen, welches beispielhaft an Stelle aller unterstützten Module steht. Innerhalb dieser aufgerufenen Methode werden zunächst die spezifischen Informationen des Moduls im Log gespeichert.
Danach wird die spezifische “execute” Methode des Moduls aufgerufen, welche in Abhängigkeit zu dem benötigten Betriebssystem die jeweilige Shell Funktion innerhalb des “interact” Packages zugreift.
Zudem verweist die “execute” Funktion des Moduls noch auf die “SaveString” Funktion im Package Artifact, diese ist dafür zuständig, dass die Daten, auf die zugegriffen wurden, als Artefakt gespeichert werden.
Wurde in der Konfigurationsdatei global oder lokal ein Pattern zur Sanitarisierung als Regex abgegeben, so wird sowohl das Ergebnis als auch das jeweilige Artefakt danach bereinigt.
Dabei können zwei Fälle eintreten, zum einen kann beim Executen ein Fehler auftreten, dann wird der spezifische Fehler, welches das Modul zurückliefert, im Log festgehalten. Zum anderen, bei erfolgreichem executen wird das zurückbekommene Ergebnis des Moduls im Log gespeichert.
In beiden Fällen wird, falls in der Konfigurationsdatei eine Sanitarisierung hinterlegt ist, das Ergebnis bereinigt.
Außerdem ist zu beachten, dass in der Konfigurationsdatei eine maximale Länge für das zurückgelieferte Ergebnis festgelegt werden kann, wird dieses überschritten tritt

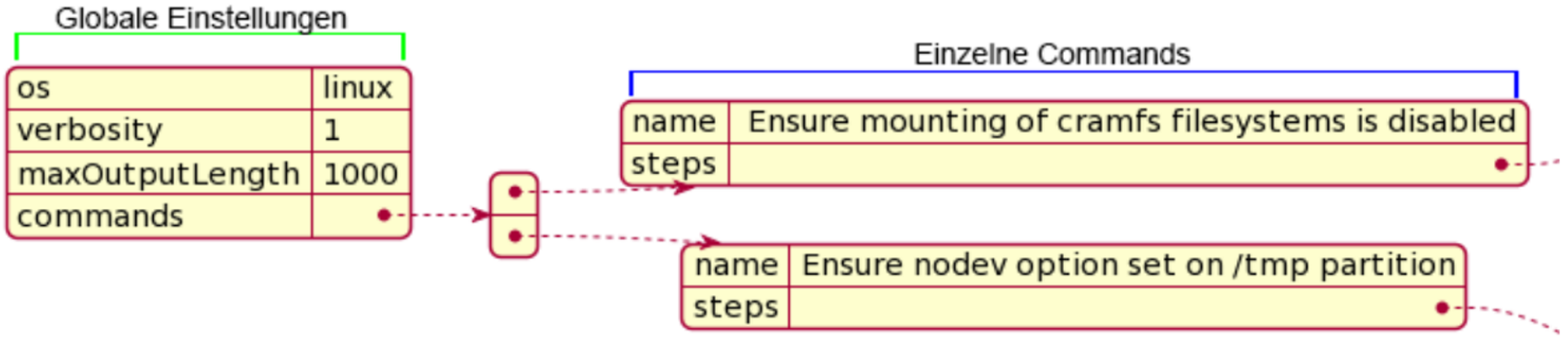
an dieser Stelle die Funktion “CheckLength” in Kraft und speichert den Output als eigenständiges Artefakt.
Dieser Ablauf wird für jeden einzelnen Step wiederholt.
(s. 8.4 Sequenzdiagramm zu ExecuteAllCommands)

4. Aufbau der Konfigurationsdatei



Dieses Diagramm dient als Überblick und wird im weiteren Verlauf konkreter betrachtet.

