INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Test-Umgebung zur Evaluierung von Schwachstellenscannern

Robin Würz

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Masterarbeit

Test-Umgebung zur Evaluierung von Schwachstellenscannern

Robin Würz

Aufgabensteller:	Prof. Dr. Helmut Reiser
Betreuer:	Tobias Appel (LRZ)
Abgabetermin:	14. Juli 2020

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14. Juli 2020

.....

(Unterschrift des Kandidaten)

Abstract

Schwachstellenscanner ermöglichen es Zielsysteme auf das Vorhandensein bekannter Sicherheitslücken zu untersuchen. Studien zeigen, dass die Wahl des richtigen Schwachstellenscanners essenziell für die Qualität der Scan-Ergebnisse ist und ein falscher Scanner oder eine falsche Konfiguration unzureichende und fehlerhafte Ergebnisse erzeugen können. Unerkannte Schwachstellen können als Einfallstor für Hacker dienen und stellen somit ein Risiko für die Sicherheit einer Anwendung dar. Ein passender Scanner ist abhängig von szenariospezifischen Anforderungen. So ist nicht jeder Scanner für jeden Anwendungsfall ähnlich geeignet oder verschlechtert sogar die Scan-Resultate, wenn er falsch angewendet wird. Konkret ist ein Scanner von Datenbanken nicht ebenso gut für Scans von Webseiten geeignet.
Diese Masterarbeit entwickelt das Konzept einer universellen automatisierten
Test-Umgebung für Schwachstellenscanner, welche Unternehmen und Forschenden dabei hilft verschiedene Schwachstellenscanner miteinander zu vergleichen. Der Vergleich geschieht anhand von Qualitätsmerkmalen eines solchen Scanners. Die Erkennungsrate der vorhandenen Schwachstellen sowie Nebenwirkungen der Scans auf die Zielsysteme sind beispielhafte Merkmale. Die zu beantwortende Kernfrage lautet: "Wie sieht das Konzept einer gesamtheitlichen Test-Umgebung für Schwachstellenscanner aus und lässt sich dieses anhand einer Referenzimplementierung in der Praxis anwenden?"
In dieser Arbeit werden zunächst Anforderungen an eine solche Test-Umgebung aus Anwendungsfällen abgeleitet, welche im Folgenden die Grundlage eines Gesamtkonzepts bilden. Eine anschließende Referenzimplementierung zeigt, wie eine Test-Umgebung es einem Anwender ermöglicht eine informierte Entscheidung bezüglich eines passenden Schwachstellenscanners zu treffen.

Abstract

Vulnerability scanners provide the means to test for existing security flaws in target systems. Studies show that the choice of the right vulnerablilty scanner is essential regarding the quality of the scan results and that a wrong scanner or configuration leads to insufficient and faulty outcomes. Unrecognized security flaws serve as gateways for hackers and constitute a risk for the security of an application. An apt scanner is dependent of scenario specific requirements. That is why not every vulnerablility scanner fits every use-case and in the worst case even impairs the scan-results. Therefore a scanner for databases is not necessarily suitable for the scan of websites and vice versa. This masters thesis develops a concept of an universal automated testbed for vulnerablilty scanners that helps companies and researchers compare different vulnerablility scanners. The comparison takes place regarding serveral quality characteristics of such a scanner. The detection rate of present vulnerabilities as well as side effects of a scan on a target system are exemplary characteristics. The key question of this thesis is as follows: "What are the features of a whole testbed for vulnerability scanners and can they be realized through the development of a prototype?"
This thesis first states the requirements for such a testbed, inferred from use-caes. Those furthermore serve the purpose of a basis for an overall concept. A subsequent reference implementation shows, how a testbed facilitates a user to make an informed desicion regarding an appropriate vulnerability scanner.

Inhaltsverzeichnis

1 Einleitung und Fragestellung

Hacker verfolgen typischerweise das Ziel an einem Opfersystem eine oder mehrere der drei Säulen der Informationssicherheit zu verletzen. Diese sind unter dem Akronym **CIA** bekannt und umfassen die Vertraulichkeit, Integrität und Verfügbarkeit (eng. confidentiality, integrity and availability). Dabei kann das Entwenden von vertraulichen Informationen, das Schädigen oder die Verhinderung der Verfügbarkeit des angegriffenen Systems Absicht der Angreifer sein. Als Ziele kommen meist "IT-Anwendungen (Webanwendungen, Mailserver) beziehungsweise die zugrundeliegenden Trägersysteme (Betriebssysteme, Datenbanken)" ([Inf16a]) in Frage.
Die Angreifer - die **Cracker** oder **Black-Hat-Hacker** [Ste18]- bedienen sich unterschiedlicher Methoden um Informationen über das Ziel in Erfahrung zu bringen, in Systeme einzudringen und deren Verbleib im System zu sichern. Ein klassischer Angriff läuft folgendermaßen ab und wird in der Literatur häufig als **Kill-Chain** (vgl. [Ste18]) bezeichnet:

1. Auskundschaften des Opfers
2. Ausbringen des Schadcodes
3. Ausführen des Schadcodes
4. Laterale Bewegung im System oder Netzwerk (infizieren weiterer Systeme)
5. Exfiltration von Informationen
6. Verwischen der Spuren

Ein Beispiel eines klassischen Angriffes stellt die Analyse des Vorgehens der Gruppe **Lotus Blossom** dar, welche mithilfe von falschen Einladungen zu einer Sicherheitskonferenz in Palo Alto Zugriff auf die Systeme ihrer Opfer erlangt haben. Diese bestanden dabei hauptsächlich aus IT-Sicherheitsexperten, was verdeutlicht, dass selbst Spezialisten nicht vor raffinierten Angriffen gefeit sind (vgl. [Pag16]).
Mithilfe von Exploits werden dabei gezielt Schwachstellen im System ausgenutzt, um in dieses einzudringen. Schwachstellen sind meist seit der Entwicklung im System enthalten (Zero-Day-Exploits) und wurden entweder von den Herstellern nicht entdeckt oder ein angebotener Fix (Hard- oder Software Update) wurde bisher von den Anwendern der Software noch nicht eingespielt.
Im Gegensatz dazu arbeiten **Ethische- oder White-Hat-Hacker** [Ste18] als externe oder interne IT-Sicherheits-Angestellte, die Systeme auf Schwachstellen testen und Fehler im System aufdecken. Diese werden dokumentiert und an die Auftraggeber berichtet. Sie führen unter anderem Penetrationstests durch, also Scans des Zielsystems, welche das Vorgehen von Angreifern nachstellen, um die gravierendsten Schwachstellen aufzudecken und potenzielle Angriffsvektoren zu identifizieren [Dre13]. Das durchführende Individuum ist der Penetrationstester. Es gilt dabei zu bedenken: Perfekte Sicherheit existiert nicht und auch ein Penetrationstest wird nicht alle möglichen Schwachstellen identifizieren.
Nachdem Penetrationstester den Angriffsweg der Black-Hat-Hacker nachstellen, verwenden

sie ähnliche Methoden, um Schwachstellen aufzudecken. Um Zeit einzusparen und Aufwand zu verringern, bedient man sich Schwachstellenscannern, um automatisiert die Systeme auf bekannte Fehler zu untersuchen, ohne dass der Sicherheitsmitarbeiter diesen Prozess per Hand durchführen muss. Scans können parallel zu weiteren Untersuchungen stattfinden. Die dabei gewonnene Zeit bleibt für Detail-Analysen, zusätzliche Tests oder Dokumentation. [AV10]
Neben passiven Scans gehen einige Schwachstellenscanner teils einen Schritt weiter und ermöglichen es, die auf dem Zielsystem gefundenen theoretischen Schwachstellen auszunutzen. Dabei bedient man sich Exploits aus speziellen Datenbanken (z.B. VulDB, exploit-db, cvedetails), um das System aktiv zu prüfen.
Schwachstellenscanner sind jedoch nicht die Universallösung für Penetrationstests sondern weisen ebenso Schwächen auf. Diese können logische Schwächen, wie das Vorhandensein von Falsch-Positiven und -Negativen Ergebnissen sein, welche die Verlässlichkeit von Scans vermindern und diese weniger aufschlussreich machen (siehe: [TTSS14a]).
Aber auch technische Schwächen wie beispielsweise eine schwache Crawling-Abdeckung (siehe: 3.2, [ERRW18]), sind ein Aspekt, welcher die Effektivität von Scannern schmälert. Schwachstellenscanner verwenden in verschiedenen Konfigurationen teils Exploits, um neben einfachen Scans auch aktiv in Systeme einzudringen. Dabei kann die Integrität und Verfügbarkeit der Testmaschinen nicht zwingend gewährleistet werden, was vor allem in Live-Umgebungen unerwünscht ist. Nebenwirkungen könnten unentdeckt bleiben und unerwünschtes Verhalten zur Folge haben. Dieses Problem existiert, da Scanner teils Exploits nutzen, welche zuvor nicht getestet wurden oder im speziellen Anwendungsfall unerwartet reagieren.
Kennzahlen welche die Qualität eines Schwachstellenscanners bestimmen sind in der Literatur gut untersucht [ERRW18] [AV10] [SS10] und unterscheiden eine große Fülle an Merkmalen von denen die Erkennungsrate nur Eine ist. Ebenfalls wird deutlich, dass die Qualität der Scan-Ergebnisse zu großen Teilen von dem Anwendungsfall abhängen, also die Resultate je nach verwendetem Scanner und zugehöriger Konfiguration unterschiedlich ausfallen [ERRW18] [LS10].
Möchte ein Anwender somit eine Entscheidung bezüglich eines für seinen Use-Case passenden Schwachstellenscanners treffen oder Forschungsfragen bezüglich der Funktionsweise und Qualität eines solchen Scanners beantworten, ist eine **Test-Umgebung für Schwachstellenscanner** sinnvoll.
Diese ermöglicht es dem Anwender verschiedene Schwachstellenscanner anhand von subjektiven Kriterien miteinander zu vergleichen, deren Funktionalität zu bestätigen oder Nebenwirkungen von Scans zu bestimmen.
Generell gilt - Eine Test-Umgebung für Schwachstellenscanner bietet einen Ort, an dem gebündelt verschiedene Tests durchgeführt werden können, welche es dem Anwender ermöglichen eine Einschätzung oder Entscheidung bezüglich eines oder mehrerer Scanner zu treffen. Diese kann getroffen werden, indem die Umgebung entsprechend der Anforderungen oder Forschungsfragen des Anwenders konfiguriert wird. Als mögliche Konfigurationen kommen das absichtliche Einbauen von Schwachstellen auf den Zielsystemen, das Einbinden eigener Test- und Scan-Maschinen sowie das Verwenden von anwendungsspezifischen Scan-Skripten in Frage.
Im Laufe der Arbeit werden die Schwachstellenscanner, von welchen Angriffe ausgehen als **Scanner**, sowie Maschinen, welche Ziele dieser Scans sind als **Vulnerables** bezeichnet.

Forschungsfrage der Arbeit

Diese Arbeit behandelt die Entwicklung und Evaluation einer Test-Umgebung für Schwachstellenscanner, wie sie in vorherigem Abschnitt grob beschrieben wurde.
Es ergeben sich dabei folgende Fragestellungen:

- Welche Anforderungen werden an eine Test-Umgebung für Schwachstellenscanner gestellt?
- Wie sieht ein gesamtheitliches Konzept einer Test-Umgebung für Schwachstellenscanner im Detail aus?
- Welche Schnittstellen und Testparameter müssen für eine funktionale Test-Umgebung zur Verfügung gestellt werden, um eine angemessene Handhabung und Erweiterbarkeit gewährleisten zu können?
- Welche existierende Software kann bei der Entwicklung der Test-Umgebung zu Hilfe genommen oder zur Vergrößerung des Funktionsumfangs eingebunden werden?
- Lässt sich das erstellte Konzept erfolgreich in einen Prototypen überführen, welcher die Anforderungen der Test-Umgebung erfüllt?

Aufbau der Arbeit

Die Arbeit weist folgende Struktur auf:
In Kapitel 2 werden zunächst alle notwendigen theoretischen Grundlagen der Arbeit wie Definitionen und Begriffsabgrenzungen und definiert einige Use-Cases.
Im Weiteren folgt eine aus den Szenarien gewonnene Beschreibung aller Anforderungen (Kapitel 3), welche für eine Test-Umgebung von Schwachstellenscannern notwendig sind.
Nachdem alle Anforderungen definiert sind findet eine Marktanalyse statt, um bereits existierende Dienste und deren Konzepte daraufhin zu untersuchen, inwiefern diese hilfreich bei der Entwicklung einer Test-Umgebung sein können.
In Kapitel 4 folgt die Definition eines Gesamtkonzepts, sowie eine detailliertere Ausarbeitung notwendiger Schnittstellen und Testparameter auf Basis der zuvor gewonnenen Anforderungen.
Schließlich folgt eine prototypische Referenzimplementierung der definierten Test-Umgebung mithilfe einer Kombination eigener und bestehender Komponenten (Kapitel 5).
Gegen Ende bildet eine Bewertung des Prototypen anhand der zuvor definierten Anforderungen mithilfe einer manuellen Auswertung der Testergebnisse und deren subjektiven Aussagekraft (Kapitel 6) die Überleitung zum Gesamt-Fazit der Arbeit (Kapitel 7).

2 Grundlagen und Use-Cases

Dieses Kapitel erläutert zunächst einige Grundbegriffe und Konzepte, welche die wissenschaftlichen Grundlagen für das Verständnis dieser Arbeit darstellen.
Die Motivation dieser Arbeit basiert auf einer Reihe von Anwendungsfällen, welche teils bereits in der Einleitung angeschnitten wurden. Dieses Kapitel konkretisiert diese weiterhin anhand von Szenarien, aus welchen im Kapitel 3 Anforderungen an eine Test-Umgebung abgeleitet werden.

2.1 Grundlagen der Arbeit

Schwachstellen vs Exploits

Viele Systeme weisen Schwachstellen auf, also "sicherheitsrelevante Fehler eines IT-Systems oder einer Institution"[Inf19], die unter den richtigen Voraussetzungen von einem Angreifer ausgenutzt werden können. Damit erlangen diese Zugriff auf verschiedene Aspekte des Zielsystems und können die Vertraulichkeit, Integrität oder Verfügbarkeit des Systems verletzen. Befinden wir uns im IT-Umfeld, so werden diese zu IT-Sicherheitslücken.
In Abgrenzung dazu ist ein Exploit eine "Befehlsfolge, um bekannte Sicherheitslücken auszunutzen. Dabei handelt es sich im Allgemeinen um Skripte, die von verschiedenen Quellen, häufig frei über das Internet zur Verfügung gestellt werden" [Inf16b]. Schadsoftware ist in diesem Kontext ein Programm, welches einen oder mehrere Exploits, sowie eine dokumentierte Bedienungsanleitung besitzt oder unter Umständen eine Binärdatei umfasst und eine definierte Funktion aufweist. Der Exploit nutzt somit zunächst ohne weitere Absicht Schwachstellen im System aus und eine Schadsoftware integriert Exploits, um einen Zweck (z.B. finanziellen Gewinn) zu dienen. Schwachstellen im System können unter anderem durch Penetrationstests erkannt werden.

Penetrationstests

Unter einem Penetrationstest versteht man "den kontrollierten Versuch von 'außen' in ein bestimmtes Computersystem bzw. -netzwerk einzudringen, um Schwachstellen zu identifizieren" [EY03]. Der Ablauf eines Penetrationstests kann in verschiedene Phasen gefasst werden, wie sie in Grafik 2.1 gezeigt werden. Dabei können Parallelen zur Killchain (vgl. 1), also dem Vorgehen der Angreifer erkannt werden.
Fünf Phasen beschreiben den Penetrationstest: Recherche nach Informationen über das Zielsystem, Scan der Zielsysteme auf angebotene Dienste, System- und Anwendungserkennung, Recherche nach Schwachstellen und Ausnutzen der Schwachstellen [EY03].

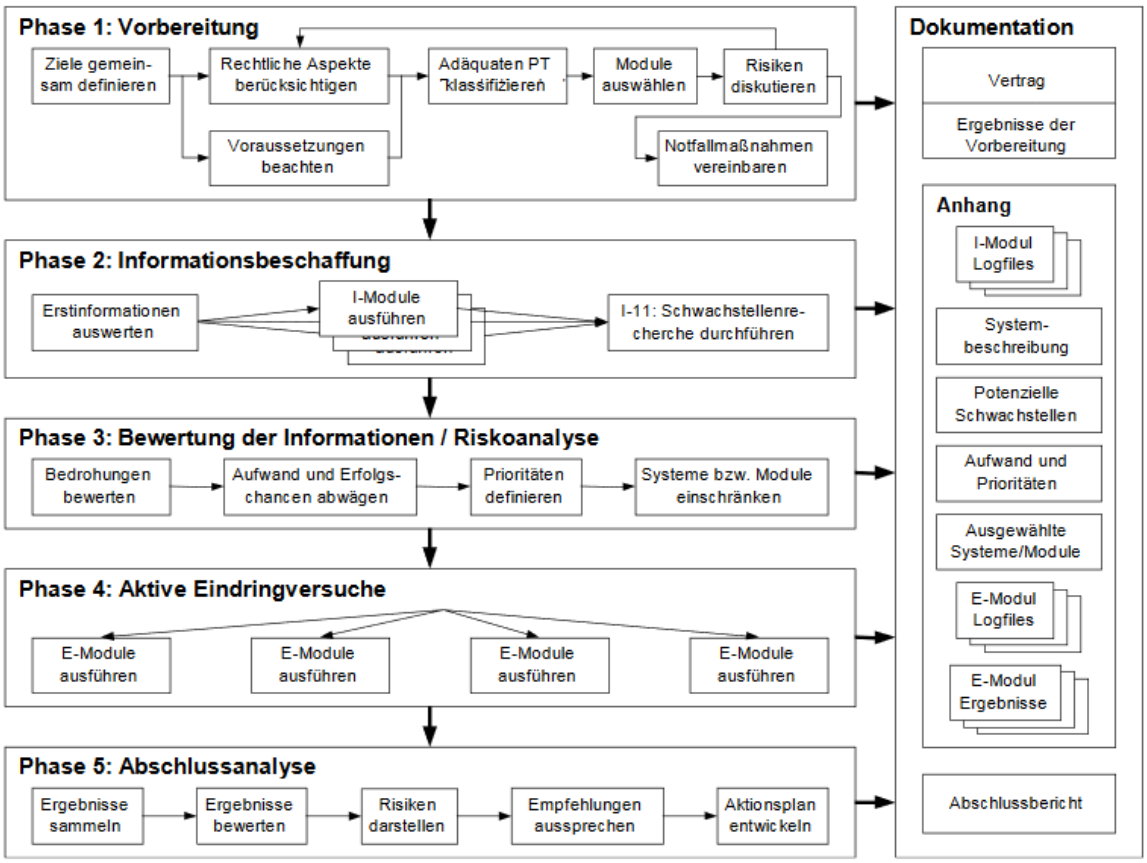

Abbildung 2.1: Die 5 Phasen eines Penetrationstests nach einer vom BSI durchgeführten Studie [EY03]

Penetrationstests werden von Firmen in Auftrag gegeben, um ihr System auf Schwachstellen bezüglich der Informationssicherheit zu Überprüfen. Dabei werden entweder interne Penetrationstester angestellt oder die Arbeit extern ausgelagert. Penetrationstests haben verschiedene Ziele:

- Sie testen ein System auf bekannte Schwachstellen und identifizieren Sicherheitslücken
- Sie dienen als Mittel des Risikomanagements, indem sie Schwachstellen und deren Eintrittsrisiko verdeutlichen
- Sie dienen als Nachweis für Auditoren, welche eine Firma nach bestimmten Standards Zertifizieren sollen (z.B. ISO27001) genauer ebenfalls beschrieben in Kapitel 12.6 der ISO/IEC 27002 [ISO17]
- Sie dienen vor Gericht im Schadensfall zum Nachweis eines angemessenen Umgangs mit der Informationssicherheit
- Der BSI Grundschutz beschreibt Penetrationstests als angemessenes Mittel im Zuge der 'Überprüfung der Eignung und Wirksamkeit von Sicherheitsmaßnahmen' [Inf19].

Diese Arbeit lässt sich in die Phasen 2 & 4 einordnen. Phase 5 wird ebenfalls angeschnitten.

Schwachstellenscanner

Penetrationstester können durch Schwachstellenscanner in ihrer Arbeit unterstützt werden. Diese stellen Tools dar, welche automatisiert Zielsysteme auf die bekanntesten Schwachstellen untersuchen. Schwachstellen werden öffentlich in Datenbanken dokumentiert (z.B. National Vulnerablilty Database [Com20]) und die Qualität der Scan-Ergebnisse ist abhängig von der Aktualität und Vollständigkeit dieser Datenbanken.
Ein Zielsystem kann auf verschiedene Arten überprüft werden. So können zum einen Informationen über das Zielsystem beschafft werden, andererseits aber ebenfalls Schwachstellen gefunden werden. Theoretisch gefundene Schwachstellen können ebenfalls mithilfe von Schwachstellenscannern ausgenutzt werden. Dabei bedient man sich Exploits aus speziellen Datenbanken (z.B. VulDB, exploit-db, cvedetails), um das System aktiv zu prüfen.

Passiver vs Aktiver Schwachstellenscan

Bei einem passiven Schwachstellenscan scannt ein Prüfer "mit eigenen Geräten im Zielnetz nach bekannten Schwachstellen. Er setzt hierzu Schwachstellenscanner ein, nutzt die Schwachstellen aber nicht aus" [Inf16b]. In Grafik 2.1 entspricht der passive Schwachstellenscan der Phase 'Informationsbeschaffung und -auswertung'.
Ein aktiver Schwachstellenscan nutzt zusätzlich Exploits, um in das System einzudringen und zuvor entdeckte Schwachstellen auszunutzen. Dabei ist zu bedenken, dass an den Zielsystemen destruktive Nebenwirkungen nicht auszuschließen sind. Diese Methode entspricht der Phase 4 'Aktive Eindringversuche' aus Grafik 2.1. Abstufungen zwischen beiden Extremen sind möglich, jedoch ist oft schwer zu erkennen, welche Nebenwirkungen tatsächlich auftreten. Ein Tester sollte somit auf Exploits gegebenenfalls verzichten, falls er die diese nicht zuvor bereits auf unerwünschte Nebenwirkungen getestet hat [Inf16b]. Das BSI definiert unterschiedliche Prüftiefen bei einem Penetrationstest, nicht invasive Schwachstellenscans (passiv) und invasive Schwachstellenscans (aktiv) [Inf16b]. Aktive Scans sind insofern aussagekräftiger, dass sich erst bei "aktiven Eindringversuchen zeigt, ob die in der Informationsbeschaffungsmaßnahme identifizierten vermeintlichen Schwachstellen auch tatsächlich genutzt werden können" [EY03].
Nebenwirkungen können trotz angeblich passiver Tests ebenfalls auftreten. Diese sind zu identifizieren; eine vorherige Überprüfung der Scans ist demnach auch bei scheinbar passiven Tests zu empfehlen [SKBG18, YCLS04, Acu14].

Test-Umgebung

Eine Test-Umgebung ist "eine [kontrollierte] Umgebung, in der Konfigurations Items, Releases, IT Services, Prozesse usw. getestet werden" [AXE13]. Sie muss von der Produktionsumgebung getrennt sein, soll jedoch der Produktionsumgebung ähneln, damit Probleme bereits im Test erkannt und behoben werden können (vgl. [VV13]). Eine Test-Umgebung ist dem Wort ensprechend gekennzeichnet durch Tests und Umgebungen. Systemtest, Integrationstest, Abnahmetest, Last- und Performancetest, Komponententest bilden mögliche Ausprägungen von Tests. Die zu überprüfenden Umgebungen bilden den zweiten Teil. Diese können aus Hardware, Software, Middleware und mehr bestehen.
Ein Testziel stellt die Grundlage einer Test-Umgebung dar. Ist dieses definiert, können die entsprechenden Umgebungen und Tests gewählt werden. Das Ziel bestimmt ebenfalls das Szenario unter welchem Kontext die Tests stattfinden. Dabei werden spezielle Tests bezüglich

der gewählten Umgebungen erstellt. Tests und Umgebungen bilden zusammen eine Test-Umgebung. (siehe: [DB11])
Eine gute Test-Umgebung ist dadurch definiert, dass sie erweiterbar, portabel aber vor allem wiederverwendbar ist [ER96]. Diese Arbeit beschreibt eine Test-Umgebung für Schwachstellenscanner, welche Bertiebstests, Komponententests, Last- und Performancetests aber auch einen Abnahmetest bezüglich einer konkreten Software ermöglicht. Das Ziel wird durch die nachfolgenden Use-Cases spezifiziert.

2.2 Use-Cases

Vergangene wissenschaftliche Arbeiten und Software-Tools verdeutlichen [Lau19, Dmi19, Jos19, Khr19, Mic17], dass in der IT-Sicherheit speziell Schwachstellenscans eine besondere Bedeutung für Forschung und Industrie genießen. Grund dafür ist unter anderem das immer mehr in den Mittelpunkt der Gesellschaft rückende Interesse des Datenschutzes [Uni18], beflügelt durch mediale Aufmerksamkeit und (über)nationale Gesetzgebung (Staat und EU) [Deu15, Deu20].
Im Zuge diese Themas tauchen folgende klassische sowie speziellere Anwendungsfälle immer wieder auf. Diese Use-Cases sollen verdeutlichen, welchem Anwendungsbereich eine Test-Umgebung für Schwachstellenscanner unterliegt und welche Anforderungen von einer solchen Test-Umgebung abgedeckt werden müssen.