4.1 Konfigurationsdatei



Der Aufbau der Konfigurationsdatei besteht aus drei wesentlichen Bestandteilen:

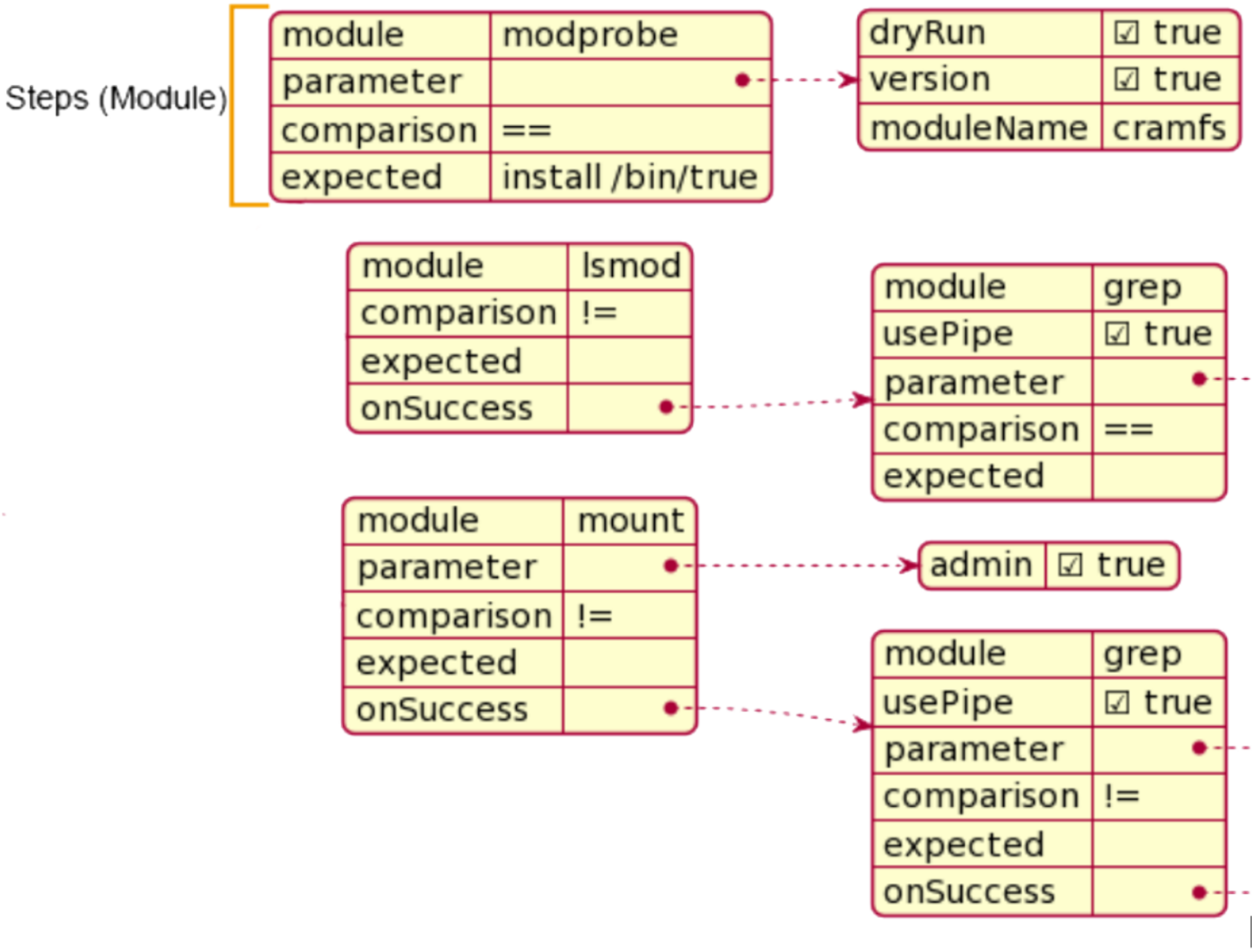
- den globalen Einstellungen
- den einzelnen Commands
- den dazugehörigen Steps mit ihren Modulen und Parametern

4.2 Globale Einstellungen

Wie in der oberen Grafik dargestellt, kann man in den globalen Einstellungen das Ziel Betriebssystem des zu auditierenden Systems, den Detailgrad der Logs (verbosity) und die maximale Output Länge bevor etwas separat als Artefakt abgespeichert wird, einstellen.