- Wahl eines geeigneten Schwachstellenscanners (siehe: 2.2.1)
- Erstellen einer Testumgebung für wissenschaftliche Zwecke (umfasst auch die Vulnerables und nicht nur Schwachstellenscanner) (siehe: 2.2.2)

2.2.1 Wahl eines geeigneten Schwachstellenscanners

Sollen ein oder mehrere Systeme auf Schwachstellen untersucht werden, so ist die Wahl des richtigen Schwachstellenscanners bedeutsam. Zusätzlich ist die Qualität der Ergebnisse eines Schwachstellenscans davon abhängig mit welchen Konfigurationen dieser eingesetzt wird [AV10, ERRW18].
Bevor somit die oft teure und aufwändige Anschaffung eines Schwachstellenscanners getätigt wird, muss eine Wahl zwischen verschiedensten Marktteilnehmern getroffen werden. Beispiele sind: Nessus, OpenVAS, Nexpose, arachni und viele mehr. Daneben besteht die Möglichkeit aus einer Vielzahl von Einzelsoftware auszuwählen, welche zwar keinen gesamt-umfänglichen Schwachstellenscanner darstellen, jedoch kombiniert ebenfalls eine ausreichende Testabdeckung liefern können: nmap, Postman, BurpSuite und Metasploit sind Kandidaten für solche Anwendungen und werden teils von den Scannern selbst eingesetzt.
Damit eine informierte Entscheidung bezüglich des zu verwendenden Schwachstellenscanners getroffen werden kann, müssen diese untereinander vergleichbar gemacht werden. Dazu kommt der Vergleich bezüglich unterschiedlichster Anforderungen in Frage, die alle abhängig vom Anwendungsfall des Nutzers mehr oder weniger ausschlaggebend für die Entscheidung sind. Einige Anforderungen an einen Schwachstellenscanner werden im Folgenden genannt:

- Erkennungsrate der vorhandenen Schwachstellen
- Anzahl der False-Positives und -Negatives

- Qualität der Ergebnisse in Abhängigkeit der verwendeten Scan-Konfigurationen
- Erkennen kürzlich veröffentlichter Schwachstellen (Qualität der verwendeten Schwachstellen-Datenbanken)
- Identifikation der Auswirkungen der Anwendung von Scannern und Exploits auf dem Zielsystem (Unerwünschte Nebenwirkungen, Hinterlassen von Spuren)
- Erfolgschancen bei der Verwendung von Exploits zum Ausnutzen gefundener Schwachstellen
- Zuverlässigkeit der Versions- und Diensterkennung
- Qualität von weiteren Scan-Modi (Lokale Scans, Authentifizierte Scans, Grey- / Whitebox-Scans)
- Deterministisches Verhalten bei wiederholten Überprüfungen
- Auslastung des Netzes und der Zielmaschinen
- Vergleich von Scannern in Live- und Test-Umgebungen
- Überprüfung der Möglichkeit zur Automatisierung von Scans
- Portabilität - Verschiedene Betriebssysteme müssen überprüft werden können
- Auswirkungen bei Kombination mehrerer Scanner oder IT-Sicherheitstools auf die Scan-Ergebnisse

Diese Liste zeigt nur einige Qualitätsmerkmale bezüglich welcher in der Vergangenheit getestet wurde und ist nicht erschöpfend.

2.2.2 Erstellen einer Test-Umgebung für wissenschaftliche Zwecke

Die wissenschaftliche Arbeit mit Schwachstellenscannern umfasst mehrere Szenarien. Zum einen werden ebenfalls Schwachstellenscanner miteinander verglichen, um die Qualität von Scannern zu bewerten aber in diesem Fall aus wissenschaftlichem Interesse und weniger aufgrund einer späteren Anschaffungsempfehlung. Aus demselben Grund werden Scanner verwendet, um Zielsysteme zu überprüfen und die wissenschaftliche Bewertung von Vulnerables zu vereinfachen. Diese Aspekte entsprechen zu großem Teil dem ersten Use-Cases aus Abschnitt 2.2.1, wobei die Anwender nun keine Firmen, sondern Wissenschaftler sind.
Dennoch besteht ein wissenschaftliches Interesse an weiteren Fragen zu Schwachstellenscannern. Einige werden im Folgenden genannt:

- Auswirkung und Funktionsweise von Exploits
- Wiederholtes Durchführen ähnlicher Tests (Überprüfen neuer Scanner-Versionen, Test neuer Scanner auf dem Markt, Direktvergleich mehrerer Scanner)
- Auswirkungen bei Kombination mehrerer Scanner oder IT-Sicherheitstools auf die Scan-Ergebnisse
- Qualität von Plug-Ins und Erweiterungen und deren Auswirkung auf die Scan-Ergebnisse
- Erstellen von Testmaschinen Vulnerables und Überprüfung der Qualität dieser.
- Erstellen von Vulnerables für Lehrzwecke

- Erstellen und Überprüfen von Eigenentwicklungen (Scanner, Vulnerables, Software, Plug-Ins)
- Möglichkeit des Ausnutzens von Fehlern in Exploits

Dieses Kapitel hat einige Szenarien aufgezeigt, wie sie in Betrieben und Forschungseinrichtungen entstehen können und was deren Relevanz im Kontext der aktuellen Gesetzgebung ist. Aus den spezifischen Anwendungsfällen werden im Folgenden Anforderungen an eine Test-Umgebung für Schwachstellenscanner abgeleitet.

3 Anforderungen an die Test-Umgebung

Kapitel 2 erläutert die notwendigen theoretischen Grundlagen bezüglich der Thematik der Schwachstellenscanner und beschreibt Anwendungsfälle, wie sie in der Industrie und Forschung auftreten. Dieses Kapitel beantwortet die Frage, welche Anforderungen an eine Test-Umgebung für Schwachstellenscanner gestellt werden müssen. Nachdem diese bekannt sind, wird eine Analyse verwandter Arbeiten und des aktuellen Marktes gegeben und der Beitrag der Masterarbeit für die aktuelle Forschung festgelegt.
Die hier beschriebenen funktionalen, wie nicht-funktionalen Anforderungen dienen als Grundlage des in Kapitel 4 folgenden Gesamtkonzepts einer Test-Umgebung für Schwachstellenscanner.

3.1 Anforderungen

Dieser Abschnitt beschreibt die identifizierten Anforderungen an eine Test-Umgebung, welche unter anderem einer Analyse der Use-Cases des letzten Kapitels entstammen. Es gibt einen allgemeinen Überblick über die Bedeutung der Anforderung sowie eine Beschreibung dieser. Die Anforderungen werden im Folgenden abstrakt definiert und im Zuge des nächsten Kapitels im Gesamtkonzept (Kapitel 4) detaillierter beschrieben und ausdefiniert.
Die Anforderungen A-1 bis A-12 in Tabelle 3.1 beschreiben eine Infrastruktur wie sie der Grafik 3.1 entnommen werden kann. Abstrahiert kann man die Architektur folgendermaßen beschreiben: Ein Nutzer startet über ein User Interface einen Scan-Prozess zwischen einer beliebigen Anzahl an Schwachstellenscannern und Vulnerables. Die Scan-Ergebnisse werden nach Abschluss der Scans persistent abgespeichert.

Nr.	**Name der Anforderung**
A-1	Analyse von Testmaschinen mithilfe von Schwachstellenscannern
A-2	Auswahl von Schwachstellenscannern
A-3	Auswahl von Vulnerables
A-4	Auswahl von Software für die Vulnerables
A-5	Auswahl benutzerdefinierter Scan-Modi
A-6	Überwachung der Laufzeitumgebung
A-7	Erstellen von Reports
A-8	Wiederholbarkeit und Persistenz
A-9	Modularität und Erweiterbarkeit
A-10	Automatisierung
A-11	Status-Informationen
A-12	Portabilität

Tabelle 3.1: Übersichtstabelle über alle Anforderungen

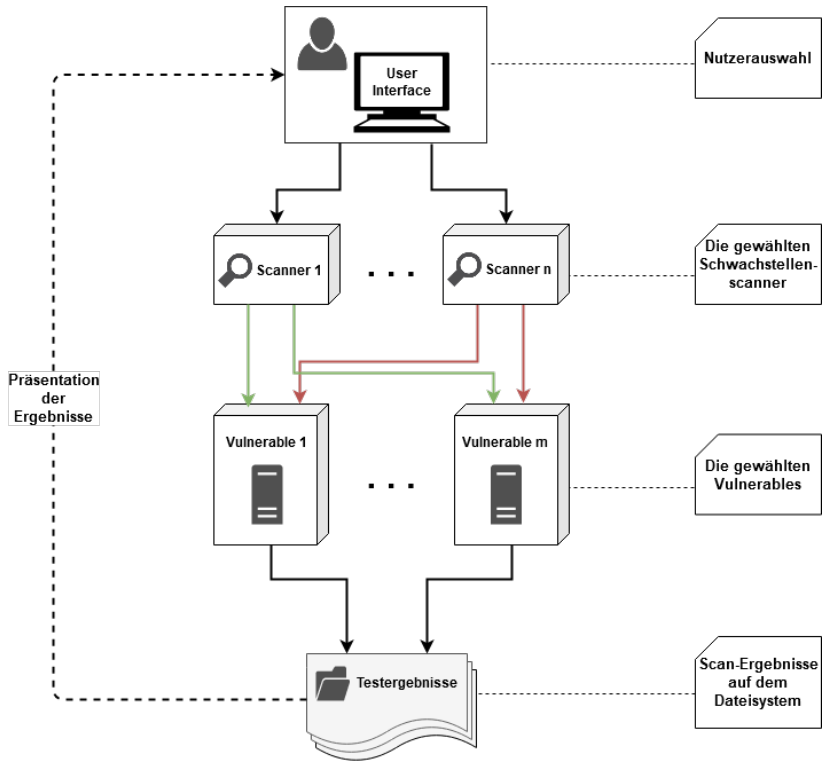

Abbildung 3.1: Architektur der Test-Umgebung

A-1 - Analyse von Testmaschinen mithilfe von Schwachstellenscannern

Die Test-Umgebung muss es erlauben folgende Szenarien abzudecken:

- Scan einer Testmaschine mithilfe eines Schwachstellenscanners
- Scan einer Testmaschine mithilfe mehrerer Schwachstellenscanner
- Scan mehrerer Testmaschinen mithilfe eines Schwachstellenscanners
- Scan mehrerer Testmaschinen mithilfe mehrerer Schwachstellenscanner

Der Scan-Prozess darf dabei nicht verfälscht werden was bedeutet, dass Ziel-VM und Scanner-VM durch unterschiedliche Maschinen repräsentiert werden müssen.
Ebenfalls muss der Scan-Prozess über das Netz stattfinden, also einen Black-Box-Test simulieren, wobei der Scanner keine Vorinformationen außer der IP-Adresse des Zielsystems erhält.
Wenn eine Maschine multiple Scans durchführen soll oder mehrmals gescannt wird, ist es notwendig, die Maschinen zunächst in einen neutralen Ausgangszustand zu bringen. Dies bedeutet, dass keine Maschine (Scanner oder Vulnerable) für mehr als einen Scan-Prozess existiert. Für weitere Scan-Prozesse werden neue Maschinen instanziiert, um gleiche Voraussetzungen zu schaffen.

A-2 - Auswahl von Schwachstellenscannern

Es muss dem Anwender die Möglichkeit gegeben werden, die Schwachstellenscanner in Anzahl und Ausstattung zu variieren. Zusätzlich muss eine Vorauswahl an Scannern gegeben sein, auf die zurückgegriffen werden kann. Der Scanner muss dabei keine Spezielle Software installiert haben, da diese ebenfalls während des Provisionierungs-Prozesses nachinstalliert werden kann. Zusätzlich muss es möglich sein neue Schwachstellenscanner in das Gesamt-System einzuspeisen. Dies ermöglicht es dem Anwender, eigene Fragestellungen zu beantworten.

A-3 - Auswahl von Vulnerables

Die Test-Umgebung muss es dem Anwender ermöglichen, seine zu testenden Maschinen individuell auszuwählen. Betriebssystem-Art und Leistung der Zielsysteme sowie die installierte Software und Konfigurationen müssen bestimmt werden können.
Eine einfache Erweiterbarkeit der Test-Umgebung mit neuen, benutzerdefinierten Vulnerables muss gegeben sein.

A-4 - Auswahl von Software für die Vulnerables

Die Test-Umgebung muss es dem Anwender ermöglichen, seine Vulnerables mit benutzerdefinierter Software auszustatten, diese kann ebenfalls zu testende Schwachstellen repräsentieren. Folgende Ausprägungen stellen einige Szenarien der nachträglich eingespielten Software dar:

- Software für Reporting
- Eigen-Entwickelte Software für Testzwecke
- Software mit Fehlkonfigurationen oder bewussten Schwachstellen
- Software zur Repräsentation eines Anwender-Szenarios (z.B. Onlineshop)

A-5 - Auswahl benutzerdefinierter Scan-Modi

Es muss möglich sein die Scans benutzerdefiniert durchzuführen, also verschiedene Parameter individuell zu setzen, wie zum Beispiel die Prüftiefe, welche auch die Laufzeit beeinträchtigt. Gleichzeitig soll die Möglichkeit bestehen weitere Scan-Methoden hinzuzufügen, um gegebenenfalls Scan-Profile anzulegen, welche auf einen speziellen Anwendungsfall zugeschnitten sind (z.B. Onlineshop).

A-6 - Überwachung der Laufzeitumgebung

Laufzeitdaten der Test- und Zielsysteme müssen überwacht werden können. Während und nach dem Abschluss des Scan-Prozesses müssen die Informationen dem Anwender zur Verfügung gestellt werden, damit er eine Auswertung vornehmen kann.
Zu den Daten zählen unter anderem die wichtigsten Log-Dateien und Informationen über Veränderungen an beiden Maschinen während der Laufzeit. Zusätzlich müssen Leistungsparameter überwacht werden, welche aus Netzauslastung sowie Prozessorauslastung und weitere Statusinformationen der Maschinen bestehen.

A-7 - Erstellen von Reports

Alle gefundenen Scan-Ergebnisse müssen dem Anwender verwertbar zur Verfügung gestellt werden. Dem Anwender wird somit die Möglichkeit gegeben, die Schwachstellenscanner anhand eigener Testparameter zu bewerten. Die Reports fassen dabei alle Konfigurationsdetails der verwendeten Maschinen, Scan-Ergebnisse, aufgetretene Fehler sowie Meta-Informationen der Scans.

A-8 - Wiederholbarkeit und Persistenz

Die Test-Umgebung muss alle Scan-Ergebnisse und Konfigurationen persistent speichern, damit Tests erneut zu späterem Zeitpunkt ausgeführt werden können und Ergebnisse zur Nachweisbarkeit weiter verfügbar bleiben.

A-9 - Modularität und Erweiterbarkeit

Die Test-Umgebung muss modular aufgebaut sein, damit die Scans bei Bedarf um weitere Komponenten wie Plug-Ins oder eine Datenbank erweitert und Einzelkomponenten ausgetauscht werden können.

A-10 - Automatisierung

Das automatisierte Aufsetzen von Infrastruktur erleichtert aussagekräftige Studien, minimiert den Arbeitsaufwand für repetitive Prozesse und muss somit möglich sein. Die automatisierte Durchführung von Scans muss möglich sein. Bei Bedarf muss es jedoch ebenfalls möglich sein den Scan Manuell durchzuführen.

A-11 - Status-Informationen

Der aktuelle Status eines Scans muss laufend an den Anwender berichtet werden. Dies ist begründet mit der Annahme, dass der Scan zum Teil mehrerer Scanner-Vulnerable-Paare eine sehr hohe Laufzeit annehmen kann. Ein Nachverfolgen verbessert die Nutzer-Erfahrung und die Fehleranalyse.

A-12 - Portabilität

Die Testumgebung muss Plattform- und Betriebssystem-Unabhängig sein. Die Unterstützung der etabliertesten Betriebssysteme Windows und Linux muss gegeben sein.

3.2 Verwandte Arbeiten

Themen rund um Schwachstellenscanner sind ein viel untersuchtes Gebiet. Es gibt Arbeiten, welche sich mit der Bewertung von Schwachstellenscannern beschäftigen, solche die Scanner miteinander vergleichen. Vorwiegend zu finden sind Arbeiten, welche Einzelaspekte eines Scanners z.B. die Qualität des Crawlings [ERRW18] oder der SQL-Injektions [BBGM10] untersuchen.

Bewertungen von Schwachstellenscannern

Generell wird zwischen zwei Methoden unterschieden, um die Qualität von Schwachstellenscannern zu Bewerten. Zum einen werden bestehende Applikationen (vorwiegend Web-Applikationen) verwendet, bei welchen die Schwachstellen bereits vor der Überprüfung meist aufgelistet vorhanden sind (siehe: metasploitable3, mutillidae) um diese gegen die Liste der gefundenen Schwachstellen der Scanner zu testen. Zum anderen werden Applikationen entwickelt, welche je nach Konfiguration unterschiedliche Schwachstellen beinhalten, um die Tests an bestimmte Szenarien anzupassen ([Sch20], [She10], [ERV+08]). Diese Arbeit ermöglicht die Verwendung beider Methoden, indem eine Infrastruktur vorgestellt wird, welche je nach Anwendungsfall beide Arten von Vulnerables als virtuelle Maschinen akzeptiert.
Schwachstellenscanner können anhand multipler Merkmale klassifiziert und vor allem bewertet werden. Dies kann von Schwachstellen-Abdeckung über Zeitaufwand hin zu Präzision [TTSS14b] reichen. Somit findet man auch viele Arbeiten, welche diese Aspekte individuell betrachten.
[AV10] analysieren verschiedene Scanner bezüglich ihrer Fähigkeit SQL-Injections zu erkennen. [FVM07] untersucht zusätzlich ebenfalls die Erkennungsrate von Cross-Site-Scripting-Angriffen. Das die Effektivität von Web-Scannern stark von deren Crawling-Befähigung (das automatische Durchsuchen des Internets zur Indexierung von Webseiten) und Möglichkeit abhängt, authentifizierte Angriffe (übergeben von bekannten Zugangsdaten) durchzuführen zeigen Esposito et al. [ERRW18]. Unter authentifizierten Angriffen versteht man in diesem Kontext die Fähigkeit des Scanners sich mithilfe von Zugangsdaten oder Schlüsseln bei Diensten zu authentifizieren ohne, dass sie sich im weiteren Ablauf des Scan-Prozesses durch das betätigen einer Log-Out- oder Löschen-Routine selbst wieder abmelden und somit die Ergebnisse verfälschen.
Und nachdem auch die Studie von Suto und San [SS10] - welche die Effektivität und Abdeckung von Web Schwachstellenscannern bezüglich der Genauigkeit der Ergebnisse und deren zeitlichem Aufwand überprüft - ihre Tests so anpasst, dass Tests nicht durch die Crawling-Fähigkeit beeinflusst werden können, wird verdeutlicht, dass bei automatisierten Untersuchungen ein besonderes Augenmerk auf Einschränkungen der Scanner selbst gelegt werden muss.
Die Ergebnisse dieser Studien zeigen, dass die Erkennungsraten beim Gros der Untersuchten Dienste in der Standardkonfiguration niedrig sind und geprägt von vielen Falsch-Positiven. Die Erweiterung der Scanner um Plug-Ins in dem jeweiligen Bereich kann dabei helfen die Ergebnisse signifikant zu verbessern, bedürfen jedoch einem größeren manuellen Aufwand. Auch die Kombination mehrerer Scanner und Tools ist ein valides Mittel um Ergebnisse zu verbessern [ERRW18]. Auch ist die verwendete Schwachstellendatenbank ein wichtiger Faktor. Ein Vorteil, den kommerzielle Tools bieten, ist eine aktiv verwaltete Datenbank, welche auch aktuelle Schwachstellen beinhaltet (siehe: [Nes20]), wohingegen man bei freien Tools meist - trotz aktiver Community - mit einer zeitlichen Verzögerung rechnen muss.
Die Analyse von Referenz Literatur unterstreicht die Annahme, dass bei der Verwendung von Schwachstellenscannern darauf zu achten ist, dass sowohl Scanner als auch Scanner-Konfigurationen und Plug-Ins mit Bedacht gewählt werden müssen, da die Qualität der Ergebnisse und somit zukünftige Sicherheits-Management-Entscheidungen davon abhängen. Diese Erkenntnis macht den Sinn einer Test-Umgebung greifbarer. Dienst- und Versions-Erkennungen sind im Fall von Penetrationstests meist der erste Schritt um mögliche Angriffsvektoren auf Zielsysteme zu identifizieren. Tools wie **nmap** dienen diesem Zweck und

werden ebenfalls von Schwachstellenscannern verwendet. Die Qualität dieser Tools ist somit ausschlaggebend, wenn es um die Aussagekraft der Scanergebnisse geht. Dennoch unterscheiden sich auch diese Tools maßgeblich in deren Ergebnisse. Besonders wenn es um die Analyse von Software mit abgeänderter Bannerantworten geht, so lassen sich viele Dienste einfach täuschen. Auch neuere Software wie beispielsweise eine MongoDB wird von Scannern selten erkannt. [Dmi19] [Khr19]

Test-Umgebungen
Forscher der Asia-University haben eine Test-Umgebung geschaffen, um vorwiegend False-Positives und -Negatives zu identifizieren [TTSS14b]. Besonders interessant ist der Aufbau des 'Web Vulnerability Scanner Testbeds' (W-VST), welches ausgehend von vier verschiedenen Web-Schwachstellenscannern drei Vulnerables untersucht. Diese repräsentieren unterschiedliche Szenarien. Es findet sich ein OWASP Broken Web Applications [OWA15], eine WebGoat [Goa20] Applikation und eine Broken-Wordpress-Anwendung. Die Scan-Ergebnisse werden gespeichert und ein Monitor überwacht die Vulnerables. Ergebnisse werden zusammengetragen und von Experten manuell auf Redundanzen überprüft.

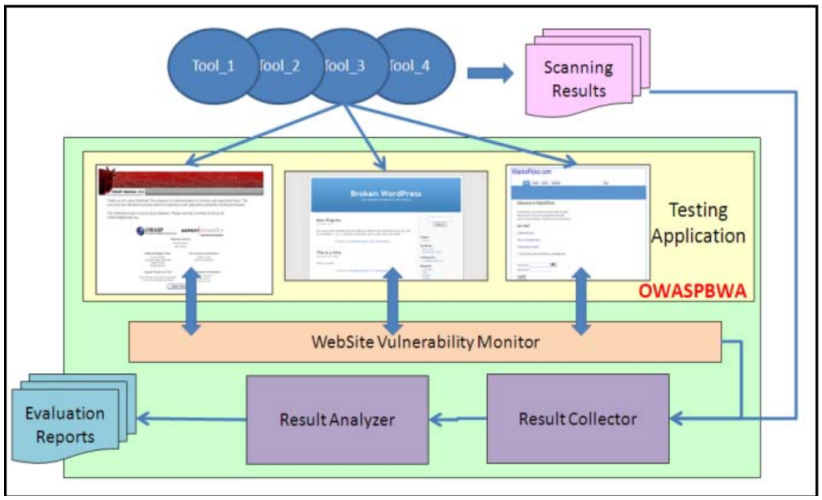

Abbildung 3.2: Der Aufbau der W-VST Test-Umgebung für Web-Schwachstellenscanner

Eine weitere Arbeit untersucht ebenfalls die Qualität von drei Schwachstellenscannern (Acunetix, Netsparker, Burp Suite) und testet deren Erkennungsrate der OWASP Top10 durch die Bereitstellung eines eigenen Vulnerables mit eigens implementierten Schwachstellen [JS16]. Die Auswertung erfolgt ebenfalls durch eine Manuelle Untersuchung der Ergebnisse. Eine weitere Test-Umgebung findet man in der Arbeit von David Shelly [She10]. Er nutzt diese um Einschränkungen von Schwachstellenscannern zu identifizieren.
Diese Arbeit erstellt eine Test-Plattform für die Erkennung der OWASP Top 10 [OWA17]. Dabei entwickelt der Autor eine 'Benchmark Web Application', welche es ermöglicht bestimmte Aspekte des Vulnerables benutzerdefiniert zu wählen und somit unterschiedliche Szenarien zu kreieren. Ebenfalls ist die Herangehensweise an einzelne Probleme von Interesse. So bietet die Idee des Einbindens eines Proxys zwischen Scanner und Vulnerable die Möglichkeit, Befehle abzuändern, blockieren und zur Nachvollziehbarkeit auszulesen (siehe:

3.3) [ERRW18].
Neben den Schwachstellenscannern ist der Aufbau der Vulnerables für eine Test-Umgebung

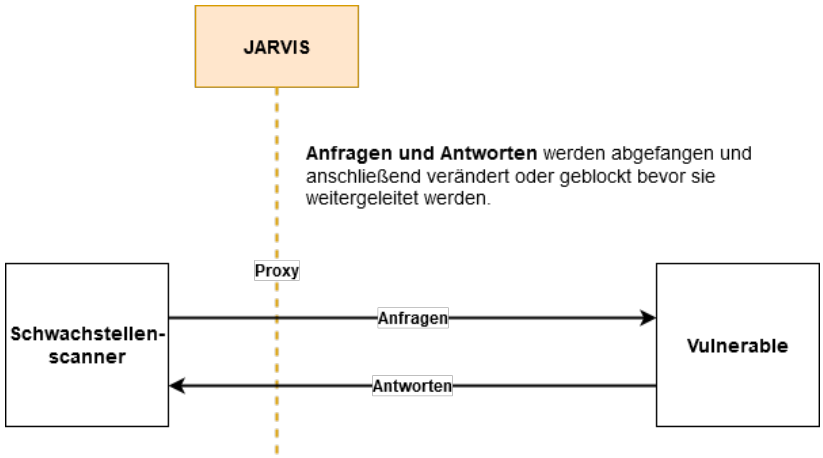

Abbildung 3.3: Das Tool JARVIS als Proxy zwischen Scanner und Vulnerable ermöglicht es anfragen abzuändern oder blockieren, um die Ergebnisse zu verbessern. [ERRW18]

von Interesse. Geht es um die Überwachung dieser, also entweder das Erkennen von Eindringlingen im System (eng. intrusion detection) oder das Nachträgliche erkennen von Veränderungen am System also der forensischen Untersuchung, so dienen Tripwire [KS94] und Forschungsmethoden des JPCERT [JPC17a][JPC17b] als Referenz. Ähnliche Methoden werden in dieser Arbeit verwendet.
Kombiniert man Techniken der genannten Arbeiten, entsteht die Infrastruktur einer Test-Umgebung.

Eigener Beitrag
Bei der Analyse der verwandten Arbeiten lässt sich eine Tendenz zur spezifischen Analyse der Einzelaspekte von Schwachstellenscannern erkennen. Man erkennt jedoch, dass für jede Arbeit neue individuelle auf den Anwendungsfall zugeschnittene Umgebungen erzeugt werden. Dabei werden Vulnerables und Schwachstellenscanner oft per Hand konfiguriert und Scans ebenso manuell durchgeführt. Gibt es Änderungen am Szenario, so ist dies mit einer aufwändigen Anpassung der Umgebung verbunden und standardmäßig nicht vorgesehen. Ebenso ist ein erneutes Durchführen der Tests zu späterem Zeitpunkt meist nicht erwartet, da eine Momentaufnahme aktueller Software und deren Konfiguration durchgeführt werden soll. Der Beitrag dieser Arbeit ist es eine Umgebung zu schaffen, welche diese Schwächen abfängt. Es ist möglich Scans wiederholt zu späterem Zeitpunkt durchzuführen, das Aufsetzen und die Verwaltung der virtuellen Maschinen zu automatisieren und den Prozess der Konfiguration, Provisionierung und Orchestrierung auszulagern. Zusätzlich erlaubt es die Test-Umgebung, mehrere Aspekte der Test-Umgebung simultan aufzuzeichnen und schließlich zu untersuchen. Da das Konzept nicht auf einen spezifischen Aspekt (z.B. Cross Site Scripting) von Schwachstellenscans ausgelegt ist und erweiterbar ist, besteht die Möglichkeit, bei Bedarf weitreichendere Untersuchungen vorzunehmen.