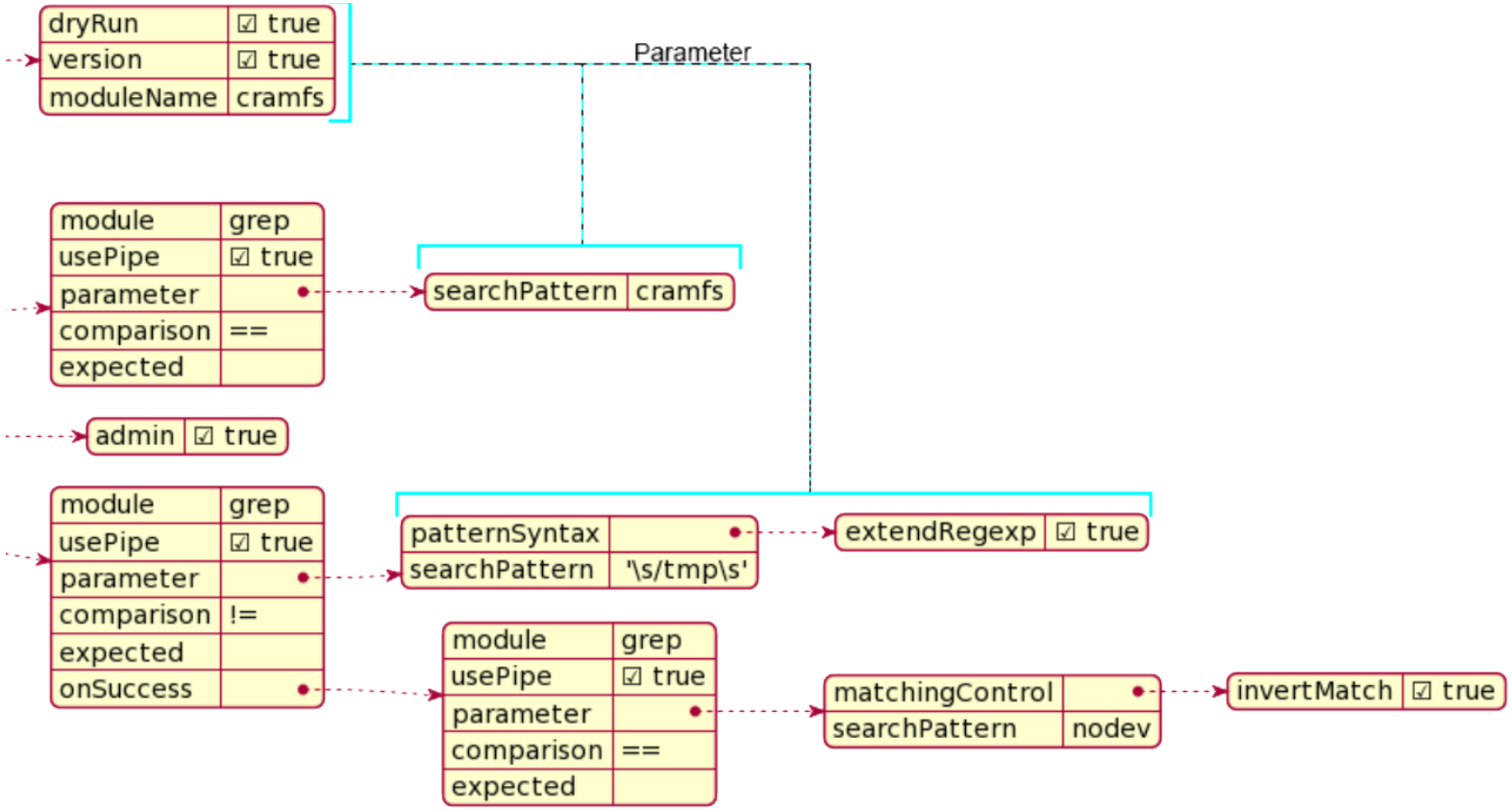
4.3 Einzelne Commands

Die jeweiligen Commands enthalten alle einen Namen und können einen oder mehrere Steps beinhalten. Dabei unterscheidet man, ob ein Step bei einem Command voneinander abhängig ist (siehe Grafik 3 “onSuccess”) oder nicht.



4.4 Step

Ein Step entspricht einem Shell/Terminal Command und spricht dabei immer direkt ein Modul an.
Die Module erwarten spezifische Parameter, um ordnungsgemäß zu funktionieren. Comparison bekommt einen Vergleichsoperator zugeordnet, so dass das tatsächliche Ergebnis mit dem erwarteten (“expected”) Wert abgeglichen werden kann.
Diese Bedingung ist sowohl für den Status eines Steps ausschlaggebend, wie in Grafik 2 & 3 zu sehen ist, als auch für die Parameter “onSuccess” und “onFailure”. Diese können wie ein If-Statement gesehen werden und bei Eintritt weitere Module aufrufen.


dryRun
☑ true
version
☑ true
moduleName
cramfs
Parameter
module
grep
usePipe
☑ true
parameter
comparison
==
expected
searchPattern
cramfs
admin
☑ true
module
grep
usePipe
☑ true
parameter
comparison
!=
expected
onSuccess
patternSyntax
searchPattern
'\s/tmp\s'
extendRegexp
☑ true
module
grep
usePipe
☑ true
parameter
comparison
==
expected
matchingControl
searchPattern
nodev
invertMatch
☑ true

4.5 Parameter

Jedes Modul kann spezifische Parameter entgegennehmen, um ordnungsgemäß zu funktionieren.
Eine Liste, welches Modul welche Parameter entgegennimmt, können Sie detailliert dem Benutzerhandbuch entnehmen.

5. Aufbau der Ausgabe:

5.1 ResultReport.json

```
"auditSummary": {
  "passedPercentage": "20.75%",
  "passed": 11,
  "failed": 25,
  "errors": 17
},
```

Im obersten Teil der JSON Datei werden allgemeine Daten ausgegeben, das beinhaltet die Prozentzahl an erfolgreich durchgelaufenen Steps, sowie die Anzahl an Steps, die Fehler hatten oder nicht erfolgreich waren.

```
 "general": {
  "date": "2021-06-22 11:33:05.02574 +0200 CEST
m=+0.024999401",
  "executionTime": "Elapsed time 6.31s",
  "interact": "Windows operating system | Product Name:
Microsoft
              Windows 10 Pro | Current Build Version: 19042 |
              Version: 10.0.19042",
  "admin": true,
  "user": " (ID: … )",
  "userName": "T490S\\..."
 },
```