4 Gesamtkonzept der Test-Umgebung

Aus den identifizierten Anforderungen soll im Folgenden ein Konzept erarbeitet werden, welches die praktische Implementierung einer Test-Umgebung, beschrieben in Kapitel 5 ermöglicht. Das Konzept gründet auf der Voraussetzung, dass ein Standard Betriebssystem wie Windows oder Unix bei dem Anwender vorhanden ist.
Auf logischer Ebene unterscheidet die Test-Umgebung zwischen dem **globalen Kontext** und den **Laufzeitumgebungen**. Der Kontext ist dabei verantwortlich für die Verwaltung von Informationen über virtuelle Maschinen, Ordnerstruktur, Namensgebung und von Blaupausen-Skripten. Plug-Ins werden ebenfalls von der Umgebung verwaltet. Diese Komponenten sind für die Durchführung eines Scan-Prozesses Voraussetzung.

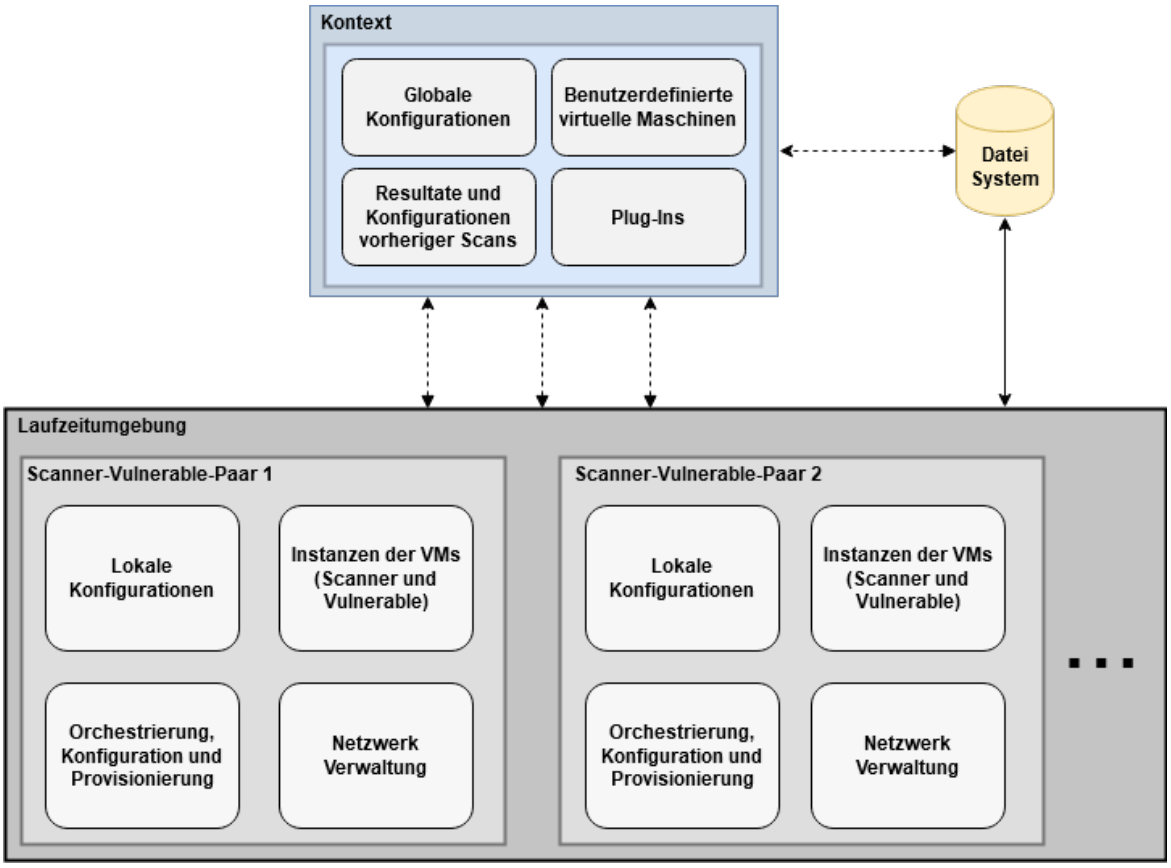

Abbildung 4.1: Aufbau der Test-Umgebung mit den einzelnen Komponenten

Die Laufzeitumgebungen stellen die Scan-Prozesse selbst dar (siehe: 4.1), und reichen von den Nutzereingaben über das Durchführen des Schwachstellenscans bis zu dem Auslesen der

Scan-Ergebnisse. Die Laufzeitumgebungen werden nach der Durchführung der Scan-Prozesse zerstört.
Diese Logik wird von der Test-Umgebung mittels Modulen umgesetzt. Die Test-Umgebung besteht aus den Modulen: 'Kontext', welches den globalen Kontext auf logischer Ebene umsetzt und den Modulen 'User-Interface, Scheduler & VM-Management-Umgebung', welche die logischen Laufzeitumgebungen abbilden.
Dieses Kapitel beschreibt zunächst den klassischen Ablauf der Verwendung der Test-Umgebung 4.1 und erläutert diesen anhand eines Beispiels. Die Beschreibung des logischen Kontexts 4.2 und der Laufzeitumgebung 4.3 im Generellen dienen dem besseren Verständnis des Zusammenspiels einzelner Komponenten. Schließlich werden die Module und deren Kommunikation untereinander dargestellt 4.4.
Wichtige Konzepte des Test-Ablaufs werden im Abschnitt Systemarchitektur 4.5 im Detail erläutert.
Kombiniert bietet das Konzept somit eine Blaupause zur Entwicklung einer Test-Umgebung für Schwachstellenscanner basierend auf den Anforderungen aus Kapitel 3. Im Folgenden wird von den Laufzeitumgebungen im Singular gesprochen, da jede Umgebung dieselben Komponenten und Abläufe beinhaltet.

4.1 Test-Umgebungs-Ablauf

Der standardmäßige Ablauf der Test-Umgebung wird in Abbildung 4.2 veranschaulicht.

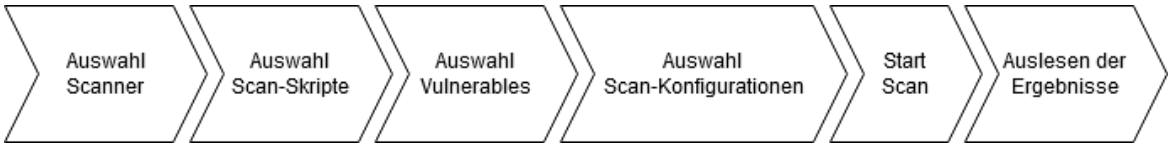

Abbildung 4.2: Der Basis-Ablauf der Test-Umgebung.

Der Basis-Ablauf besteht zunächst aus der Auswahl aller zu verwendenden Schwachstellenscanner. Im Zuge dessen wird ein Szenario festgelegt, welches durch die Auswahl eines Scan-Skripts bestimmt wird. Schließlich folgt die Wahl der Vulnerables. Diese können mithilfe eines eigenen Skriptes erweitert werden, so kann Software nachinstalliert werden oder notwendige Konfigurationen vorgenommen werden (z.B. Firewall Einstellungen). Wird schließlich der Scan-Prozess gestartet, werden alle Maschinen instanziiert, provisioniert und konfiguriert (z.B. Netzwerkkonfiguration, Meta-Informationen). Es werden Ordner für die entsprechenden Scan-Paarungen angelegt und die Scans gestartet (z.B. Scanner1 - Vulnerable1; Scanner2 - Vulnerable1). Während des Prozesses werden Statusinformationen fortlaufend an den Anwender kommuniziert. Nachdem die Scans abgeschlossen sind werden die jeweiligen Laufzeitumgebungen terminiert und zerstört. Ergebnisse werden an den entsprechend vorgesehenen Orten abgelegt und gebündelt präsentiert. Detailliertere Informationen finden sich in Abschnitt 4.3.
Folgendes Szenario kann ein Beispiel für die Nutzung der Test-Umgebung sein:

Der Nutzer hat das Ziel zwei Schwachstellenscanner bezüglich ihrer Erkennungsrate zu vergleichen. Ein metasploitable3 soll als Referenz-Vulnerable für beide Maschinen dienen, da

dem Anwender die vorliegenden Schwachstellen bereits bekannt sind.
Dazu hat er bereits vor dem Start des Scan-Prozesses ein 'metasploitable3' sowie ein 'Nessus' und ein 'OpenVAS' Image in den Kontext hinzugefügt.
Nun startet er die Software und wählt zunächst seine beiden zu testenden Schwachstellenscanner (Nessus und OpenVAS) aus einer Liste aus.
Außerdem wählt er für jeden Scanner ein Scan-Skript, welches das Szenario: 'Vollständiger Test der Zielsysteme' abbildet. Er wählt ein Skript welches die Scan-Ergebnisse als PDF Dateien ausgibt, damit er diese später auswerten kann.
Schließlich wählt er die Maschine metasplotable3 aus der Liste der verfügbaren Vulnerables aus und wählt ein Skript welches die Firewall der Maschine öffnet, da diese per Default keine ICMP-Requests akzeptiert. Bevor er den Scan schließlich startet, wählt er noch ein Plugin aus, welches die Virtualbox-Guest-Additions der virtuellen Maschinen auf den neusten Stand bringt, damit diese ohne Probleme mit der gewählten Virtualisierungssoftware VirtualBox interagieren können. Außerdem möchte er sehen, ob die Scans auf dem Vulnerable irgendwelche unerwünschten Nebenwirkungen zur Folge haben. Dazu wählt er ein Plug-In, welches die Integrität des Vulnerables während des Scan-Prozesses überprüft. Schließlich ist er zufrieden mit der Auswahl und startet den Scan. Dieser nimmt einige Zeit in Anspruch und gibt laufend Statusinformationen über den aktuellen Stand des Scan-Prozesses zurück.
Bedingt durch die getroffene Auswahl werden die Scan-Kombinationen: Nessus-metasploitable3 und OpenVAS-metasploitable3 durchgeführt. Die jeweiligen Ergebnisse finden sich in Ordnern mit den Entsprechenden Namen auf dem Datei-System. Darin befindet sich ebenfalls eine Datei, welche dem Nutzer übersichtsartig Informationen über alle erstellten Ergebnisse gibt.

4.2 Kontext

Bevor dieser Ablauf durchgeführt werden kann, müssen alle notwendigen Voraussetzungen erfüllt sein. Dies ist die Aufgabe des logischen Kontexts welcher für die Verwaltung von virtuellen Maschinen, Ordnerstrukturen, Blaupausen-Skripten und Plug-Ins verantwortlich ist. Er sorgt dafür, dass alle Voraussetzungen, welche für einen reibungslosen Ablauf der eigentlichen Tests - die von der Laufzeitumgebung durchgeführt werden - notwendig sind, erfüllt sind. Die Komponenten werden im Folgenden detaillierter beschrieben:

(Benutzerdefinierte) virtuelle Maschinen
Die Test-Umgebung basiert auf virtuellen Maschinen als Kernmodulen. Dabei übernehmen sie sowohl die Rolle von zu testenden Maschinen, als auch die der Scanner selbst. Der Nutzer hat dabei die Möglichkeit seine eigenen VMs in das System einzuspeisen sowie auf vordefinierte Maschinen zurückzugreifen. Dabei kann jede Maschine des Scan-Prozesses individuell gewählt werden. Somit unterstützt die Umgebung die Funktion, virtuelle Maschinen in das System einzufügen. Die Maschinen können Remote oder Lokal auf dem Host vorliegen. Einmal hinzugefügt ist die Maschine persistent vorhanden, sodass sowohl die Geschwindigkeit der nächsten Scan-Prozesse beschleunigt wird, als auch die erneute Ausführung eines Scans zu späterem Zeitpunkt gewährleistet werden kann. Dies kann zum Beispiel im Zuge eines Vergleichs einer alten und neuen Server-Version zu dem Zeitpunkt des Releases der aktualisierten Anwendung geschehen.
Ebenfalls wird gewährleistet, dass die virtuellen Maschinen jedes Mal dieselbe Startkonfi-

guration, Einstellungen und Speicherstände aufweisen, damit eine deterministische Untersuchung der Ergebnisse möglich ist. Ein Weiterarbeiten auf bereits verwendeten Maschinen ist nicht gewünscht, da Ergebnisse verfälscht werden und zu fehlerhaften Annahmen führen können. Auch wenn ein vermeintlich harmloser Scan durchgeführt wird, können sowohl Veränderungen auf dem Scanner als auch auf der Zielmaschine passieren,aufgrund von fehlerhafter Software, Anwendung oder böswilligem Verhalten beider Seiten. Dies kann nur verhindert oder erkannt werden, wenn bei jedem erneuten Scan beide Basis-Maschinen neu instanziiert oder im Grundzustand wiederhergestellt werden. Dadurch verhindert das System den Verbleib von zurückgebliebenen Fragmenten. Möchte der Nutzer seine Virtuellen Maschinen versionieren, um dauerhafte, regelmäßige Tests durchführen zu können - angemessener Weise bei jedem neuen Release - so ist dies ebenfalls möglich.

Ordnerstuktur und Persistenz

Persistenz der Ergebnisse der Scans sind essenziell, um einen Mehrwert aus einer Test-Umgebung ziehen zu können. In diesem Fall beschreibt die Persistenz die Gewährleistung der Verfügbarkeit von Ergebnissen und Konfigurationen einzelner Scans, auch nach längerem Zeitraum. Die erneute Ausführung von Tests mit denselben Parametern sowie die Nachvollziehbarkeit von Ergebnissen muss auch zu späterem Zeitpunkt gegeben sein. Dazu besitzt jeder Scan einen abgegrenzten Ausführungsbereich, welcher nicht von anderen Scans beschrieben werden kann. Dies dient zusätzlich der Übersichtlichkeit und einfacheren Bedienung. Je nach Anwendungsfall ist es sinnvoll diesen Bereich nach Abschluss eines Scans mit einem Schreibschutz zu versehen. Somit bekommt jeder Scan einen eigenen Ordner in welchem alle spezifischen Konfigurationsparameter, Provisionierungs-Skripte, Nutzerdaten und ähnliches liegen. Ein Unterordner für Scan-Ergebnisse und Reports von Log-Dateien ist hier ebenfalls zu finden. Eine Datenbank kann als Hilfe zur Verwaltung der Daten bei längerfristiger Verwendung und bei der automatisierten Verarbeitung der Daten helfen. Die Ordner weisen einfach identifizierbare Namen auf und sind eindeutig, wodurch die Gefahr von Verwechslung und somit Fehlern minimiert wird.

Blaupausen-Skripte

Die Test-Umgebung hat als Aufgabe virtuelle Maschinen aufzusetzen, zu provisionieren, zu scannen und die Ergebnisse auszulesen. Somit werden eine große Anzahl von Skripten ausgeführt sowohl auf der Host- als auch auf den Guest-Maschinen. Diese können individuell eingespeist werden. Eine Auswahl liegt ebenfalls bereits vor, um eine Out-Of-The-Box Funktionalität zu gewährleisten.
Damit die Skripte automatisiert und wiederholbar auf verschiedenen Maschinen ausführbar sind, liegen sie als Blaupausen vor, welche zur Ausführungszeit mit Informationen befüllt werden. Die Informationen zur Laufzeit bestehen aus IP-Adressen, Benutzerdaten, Schlüsseln, Namen der Maschinen und weiteren Konfigurationsparametern. Die Kontext-Umgebung verwaltet also sowohl Provisionierungs-Skripte, Konfigurations-Skripte, Meta-Daten der VMs & Scan-Profile, sowie Management-Skripte, welche die Scan-Prozesse steuern, überwachen und deren Status ausgeben können. Die Skripte sind an entsprechenden Orten abgelegt, um einen einfachen Zugriff und einfache Erweiterbarkeit zu ermöglichen.

Reports

Auf der Kontext-Ebene ist es ebenfalls möglich sich alle Ergebnisse auszugeben. Dies ist zum einen durch Auswahl innerhalb der Software - zum Zweck der Nutzerfreundlichkeit - aber ebenfalls durch einen direkten lesenden Zugriff auf Dateisystem oder gegebenenfalls eine Datenbank möglich, um weiterführende Automatisierung zu ermöglichen. Die Ergebnisse können nach verschiedenen Informationen ausgewählt werden. Diese bestehen aus Datum, dediziertem Name oder ID des Scans. Die Ordner beinhalten neben Start-Konfigurationen und Skripten ebenfalls alle Ergebnisse und Überwachungs-Parameter der Scans. Nach der Auswahl eines übergeordneten Ordners kann aus allen Scanner-Vulnerable-Paarungen ausgewählt werden sofern mehrere existieren. Diese beinhalten schließlich die Ergebnisse der jeweiligen Scan-Paarungen. Diese müssen klar identifizierbar sein. Somit sind die Ergebnisse zwar direkt zugreifbar es wird jedoch ebenfalls eine visuelle Ergebnisübersicht bereitgestellt, welche Nutzern einen schnellen Gesamtüberblick über die Konfigurationen und einzelnen Scan-Ergebnisse bietet.

Plug-Ins

Die Test-Umgebung kann um weitere Funktionen erweitert werden. Dies geschieht über Plug-Ins, welche vor dem Start des Scans in den Kontext eingefügt werden.
Erweiterungen können in jedem Bereich der Test-Umgebung geschehen. Dies ist durch den modularen Aufbau und die Verwendung von APIs möglich. Plug-Ins können den Scan-Prozess erweitern, das Reporting verbessern, aber auch die Infrastruktur erweitern. Auf Anfrage liefert der Kontext die Plug-Ins aus oder aktiviert diese.

4.3 Laufzeitumgebung

Dem Ablauf-Schaubild 4.2 folgend, soll im Folgenden die Laufzeitumgebung detailliert beschrieben werden. Dazu wird zunächst erklärt, wie der Nutzer bestimmte Prozesse anstoßen kann und schließlich, welche Hintergrundprozesse dabei ablaufen.

Auswahl der Scanner

Bei der Auswahl der Scanner können alle im Kontext hinzugefügten Scanner ausgewählt werden. Diese wurden durch eine Vorauswahl an Scannern erweitert, welche ein sofortiges Ausführen Out-Of-The-Box ermöglichen. Die Auswahl der Scanner kann über eine eindeutig identifizierbare ID-Name-Kombination vollzogen werden. Den benutzerdefinierten Maschinen wurde zuvor bereits ein eindeutiger Name vergeben dieser besteht aus der Betriebssystemversion verbunden mit dem installierten Scanner (z.B. Debian 10.3 - OpenVAS-9). Der Name aber auch die Metadaten beinhalten das zugrundeliegende Betriebssystem, welches es ermöglicht, die Auswahl der in den nächsten Schritten folgenden ausführbaren Skripte nach betriebssystem-Typ zu filtern, sowie den Scanner-Typen um ebenfalls die ausführbaren Skripte anzuzeigen.
Es ist möglich mehrere Scanner zu wählen. Eine Begrenzung der Anzahl ist nicht vorhanden, es gilt jedoch zu bedenken, dass die Zahl der Scans mit der Laufzeit und der Auslastung auf dem Host zunimmt.

Auswahl der Scan-Skripte

Da die Anwendungsgebiete der meisten Schwachstellenscanner so breit gefächert sind, dass man kein allgemeingültiges Skript pro Scanner festlegen kann und die Ergebnisse stark von dem subjektiven Anwendungsfall abhängen ist es im nächsten Schritt möglich sein Scan-Skript für jeden Scanner spezifisch zu wählen. Dieses kann aus einer Liste ausgesucht werden, welche aus einigen Vorschlägen (z.B. Basic nmap-Scan) besteht, sowie benutzerdefinierte Skripte des Anwenders beinhaltet. Die mögliche Auswahl der Skripte ist abhängig von der Art des Betriebssystems, da sich auf Skript-Ebene Befehle voneinander unterscheiden. Die Auswahl der Scanner und Scan-Skripte kann in ihrer Reihenfolge variiert werden. Dies unterstützt die Möglichkeit des universellen Einsatzes der Test-Umgebung. So kann entweder die Fragestellung ausgehend von einem Schwachstellenscanner oder einem Szenario (bzw. Scan-Skript) gestellt werden. Die Scan-Skripte können zusätzlich dazu benutzt werden, die Scanner-Software während der Laufzeit zu installieren. Dies erfordert jedoch bei vielen Schwachstellenscannern einen enormen Zeitaufwand (z.B. aktuell fast eine Stunde bei OpenVAS), da vor allem Schwachstellendatenbanken aber auch die Software selbst mit hohem Installations-Aufwand verbunden sind.Es wird empfohlen eigene Scanner bereits mit installierter Software einzuspeisen und einzig bei Bedarf die Aktualisierung der Schwachstellendatenbank zur Laufzeit durchzuführen. Es besteht ebenfalls die Möglichkeit ein Skript zu wählen welches einfach die entsprechenden virtuellen Maschinen startet und es dem Anwender ermöglicht selbst einen Scan, direkt auf der Maschine durchzuführen.

Auswahl der Vulnerables

Bei der Auswahl der Vulnerables kann zwischen einigen Vorlagen unterschieden werden, die von Beginn an implementiert sind, um eine schnelle intuitive Verwendung zu gewährleisten. Diese Liste an Vulnerables kann einfach im Kontext um neue Elemente erweitert werden. Es können mehrere Vulnerables ausgewählt werden. Die maximale Anzahl der Vulnerables soll theoretisch unbegrenzt sein. Die Vulnerables benötigen kein eigenes Scan-Szenario, da die Maschinen wie im realen Umfeld als Black-Boxen betrachtet werden, welche ebenfalls nicht extra für Scans im Vorhinein präpariert werden.
Es wird dennoch die Möglichkeit geboten, vor der Ausführung der Scans per Skript Veränderungen an dem Vulnerable vorzunehmen. Dies ist darauf zurückzuführen, dass es einen zu großen Overhead darstellen würde, eine eigene virtuelle Maschine zu bauen, obwohl nur einzelne Software-Veränderungen oder Konfigurationen durchgeführt werden sollen (z.B. Änderung der Firewall-Einstellungen, Installieren von einzelnen Softwarepaketen) und das System ansonsten identisch ist.
Die Auswahl eines solchen Skripts kann individuell für jedes Vulnerable getroffen werden.
Es sind ebenfalls sogenannte Basis-VMs vorhanden, welche ein unbearbeitetes Betriebssystem darstellen (z.B. ein Ubuntu-System der neusten Version). Diese können sowohl als Referenz bei Tests dienen [She10], aber ebenfalls mithilfe der eben genannten Skripte während der Laufzeit um Software erweitert werden.

Auswahl von Scan-Konfigurationen

Zusätzlich zu allen verwendeten Skripten können weitere Konfigurationen für die Laufzeitumgebung vorgenommen werden. Diese werden im Folgenden aufgeführt:

Überwachung der Zielsysteme auf Veränderungen (Logs, Basislinie)
Eine der möglichen Konfigurationen besteht aus der Entscheidung, dass die Vulnerables überwacht werden sollen. Dies dient der Erkennung von unerwünschten Nebenwirkungen durch den Scan-Prozess. Dazu wird eine Basislinie festgelegt, anhand welcher abgeglichen wird, ob das System an unerlaubter Stelle Abänderungen erfahren hat. Eine Überwachung der Log-Dateien auf Veränderungen und vor allem unerwünschte Veränderungen ist ebenfalls möglich

Festlegen der gewünschten Reports
Die Test-Umgebung ermöglicht es anzugeben welche der erstellten Reports schließlich ausgegeben werden. Dies ermöglicht es dem Anwender je nach Szenario nur Kriterien von Interesse zu untersuchen.

Performance-Messungen
Die Systeme können bezüglich ihrer Leistungsparameter überwacht werden. Dazu zählen unter anderem: CPU-Auslastung, RAM-Nutzung und Netzauslastung. Die Dauer eines Scan-Prozesses wird ebenfalls an den Anwender ausgegeben.

Überwachung der gesendeten Befehle

Eine Überwachung der Netz-Schnittstelle ermöglicht es die tatsächlichen Befehle welche von dem Scanner an die Vulnerables geschickt werden aufzuzeichnen. Dies ermöglicht einen Abgleich mit den eigenen Erwartungen oder Herstellerangaben oder dem Finden von Redundanzen.

Start Scan

Hat der Anwender alle entsprechenden Konfigurationen vorgenommen kann er den Scan-Prozess starten. Dabei werden alle notwendigen Scan-Informationen zusammengetragen, die Maschinen instanziiert, Skripte ausgeführt und der Scan selbst angestoßen. Ist der Start erfolgt, können keine weiteren Konfigurationen vorgenommen werden und der Prozess wird automatisiert durchgeführt ohne ein weiteres Eingreifen des Nutzers zu benötigen.

Auslesen der Ergebnisse

Nach Abschluss eines Scans werden die verwendeten Instanzen der virtuellen Maschinen zerstört. Für weitere Scans werden jeweils neue Instanzen erzeugt um vergleichbare Ergebnisse ohne unerwünschte Nebenwirkungen zu erzeugen. Die Scan-Ergebnisse und Reports werden zuvor persistent gespeichert, sodass sie auffindbar auf dem Dateisystem abgelegt sind. Die persistenten Daten bestehen aus Scan-Ergebnissen, Logdateien, Konfigurationsdateien (z.B. verwendete Maschinen und deren Konfigurationen), Protokollen, Diff-Dateien (Veränderungen am Zielsystem). Es wird automatisiert eine Übersichtsdatei erstellt, welche die wichtigsten Informationen, deren Dateiformat und ihren Speicherort gebündelt ausgibt. Dies dient der einfachen Auswertung durch den Nutzer. Die Daten liegen ebenfalls in ihrem Rohformat vor, um weitere maschinelle Auswertungen zu ermöglichen.

Kontinuierliche Statusmeldungen

Während des gesamten Scan-Prozesses werden laufend Statusmeldungen an den Anwender ausgegeben. Diese umfassen den aktuellen Fortschritt in der Scan-Pipeline, eventuelle Fehlermeldungen, aber auch Informationsangaben über den aktuellen Prozess. Der erfolgreiche Abschluss eines Scans sowie das Zerstören alter Laufzeitumgebungen kann mithilfe der Meldungen verfolgt werden.

4.4 Module und Schnittstellen

Die Test-Umgebung gliedert sich in unterschiedliche Module, welche den logischen Kontext und die Laufzeitumgebungen technisch umsetzen. Jedes Modul besitzt eine Reihe von Aufgaben, welche im Folgenden beschrieben werden. Die Module sind **Kontext**, **User-Interface**, **Scheduler** und **VM-Managementumgebung** (siehe: Abb.4.3). Dabei übernimmt der Kontext die Verwaltung aller Informationen zum reibungslosen Ablauf der Laufzeitumgebung 4.2. Die weiteren Module setzen die Laufzeitumgebung 4.3 in die Praxis um. Das User-Interface stellt die visuelle Schnittstelle für den Anwender zu Verfügung, über welches alle Konfigurationen vorgenommen werden können und der Scan-Prozess gestartet werden kann. Der Scheduler nimmt die Anfragen des User-Interfaces entgegen und wandelt sie in die entsprechenden Befehle um. Er übernimmt die Verwaltung der Logik und arbeitet im Hintergrund. Aufgaben wie das Anstoßen des Scan-Prozesses werden von dem Scheduler übernommen.