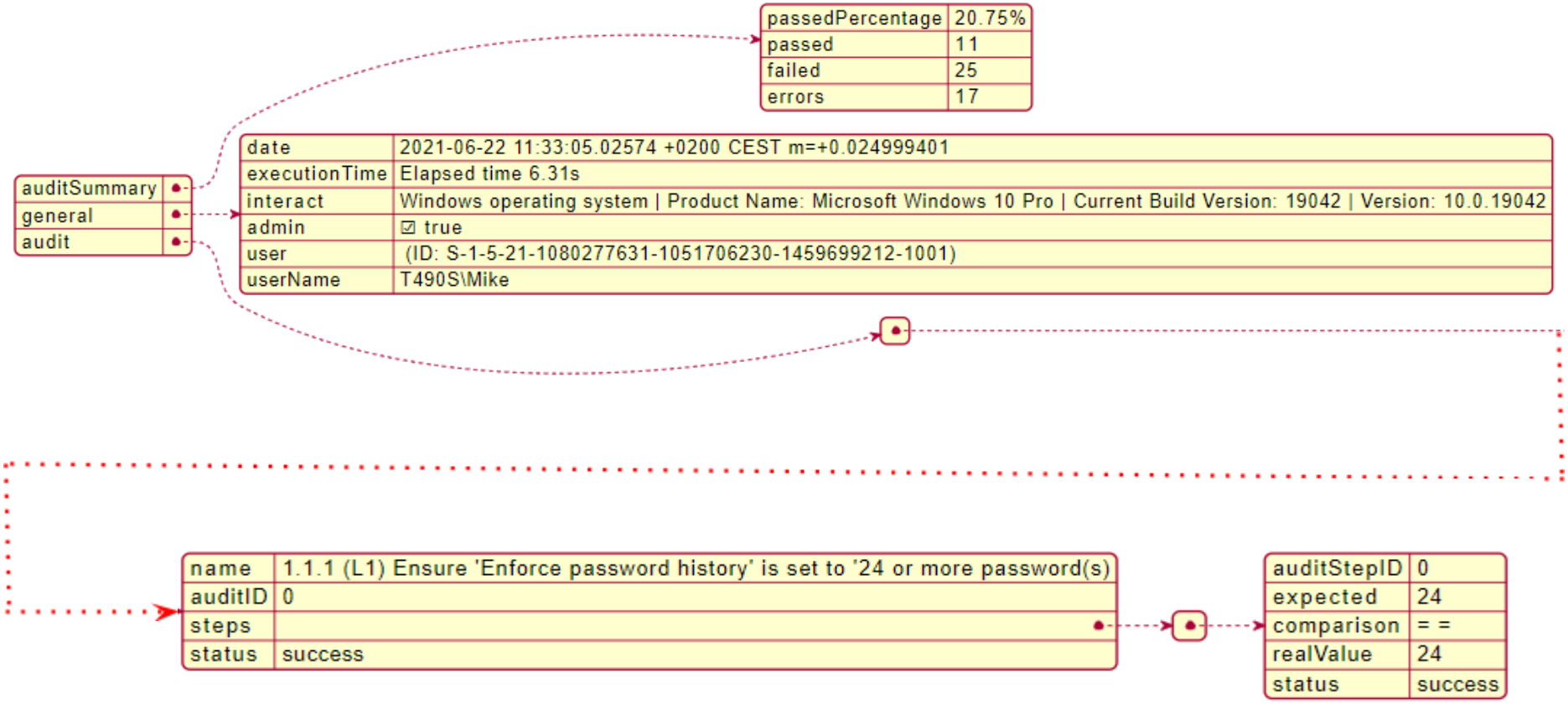
Die Infos über das auditierte System werden hier nochmal ausgegeben, damit bei der späteren Auswertung nachvollzogen werden kann, welche Datei zu welchem System gehört.

```
"audit": [
  {
   "name": "1.1.1 (L1) Ensure 'Enforce password history' is
set to '24 or more password(s)",
   "auditID": 0,
   "steps": [
```

```
    {
     "auditStepID": 0,
     "expected": "24",
     "comparison": "==",
     "realValue": "24",
     "status": "success"
    }
   ],
   "status": "success
}
```


Nun werden die auditSteps einzeln ausgegeben und die einzelnen Parameter aufgelistet, welche das System einerseits durch die Config bekam und andererseits die Vergleichswerte , die vom auditierenden System kamen.

5.2 ResultReport.json Visualisierung



5.3 Debug.log

```
INFO:   2021/06/22 11:33:05 main.go 24:
Starting with flags:
config=C:\Users\Mike\go\src\Just-Go-IT\Audit-Framework\configFiles\configW.
json, verbosity=5, noZip
INFO:   2021/06/22 11:33:05 registry.go 69:
            Config requires elevation: false
            Program ran elevated: true
INFO:   2021/06/22 11:33:05 interact_windows.go 27:
            Get-CimInstance -ClassName Win32_OperatingSystem | SELECT *
################################################################################
Command 0   1.1.1 (L1) Ensure 'Enforce password history' is set to '24 or
more password(s)
      Step 0
            DEBUG:  2021/06/22 11:33:06 executer.go 77:
                        Module &windows.Secedit:
                              Pattern:"PasswordHistorySize"
            INFO:   2021/06/22 11:33:06 secedit.go 65:
                        c:\windows\system32\secedit.exe /export /cfg
C:\Users\Mike\go\src\Just-Go-IT\Audit-Framework\EZAuditResult_22-Jun-2021_1
1-33-05\artifacts\mainResources\secedit\secedit.txt
            DEBUG:  2021/06/22 11:33:06 executer.go 96:
                        global.AuditStep:
                              ID:0,
                              Expected:"24",
                              Comparison:"==",
                              RealValue:"24",
                              Description:"",
                              Status:"success",
                              OnSuccess:(*global.AuditStep)(nil),
                              OnFailure:(*global.AuditStep)(nil)
################################################################################
```

Der Debug.log hängt vom in der Konfigurationsdatei angegebenen Verbosity-Level ab. Je nachdem, welches Level gewählt wurde, werden mehr oder weniger Informationen ausgegeben.
Es skaliert sich wie folgt:

Level 1: Fatal, loggt fatale Fehler, die das Programm beenden.
Level 2: Error, loggt zusätzlich Fehler innerhalb der Module.
Level 3: Warning, loggt zusätzlich noch Warnings im Code.
Level 4: Information, loggt zusätzlich die einzelnen Ergebnisse und die einzelnen Commands/ Audit Schritte.

Level 5: Debug, loggt alles was möglich ist, vollständige Ausgabe aller Informationen.

Nach der Globalen Ausgabe ganz oben folgt eine Reihe Hashtags und grenzt den nächsten Abschnitt ab. Dieser Abschnitt geht nun auf jeden AuditSchritt einzeln ein und gibt zu diesem eine detaillierte Ausgabe mit allen Paths und Werten zurück.

5.4 Artifacts Folder

In dem Artifacts Folder befinden sich alle Ausgaben, die zu groß für den Result Report waren und zudem sind diese numerisch sortiert.

Die Artifacts befinden sich in dem “Audit” Ordner innerhalb des “EZAuditResult” Ordners. Bei speziellen Dateien wie secedit oder auditpol, die mehrach ausgelesen werden, wird diese extra in einem Ordner “mainResources” abgelegt.
Andere Audits werden durchnummeriert und mit dem Audit Name gespeichert.
Innerhalb der Audits befinden sich die Steps. Falls hier mehrere vorhanden sind werden diese als .txt ebenfalls durchnummeriert und gespeichert.

6. Bewertung von Verfahren und Notationen

Das Use-Case Diagramm veranschaulichte die Kundenwünsche und unterstützte bei der Konkretisierung der Anforderungen. Aufgrund dessen und der schnellen Erstellung erwies sich das Use-Case Diagramm als äußerst hilfreich.

Bei Kundengesprächen, wurde vermehrt das Komponentendiagramm verwendet, um das immer komplexer werdende Programm darzustellen und den Aufbau zu verdeutlichen. Dieses Diagramm wurde zur Visualisierung der Programmarchitektur eingesetzt und diente den Developern lediglich als Überblick.

Die Sequenzdiagramme wurde hauptsächlich zu Dokumentationszwecken eingesetzt. Diese waren für die Developer und den Code nicht von weiterem Nutzen.

Weiterführend dienten die visualisierten JSONs als Bauplan, sodass unter anderem Konfigurationsdateien nach der Übergabe des EZ-Audit Frameworks weiter ausgebaut und ergänzt werden können. Zudem sind die komplexen JSON Dateien durch die Visualisierung leichter nachzuvollziehen.

Da es in Golang keine Klassen gibt, kamen keine Klassendiagramme zum Einsatz.

Außerdem hatte das Verteilungsdiagramm keinen Mehrwert, da sich das Framework ausschließlich auf der Zugriffsschicht bewegt und lediglich zwischen Framework und Kundensystem ein Austausch erfolgt. Da keine komplexen Beziehungen beziehungsweise Verteilungen dargestellt werden sollen und für die Programmentwicklung entbehrlich ist.

7. Schnittstellen aus anderen Repositories

Im Programm wird mit zwei externe Repositories gearbeitet. Zum einen die Goja Bibliothek (github.com/dop251/goja) um Scripting zu ermöglichen. Und zum anderen kommt eine externe Progressbar zum Einsatz (github.com/schollz/progressbar). Beide unterstehen einer MIT Lizenz und stehen somit jedem zur freien Verfügung.

7.1 Goja Scripting

Das Goja Scripting wird ausschließlich in dem dafür eigenen Modul Ecma verwendet und verarbeitet. Dadurch, dass innerhalb des Moduls weitere Steps aufgerufen werden können, können hier auch andere Module innerhalb des Skripts aufgerufen werden. Bei einem dry Run wird die Syntax des Skripts im Parser überprüft.

7.2 Progressbar

Die Progressbar wird ausschließlich im executer verwendet. Jedes mal, wenn ein neuer Command ausgeführt wird, wird dieser der Progressbar hinzugefügt und so füllt sie sich nach und nach, um den aktuellen Status des laufenden Audits auf der Kommandozeile auszugeben.