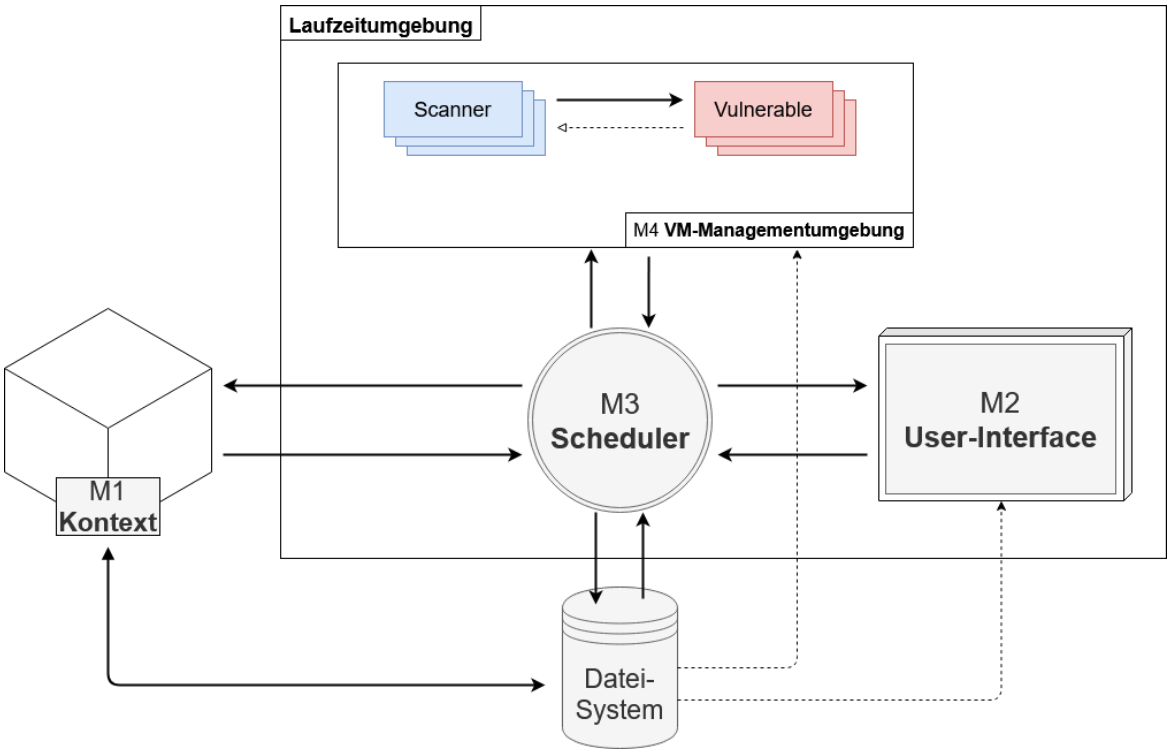

Abbildung 4.3: Module und Schnittstellen der Test-Umgebung

Die VM-Managementumgebung nimmt Befehle des Schedulers entgegen und übernimmt die Verwaltung aller beteiligten virtuellen Maschinen.
Das Datei-System hält alle persistenten Daten sowie zur Laufzeit ebenfalls die Daten der Ausführungsumgebung bis zu deren Zerstörung. Der Scheduler als zentrale Verwaltungseinheit hat schreibende Berechtigung auf das Datei-System. Alle weiteren Module greifen lesend auf die Ergebnisse zu, so kann das User-Interface auf Informationen über Scan-Ergebnisse zugreifen, sowie die VM-Umgebung Konfigurationsdateien einlesen kann.
Die Module, deren Aufgaben und Schnittstellen sowie den vorliegenden Datenfluss findet man im folgenden Kapitel. Die Zahlen auf den Bildern entsprechen den beschriebenen Aktivitäten des jeweiligen Kapitels.

M1 - Kontext

Das Modul **Kontext** verwaltet alle Informationen, virtuellen Maschinen und Dateien, welche später notwendig sind um einen reibungslosen Ablauf der Laufzeitumgebung gewährleisten zu können. Es entspricht dem logischen Kontext dieser Arbeit und wurde bereits ausführlich in 4.2 beschrieben. Die Grafik 4.4 zeigt nochmals alle Komponenten sowie den Datenfluss zwischen dem Kontext und weiteren Komponenten der Test-Umgebung.

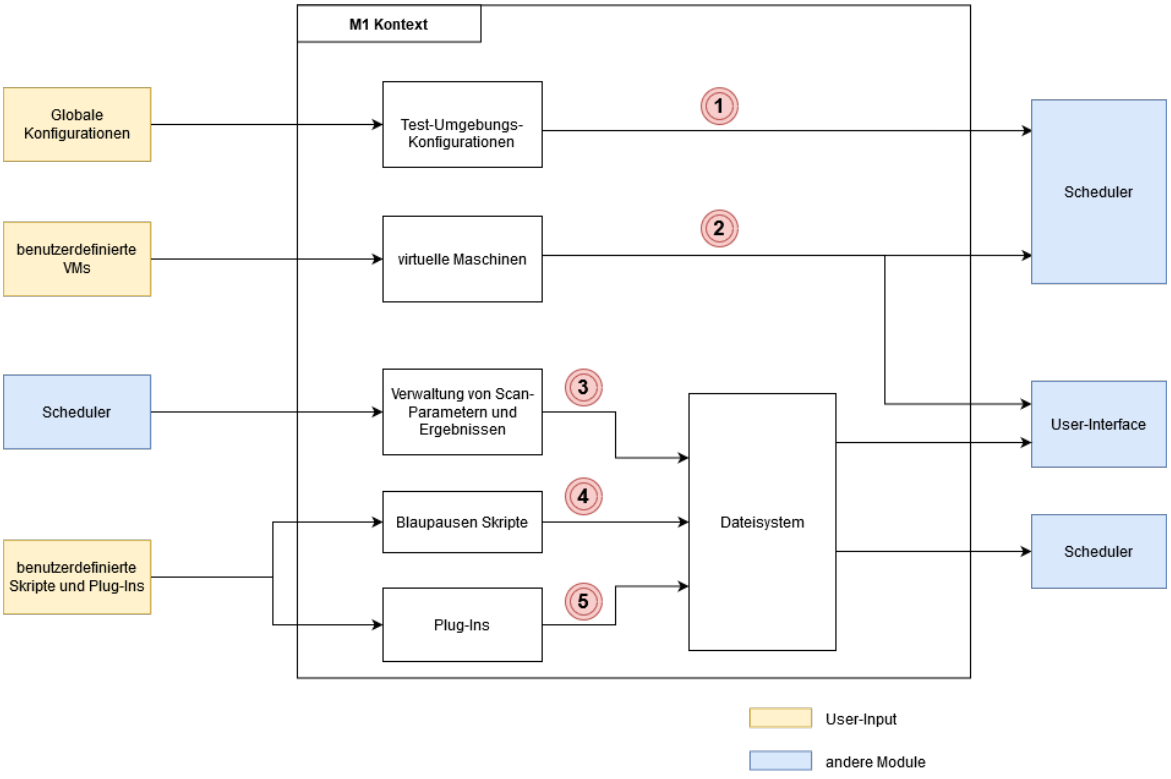

Abbildung 4.4: Modul 1 - Kontext

Aktivitäten

1. **Test-Umgebungs-Konfigurationen**
   Als User-Input empfängt der Kontext globale Konfigurationen. Diese werden im Kontext abgespeichert und verwaltet. Möchte der Nutzer eine Laufzeitumgebung starten, fordert er Informationen über den globalen Kontext an und dieser gibt die entsprechenden Informationen weiter.

2. **Virtuelle Maschinen**
   Der Nutzer kann über eine Schnittstelle eigene virtuelle Maschinen in die Test-Umgebung einspeisen. Diese müssen als Images auf dem Host-Dateisystem abgelegt werden. Außerdem ist es möglich Referenzen auf Remote-Maschinen (per URL) anzugeben. Der Kontext verwaltet die Meta-Informationen über die virtuellen Maschinen und deren Speicherort. Auf Anfragen werden die Informationen an den Scheduler weitergegeben. Möchte der Nutzer eine Auswahlbezüglich der zu verwendenden VMs treffen, werden diese an das User-Interface ausgegeben.

3. **Verwaltung von Scan-Parametern und Ergebnissen**
   Der Kontext erhält vom Scheduler die Informationen über Scan-Ergebnisse. Weiterhin werden nach Abschluss eines Scans Informationen über Scan-Konfigurationen, Log Dateien und weitere Berichte von Zusatzmodulen gespeichert. Der Kontext verwaltet diese Daten, welche auf dem Dateisystem abgelegt sind.

4. **Blaupausen Skripte**
   Der Kontext speichert Informationen über Blaupausenskripte (Provisionierungs-Skripte für Scanner und Vulnerables, Scan-Szenarios, VM-Konfigurations-Skripte). Diese können ebenfalls vom Nutzer in die Test-Umgebung eingebracht werden. Die Skripte selbst werden auf dem Dateisystem abgelegt. Möchte der Nutzer eine Auswahl bezüglich der zu verwendenden Skripte tätigen, so werden alle verfügbaren Auswahlmöglichkeiten vom Kontext an das User-interface weitergegeben.

5. **Plug-Ins**
   Dem Nutzer wird die Möglichkeit geboten Plug-Ins in das System einzubringen. Diese sowie bereits vorhandene Erweiterungen werden vom Kontext verwaltet. Auf Anfrage vom User-Interface werden die verfügbaren Plug-Ins dem Anwender als Auswahlmöglichkeiten präsentiert oder an den Scheduler ausgeliefert.

M2 - User Interface (UI)

Das User Interface dient als visuelle Schnittstelle zwischen Anwender und Test-Umgebung. Der Nutzer nimmt alle Scan-Konfigurationen vor und bestimmt den Scan-Ablauf. Dieser ist bereits ausführlich in 4.3 beschrieben. Die einzelnen Schritte des Ablaufs können über das User Interface konfiguriert und gestartet werden (siehe: 4.5).

Aktivitäten

1. **Anzeige**
   Das User-Interface dient als Anzeige für mögliche Konfigurationseinstellungen des Scan-Prozesses. Die Informationen über verfügbare VMs und Skripte sowie mögliche Plug-Ins werden vom Kontext geliefert und dem User präsentiert.

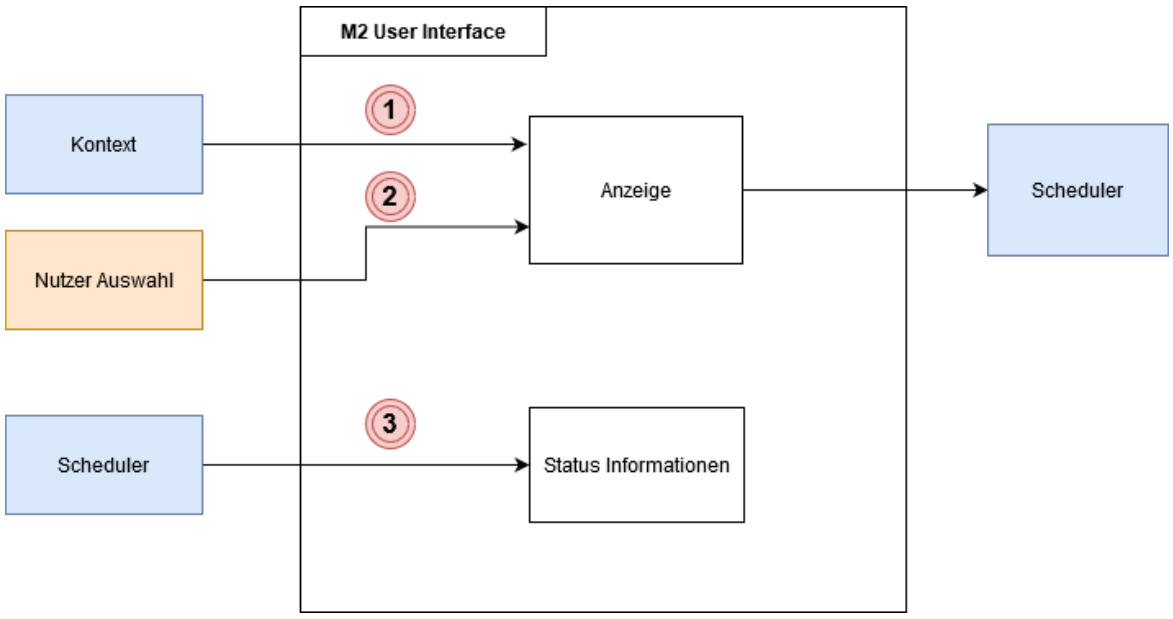

Abbildung 4.5: Modul 2 - User Interface

2. **Nutzer-Konfigurationen**
   Der Anwender trifft eine Auswahl bezüglich der Konfiguration des Scan-Prozesses (VMs, Scan-Konfigurationen, Skripte, Plug-Ins) und startet diesen schließlich. Diese Nutzer-Konfigurationen werden dem Scheduler übergeben.

3. **Status-Informationen**
   Das User-Interface empfängt aktuelle Status-Informationen über den laufenden Scan-Prozess und präsentiert diese dem Anwender.

M3 - Scheduler

Der Scheduler ist das Modul welches die Laufzeitumgebung verwaltet. Es übernimmt die Aufgabenzuteilung, startet die Aufträge und verwaltet die Kommunikation zwischen den anderen Modulen. Er sorgt für einen reibungslosen, nicht-blockierenden Ablauf der Pipeline, gibt Statusmeldungen und gegebenenfalls Fehlermeldungen aus.
Der Scheduler interagiert dabei nicht direkt mit dem Anwender, sondern ist im Hintergrund aktiv. Abbildung 4.6 zeigt alle entsprechenden Aktivitäten.

Aktivitäten

1. **Scanner-Vulnerable-Paarung**
   Der Scheduler empfängt die Nutzer -Konfigurationen vom User-Interface und erstellt daraus Scanner-Vulnerable-Paarungen, welche aus einer VM mit Schwachstellenscanner und der zu testenden Zielmaschine bestehen. Weitere Informationen wie Meta-Daten der gewählten virtuellen Maschinen und deren Speicherorte werden vom Kontext abgerufen. Ebenfalls werden alle notwendigen Blaupausen vom Kontext angefordert und in Laufzeit-Skripte übersetzt. Alle Konfigurationen und Skripte sowie die

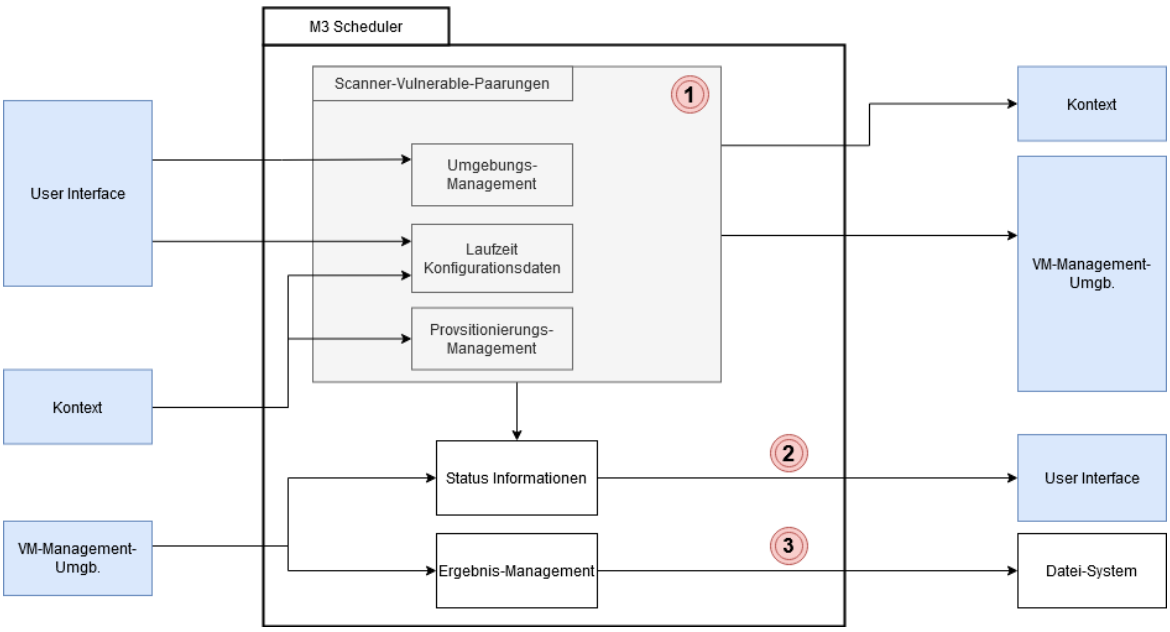

Abbildung 4.6: Modul 3 - Scheduler

VMs der aktuellen Paarung werden an die VM-Managementumgebung weitergegeben, welche für das Durchführen des Scans und das Aufzeichnen von Ergebnissen verantwortlich ist. Die erstellten Konfigurationsdaten und Skripte werden an den Kontext weitergeleitet damit diese persistent gespeichert werden. Dies ermöglicht das Nachvollziehen von Scan-Ergebnissen oder das erneute Ausführen von Scans zu späterem Zeitpunkt.
Der modul-übergreifende Scan-Prozess wird detaillierter in Kapitel 4.5 beschrieben.

2. **Status-Informationen**
   Status-Informationen über den aktuellen Scan-Prozess werden laufend an das User-Interface berichtet. Diese bestehen aus Informationen zum aktuellen Fortschritt aber ebenso aus Fehler-Informationen und Abbruchmeldungen. Auch Laufzeiten der einzelnen Abschnitte können hier überwacht werden. Weiterhin empfängt der Scheduler Status-Informationen der VM-Management-Umgebung und leitet diese ebenfalls an den Anwender weiter.

3. **Ergebnis-Management**
   Der Scheduler empfängt alle Scan-Ergebnisse und Log-Daten sowie Plug-In Informationen von der VM-Management-Umgebung und speichert diese persistent auf dem Datei-System ab.
   Mehr Details zum Ergebnis-Management finden sich in Kapitel 4.5.

M4 - VM-Management-Umgebung

Die VM-Managementumgebung ist für die Verwaltung aller virtuellen Maschinen verantwortlich. Sie nimmt ihre Anweisungen vom Scheduler-Modul entgegen. Folgende Aktivitäten können der Laufzeitumgebung zugeordnet werden (siehe: 4.7)

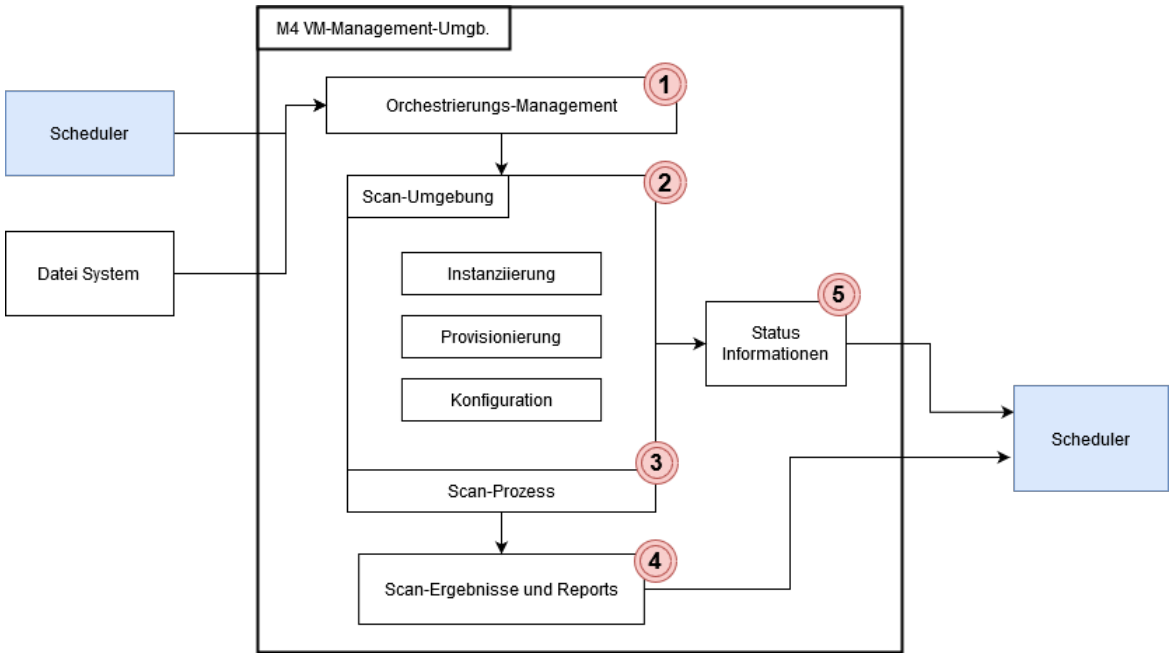

Abbildung 4.7: Modul 4 - VM-Management-Umgebung

Aktivitäten

1. **Orchestrierungs-Management**
   Die VM-Management-Umgebung erhält vom Scheduler alle Informationen über die zu startenden VMs. Diese bestehen aus Speicherort, Konfigurationen und Skripten. Vom Datei-System werden die entsprechenden Dateien ausgelesen. Der Orchestrierungs-Manager übernimmt die Verwaltung und Konfiguration alle virtuellen Maschinen und verwaltet diese in separaten Scan-Umgebungen.

2. **Scan-Umgebung**
   Die Scan-Umgebung wird vom Orchestrierungs-Management verwaltet. Diese instanziiert zunächst alle im kontext benörigten virtuellen Maschinen, provisioniert diese und konfiguriert diese.

3. **Scan-Prozess**
   Schließlich wird der Scan-Prozess selbst gestartet und es wird auf einen Abschluss des Prozesses gewartet.

4. **Scan-Ergebnisse und Reports**
   Nach Abschluss des Scan-Prozesses werden alle Scan-Ergebnisse und weitere Reports

(Log-Dateien, Plug-In Aufzeichnungen) and den Scheduler ausgegeben, wessen Aufgabe es ist die Daten zu speichern.

5. **Status-Informationen**
   Während des gesamten Prozesses werden Status-Informationen an den Scheduler zurückgegeben, sodass diese an den Nutzer weitergegeben werden können.

Der modul-übergreifende Scan-Prozess wird detaillierter in Kapitel 4.5 beschrieben.

4.5 Intermodulare Aktivitäten

Einige Aktivitäten lassen sich nicht innerhalb eines Moduls durchführen sondern ein Zusammenspiel verschiedener Module ist notwendig. Dieses Kapitel fasst diese Aktivitäten, und beschreibt diese Detaillierter wobei Referenzen auf die verwendeten Module gegeben werden. Besonders das Zusammenspiel von Scheduler und VM-Management-Umgebung ist hier relevant. Folgende Aktivitäten werden genauer betrachtet:

- Erstellung einer Scanner-Vulnerable-Paarung
- Konfiguration der virtuellen Maschinen
- Durchführung eines Scan-Prozesses
- Verwaltung der Scan-Ergebnisse und weiterer Dateien
- Einbindung gewählter Plug-Ins und Zusatzkomponenten

Erstellung einer Scanner-Vulnerable-Paarung

Wird ein Scan-Prozess angestoßen so wird mindestens eine Kombination aus Scanner und Vulnerable erstellt. Diese bilden zusammen eine Paarung, also ein logisch geschlossenes System, welches das Szenario ‘Scan eines Vulernables mithilfe eines Scanners’ abbildet. Wurden vom Anwender mehrere Scanner oder Vulnerables gewählt, so müssen mehrere solcher Paarungen erstellt und verwaltet werden. Jede dieser Umgebungen erhält eine eigene Umgebung, Laufzeit-Skripte, virtuelle Maschinen und Konfigurationen. Der Nutzer hat während der Interaktion mit dem User-Interface bereits ein Szenario gewählt, welches er abbilden möchte. Zunächst erstellt der Scheduler eine Umgebung für jede Paarung. Sowohl physisch als auch logisch wird eine Konfiguration des Systems vorgenommen.
Physisch wird eine Ordnerstruktur angelegt, die entsprechenden Skripte von Blaupausen vom Kontext angefordert und in ausführbare Skripte übersetzt und diese schließlich an den richtigen Stellen ablegt. Es finden sich Skripte für die Provisionierung und Konfiguration von VMs, den Scan-Prozess, Plug-Ins und eine Laufzeit-Konfigurationsdatei.
Logisch wird ein virtuelles Netzwerk für die Paarung erstellt, IP-Adressen vergeben, Credential-Management betrieben, eine Port Weiterleitung erstellt und Speicherverwaltung betrieben. Ebenfalls wird ein Ordner zur Synchronisation von Ergebnissen festgelegt und jede Paarung bekommt einen eindeutigen Namen.
Zusätzlich wird ein Ablauf festgelegt, nach welchem die Skripte ausgeführt werden sollen, virtuelle Maschinen gestartet und konfiguriert werden, welche Plug-ins eingebunden werden und wann die Ergebnisse ausgelesen werden.

Zuletzt wird für jede Paarung der Scan-Prozess angestoßen, welcher von der VM-Management-Umgebung durchgeführt wird.

Konfiguration der virtuellen Maschinen

Wenn der Scheduler einen Scan-Prozess in Auftrag gibt, dann werden zunächst die notwendigen virtuellen Maschinen instanziiert, konfiguriert und provisioniert.
Eine Orchestrierungssoftware (z.B. Vagrant, Multipass) nimmt dabei die Laufzeit-Konfigurationen und führt entsprechend des festgelegten Ablaufs die Konfiguration der virtuellen Umgebung vor. Zunächst werden mithilfe einer Virtualisierungssoftware (z.B. VirtualBox, VMWare) und deren API automatisiert Instanzen der vorhandenen Images erstellt. Diese werden dann konfiguriert und stellen im folgenden den Scanner und das Vulnerable dar. Eine Erweiterung um weitere Maschinen mithilfe von Plug-Ins (z.B. ein Management-Server) ist ebenfalls möglich und muss der Orchestrierungssoftware mitgeteilt werden. Diese VMs stellen ein geschlossenes System ohne beeinflussung durch andere Maschinen dar. Dazu befinden sie sich in einem eigenen Subnetz. Der Scanner bekommt im Standardfall nur die Vulnerable-IP-Adresse mitgeteilt (Black-Box-Test). Diese Vorinformationen können mithilfe benutzerdefinierter Skripte erweitert werden um weitere Szenarien zu repräsentieren (z.B. Grey-Box-Test). Schließlich werden alle benötigten Laufzeit-Skripte in der richtigen Reihenfolge ausgeführt. Diese unterscheiden sich in Skripte für die Scanner und solche für die Vulnerables.

Scanner-Skripte

Die Scanner können vor dem Start der Scans provisioniert und konfiguriert werden. So kann die Scanner-Software installiert werden (sofern noch nicht in den Images vorhanden) sowie weitere Software wie Plug-Ins oder Zusatzkomponenten notwendig für die Scans. Auch Reporting-Tools und Tools zur Auswertung von Ergebnissen sind hier sinnvoll. Software zur Überwachung der Leistungsparameter (z.B. CPU-Auslastung, Netzwerkauslastung) sowie Integritäts-Checker können ebenfalls auf Wunsch des Anwenders installiert werden.
Zur Konfiguration und Provisionierung kann entweder das klassische SSH oder Tools wie Chef oder Ansible verwendet werden. Dies ist dem Anwender selbst überlassen und die bekanntesten Tools werden generell von der Orchestrierungs-Software unterstützt.

Vulnerable-Skripte

Im Gegensatz zu den Scanner-Skripten muss auf den Vulnerables keine Scanner-Software installiert werden. Dennoch gibt es Szenarien in denen der Anwender Veränderungen an den Vulnerables vor den Scans vornehmen will. Es kann klassisch Software nachinstalliert werden, indem diese von Repositorys wie dem 'Advanced Packaging Tool (apt)' installiert werden. Diese Software kann dazu dienen ein bestimmtes Szenario darzustellen (z.B. ein Onlineshop als Referenz zum erkennen von Web-Schwachstellen). Es kann ebenfalls Software mit bekannten Schwachstellen installiert werden. Dies kann als Referenz für die Erkennungsrate von Schwachstellenscannern dienen. Möchte der Anwender die Veränderungen auf dem Vulnerable beobachten, welche durch einen Scan-Vorgang geschehen, so ist es sinnvoll ein Tool zu installieren, welches die Softwareintegrität überwacht (z.B. tripwire-open-source). Dies geschieht durch den Vergleich mit einer festgelegten Baseline. Diese markiert alle Veränderungen die erkannt werden sollen (z.B. Einträge in der passwd-Datei). So kann der Erfolg und die Integrität der Vulnerables bei der Verwendung von Scannern, welche aktiv Exploits

einsetzen getestet werden.
Ebenfalls können bestimmte Konfigurationen wie Firewall-Einstellungen vorgenommen werden.

Durchführung eines Scan-Prozesses

Die Kombination vorher beschriebener Techniken führt zu folgender Pipeline, welche während eines Scan-Prozesses mithilfe dieser Test-Umgebung durchgeführt wird (siehe: 4.8).
Bevor der Scan durchgeführt werden kann, müssen zunächst einige Prozesse ablaufen um die notwendige Infrastruktur zu schaffen. So beginnt ein Scan mit der Erstellung eines Paars aus Scanner und Vulnerable. Ein logisches Netzwerk wird definiert, die vom Anwender spezifizierten Skripte werden als Blaupausen angefordert und Informationen über die virtuellen Maschinen eingeholt (0).
Mithilfe dieser Informationen kann schließlich das Vulnerable erstellt und konfiguriert werden. Im ersten Schritt wird mithilfe der Virtualisierungssoftware eine Instanz eines im Kontext verwalteten Images als Vulnerable erstellt (1). Diesem werden über die API der Virtualisierungssoftware bestimmte zuvor in (0) definierte Konfigurationen wie die IP-Adresse, ein Name, Credentials aufgespielt (2). Schließlich werden alle Blaupausen in ausführbare Skripte übersetzt und mit den Notwendigen Informationen angereichert (z.B. IP der Zielmaschine, Betriebssystem-Spezifische-Befehle) (3). Die Skripte werden auf das Vulnerable aufgespielt und schließlich ausgeführt. Die Provisionierung und Konfiguration der VM sind somit abgeschlossen.
Im nächsten Schritt (5) wird derselbe Prozess für den Scanner durchgeführt. So wird für diesen ebenfalls eine Instanz erstellt und die Schritte 2-4 durchgeführt. Während dieses Prozesses wird gegebenenfalls ebenfalls der Schwachstellenscanner installiert und Schwachstellen-Datenbanken aktualisiert. Dem Scanner wird zusätzlich das Scan-Skript übertragen, welches zuvor ebenfalls in eine Ausführbare Datei übersetzt und mit der IP-Adresse des Vulnerables angereicht wird (6). Schließlich wird der Scan gestartet (7). Das Skript übernimmt die Kommunikation mit dem installierten Schwachstellenscanner und dessen API. Es legt einen Nutzer an, erstellt einen Scan basierend auf den mitgegebenen Konfigurationen, legt die Ergebnis-Parameter fest und startet schließlich den Scan..
Zum Schluss werden die Scan-Ergebnisse ausgelesen und in dem Datei-System abgelegt (8). Weiterhin werden natürlich alle weiteren Dateien von Belang nach Abschluss des Scans sowie die Konfigurationsdateien auf das Datei-System gespeichert, bevor die gesamte Umgebung wieder zerstört wird (9). Nun kann der Scheduler - falls notwendig - weitere Scanner-Vulnerable-Paarungen ausführen (0).

Verwaltung der Scan-Ergebnisse und weiterer Dateien

Eine Test-Umgebung kann nur sinnvoll genutzt werden, wenn es möglich ist Schlüsse aus den durchgeführten Tests zu ziehen. Dazu werden Scan-Ergebnisse und alle weiteren Informationen aufgezeichnet und zur späteren Auswertung ausgegeben. Die Ergebnisse liegen dazu persistent vor und müssen somit außerhalb der Laufzeitumgebung gespeichert werden, da diese nach Abschluss eines Scan-Prozesses zerstört wird. Das Dateisystem mit einer entsprechenden Struktur dient als solche persistente Umgebung, verwaltet von dem Kontext, welcher Referenzen auf alle Daten hält und auf Anfrage ausgibt. Die Daten werden während bereits während der Laufzeit aber mindestens vor Zerstören der Umgebung auf das Host-Datei-System synchronisiert oder Ergebnisse erstellt. Folgende Daten werden während einem

Scan-Prozess erzeugt und gespeichert:

- Die Ergebnisse der Scans in dem gewünschten Format. Diese liegen menschenlesbar und maschinenlesbar vor.
- VM-Konfigurations-Dateien, welche ein Wiederholen des Scan-Prozesses zu späterer Zeit ermöglichen.
- Provisionierungs-Skripte zur späteren Nachvollziehbarkeit und Wiederholbarkeit
- Scan-Skripte der jeweiligen Paarungen
- (Optional) Integritäts-Verstöße bei den VMs
- (Optional) Aufzeichnungen der Scanner Befehle über das Netzwerk
- (Optional) Log-Dateien verschiedener Plug-Ins
- Status-Informationen des Scan-Prozesses zur Fehleranalyse

Die Fülle an Ergebnissen und Dateien, beugt der Übersichtlichkeit und einfachen Handhabung der Umgebung vor. Dementsprechend wird vor Beendigung des Prozessen, sobald alle Ergebnisse vorliegen eine Übersichtsseite erstellt, welche Links mit Beschreibungen zu den entsprechenden Ergebnissen hält oder diese menschen-lesbarer darstellen. Die Daten liegen dennoch ebenfalls im Rohformat vor um eine weitere Verarbeitung zu gewährleisten. Ein Bild der Übersichtsdatei ist im Anhang zu sehen (siehe: 7.2).

Einbindung gewählter Plug-Ins und Zusatzkomponenten

Sollen beispielsweise Scan-Ergebnisse weiter verarbeitet werden können, nachdem der Scan-Prozess abgeschlossen ist oder weitere nicht in der Basis-Test-Umgebung enthaltene Untersuchungen durchgeführt werden, ist es möglich eigene Plug-Ins in das System einzubringen. Das Einbinden des Integrations-Checkers könnte bereits als Zusatzkomponente bezeichnet werden. Ein Beispiel für ein weiteres Plug-In kann das einbauen eines Proxys zur allgemeinen Verbesserung der Scan-Ergebnisse sein, mit dem Zweck dessen Effizienz mithilfe der Test-Umgebung zu überprüfen [ERRW18]. Außerdem könnten Skripte zur automatisierten Ausnutzung gefundener Schwachstellen eingebaut werden [Jos19]. Der Ausblick 7 zeigt weitere Anwendungsmöglichkeiten der Test-Umgebung sowie mögliche Plug-Ins, um welche diese erweitert werden kann.
Damit das Einbinden von Plug-Ins einfach umsetzbar ist, wurde die Test-Umgebung modular gestaltet. Somit wäre es ebenfalls möglich, bei bedarf das User-Interface oder weitere Komponenten zu ersetzen. Eine gute Dokumentation ist ebenfalls essenziell für die Integration von Plug-Ins. Der Quellcode ist ebenfalls offen verfügbar, um den breiten Einsatz der Umgebung zu fördern.. Laufzeitparameter finden sich in festgelegten Orten und können von dem Scheduler abgefragt werden. Die Nutzung diverser APIs erlaubt eine weitere Konfiguration. Die Verwaltung der VMs über eine Virtualisierungs-Software und der Verwendung deren APIs ermöglicht es ebenfalls jeden Aspekt der Umgebung benutzerdefiniert zu erweitern.

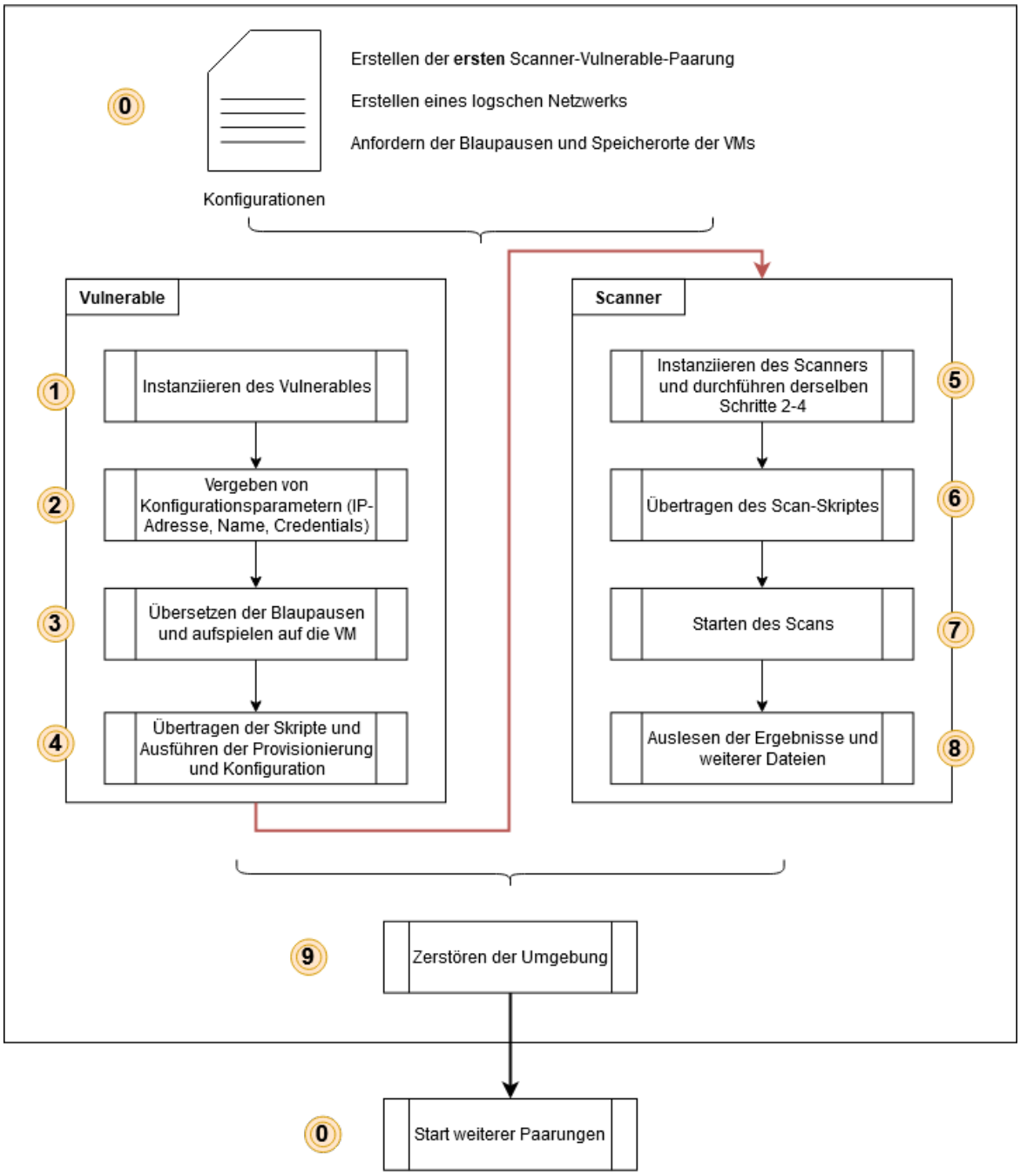

Abbildung 4.8: Übersicht des Ablaufs eines Scans mithilfe der Test-Umgebung

5 Referenzimplementierung einer Test-Umgebung

Die prototypische Referenzimplementierung basiert auf dem in Kapitel 4 präsentierten Konzept. Sie setzt alle genannten Aspekte praktisch um und erfüllt somit die in Kapitel 3 identifizierten Anforderungen.
Dieses Kapitel beschreibt die Implementierung der Test-Umgebung für Schwachstellenscanner auf Basis des Windows Betriebssystems mithilfe der Software Vagrant und programmatisch basiert die Test-Umgebung auf PowerShell und Ruby. Die VM-Spezifischen Skripte sind in der jeweiligen Shell-Sprache verfasst.
Der Prototyp wird schließlich in Kapitel 6 anhand verschiedener Szenarien auf seine Funktionalität und Vollständigkeit überprüft.

5.1 Aufbau des Prototypen

Der Prototyp folgt dem modularen Aufbau des Konzeptes. Es ist in die vier Module Kontext, User-Interface, Scheduler und VM-Managementumgebung gegliedert, welche folgend beschrieben werden.

5.1.1 Kontext

Der Kontext (Modul 1) bietet dem Anwender die Möglichkeit die Test-Umgebung um benutzerdefinierte Einstelllungen, Skripte oder Images zu erweitern. Möchte man diese in den Kontext einspeisen, so ist dies über zwei Methoden möglich. Entweder nutzt man ein entsprechendes Skript mit dem Namen kontext.ps1 (PowerShell-Skript) und lässt sich interaktiv durch den Prozess leiten. Oder man legt die Daten an den richtigen Stellen auf dem Datei-System ab. Dies ist möglich, da der Kontext bei jeder Anfrage von anderen Modulen das Datei-System durchsucht und aktuelle Informationen zurückgibt (z.B. wählbare Scan-Skripte). Somit können Informationen bis zu dem Zeitpunkt an dem der Scan gestartet - und somit alle Daten bereits allokiert wurden - in den Kontext eingefügt werden, ohne die Laufzeitumgebung neu starten zu müssen. Es muss einzig das User-Interface aktualisiert werden.
Der Prototyp beinhaltet bereits einige Scanner, Vulnerables sowie Scan-Skripte und Konfigurations- und Provisionierungs-Skripte, damit das System bereits Out-Of-The-Box verwendet werden kann. Die Skripte liegen als Blaupausen vor und werden während der Laufzeit in ausführbare Skripte übersetzt.
In der Test-Umgebung findet sich beispielsweise ein OpenVAS und nikto Scanner, ein metasploitable3 Vulnerable, sowie ein OpenVAS Full and Fast Scan-Skript.
Folgender Code (siehe: 5.1) zeigt den Ausschnitt eines Blaupausen Skriptes, welches zur Laufzeit übersetzt wird.
Man erkennt die PowerShell Umgebungsvariablen anhand des Präfix “**$env:**.

```
config.vm.define "vulnerable" do |v|
v.vm.box = "$env:VULNERABLE_BOXNAME"
v.vm.network "private_network", ip: "$env:IP_VULNERABLE"
v.vm.synced_folder "sync/", "$env:PATH_VULNERABLE_SYNC", create:
    true
v.vm.provision "shell", path: "$env:
    VULNERABLE_PROVISIONING_FILE_NAME"
if $env:monitoring == 1
  v.trigger.before :destroy do |trigger|
        trigger.run_remote = {inline: "tripwire --check > /vagrant
           /tripwire_log.txt; echo 'Tripwire log copied'"}
  end
end
```

Listing 5.1: Ausschnitt aus einem Skript mit Umgebungsvariablen

Hat bereits ein Scan stattgefunden, kann der Nutzer sich dessen Ergebnisse ausgeben lassen, indem er den gewünschten Scan aus einer Übersicht aller bisheriger Scans auswählt. Er erhält die Möglichkeit sich ein Ergebnis-HTML-Dokument anzeigen zu lassen, welches Übersichten und Links zu allen Files und Logs enthält (z.B. Informationen über Veränderungen auf den Zielmaschinen durch den Schwachstellenscanner)

5.1.2 User-Interface

Die nächste Komponente ist das User-Interface (Modul 2). Aufgrund von überschaubaren Konfigurationen wurde ein interaktives Menü auf Shell-Basis gewählt. Diese reicht aus um die Konfiguration durch den Nutzer zu genüge abzudecken, sodass dieser in der Lage ist die Funktionalität der Software zu verstehen und produktiv zu verwenden.
Die Software kann durch einen Konsolen-Befehl ausgeführt werden. Eine interaktive Shell öffnet sich und der Nutzer nimmt seine Konfigurationen vor.

Beispielhafte Interaktion:

- Nutzer startet die Software in der Konsole
- Der Nutzer entscheidet sich über ein Auswahlmenü für einen oder mehrere VMs mit installiertem Schwachstellenscanner (z.B. OpenVAS)
- Weiterhin wählt der Nutzer einen Scan Modus aus (z.B. OpenVAS Full and Fast)
- Nutzer wählt nunmehr zunächst einen oder mehrere Vulnerables (z.B. Ubuntu 13 & 14)
- Schließlich startet der Nutzer die den Scan-Prozess und wählt ein Plug-In aus, dass Veränderungen auf den Vulnerables aufzeichnet (z.B. durch tripwire-open-source)
- Nutzer wartet auf Abschluss der Scans und kann den Prozess in der Konsole über Fortschritts-Anzeigen verfolgen.

```
Testbed Menu: Choose a number to start the configurations.
When you are ready start the scan!
1: Choose a Scanner.
2: Choose a Vulnerable.
3: Choose Scan-Configurations.
4: Start the Scan-Process.
```

Abbildung 5.1: User-Interface der Test-Umgebung

5.1.3 Scheduler

Der Scheduler (Modul 3) funktioniert im Hintergrund, nimmt die Konfigurationen des Nutzers aus dem User-Interface entgegen. Anschließend verwandelt er diese in einen Programmfluss, welcher sequenziell abgearbeitet wird. Jede Scanner-Vulnerable-Paarung wird nacheinander durchgeführt. Dazu erstellt der Scheduler zunächst eine Ordnerstruktur auf dem Datei-System mit einem eindeutigen Namen. Schließlich werden die Konfigurationsdateien an diesen Ort erstellt, Blaupausen mit Laufzeit-Informationen angereichert und diese Dateien schließlich in die Ordner kopiert. Ebenfalls wird ein Ordner zur Synchronisation von Ergebnis-Daten bestimmt und angelegt. Diese Daten sind persistent und dienen später der Wiederholbarkeit des Scan-Prozesses oder der Auswertung der Ergebnisse.

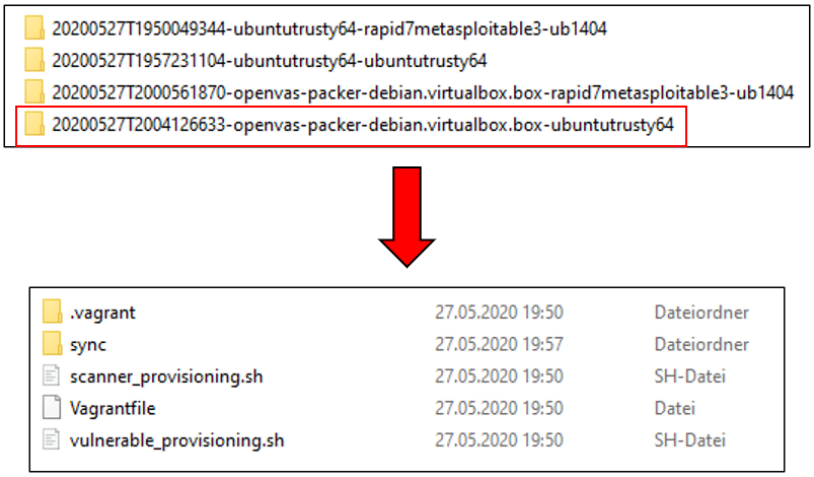

Abbildung 5.2: Die Ordnerstruktur eines Scans mit zwei Scannern und zwei Vulnerables

Nun gibt der Scheduler den Scan-Prozess in Auftrag, indem das entsprechende Pipeline-Skript ausgeführt wird. Dieser Prozess wird von der VM-Management-Umgebung übernommen.
All diese Schritte werden für jede Paarung durchgeführt.
Nach Abschluss der Scans speichert der Scheduler alle Ergebnisse, erstellt eine Übersichts-HTML-Datei (siehe: 7.2) und zerstört schließlich die Laufzeitumgebungen, um den Speicher wieder frei zu machen.

5.1.4 VM-Management-Umgebung

Wenn die VM-Management-Umgebung (Modul 4) vom Scheduler den Befehl erhält einen Scan durchzuführen, wird ihm der entsprechende Ordner mit den Dateien mitgeteilt, welcher alle Konfigurationsdateien sowie das Pipeline-Skript enthält. In diesem Fall übernimmt die Orchestrierungssoftware Vagrant die Aufgabe, die virtuellen Maschinen und den Scan-Prozess zu verwalten. Dementsprechend erstellt der Scheduler eine sogenannte **Vagrantfile**, welche das Pipeline-Skript im Vagrant-Kontext repräsentiert. Diese wird aus einer Vagrantfile-Blaupause generiert und mit Informationen angereichert, welche zur Laufzeit notwendig sind. Der folgende Code (siehe: 5.2) zeigt den Ausschnitt der vorherigen Blaupause (siehe: 5.1) angereichert mit Laufzeitinformationen.

```
config.vm.define "vulnerable" do |vulnerable|
  vulnerable.vm.box = "rapid7/metasploitable3-ub1404"
  vulnerable.vm.network "private_network", ip: "172.16.16.3"
  vulnerable.vm.synced_folder "sync/", "/vagrant"
  vulnerable.vm.provision "shell", path: "vuln_provisioning.sh"
  if 1 == 1
        vulnerable.trigger.before :destroy do |trigger|
         trigger.run_remote = {inline: "tripwire --check > /
            vagrant/tripwire_log.txt; echo 'Tripwire log copied'"}
    end
  end
end
```

Listing 5.2: Ausschnitt aus einem generierten Vagrantfile

Mithilfe einer Virtualisierungssoftware (hier VirtualBox) und dessen API instanziiertVagrant zunächst die Scanner-VM aus einem lokalen oder remote Image und schließlich die Vulnerable-VM.

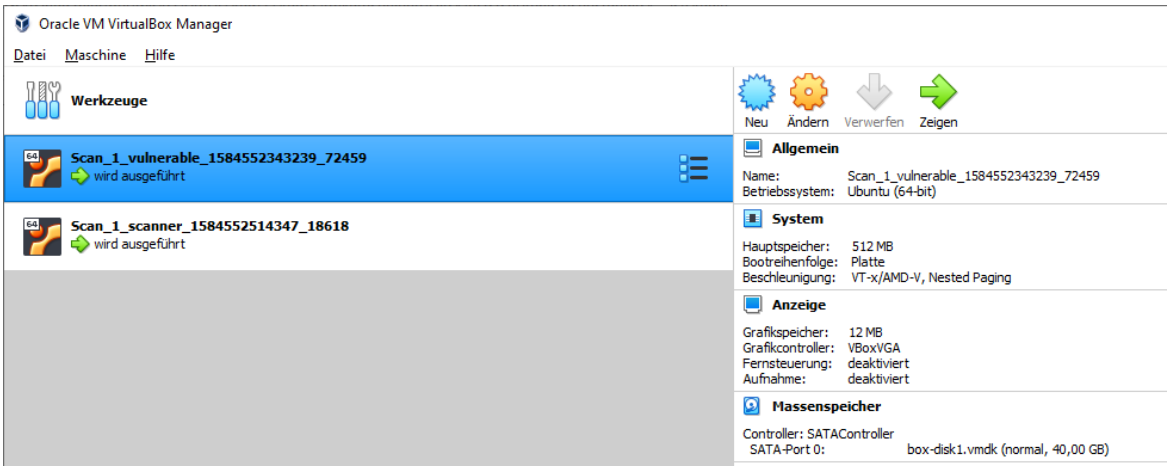

Abbildung 5.3: Instantiierte VMs während einem Scan-Prozess

Diese werden konfiguriert (z.B. Zuweisung einer IP-Adresse) und provisioniert (z.B. aktualisieren der Schwachstellendatenbank).
Alle notwendigen Dateien (z.B. Skripte) werden dabei ebenfalls auf die VMs aufgespielt und an den entsprechenden Orten abgelegt. Dies funktioniert über einen SSH Zugriff. Sind alle Maschinen gestartet und konfiguriert, werden gegebenenfalls Plug-Ins ausgeführt (z.B. erstellen einer Baseline für Tripwire).
Genannte Schritte sind abhängig von dem gewählten Scanner-Szenario und den gewählten Plug-Ins durch den Nutzer.
Schließlich wird der Scan-Prozess selbst gestartet, indem das Scanner-Skript auf der Scanner-VM ausgeführt wird. Die dabei entstehenden Ergebnisse werden an dem zuvor bestimmten Synchronisations-Ordner abgelegt und somit persistent gemacht.
Bevor der Prozess beendet wird, werden nochmals etwaige Plug-In Skripte mithilfe sogenannter Vagrant-Trigger ausgeführt. So können Log-Dateien oder Status-Information vor dem zerstören der Maschinen vom System kopiert werden (z.B. Leistungsparameter-Daten während des Scan-Prozesses).
Während der Laufzeit gibt die VM-Management-Umgebung laufend Statusinformationen über den Scan-Prozess an den Scheduler zurück. Die Informationen sind farbig voneinander abgehoben, um dem Nutzer einen umfassenden Überblick zu geben. Fehler werden ebenfalls auf der Konsole ausgegeben.

```
==> scanner: Adding box 'openvas-packer-debian.virtualbox.box' (v0) for provider: virtualbox
    scanner: Downloading: openvas-packer-debian.virtualbox.box
    scanner:
An error occurred while downloading the remote file. The error
message, if any, is reproduced below. Please fix this error and try
again.

Couldn't open file C:/Users/robin/Desktop/vagrant_projects/meta_double/Scans_20200527T1950048964/20200527T2
004126633-openvas-packer-debian.virtualbox.box-ubuntutrusty64/openvas-packer-debian.virtualbox.box
All machines are being destroyed (some scripts can be triggered during this process so this could take a wh
ile before the machines are actually destroyed)
==> scanner: VM not created. Moving on...
==> vulnerable: Running action triggers before destroy ...
==> vulnerable: Running trigger...
==> vulnerable: Checking tripwire against the baseline for differences
    vulnerable: Running: inline script
    vulnerable: Tripwire log copied
==> vulnerable: Forcing shutdown of VM...
==> vulnerable: Destroying VM and associated drives...
This script took 2 minutes to run and now destroys the machines
```

Abbildung 5.4: Konsolen Ausgabe nach Fehler innerhalb eines Skriptes

5.2 Implementierungsdetails

Dieser Abschnitt beschreibt den Implementierungsprozess genauer und geht auf Besonderheiten, Probleme und Details des Prototypen ein.

Voraussetzungen und verwendete Software
Damit die Test-Umgebung verwendet werden kann muss folgende Software installiert sein:

- Windows 10 (v1903) mit PowerShell (root)
- Vagrant (v2.2.6) mit vagrant-vbguest Plug-In

- VirtualBox (v6.1) mit installierten Guest-Additions
- Internetzugang

Windows 10 & PowerShell
Die Module Kontext, Scheduling und User-Interface sind mit PowerShell programmiert. PowerShell muss als Administrator auf der Host-Maschine laufen. Es werden Windows-Umgebungsvariablen verwendet, um Blaupausen mit Informationen anzureichern. Diese werden während dem Konfigurationsprozess durch den Nutzer im Hintergrund entsprechend der Auswahl gesetzt. Der folgende Befehl (siehe: 5.3) übersetzt eine Blaupause durch die entsprechenden Werte. Das vollständige Blaupausen Vagrantfile des Prototypen findet sich im Anhang (siehe: 7.2)

```
Get-Content $global:working_dir\env_vagrantfile.sh |
ForEach-Object { $ExecutionContext.InvokeCommand.ExpandString($_) }
> "$scope_dir\Vagrantfile"
```

Listing 5.3: Der PowerShell-Befehl um eine Blaupause mithilfe von Umgebungsvariablen in ein ausführbares Skript zu übersetzen.

Entwickelt und getestet ist die Software auf Windows 10 Home und Education. Der Windows Update Prozess hat während der Entwicklung keine Probleme verursacht.

Vagrant
Ist eine Anwendung von HashiCorp zum Erstellen und Verwalten von virtuellen Maschinen (siehe: `vagrantup.com`). Einfache Provisionierung und Konfiguration ist mithilfe von Software wie Chef, Docker oder Ansible aber auch durch die Shell unterstützt. Vagrant basiert auf der Programmiersprache Ruby und die wichtigste Konfigurations-Datei - die **Vagrantfile** - unterstützt ebenfalls Ruby Syntax und lässt sich somit programmatisch anpassen. Die Vagrantfile liegt zunächst als Blaupause vor und wird zur Laufzeit mit Informationen angereichert. Diese umfassen wichtige Konfigurierungsdetails wie IP-Adressen oder zu verwendende Plug-Ins.
Alle genannten Standards basieren auf Version 2.2.6. Die Konfiguration und Provisionierung des Scanner-Vulnerable-Paars sowie weiterer virtueller Maschinen (z.B. Management-Server)wird über die Vagrantfile gesteuert. Die Test-Umgebung wird zudem durch das Plug-In 'vagrant-vbguest' erweitert, um während der Laufzeit die Virtualbox-Guest-Additions der Gast-Maschinen auf die von VirtualBox unterstützte Version zu bringen. Dies beugt Fehlern vor allem bei der Synchronisation von Ordnern vor.
Probleme entstehen, wenn entweder VirtualBox auf eine neue Version aktualisiert wird, oder VMs mit veralteten Guest-Additions verwendet werden.
Nachdem VirtualBox eine neue Version veröffentlicht hat kann es bis zu einem halben Jahr dauern, bis Vagrant seine Software auf den aktuellen Stand gebracht hat. Eigene Workarounds sind jedoch möglich. Vagrant dient der Test-Umgebung als VM-Management-Umgebung und Orchestrierungssoftware zum Starten und Verwalten aller Virtuellen Maschinen.
Die Software verwendet sogenannte 'Vagrant Boxen' (angepasste Virtuelle Maschinen im .box Format). Sollen individuelle Virtuelle Maschinen angeboten werden, so müssen diese

als Vagrant Boxen entweder in der Vagrant Cloud oder als .box-Datei auf dem Host System vorliegen.
Eigene Boxen können mithilfe der ebenfalls von HashiCorp stammenden Software Packer erstellt werden. Im Anhang wird eine Anleitung zum Erstellen eines eigenen Images gegeben (siehe: 7.2). Dies ist vor allem dann sinnvoll, wenn die der zu verwendenden Software (z.B. OpenVAS) lange Zeit beanspruchen würde. Dies verlangsamt den Scan-Prozess enorm. Ebenfalls sinnvoll ist es, die Guest-Additions auf aktueller Version zu verwenden, da die Nachinstallation ebenfalls einen hohen Zeitaufwand bedeutet.
Um weiterhin Zeit einzusparen ist es sinnvoll die Pipeline von Vagrant zu betrachten (siehe: 5.5) und die Konfigurationen entsprechend vorzunehmen. So reicht der einmalige Import einer Vagrant Box mittels des Befehls “vagrant box add”. Dabei kann entschieden werden, ob das Image aus der Vagrant Cloud stammen soll, von einem ‘Remote Repository’ oder von der Lokalen Maschine. Anschließend muss zur Laufzeit einzig eine Instanziierung der Maschinen vorgenommen werden. Die Zeit des Downloads eingespart werden kann. Ist die

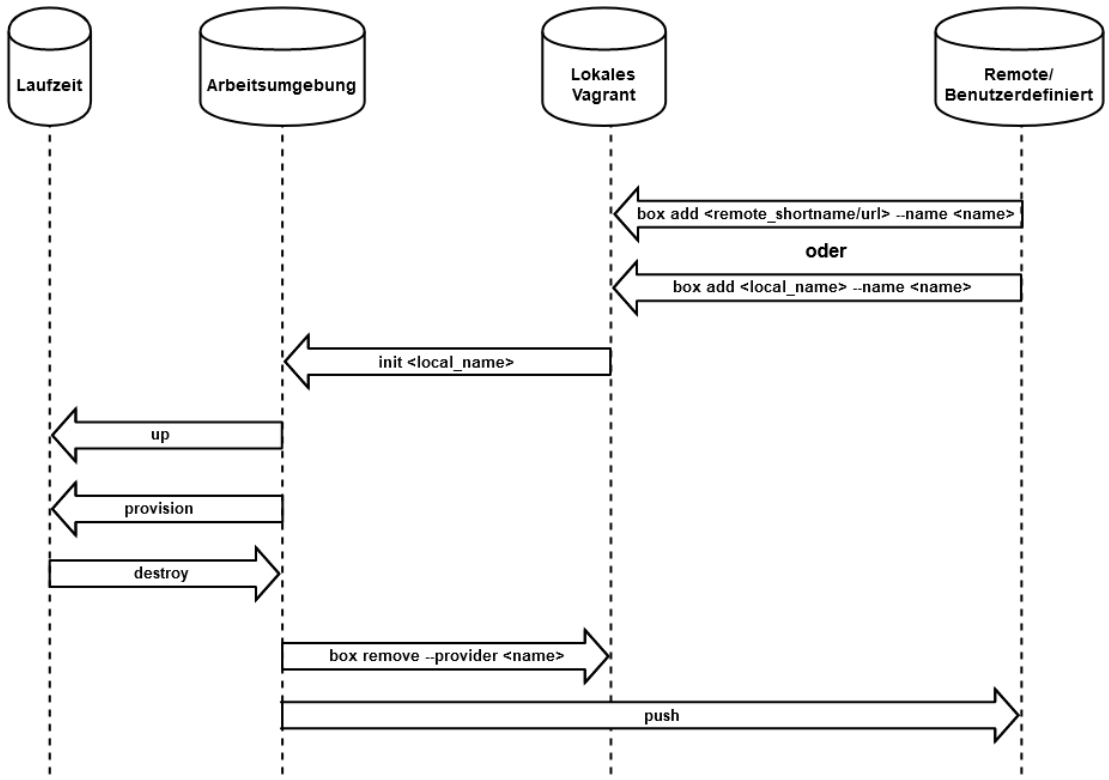

Abbildung 5.5: Zeigt die Pipeline von Vagrant und wie die Verwaltung der Maschinen geschieht.

zu verwendende virtuelle Maschine bereits als Image auf der Arbeitsumgebung vorhanden ist die Laufzeit deutlich verringert, da diese nicht aus dem Internet kopiert werden muss.Hängt ebenfalls keines der verwendeten Skripte von Remote Paketverwaltungen (z.B. apt) ab ist sogar eine Nutzung ohne Internet möglich, was besonders in kritischen Umgebungen sinnvoll ist.

Konfiguration und Provisionierung
Die Konfiguration und Provisionierung sind abhängig von den verwendeten Skripten, welche vor dem Start der Tests durch den Nutzer gewählt werden können. Diese basiert standardmäßig auf der einfachen Ausführung eines dem Guest-Betriebssystem entsprechenden Shell-Skriptes.Es ist jedoch ebenfalls möglich - durch leichte Anpassungen bei der Generierung der Vagrantfile - Software wie Chef, Docker oder Ansible als Konfigurations- und Provisionierungstools zu verwenden.

Eingebundene Plug-Ins
Plug-Ins erweitern die Funktionalität der Basis-Testumgebung. Um zu demonstrieren, dass und wie es möglich ist die Test-Umgebung zu erweitern, wurden bereits folgende Plug-Ins eingebunden. Abbildung 5.6 zeigt die Plug-Ins und deren korrespondierenden Module grafisch auf.

- vagrant-vbguest (VirtualBox Guest Addition Updater)
- tripwire (Integritäts-Checker)
- Wireshark (Netzwerkanalyse-Tool)
- OpenVAS-Konfig (OpenVAS Scan-Konfigurations-Manager; Eigenentwicklung)
- nmon (Performance Monitoring Tool)

Dabei erweitert vagrant-vbguest die VM-Management-Umgebung (Modul 4.4), OpenVAS-Konfig das User-Interface (Modul 4.4) und die Scan-Skripte, Wireshark, nmon und tripwire die Guest-Maschinen selbst. Man sieht, dass eine Erweiterung der Test-Umgebung auf allen Ebenen möglich ist.
Soll ein neues Plug-In eingebunden werden, müssen an folgenden Stellen Veränderungen vorgenommen werden. Je nachdem welches Modul das Plug-In unterstützt, kann diese Konfiguration etwas abweichen. Eine detaillierte Beschreibung des Vorgangs befindet sich im Anhang (siehe: 7.2).
Zunächst muss eine globale Umgebungsvariable für das Plug-In angelegt und initialisiert werden, damit später überprüft werden kann, ob das Plug-In aktiv ist. Schließlich muss der Plug-In Abschnitt in die Blaupausen Vagrantfile eingefügt werden, welcher mittels der gesetzten Umgebungsvariable zur Laufzeit aktiviert werden kann. Dieser Teil beinhaltet etwaige Skripte oder Befehle, welche an den entsprechenden Stellen ausgeführt werden sollen. Zuletzt muss der Abschnitt zur Interaktion für den Nutzer hinzugefügt werden. Dies entspricht einer Abfrage, ob das Plug-In aktiviert werden soll, verbunden mit der Aktivierung der Umgebungsvariablen. Außerdem wird somit das Plug-In einem String Array hinzugefügt, welches den Nutzer vor dem Start des Scans alle getätigten Konfigurationen aufzählt. Werden Ergebnis-Dateien erzeugt (z.B. Logs) müssen diese in dem Synchronisierten Ordner abgelegt werden, damit diese in der Ergebnis-Gesamtübersicht erscheinen.
Diese Test-Umgebung beinhaltet die beschriebenen Plug-Ins Out-Of-The-Box. Dennoch sind natürlich diverse weitere Plug-Ins sinnvoll, welche jedes Modul der Test-Umgebung um weitere Funktionen anreichern können. Einige denkbare Erweiterungen sind im Anhang 7.2 aufgelistet.

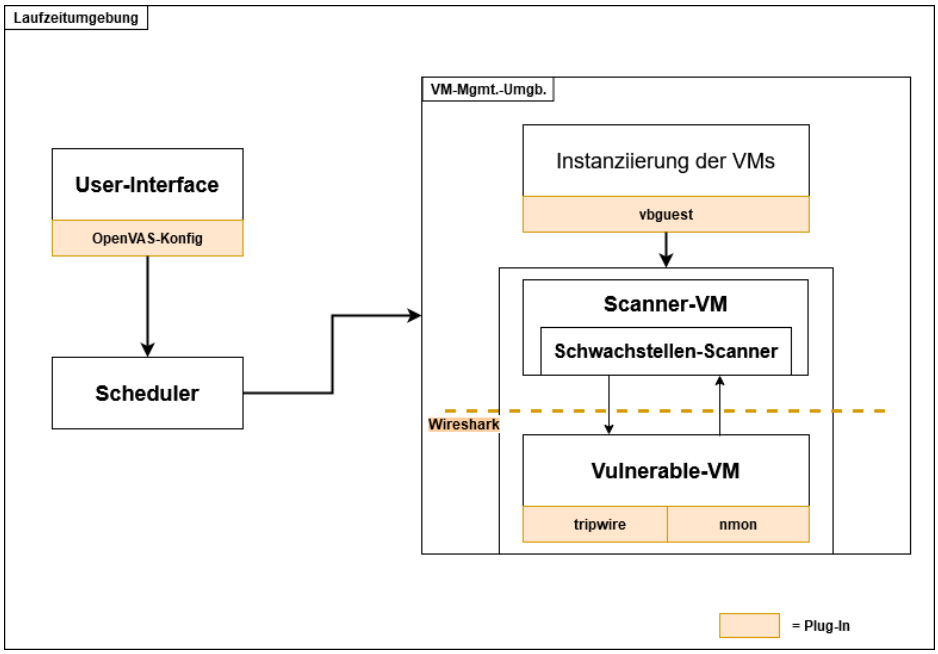

Abbildung 5.6: Verwendete Plug-Ins und die beeinflussten Komponenten der Test-Umgebung

6 Auswertung

Die letzten Kapitel haben Anforderungen an eine Test-Umgebung definiert, sowie aus diesen ein Konzept abgeleitet, wie eine solche Umgebung umgesetzt werden kann. Schließlich wurde die darauffolgende Referenzimplementierung durchgeführt.
Dieses Kapitel überprüft, ob die Referenzimplementierung alle definierten Anforderungen umsetzt. Nachdem der Prototyp nach dem Konzept umgesetzt ist bedeutet dies auch, dass das Gesamt-Konzept vollständig ist und die eingangs gestellten Forschungsfragen 1 abschließend beantwortet werden können.
In Abschnitt 6.1 werden die in Kapitel 3 beschriebenen Anforderungen an eine Test-Umgebung mit dem Prototypen abgeglichen und eine Auswertung erstellt, inwiefern die Referenzimplementierung diese erfüllt 6.1. Anschließend definiert Abschnitt 6.2 zwei Use-Cases, welche die Funktionalität der Test-Umgebung qualitativ bewerten und Schlüsse auf die sinnvolle Verwendbarkeit der Test-Umgebung geben.

6.1 Überprüfen der Anforderungen

In diesem Abschnitt wird überprüft, ob der Prototyp alle Anforderungen erfüllt, welche in Kapitel 3 definiert wurden. Dazu folgt eine Aufzählung der Anforderungen und eine Beschreibung, inwiefern die Test-Umgebung diese umsetzt. Die Anforderungs-Bestätigungen AB-1 bis AB-12 stellen die Überprüfung der Anforderungen A-1 bis A-12 aus Kapitel 3 dar. Im Anschluss wird eine Liste mit Schwächen der Test-Umgebung aufgeführt, bevor eine Gesamtübersicht den Abschnitt abschließt.

AB-1 - Analyse von Testmaschinen mithilfe von Schwachstellenscannern

Mithilfe der Testumgebung ist es möglich mit beliebig vielen Scanner mit beliebig viele Vulnerables zu scannen. Jedem Scan wird eine neue Instanz des Scanners und Vulnerables zugeordnet, sodass keine Beeinflussung stattfindet. Es wird ein virtuelles Netzwerk erstellt, um einen Black-Box-Test zu simulieren. Der Nutzer hat die Kontrolle über die verwendeten Skripte. Somit kann er bestimmten, welche Vorinformationen dem Scanner übergeben werden. Im Standardfall eines Black-Box-Tests ist nur die IP-Adresse notwendig. Diese wird von der Test-Umgebung automatisch vergeben.

AB-2 - Auswahl von Schwachstellenscannern

Man kann im User-Interface zwischen allen verfügbaren Schwachstellenscannern unterscheiden und beliebig viele auswählen. Es werden einige virtuelle Maschinen mit vorinstallierten Scannern angeboten, sodass die Umgebung Out-Of-The-Box bereits verwendet werden kann. Man hat die Möglichkeit über den Kontext eigene VMs in das System einzuspeisen. Somit kann man benutzerdefinierte Scanner und Konfigurationen verwenden. Die Scanner können

sowohl bereits im Image der VMs enthalten sein oder zur Laufzeit mithilfe von Skripten nachinstalliert oder aktualisiert werden (z.B. aktualisieren der Schwachstellendatenbank).

AB-3 - Auswahl von Vulnerables

Ebenso besitzt man die Möglichkeit im User-Interface zwischen verschiedenen Vulenerables auszuwählen. Man besitzt auch hier die Möglichkeit eigene virtuelle Mashcinen über den Kontext in das System einzubringen. Dadurch und mithilfe eines Vulnerable-Skriptes ist es möglich, benutzerdefinierte Software, Einstellungen und weiteres vorzunehmen.

AB-4 - Auswahl von Software für die Vulnerables

Die Vulnerables können zur Laufzeit provisioniert und konfiguriert werden. Dazu kann man ein eigenes Vulnerable-Skript einbinden oder im User-Interface aus einer Vorlage auswählen. Die installierte Software kann beispielsweise beabsichtigte Schwachstellen oder ähnliches beinhalten. Die Skripte werden ausgeführt bevor der eigentliche Schwachstellenscan gestartet wird.

AB-5 - Auswahl benutzerdefinierter Scan-Modi

Man besitzt die Möglichkeit aus einer Liste für jeden Scanner einen Scan-Modus auszuwählen. Die Liste ist um eigene Scan-Modi erweiterbar, indem der Anwender diese im Kontext einfügt. Dies ist immer automatisiert möglich, wenn der Schwachstellenscanner eine entsprechende API besitzt (z.B. bei OpenVAS). Ist dies nicht der Fall kann dennoch mithilfe des manuellen Modus der Scan selbst ausgeführt werden, indem per SSH auf die Maschine und z.B. die GUI des Scanners zugegriffen werden kann. Weitere Scan-Modi (z.B. aktives verwenden von Exploits) können auf die selbe Weise hinzugefügt werden.

AB-6 - Überwachung der Laufzeitumgebung

Die Laufzeitumgebung kann auf verschiedene Weisen überwacht werden. So ist es möglich durch Einbinden des tripwire Plug-Ins das Vulnerable auf Integrität zu überwachen, um Nebenwirkungen der Scans zu beobachten. Man kann mithilfe von der Auswahl des Wireshark-Plug-Ins den Netzwerkverkehr überwachen aber auch die Auslastung des Zielsystems während dem Scan-Prozess beobachten. Während des gesamten Prozesses werden Statusinformationen auf der Konsole ausgegeben. Diese umfassen den aktuellen Stand des Scan-Prozesses, eventuelle Fehlermeldungen, aber auch Daten zur Dauer des Scan-Prozesses. Die genannten Plug-Ins ermöglichen es somit Veränderungen am Zielsystem zu erkennen und dokumentieren diese umfassend.

AB-7 - Erstellen von Reports

Die Ergebnisse des Scans werden persistent auf dem Dateisystem abgespeichert. Es wird ebenfalls eine HTML Ergebnisübersicht erstellt, welche direkte Links zu den Ergebnissen enthält. Die Konfigurationsdetails der VMs werden gespeichert (z.B. Vagrantfile). Fehler während des gesamten Ablaufs können auf der Konsole beobachtet werden und sind bis zum Schließen des Fensters vorhanden.

AB-8 - Wiederholbarkeit und Persistenz

Es ist möglich einen Scan erneut auszuführen, indem man diesen über das User-Interface auswählt und startet. Alle relevanten Informationen eines Scan-Prozesses (Ergebnisse, Konfigurationsdateien, VMs) werden persistent gespeichert. Somit ist es ebenfalls möglich, Ergebnisse zu späterem Zeitpunkt nachzuvollziehen und zu wiederholen.

AB-9 - Modularität und Erweiterbarkeit

Die Modularität ist auf Software-Ebene gegeben. Somit ist es möglich bei Bedarf einzelne Komponenten (z.B. User-Interface oder VM-Managementumgebung) auszutauschen, ohne dass der Rest der Umgebung beeinflusst wird. Eine Erweiterbarkeit ist dank der Modularität ebenfalls möglich. Man ist in der Lage eigene Plug-Ins über Skripte einzubinden. Die Ergebnisse liegen persistent und als Originale vor, weshalb diese einfach weiterverarbeitet werden können, der Kontext speichert die Dateien im Datei-System und die Auswahlmöglichkeiten an Scan-Skripten oder VMs kann benutzerdefiniert erweitert werden. Die Provisionierungs- und Konfigurations-Skripte können ebenfalls einfach erweitert werden.

AB-10 - Automatisierung

Nach der initialen Konfiguration durch den Nutzer läuft die Test-Umgebung vollständig automatisiert ab, ohne dass ein Eingreifen notwendig ist. Dies kann einzig durch die Wahl des manuellen Modus umgangen werden, welcher es Ermöglicht die Scans selbst durchzuführen. Dabei werden die Maschinen identisch konfiguriert, einzig das Scan-Skript wird nicht ausgeführt sondern auf dem Scanner abgelegt.

AB-11 - Status-Informationen

Es werden laufend Status-Informationen über den aktuellen Prozess an den Nutzer über die Konsole zurückgegeben. Somit wird dieser sofort über Fehler informiert und er kann den Prozess verfolgen, um z.B. eine zeitliche Abschätzung über die weitere Dauer des Prozesses vornehmen zu können. Die Informationen werden teils farblich unterschieden und Aktionen der VM-Umgebung, der Maschinen selbst, und des Schedulers werden sichtbar voneinander abgegrenzt. Der Nutzer sieht seine Auswahl vor dem Start des Scans, kann diese somit bei Bedarf über das Beenden und Neu-starten des Programms verändern.

AB-12 - Portabilität

Sofern die Voraussetzungen gegeben sind ist es möglich die Test-Umgebung auf verschiedenen Plattformen zu verwenden. Die notwendigen Programme (PowerShell, Vagrant, VirtualBox) sind alle sowohl für Windows als auch für Linux verwendbar. Der Aufwand die Testumgebung auf Linux auszuführen ist höher als er das auf Windows wäre, da PowerShell nicht bereits vorinstalliert ist. Dennoch ist es ohne Probleme möglich die Test-Umgebung auf Linux zu verwenden. Der Installationsprozess und die Bedienung der Software unterscheiden sich ebenfalls nicht.

Einschränkungen

Die meisten der Anforderungen wurden durch den Prototypen vollständig abgedeckt. Anforderung A-7 bis A-9 sind jedoch nur als ausreichend beschrieben. Diese Entscheidung soll in folgendem Abschnitt begründet werden.
Ein guter Report stellt alle gefundenen Daten übersichtlich dar und unterstützt den Anwender bei der Auswertung der Ergebnisse. Die Übersichts-Datei mit den Ergebnissen kann als rudimentär bezeichnet werden, da sie teils nur auf Ergebnisse verweist und es dem Nutzer überlässt weitere Auswertungen vorzunehmen. Diese Freiheit ist zwar gewollt, dennoch wäre es möglich die Ergebnisse weiter aufzubereiten, ohne den Anwender in seiner Ergebnisfindung zu beeinflussen. Dementsprechend wird der Erfüllungsgrad dieser Anforderung nur als ausreichend festgesetzt.
Des weiteren ist es aktuell nicht möglich alte Scans in abgewandelter Form erneut durchzuführen dazu müsste der Scan-Prozess neu konfiguriert und durchgeführt werden. Diese Einschränkung an Komfort hat zur Folge, dass diese Anforderung nur als ausreichend angesehen wird.
Die Modularität der Test-Umgebung nicht vollumfänglich gegeben da einige Module innerhalb des Codes voneinander abhängen. Ebenfalls ist die Erweiterbarkeit ausbaubar. So ist es noch mit einem manuellen Aufwand verbunden Plug-Ins in das System einzubringen. Somit wird diese Anforderung zwar als erfüllt jedoch nur als ausreichend bewertet.

Tabelle 6.1 zeigt schließlich alle Anforderungen und ihren Erfüllungsgrad.

Nr.	Name der Anforderung	Erfüllungsgrad
A-1	Analyse von Testmaschinen mithilfe von Schwachstellenscannern	vollständig
A-2	Auswahl von Schwachstellenscannern	vollständig
A-3	Auswahl von Vulnerables	vollständig
A-4	Auswahl von Software für die Vulnerables	vollständig
A-5	Auswahl benutzerdefinierter Scan-Modi	vollständig
A-6	Überwachung der Laufzeitumgebung	vollständig
A-7	Erstellen von Reports	ausreichend
A-8	Wiederholbarkeit und Persistenz	ausreichend
A-9	Modularität und Erweiterbarkeit	ausreichend
A-10	Automatisierung	vollständig
A-11	Status-Informationen	vollständig
A-12	Portabilität	vollständig

Tabelle 6.1: Übersichtstabelle über alle Anforderungen und deren Erfüllungsgrad

6.2 Exemplarische Use-Cases

Die Test-Umgebung soll anhand von zwei Use-Cases auf Funktionalität bewertet werden. Diese Beispiele entsprechen Ausprägungen der in Kapitel 2 definierten Use-Cases (siehe: 2.2).

- Szenario 1: Eine Institution möchte einen Schwachstellenscanner erwerben. Sie besitzt mehrere Server, welche in regelmäßigen Abständen auf Schwachstellen überprüft werden sollen. Zur Auswahl stehen zwei Schwachstellenscanner. Aus einem vorherigen umfangreichen Penetrationstest liegt bereits eine Liste mit Schwachstellen vor. Der Schwachstellenscanner der die besten Ergebnisse bezüglich der Referenz liefert wird erworben. Um sicher zu gehen, dass die Scanner auf verschiedenen Typen von Servern ähnliche Ergebnisse liefern sollen die Tests mehrfach mit leicht abgewandelten Konfigurationen und unterschiedlicher installierter Software durchgeführt werden.

- Szenario 2: Der Hersteller eines Schwachstellenscanners wirbt damit, dass er verschiedene Scan-Modi anbietet. Einer der Modi soll einen nicht-intrusiven und somit nebenwirkungsfreien Scan ermöglichen. Da der Nutzer den Scanner im Live-Betrieb verwenden will, ist es essenziell, dass dieser das Netz nicht unnötig belastet, keine unerwünschten Nebenwirkungen zeigt oder Rückstände auf dem System hinterlässt. Deshalb sollen vor Beginn der Verwendung im Produktivsystem die tatsächlich gesendeten Befehle, sowie Nebenwirkungen auf einem Referenzsystem überwacht werden, damit die Angaben des Herstellers bestätigt werden können. Wird eine neue Version eines Schwachstellenscanners veröffentlicht, soll dieser Scan natürlich jederzeit ohne erneutes manuelles Aufsetzen und konfigurieren der Maschinen wiederholt werden können.

6.2.1 Szenario 1 - Vergleich von Schwachstellenscannern

Szenario
Eine Institution besitzt einen Live Server, welcher laufend auf Schwachstellen überprüft werden soll. Er betreibt einen Apache Webserver und läuft auf einem Ubuntu 14. Damit die Live-Umgebung angenähert wird, soll ebenfalls ein Ubuntu-System mit Webserver und der standardmäßigen Software für solche Systeme getestet werden. Als Scanner stehen zwei Schwachstellenscanner zur Verfügung. Zwischen diesen soll eine Entscheidung getroffen werden. Zum einen steht OpenVAS als umfangreicher Scanner zur Verfügung, zum Anderen ein Nikto Webserver-Scanner. Nachdem das Institut weitere Server besitzt für welche der Scanner ebenfalls gute Ergebnisse gewährleisten soll, werden die Scans ebenfalls auf Servern mit leicht abgewandelter Konfiguration und Software durchgeführt. Damit der Zeitaufwand gering gehalten wird ist eine Automatisierung der diversen Scans essenziell.

Ablauf
Zunächst wird das Image des OpenVAS Scanners über den Kontext hinzugefügt. Außerdem werde benutzerdefinierte Scan-Skripte für den OpenVAS Scanner und den Nikto Scanner in dem Kontext abgelegt. Diese stellen zum Einen den OpenVAS Basic Scan dar und zum anderen einen Nikto Basic Scan. Es wird keine weitere Optimierung der Scanner vorgenommen, um deren Funktionsumfang bei den Standard Tests zu überprüfen. Anzumerken ist, dass der Nikto Scanner angibt einzig auf Web Server spezialisiert zu sein und der OpenVAS Scanner bei weiterer Konfiguration mehr Scan-Potential bietet.
Das Hauptprogramm wird gestartet und zunächst die beiden Scanner sowie ihre Scan Skripte gewählt. Es soll keine weitere Software außer den Scannern installiert werden. Der OpenVAS Scanner ist bereits vorinstalliert und der Nikto Scanner wird zur Laufzeit nachinstalliert.
Schließlich wird das entsprechende Referenz-Vulnerable gewählt, welches den echten Webserver im Test repräsentiert. Da der Live-Server Pingbar ist und diese Funktion bei den Firewall

Einstellungen des Vulnerables deaktiviert ist wird ein Skript gewählt, welches die Einstellungen so verändert, dass Pings zugelassen werden.
Weiterhin wird die Referenz-Maschine erneut mit leicht unterschiedlichen Konfigurationen gewählt, um die Funktionalität der Scanner bezüglich weiterer Server des Instituts zu testen. Dazu kann dank der Test-Umgebung ohne weiteren Aufwand das Vulnerable erneut gewählt werden, wobei nun unterschiedliche Konfigurations- und Provisionierungs-Skripte gewählt werden.
Bevor der Scan gestartet wird werden zwei Plug-Ins gewählt, welche bereits in der Test-Umgebung enthalten sind. Zum einen soll mittels Wireshark jeder Netzwerkverkehr zwischen Scanner und Vulnerable aufgezeichnet werden. Zum anderen werden mithilfe von nmon die Performance-Daten des Vulnerables überwacht (z.B. CPU-Auslastung, Netzwerkauslastung). Schließlich wird der Scan-Prozess gestartet. Es wird auf einen manuellen Modus verzichtet und der automatisierte Modus wird gestartet.
Die Test-Umgebung arbeitet ohne weiteres Eingreifen vollständig automatisiert ab, wie im Szenario gefordert.
Anschließend wird auf den Abschluss des Prozesses gewartet. Der aktuelle Status kann über die Konsole verfolgt werden. Nach 5 Minuten ist die erste Scanner-Vulnerable-Paarung abgeschlossen (Nikto + Vulnerable). Die Ergebnisse werden im entsprechenden Ordner abgelegt. Der zweite Scan (OpenVAS + Vulnerable) dauert mit 20 Minuten deutlich länger. Die Ergebnisse werden ebenfalls in dem entsprechenden Unterordner abgelegt.
Nach Abschluss der Scans werden die virtuellen Maschinen wieder zerstört. Der Scan kann jedoch bei Bedarf über das User-Interface erneut durchgeführt werden.
Bei beiden Scan-Ordnern findet man persistent sowohl die Scan-Konfigurationen des Scans, die Scan-Ergebnisse, sowie einmal das nmon-Ergebnis sowie die Wireshark Aufzeichungen. Diese Ergebnisse können über eine .html Datei eingesehen und ausgewertet werden.
Für die Weiteren Konfigurationen werden ebenfalls Ordner erzeugt, welche äquivalent zum Haupt-Scan die alle Ergebnisse beinhalten.

Ergebnis

Die Funktionalität des OpenVAS Scanners zum Test eines reinen Webservers ist gegebenenfalls überhöht. So werden neben den gefundenen Web-Schwachstellen ebenfalls Informationen über andere Schwachstellen ausgegeben. Der Nikto Scanner bietet einen geringeren Funktionsumfang, betrachtet jedoch einzig die Schwächen des Web-Servers selbst. Der Overhead ist dementsprechend verringert. Im Hinblick darauf, dass im Live Betrieb weitere Server als der Hauptserver überprüft werden sollen, kann dieser erweiterte Funktionsumfang des ersten Scanners gegenüber dem Overhead als Vorteil erscheinen. Der OpenVAS Scan hat eine deutlich längere Zeit in Anspruch genommen als der Nikto Scan. Die nmon Ergebnisse zeigen außerdem deutlich, dass die CPU sowie die Netzwerk-Auslastung der Vulnerables während des gesamten Scan-Prozesses deutlich höher war. Die Auswirkungen des Nikto-Scanners zeigen eine kaum erkennbare Auslastung. Eine subjektive Entscheidung für oder gegen einen der Scanner kann nach einer detaillierten Überprüfung der Ergebnisse selbst, gegebenenfalls unter Einbezug von Experten getroffen werden.
Zusammenfassed kann gesagt werden, dass man mithilfe der Test-Umgebung erfolgreich die Fähigkeiten verschiedener Scanner aufzeichnen kann um diese später zu evaluieren und eine fundierte Entscheidung bezüglich der Wahl eines Schwachstellenscanners zu treffen. Im Anhang (siehe: 7.2) kann beispielhaft eine Übersicht der Scan-Ergebnisse gefunden werden.

6.2.2 Szenario 2 - Überprüfen von Nebenwirkungen auf dem Zielsystem

Szenario

Ein Schwachstellenscanner soll in Zukunft regelmäßig einen Live-Server auf Schwachstellen überprüfen. Da der Betreiber aufgrund eines strengen SLAs einen Absturz des Servers und somit einen Verlust der Verfügbarkeit nicht riskieren darf möchte er vor dem Einsatz eines Schwachstellenscanners zunächst überprüfen, ob dieser unerwünschte Nebenwirkungen auf den Zielserver hat. Der zunächst zu testende Scanner ist ein OpenVAS Scanner. Dieser wirbt damit, verschiedene Scan-Modi anzubieten, welche entweder rein passiv ein System überprüfen oder aktiv ebenfalls versuchen, gefundene Schwachstellen mittels Exploits auszunutzen (siehe: 2.1). Zwei dieser Modi sollen zunächst überprüft werden. Der erste Modus ist ein OpenVAS "Full and Fast Scan" der zweite ein "Full and very deep Scan". Beide Modi sollen anhand derselben Referenzmaschine getestet werden. Als Vulnerable wird ein Ubuntu 12.04 (Precise Pangolin) von Hashicorp gewählt. Da dieser Server älter ist kann erwartet werden, dass bereits einige Schwachstellen gefunden werden, welche von dem Scanner direkt ausgenutzt werden können. Somit wird der Fall simuliert, dass der Scanner tatsächlich Exploits aktiv anwenden kann und das Verhalten des Servers überprüft werden kann.
Sobald eine neue Version des Schwachstellenscanners veröffentlicht wird, soll dieser Scan natürlich erneut durchgeführt werden können, ohne eine neue Test-Umgebung erstellen zu müssen oder manuell aufwändig die Maschinen konfigurieren zu müssen. Da in diesem Szenario Nebenwirkungen auf dem System betrachtet werden sollen, ist es notwendig ein neues Plug-In einzubinden, welches es ermöglicht Unterschiede am Dateisystem zu überwachen und diese an den Anwender auszugeben.

Ablauf

Zunächst wird das Image des OpenVAS Scanners in den Kontext hinzugefügt. Die beiden entsprechenden Scan-Skripte werden ebenfalls abgelegt.
Das Vulnerable wird ebenfalls in den Kontext hinzugefügt. Ein weiteres Skript für das Vulnerable wird nicht benötigt.
Nachdem dieses Szenario die Anforderung besitzt, dass das Vulnerable auf Veränderung überwacht werden soll, wird ein Tripwire-Plug-In eingebunden. Die ausführliche Anleitung dafür findet sich im Anhang (siehe: 7.2) und bedeutet dank der Modularität der Test-Umgebung keinen großen Aufwand.
Nun wird das Hauptskript gestartet. Es wird der Scanner zweimal mit dem jeweiligen Scan-Skript hinzugefügt und das Vulnerable ohne Skript ausgewählt.
Als Scan-Konfiguration werden das Wireshark- und das nmon-Plug-In sowie das hinzugefügte Tripwire-Plug-In gewählt und der Scan schließlich gestartet.
Der Prozess kann über die Konsole verfolgt werden und die Ergebnisse werden in einem entsprechenden Unterordner abgelegt.

Wird zu gegebener Zeit eine neue Verison des Schwachstellenscanners veröffentlicht, ist mithilfe der Test-Umgebung ohne weiteres Möglich den bereits durchgeführten Scan ohne erneute Konfiguration erneut durchzuführen. Sollen Veränderungen an dem Szenario vorgenommen werden, so ist dies ebenfalls möglich. Dazu kann die persistente Scan-Konfiguration vergangener Szenarien eingesehen werden und die neuen Tests leicht abgeändert durchzuführen.

Ergebnis
Wie erwartet hat der OpenVAS "Full and very deep Scan" mehr Zeit in Anspruch genommen (34 Minuten) als der "Full and Fast Scan" (24 Minuten). Mittels der Wireshark Ergebnisse können ebenfalls die Unterschiede in den durchgeführten individuellen Befehle eingesehen werden, welche sich auf 2000 zusätzliche an das Vulnerable belaufen. Die Netzauslastung war bei dem erstgenannten Scan mit 87% zu 70% leicht höher. Weitere Ergebnisse können über die nmon Ergebnisdatei eingesehen werden. Die Unterschiede am Dateisystem können über das tripwire Ergebnis betrachtet werden sind jedoch unauffällig und unterscheiden sich nicht voneinander. Dies kann auf eine fehlerhaft konfigurierte Baseline zurückgeführt werden, spricht jedoch eher dafür, dass keine Veränderungen am Dateisystem vorgenommen wurden. Ebenfalls ist der Server nicht abgestürzt, was für den Scanner spricht. Unerwünschte Nebeneffekte konnten somit nicht beobachtet werden. Die Scan-Ergebnisse selbst sind ebenfalls aussagekräftig. Ob somit die zusätzliche Zeitdauer in Kauf genommen wird und die zusätzlichen gesendeten Befehle als unbedenklich angesehen werden, sowie eine erhöhte Auslastung in Kauf genommen wird muss subjektiv entschieden werden.
Soll der Scan zu beliebiger Zeit wiederholt werden - mit denselben oder unterschiedlichen Konfigurationen - ist dies ohne größeren Aufwand mittels der Test-Umgebung möglich.
Das Hinzufügen von weiteren Szenario-Spezifischen Plug-Ins ist mit geringem Aufwand umsetzbar.
Zusammenfassend kann gesagt werden, dass es möglich ist mithilfe der Testumgebung die Unterschiede zwischen verschiedenen Scan Methoden erfolgreich zu erfassen, deren Metadaten aufzuzeichnen und Veränderungen auf dem Vulnerable zu bemerken.

7 Fazit und Ausblick

7.1 Fazit

Diese Arbeit beschreibt die Konzeption und Umsetzung einer Test-Umgebung für Schwachstellenscanner. Im Unterschied zu früheren Arbeiten bietet die Umgebung eine Plattform, durch welche es möglich ist, automatisiert mithilfe eines oder mehrerer Schwachstellenscanner beliebig viele Zielmaschinen auf Schwachstellen zu untersuchen. Besonders dabei ist ein Fokus auf Wiederholbarkeit, benutzerdefinierter Konfiguration des Test-Szenarios sowie einer Erweiterbarkeit mithilfe von Plug-Ins.
Nachdem alle Aspekte des Scan-Ablaufs individuell angepasst werden können, ist es möglich diverse Use-Cases abzubilden. Wie eine qualitative Analyse der Test-Umgebung zeigt, kann diese sowohl im wissenschaftlichen wie auch kommerziellen Umfeld produktiv eingesetzt werden. Die eingangs gestellten Forschungsfragen (siehe: Abschnitt 1) können positiv beantwortet werden.
So bestimmt die Arbeit zunächst zwölf Anforderungen, welche an eine Test-Umgebung gestellt werden müssen. Sie umfassen Qualitätskriterien guter Test-Umgebungen wie Wiederholbarkeit und Erweiterbarkeit aber auch Anforderungen, welche spezifisch auf Schwachstellenscanner zugeschnitten sind. Aus den Anforderungen wird im Folgenden ein Konzept erarbeitet. Das Konzept unterscheidet auf Ebene der Programmlogik zwischen einem Kontext und einer Laufzeitumgebung. Diese werden praktisch durch vier Module - Kontext, User-Interface, Scheduler und VM-Management-Umgebung - umgesetzt. Mithilfe dieses Konzepts war es möglich einen Prototypen basierend auf PowerShell und Vagrant zu entwickeln, welcher es ermöglicht, erfolgreich verschiedene Use-Cases zu bedienen. Es wurden einige Plug-Ins eingebunden, welche sowohl die Funktionalität des modularen Systems verdeutlichen, als auch den Funktionsumfang der Test-Umgebung deutlich erweitern (siehe: Abschnitt 5.2). Alle Anforderungen wurden mindestens ausreichend und meist vollständig mittels des Prototypen umgesetzt.
Der Prototyp lässt sich auf meinem GitHub Profil einsehen (`https://github.com/Idefixus/Vagrant/releases/latest`). Eine Installationsanleitung findet sich im Anhang (siehe: 7.2).

7.2 Ausblick

Folgender Abschnitt gibt Anregungen zu weiteren Forschungsfragen, welche in zukünftigen Arbeiten behandelt werden können. Ebenfalls werden einige Forschungsgebiete genannt, in denen die Test-Umgebung unterstützend eingesetzt werden kann.
Da es sich hier um einen Prototypen handelt, ist die Test-Umgebung zwar in der Praxis einsetzbar und die Machbarkeit ist gezeigt. Dennoch sind einige Funktionalitäten, welche entweder die Bedienung vereinfachen, die Qualität der Ergebnisse oder die Präsentation der Ergebnisse weiter verbessern können denkbar. Diese Themen sind Forschungsgegenstand späterer Arbeiten. Dennoch möchte ich einige Beispiele geben, inwiefern die Test-Umgebung

um weitere Features erweitert werden könnte und welchen Mehrwert diese bieten würden.

- Die Integration von Plug-Ins kann weiter vereinfacht werden, indem ein Standard vorgegeben wird, welcher von dem Programm interpretiert werden kann und somit den Aufwand bei der Integration vermindert. Dafür ist es notwendig eine eigene Schnittstelle für jeden Schritt des Scan-Ablaufs festzulegen, da die Plug-Ins vielfältig einsetzbar sein können und jedes der Module ein potentielles Ziel der Erweiterungen sein kann.
- Die Präsentation der Ergebnisse kann um weitere Funktionalitäten erweitert werden, wie eine Vorauswertung der Ergebnisse und einer Filterung dieser. Ein Problem dabei ist die Erkennung von Duplikaten und False-Positives und Negatives, welche automatisiert nur mit Fehlern umzusetzen sind. Damit diese verringert werden können wäre es sinnvoll äquivalent zu Exploit-Signaturen ebenfalls Schwachstellen-Signaturen zu besitzen, welche sich zuverlässig den Exploits zuweisen lassen.
- Die Test-Umgebung kann durch einen vollwertigen Monitoring-Server erweitert werden, welcher eine Scanner-Vulnerable-Paarung um einen dritten Server erweitern würde. Somit kann der Netzwerkverkehr aufschlussreicher überwacht werden (z.B. mit Icinga). Dies bietet ebenfalls die Möglichkeit die Ergebnisse visuell anschaulicher darzustellen.
- Ein Scanner-Übergreifendes Mapping verschiedener Scan-Methoden kann entwickelt werden, welches dafür verantwortlich ist, dass Scans untereinander besser vergleichbar gemacht werden können. Aktuell muss das Wissen um entsprechende äquivalente Scans bei dem Nutzer vorhanden sein, damit diese aussagekräftig sind. Ein Problem dabei ist, dass Schwachstellenscanner normalerweise bestimmte Scan-Szenarien anbieten, diese jedoch in wenigen Fällen dieselben Tests wie bei der Konkurrenz durchführen. Eine Analyse verschiedener Scanner-Methoden und der tatsächlich ausgeführten Tests ist notwendig, damit ein solches Mapping erstellt werden kann. Schließlich kann eine Schnittstelle für die Test-Umgebung entwickelt werden, welche ein solches Mapping integriert.
- Die Integrität von Zielsystemen kann überprüft werden, wenn der sich der Anwender bewusst ist, welche Bereiche der Maschine er für schützenswert betrachtet. Dementsprechend kann eine Baseline festgelegt werden, anhand welcher unerwünschte Nebenwirkungen erkannt werden können. Diese Methode erfordert viel Fachwissen und hoher Eigenaufwand bei der Konfiguration ist notwendig. Eine Methode welche die direkten Auswirkungen von Exploits auf dem Zielsystem überwacht wäre eine sinnvolle Erweiterung der Test-Umgebung und eine zukünftige Forschungsfrage.
- Die Scans selbst sowie die Verwendung von Exploits kann verbessert werden. Dies ist jedoch eher Aufgabe des Anwenders als der Test-Umgebung. Dennoch kann dem Anwender eine größere Vorauswahl an Methoden zur Verfügung gestellt werden, welche die Arbeit vereinfachen (z.B. [Jos19, ERRW18]). Weitere optionale Plug-Ins sind im Anhang aufgelistet (siehe: 7.2)

Abbildungsverzeichnis

Literaturverzeichnis

[717] 7, Rapid: *Metasploitable3*. 2017

[Acu14] Acunetix: *Negative Impacts of Automated Vulnerability Scanners and How to Prevent them*. https://www.acunetix.com/blog/articles/negative-impacts-automated-vulnerability-scanners-prevent/, 2014. – Accessed: 2020-05-19

[AV10] Antunes, N. ; Vieira, M.: Benchmarking Vulnerability Detection Tools for Web Services. In: *2010 IEEE International Conference on Web Services*, 2010, S. 203–210

[AXE13] AXELOS: *ITIL Glossar und Abkürzungen*. 2013

[BBGM10] Bau, J. ; Bursztein, E. ; Gupta, D. ; Mitchell, J.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In: *2010 IEEE Symposium on Security and Privacy*, 2010, S. 332–345

[Com20] Commerce, U.S. D.: *National Vulnerability Database*. https://nvd.nist.gov/, 2020. – Accessed: 2020-06-09

[DB11] Dr. Brandes, Christian: *Konzeption von produktionsnahen Testumgebungen*. 2011

[Deu15] Deutschland, Bundesrepublik: *Gesetz zur Erhöhung der Sicherheit informationstechnischer Systeme (IT-Sicherheitsgesetz)*. https://www.bgbl.de/xaver/bgbl/start.xav?startbk=Bundesanzeiger_BGBl&jumpTo=bgbl115s1324.pdf#__bgbl__%2F%2F*%5B%40attr_id%3D%27bgbl115s1324.pdf%27%5D__1589895757664, 2015. – Accessed: 2020-05-19

[Deu20] Deutschland, Bundesrepublik: *Gesetz zur Erhöhung der Sicherheit informationstechnischer Systeme (IT-Sicherheitsgesetz) Version 2*. https://www.br.de/nachrichten/netzwelt/it-sicherheitsgesetz-2-0-bsi-soll-aufgeruestet-werden,Rys3BlX, 2020. – Accessed: 2020-05-19

[Dmi19] Dmitrity, Elin: *Evaluierung der Dienst- und Versionserkennung von Linux-Diensten mit Open-Source und kommerziellen Netzwerkscanner*, LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN, Bachelorarbeit, 2019

[Dre13] Dreo, Prof. Dr. G.: *Grundlagen der IT-Forensik*, LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN, Bericht, 2013

[ER96] Eickelmann, N. S. ; Richardson, D. J.: An evaluation of software test environment architectures. In: *Proceedings of IEEE 18th International Conference on Software Engineering*, 1996, S. 353–364

[ERRW18] Esposito, Damiano ; Rennhard, Marc ; Ruf, Lukas ; Wagner, Arno: Exploiting the potential of web application vulnerability scanning. In: *ICIMP 2018 - The Thirteenth International Conference on Internet Monitoring and Protection*, 2018, S. 1–29

[ERV+08] Elizabeth, Fong ; Romain, Gaucher ; Vadim, Okun ; Paul, Black ; Eric, Dalci: Building a Test Suite for Web Application Scanners. In: *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, 2008, S. 478 – 478

[EY03] Ernst & Young, Bundesamt für Sicherheit in der Informationstechnik und BDO u.: *Studie - Durchführungskonzept für Penetrationstests*. 1.0.1. Godesberger Allee 185-189, 53175 Bonn, 2003. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Penetrationstest/penetrationstest.pdf?__blob=publicationFile&v=3

[FVM07] Fonseca, J. ; Vieira, M. ; Madeira, H.: Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. In: *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, 2007, S. 365–372

[Goa20] Goat, Web: *Web Goat*. https://webgoat.github.io/WebGoat/, 2020. – Accessed: 2020-05-19

[Inf16a] Informationstechnik, Bundesamt für Sicherheit in d.: *Ein Praxis-Leitfaden für IS-Penetrationstests*. 1.2. Godesberger Allee 185-189, 53175 Bonn, 2016. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Sicherheitsberatung/Pentest_Webcheck/Leitfaden_Penetrationstest.pdf?__blob=publicationFile&v=10

[Inf16b] Informationstechnik, Bundesamt für Sicherheit in d.: *Ein Praxis-Leitfaden für IS-Webchecks*. 1. Godesberger Allee 185-189, 53175 Bonn, 2016. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Sicherheitsberatung/Pentest_Webcheck/Leitfaden_Webcheck.pdf?__blob=publicationFile&v=4

[Inf19] Informationstechnik, Bundesamt für Sicherheit in d.: *IT-Grundschutz-Kompendium*. Godesberger Allee 185-189, 53175 Bonn : Reguvis, 2019 https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2019.pdf?__blob=publicationFile&v=5. – ISBN 978–3–8462–0906–6

[ISO17] ISO/IEC (Hrsg.): *DIN EN ISO/IEC 27002:2017-06*. 2017. Saatwinkler Damm 42/43, 13627 Berlin: ISO/IEC, 6 2017

[Jos19] Josef, Sedlmeir S.: *Automated Success Verification of Exploits for Penetration Testing with Metasploit*, LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN, Bachelorarbeit, 2019

[JPC17a] JPCERT: *Detecting Lateral Movement through Tracking Event Logs*. https://www.jpcert.or.jp/english/pub/sr/20170612ac-ir_research_en.pdf, 2017. – Accessed: 2020-05-19

[JPC17b] JPCERT: *Detecting Lateral Movement through Tracking Event Logs (Version 2).* `https://www.jpcert.or.jp/english/pub/sr/DetectingLateralMovementThroughTrackingEventLogs_version2.pdf`, 2017. – Accessed: 2020-05-19

[JS16] Joshi, Chanchala ; Singh, Umesh: Security Testing and Assessment of Vulnerability Scanners in Quest of Current Information Security Landscape. In: *International Journal of Computer Applications* 145 (2016), 07, S. 1–7. `http://dx.doi.org/10.5120/ijca2016910563`. – DOI 10.5120/ijca2016910563

[Khr19] Khrystyna, Struk: *Evaluierung der Dienst- und Versionserkennung von Windows-Diensten mit Open-Source und kommerziellen Netzwerkscannern*, LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN, Bachelorarbeit, 2019

[KS94] Kim, Gene H. ; Spafford, Eugene H.: The Design and Implementation of Tripwire: A File System Integrity Checker. In: *Proceedings of the 2nd ACM Conference on Computer and Communications Security.* New York, NY, USA : Association for Computing Machinery, 1994 (CCS ?94). – ISBN 0897917324, 18?29

[Lau19] Laura, Gamisch: *Kategorisierung von Exploits zur Durchfuhrung von nicht-intrusiven Penetrationtests*, LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN, Bachelorarbeit, 2019

[LS10] Loh, P. K. K. ; Subramanian, D.: Fuzzy Classification Metrics for Scanner Assessment and Vulnerability Reporting. In: *IEEE Transactions on Information Forensics and Security* 5 (2010), Nr. 4, S. 613–624

[Mic17] Michael, Kauschinger: *Penetrationstesting als Service für einen IT-Provider am Beispiel des Web-Hosting- und IaaS-Services des Leibniz-Rechenzentrum (LRZ)*, LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN, Masterarbeit, 2017

[Nes20] Nessus: *Nessus Vulnerability Scanner.* `https://de.tenable.com/products/nessus`, 2020. – Accessed: 2020-05-19

[OWA15] OWASP: *OWASP Broken Web Applications Project.* `https://github.com/chuckfw/owaspbwa/`, 2015. – Accessed: 2020-05-19

[OWA17] OWASP: *OWASP Top 10 - 2017.* `https://github.com/OWASP/www-project-top-ten`, 2017. – Accessed: 2020-05-19

[Pag16] Paganini, Pierluigi: *Lotus Blossom Chinese cyberspies leverage on fake Conference Invites in the last campaign.* `https://securityaffairs.co/wordpress/52911/cyber-warfare-2/lotus-blossom-campaign.html`. Version: 2016

[Sch20] Schreuders, Cliffe et a.: *Security Scenario Generator (SecGen).* `https://github.com/cliffe/SecGen/`, 2020. – Accessed: 2020-06-08

[She10] Shelly, David: *Using a Web Server Test Bed to Analyze the Limitations of Web Application Vulnerability Scanners*, Virginia Polytechnic Institute and State University, Diplomarbeit, 2010. – 1–100 S.

[SKBG18] Schagen, Nathan ; Koning, Koen ; Bos, Herbert ; Giuffrida, Cristiano: Towards Automated Vulnerability Scanning of Network Servers. In: *Publication: EuroSec'18: Proceedings of the 11th European Workshop on Systems Security*, 2018, S. 1–6

[SS10] Suto, Larry ; San, Consultant: Analyzing the Accuracy and Time Costs of Web Application Security Scanners. (2010), 03

[Ste18] Steffens, T.: *Auf der Spur der Hacker: Wie man die Täter hinter der Computer-Spionage enttarnt*. Springer Berlin Heidelberg, 2018 `https://books.google.de/books?id=SftKDwAAQBAJ`. – ISBN 9783662559543

[TTSS14a] Tung, Y. ; Tseng, S. ; Shih, J. ; Shan, H.: W-VST: A Testbed for Evaluating Web Vulnerability Scanner. In: *2014 14th International Conference on Quality Software*, 2014, S. 228–233

[TTSS14b] Tung, Y. ; Tseng, S. ; Shih, J. ; Shan, H.: W-VST: A Testbed for Evaluating Web Vulnerability Scanner. In: *2014 14th International Conference on Quality Software*, 2014, S. 228–233

[Uni18] Union, Europäische: *Datenschutz-Grundverordnung*. `https://www.datenschutz-grundverordnung.eu/`, 2018. – Accessed: 2020-05-19

[VV13] *Kapitel* 9. In: Vivenzio, Alberto ; Vivenzio, Domenico: *Testumgebung*. Wiesbaden : Springer Fachmedien Wiesbaden, 2013. – ISBN 978–3–8348–2142–3, 89–98

[YCLS04] Yao-Wen Huang ; Chung-Hung Tsai ; Lee, D. T. ; Sy-Yen Kuo: Non-detrimental Web application security scanning. In: *15th International Symposium on Software Reliability Engineering*, 2004, S. 219–230

Anhang A

Weitere Plug-Ins

Eine Erweiterung der Test-Umgebung durch weitere Plug-Ins kann die Ergebnisse verbessern oder übersichtlicher gestalten. Jedes Modul kann dabei bei Bedarf um weitere Features erweitert werden. Folgende Plug-Inst stellen einige Vorschläge dar, welche bei der Entwicklung des Prototypen als potentiell Sinnvoll für die Test-Umgebung identifiziert wurden.

- Automatische Erfolgs-Verifikation von Exploits [Jos19]
- Sec-Gen [Sch20] zur Generierung von Vulnerables mit (zufälligen) benutzerdefinierten Schwachstellen zur Laufzeit
- Liste an nach-installierbaren Schwachstellen für bestehende Vulnerables [717, Sch20]
- Proxys zur Verbesserung von Scan-Ergebnissen [ERRW18]
- Benutzerdefinierte Scan Typen für weitere Scanner als interaktives Menü (z.B. Nessus Scan-Generator)
- Teilauswertung und Verschönerung der Scan-Ergebnisse mithilfe von Informationsvisualisierungs-Techniken
- Mapping der Ergebnisse auf die mitgelieferten Schwachstellen
- Verringerung von Duplikaten in Scan Ergebnissen
- Erweiterung des Netzwerks um weitere VMs (z.B. Test von Lateral Movement, dedizierte DB-Server, Domain-Controllers)
- Dedizierter Management-Server für genauere Analysen von Test- und Zielsystem
- Einbinden von Szenarien, welche im Hintergrund auf bestimmte Scan-Skripte abgebildet werden (z.B. Szenario: API-Test, Onlineshop, Blog)

Plug-In Implementierungsbeispiel

Hier wird gezeigt, wie die Implementierung eines Plug-Ins im Detail aussehen würde. Dies wird am Beispiel der Wireshark Erweiterung gezeigt, welche in der Test-Umgebung mitgeliefert wird. Dieser Abschnitt stellt eine Erweiterung des in Abschnitt 5.2 beschriebenen Prozesses zum Einbinden eines Plug-Ins dar.

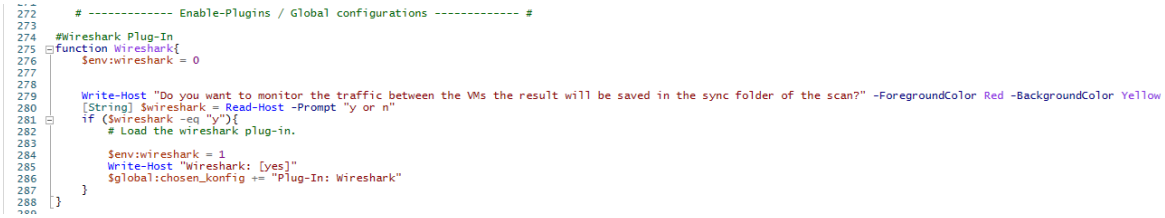

Abbildung 1: Wirshark Codeabschnitt in scheduler.ps1

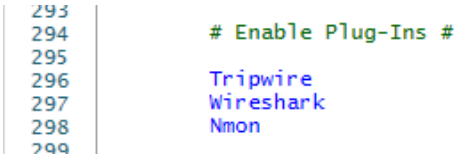

Abbildung 2: Funktionsaufruf in testbed.ps1

Anlegen einer Wireshark Funktion

Es wird eine eigene Funktion für das Plug-in in der Datei scheduler.ps1 unter dem Abschnitt 'Enable-Plugins' angelegt. Diese beinhaltet eine globale PowerShell Umgebungsvariable, welche mit 0 initialisiert wird. Eine Prompt für den Anwender wird erstellt, welches diesen dannach fragt, ob das Plug-In eingebunden werden soll. Wird diese Frage bejaht, wird die Umgebungsvariable auf 1 gesetzt. Zusätzlich sollte die Information in das '$global:chosen_konfig' Array eingefügt werden, welches dafür verantwortlich ist, dass die Information über das aktivierte Plug-In dem Nutzer in der Vorschauauswahl angezeigt wird. (siehe: 1)

Funktionsaufruf

Die eben definierte Funktion muss schließlich ausgeführt werden. Dazu ist es notwendig diese an der entsprechenden Stelle in der testbed.ps1 Datei eingefügt werden (siehe: 2).

Aktualisieren der Vagrantfile Blaupause

Schließlich kann die Funktion des Plug-Ins integriert werden. Dies geschieht in der Blaupause der Vagrantfile. Dazu wird eine Überprüfung der Umgebungsvariable eingefügt, welche des Abschnitt nur unter der Bedingung ausführt, dass die Variable auf 1 gesetzt ist. Schließlich wird die entsprechende Funktion in die Vagrantfile eingefügt. In diesem Fall wird VirtualBox um einige Funktionen erweitert. Weiterhin können an dieser Stelle benutzerdefinierte Skripte auf die VMs übertragen oder Befehle oder Skripte via der Provisionierung ausgeführt werden. Vagrant unterstützt Provisionierung mittels der Shell, Ansible, Chef, Docker, uvm. (siehe: 3).

Individuelle Plug-Ins und Konfigurationen können weitere Arbeit notwendig machen. Diese Code-Teile müssen dokumentiert werden und an den entsprechenden Abschnitten eingefügt werden. Dies sollte aufgrund der Modularität der Software ohne großen Aufwand machbar sein.

```
config.vm.define "scanner" do |scanner|
  if $env:wireshark == 1
    scanner.vm.provider 'virtualbox' do |v|
      # Do a wireshark capture of the traffic - Some machines have the subnet on the first some on the second network interface
      v.customize ['modifyvm', :id, '--nictrace1', 'on']
      v.customize ['modifyvm', :id, '--nictrace2', 'on']
      v.customize ['modifyvm', :id, '--nictracefile1', "$env:SCOPE_DIR_plugin\\sync\\trace1.pcap"]
      v.customize ['modifyvm', :id, '--nictracefile2', "$env:SCOPE_DIR_plugin\\sync\\trace2.pcap"]
    end
  end
```

Abbildung 3: Erweiterung der Vagrantfile Blaupause um die Funktionalität des Plug-Ins

Blaupausen Vagrantfile

Dieser Abschnitt enthält die vollständige Blaupausen Vagrantfile des Prototypen. Sie wird zur Laufzeit durch den Scheduler in eine ausführbare Datei übersetzt. Dies geschieht dadurch, dass die Umgebungsvariablen ($env:*VARIABLENAME*) durch Laufzeitdaten ersetzt werden und die Datei im entsprechenden Unterordner abgelegt wird.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
# encoding: UTF-8

Vagrant.configure("2") do |config|
config.vm.provision "shell", inline: "echo Start the Vagrant
    Process"
# Configuration for the ssh process
config.ssh.username = "vagrant"
config.ssh.password = "vagrant"

config.vm.define "vulnerable" do |vulnerable|
vulnerable.vm.box = "$env:VULNERABLE_BOXNAME"
vulnerable.vm.network "private_network", ip: "$env:IP_VULNERABLE"
vulnerable.vm.synced_folder "sync/", "$env:PATH_VULNERABLE_SYNC",
    create: true
# Main provisioning with shell, maybe also abstract so ansible and
     docker and so are possible.
vulnerable.vm.provision "shell", path: "$env:
    VULNERABLE_PROVISIONING_FILE_NAME"

if $env:monitoring == 1
vulnerable.vm.provision "file", source: "../../twpol.txt",
    destination: "/home/vagrant/"
vulnerable.vm.provision "shell", path: "../../tripwire_init.sh"
# Before destroying the machine get the reports - INFO: The remote
     script has to end with an echo or sth or it will fail.
vulnerable.trigger.before :destroy do |trigger|
```

```
trigger.warn = "Checking tripwire against the baseline for
    differences"
trigger.run_remote = {inline: "tripwire --check > /vagrant/
    tripwire_log.txt; echo 'Tripwire log copied'"}
end
end
if $env:performance == 1
vulnerable.vm.provision "file", source: "../../nmonchart",
    destination: "/home/vagrant/"
vulnerable.vm.provision "shell", path: "../../nmon.sh"
# Before destroying the machine get the nmonchart html file
vulnerable.trigger.before :destroy do |trigger|
trigger.warn = "Creating a html chart of the nmon results"
# Creating the chart and copying the files to the results folder
    for persistence
trigger.run_remote = {inline: "ksh nmonchart performance_result.
    nmon; mv performance_result.nmon /vagrant/; mv
    performance_result.html /vagrant/; echo 'Nmon result created
    and copied'"}
end
end
end
config.vm.define "scanner" do |scanner|
if $env:wireshark == 1
scanner.vm.provider 'virtualbox' do |v|
# Do a wireshark capture of the traffic - Some machines have the
    subnet on the first some on the second network interface
v.customize ['modifyvm', :id, '--nictrace1', 'on']
v.customize ['modifyvm', :id, '--nictrace2', 'on']
v.customize ['modifyvm', :id, '--nictracefile1', "$env:
    SCOPE_DIR_plugin\\sync\\trace1.pcap"]
v.customize ['modifyvm', :id, '--nictracefile2', "$env:
    SCOPE_DIR_plugin\\sync\\trace2.pcap"]
end
end

scanner.vm.box = "$env:SCANNER_BOXNAME"
scanner.vm.network "private_network", ip: "$env:IP_SCANNER"
scanner.vm.synced_folder "sync/", "$env:PATH_SCANNER_SYNC", create
    : true
#Openvas automation TODO: Maybe sync whole folder. Handle
    different provisioning scripts
scanner.vm.provision "file", source: "../../get_openvas_result.sh"
    , destination: "/home/vagrant/"
scanner.vm.provision "file", source: "../../
    openvas_scan_automation_start.sh", destination: "/home/vagrant/
    "
```

```
scanner.vm.provision "file", source: "../../
    openvas_scan_automation_start_deep.sh", destination: "/home/
    vagrant/"
# If there is a manuell mode set then dont trigger the custom scan
     script. But it is copied to the machine.
if $env:manuell == 0
scanner.vm.provision "shell", path: "$env:
    SCANNER_PROVISIONING_FILE_NAME"
else
scanner.vm.provision "file", source: "$env:
    SCANNER_PROVISIONING_FILE_NAME", destination: "/home/vagrant/"
end
end
end
```

Listing 1: Ein Blaupausen Vagrantfile zu Anschauungszwecken

Installationsanleitung

Folgende Schritte müssen durchgeführt werden um die Test-Umgebung erfolgreich zu installieren und auszuführen.

- Installiere Vagrant
- Installiere PowerShell (Unter Linux)
- Installiere VirtualBox
- Klone das Repository (Link: `https://github.com/Idefixus/Vagrant.git`)
- Start der kontext.ps1 Datei zur Konfiguration des Kontexts
- Start der testbed.ps1 Datei zum Konfigurieren und Starten des Scan-Prozesses
- Weitere Informationen sowie Bekannte Bugs sind auf GitHub zu finden.

Erstellen einer Vagrant Box

Der Prototyp arbeitet mit Vagrant als Orchestrierungssoftware. Daher werden Vagrant Boxen benötigt, um mit der Test-Umgebung arbeiten zu können. Einige Boxen kann man über die Seite `https://www.vagrantup.com/` finden. Möchte man jedoch sein eigenes Image einer VM erstellen, so ist es notwendig auf eine Software namens Packer zurückzugreifen. Diese ist wie Vagrant aus dem Entwicklerstudio HashiCorp. Eine genaue Anleitung zum Erstellen von eigenen Maschinen kann unter `https://www.packer.io/` gefunden werden. Im folgenden findet sich eine Datei zum erstellen eines Debian-10 Systems mit bereits installiertem OpenVAS Schwachstellenscanner und einer aktuellen Schwachstellen-Datenbank. Vor der aktiven Verwendung sollte die Datenbank auf Aktualisierungen überprüft werden. Das Skript erstellt eine .box Datei. Diese kann mittels des kontext.ps1 Skriptes in die

Test-Umgebung importiert werden. Templates für weitere packer Skripte finden sich unter `https://github.com/chef/bento`. Dort zu finden ist ebenfalls die Datei ‘preseed.cfg’, welche benötigt wird um die Box ohne Nutzereingaben automatisiert erstellen zu können.

```
1         {
2         "builders": [
3         {
4         "boot_command": [
5         "<esc><wait>",
6         "install <wait>",
7         " preseed/url=http://{{ .HTTPIP }}:{{ .HTTPPort }}/{{
             user `preseed_path`}} <wait>",
8         "debian-installer=en_US.UTF-8 <wait>",
9         "auto <wait>",
10        "locale=en_US.UTF-8 <wait>",
11        "kbd-chooser/method=us <wait>",
12        "keyboard-configuration/xkb-keymap=us <wait>",
13        "netcfg/get_hostname={{ .Name }} <wait>",
14        "netcfg/get_domain=vagrantup.com <wait>",
15        "fb=false <wait>",
16        "debconf/frontend=noninteractive <wait>",
17        "console-setup/ask_detect=false <wait>",
18        "console-keymaps-at/keymap=us <wait>",
19        "grub-installer/bootdev=/dev/sda <wait>",
20        "<enter><wait>"
21        ],
22        "boot_wait": "10s",
23        "disk_size": "{{user `disk_size`}}",
24        "guest_additions_url": "{{ user `guest_additions_url` }
             }",
25        "guest_additions_path": "VBoxGuestAdditions_{{.Version}
             }.iso",
26        "guest_os_type": "Debian_64",
27        "hard_drive_interface": "sata",
28        "headless": "{{ user `headless` }}",
29        "http_directory": "{{template_dir}}/http",
30        "iso_checksum": "{{user `iso_checksum`}}",
31        "iso_checksum_type": "{{user `iso_checksum_type`}}",
32        "iso_url": "{{user `mirror`}}/{{user `mirror_directory`
             }}/{{user `iso_name`}}",
33        "output_directory": "{{ user `build_directory` }}/
             packer-{{user `template`}}-virtualbox",
34        "shutdown_command": "echo 'vagrant' | sudo -S /sbin/
             shutdown -hP now",
35        "ssh_password": "vagrant",
36        "ssh_port": 22,
37        "ssh_username": "vagrant",
38        "ssh_wait_timeout": "10000s",
39        "type": "virtualbox-iso",
40        "memory": "{{ user `memory` }}",
```

```
41          "cpus": "{{ user `cpus` }}",
42          "virtualbox_version_file": ".vbox_version",
43          "vm_name": "{{ user `template` }}"
44          },
45          {
46          "boot_command": [
47          "<esc><wait>",
48          "install <wait>",
49          " preseed/url=http://{{ .HTTPIP }}:{{ .HTTPPort }}/{{
              user `preseed_path`}} <wait>",
50          "debian-installer=en_US.UTF-8 <wait>",
51          "auto <wait>",
52          "locale=en_US.UTF-8 <wait>",
53          "kbd-chooser/method=us <wait>",
54          "keyboard-configuration/xkb-keymap=us <wait>",
55          "netcfg/get_hostname={{ .Name }} <wait>",
56          "netcfg/get_domain=vagrantup.com <wait>",
57          "fb=false <wait>",
58          "debconf/frontend=noninteractive <wait>",
59          "console-setup/ask_detect=false <wait>",
60          "console-keymaps-at/keymap=us <wait>",
61          "grub-installer/bootdev=/dev/sda <wait>",
62          "<enter><wait>"
63          ],
64          "boot_wait": "10s",
65          "disk_size": "{{user `disk_size`}}",
66          "guest_os_type": "debian8-64",
67          "headless": "{{ user `headless` }}",
68          "http_directory": "{{template_dir}}/http",
69          "iso_checksum": "{{user `iso_checksum`}}",
70          "iso_checksum_type": "{{user `iso_checksum_type`}}",
71          "iso_url": "{{user `mirror`}}/{{user `mirror_directory`
              }}/{{user `iso_name`}}",
72          "output_directory": "{{ user `build_directory` }}/
              packer-{{user `template`}}-vmware",
73          "shutdown_command": "echo 'vagrant' | sudo -S /sbin/
              shutdown -hP now",
74          "ssh_password": "vagrant",
75          "ssh_port": 22,
76          "ssh_username": "vagrant",
77          "ssh_wait_timeout": "10000s",
78          "tools_upload_flavor": "linux",
79          "type": "vmware-iso",
80          "vm_name": "{{ user `template` }}",
81          "memory": "{{ user `memory` }}",
82          "cpus": "{{ user `cpus` }}",
83          "vmx_data": {
```

```
84          "cpuid.coresPerSocket": "1",
85          "ethernet0.pciSlotNumber": "32"
86          },
87          "vmx_remove_ethernet_interfaces": true
88          },
89          {
90          "boot_command": [
91          "<esc><wait>",
92          "install <wait>",
93          " preseed/url=http://{{ .HTTPIP }}:{{ .HTTPPort }}/{{
               user `preseed_path`}} <wait>",
94          "debian-installer=en_US.UTF-8 <wait>",
95          "auto <wait>",
96          "locale=en_US.UTF-8 <wait>",
97          "kbd-chooser/method=us <wait>",
98          "keyboard-configuration/xkb-keymap=us <wait>",
99          "netcfg/get_hostname={{ .Name }} <wait>",
100         "netcfg/get_domain=vagrantup.com <wait>",
101         "fb=false <wait>",
102         "debconf/frontend=noninteractive <wait>",
103         "console-setup/ask_detect=false <wait>",
104         "console-keymaps-at/keymap=us <wait>",
105         "grub-installer/bootdev=/dev/sda <wait>",
106         "<enter><wait>"
107         ],
108         "boot_wait": "10s",
109         "disk_size": "{{user `disk_size`}}",
110         "guest_os_type": "debian",
111         "http_directory": "{{template_dir}}/http",
112         "iso_checksum": "{{user `iso_checksum`}}",
113         "iso_checksum_type": "{{user `iso_checksum_type`}}",
114         "iso_url": "{{user `mirror`}}/{{user `mirror_directory`
               }}/{{user `iso_name`}}",
115         "output_directory": "{{ user `build_directory` }}/
               packer-{{user `template`}}-parallels",
116         "parallels_tools_flavor": "lin",
117         "memory": "{{ user `memory` }}",
118         "cpus": "{{ user `cpus` }}",
119         "prlctl_version_file": ".prlctl_version",
120         "shutdown_command": "echo 'vagrant' | sudo -S /sbin/
               shutdown -hP now",
121         "ssh_password": "vagrant",
122         "ssh_port": 22,
123         "ssh_username": "vagrant",
124         "ssh_wait_timeout": "10000s",
125         "type": "parallels-iso",
126         "vm_name": "{{ user `template` }}"
```

```
127         },
128         {
129         "boot_command": [
130         "<esc><wait>",
131         "install <wait>",
132         " preseed/url=http://{{ .HTTPIP }}:{{ .HTTPPort }}/{{
              user `preseed_path`}} <wait>",
133         "debian-installer=en_US.UTF-8 <wait>",
134         "auto <wait>",
135         "locale=en_US.UTF-8 <wait>",
136         "kbd-chooser/method=us <wait>",
137         "keyboard-configuration/xkb-keymap=us <wait>",
138         "netcfg/get_hostname={{ .Name }} <wait>",
139         "netcfg/get_domain=vagrantup.com <wait>",
140         "fb=false <wait>",
141         "debconf/frontend=noninteractive <wait>",
142         "console-setup/ask_detect=false <wait>",
143         "console-keymaps-at/keymap=us <wait>",
144         "grub-installer/bootdev=/dev/vda <wait>",
145         "<enter><wait>"
146         ],
147         "boot_wait": "10s",
148         "memory": "{{ user `memory` }}",
149         "cpus": "{{ user `cpus` }}",
150         "disk_size": "{{user `disk_size`}}",
151         "headless": "{{ user `headless` }}",
152         "http_directory": "{{template_dir}}/http",
153         "iso_checksum": "{{user `iso_checksum`}}",
154         "iso_checksum_type": "{{user `iso_checksum_type`}}",
155         "iso_url": "{{user `mirror`}}/{{user `mirror_directory`
              }}/{{user `iso_name`}}",
156         "output_directory": "{{ user `build_directory` }}/
              packer-{{user `template`}}-qemu",
157         "shutdown_command": "echo 'vagrant' | sudo -S /sbin/
              shutdown -hP now",
158         "ssh_password": "vagrant",
159         "ssh_port": 22,
160         "ssh_username": "vagrant",
161         "ssh_wait_timeout": "10000s",
162         "type": "qemu",
163         "vm_name": "{{ user `template` }}"
164         }
165         ],
166         "post-processors": [
167         {
168         "output": "{{ user `build_directory` }}/{{user `
              box_basename`}}.{{.Provider}}.box",
```

```
169         "type": "vagrant"
170         }
171         ],
172         "provisioners": [
173         {
174         "environment_vars": [
175         "HOME_DIR=/home/vagrant",
176         "http_proxy={{user `http_proxy`}}",
177         "https_proxy={{user `https_proxy`}}",
178         "no_proxy={{user `no_proxy`}}"
179         ],
180         "execute_command": "echo 'vagrant' | {{.Vars}} sudo -S
              -E sh -eux '{{.Path}}'",
181         "expect_disconnect": true,
182         "scripts": [
183         "{{template_dir}}/scripts/update.sh",
184         "{{template_dir}}/../_common/motd.sh",
185         "{{template_dir}}/../_common/sshd.sh",
186         "{{template_dir}}/scripts/networking.sh",
187         "{{template_dir}}/scripts/sudoers.sh",
188         "{{template_dir}}/../_common/vagrant.sh",
189         "{{template_dir}}/scripts/systemd.sh",
190         "{{template_dir}}/../_common/virtualbox.sh",
191         "{{template_dir}}/../_common/vmware.sh",
192         "{{template_dir}}/../_common/parallels.sh",
193         "{{template_dir}}/scripts/cleanup.sh",
194         "{{template_dir}}/../_common/minimize.sh"
195         ],
196         "type": "shell"
197         },
198         {
199         "type": "shell",
200         "inline": [
201         "sudo apt-get install openvas --yes",
202         "sudo openvas-setup",
203         "sudo openvas-start",
204         "sudo openvasmd --create-user tester --password pw"
205         ]
206         }
207         ],
208         "variables": {
209         "box_basename": "openvas-packer-debian",
210         "build_directory": "../../builds",
211         "build_timestamp": "{{isotime \"20060102150405\"}}",
212         "cpus": "1",
213         "disk_size": "65536",
214         "git_revision": "__unknown_git_revision__",
```

```
215         "headless": "",
216         "http_proxy": "{{env `http_proxy`}}",
217         "https_proxy": "{{env `https_proxy`}}",
218         "guest_additions_url": "",
219         "iso_checksum": "e43fef979352df15056ac512ad96a07b515cb8
              789bf0bfd86f99ed0404f885f5",
220         "iso_checksum_type": "sha256",
221         "iso_name": "debian-10.2.0-amd64-netinst.iso",
222         "memory": "1024",
223         "mirror": "http://cdimage.debian.org/cdimage/release",
224         "mirror_directory": "10.2.0/amd64/iso-cd",
225         "name": "debian-10.2",
226         "no_proxy": "{{env `no_proxy`}}",
227         "preseed_path": "debian-9/preseed.cfg",
228         "template": "debian-10.2-amd64",
229         "version": "TIMESTAMP"
230         }
231         }
```

Listing 2: Eine Packer Konfigurationsdatei zum erstellen einer benutzerdefinierten .box Datei

Ergebnis-Übersichtsdatei

In diesem Abschnitt sieht man eine Übersichtsdatei, wie sie nach dem erfolgreichen Beenden des Scan-Prozesses persistent auf dem Dateisystem in dem entsprechenden Scanner-Vulnerable-Unterordner unter dem Namen 'result_overview.html' existiert.

All results of the current scan

File	Description
openvas_172.16.16.3_1591958551.HTML	This is an OpenVAS scan result. Click to see the scan results and configuration details.
performance_result.html	This is a graphical overview file for the nmon Performance Analysis. Click to see more details
performance_result.nmon	This is a result file of the nmon performance measurement tool. It is unformatted. For a graphical overview look for an html file with the sam
trace1.pcap	This is a result of the network traffic. You can view it with a tool like Wireshark.
trace2.pcap	This is a result of the network traffic. You can view it with a tool like Wireshark.
tripwire_log.txt	This is a tripwire result. It shows the differences of the filesystem according to the baseline set while configuring the scan-process. tripwire c
Vagrantfile	This is the local Vagrant configuration file. It can be used to repeat the scan

Abbildung 4: Eine Beispiel Übersichtsdatei mit Links zu den Original-Dateien

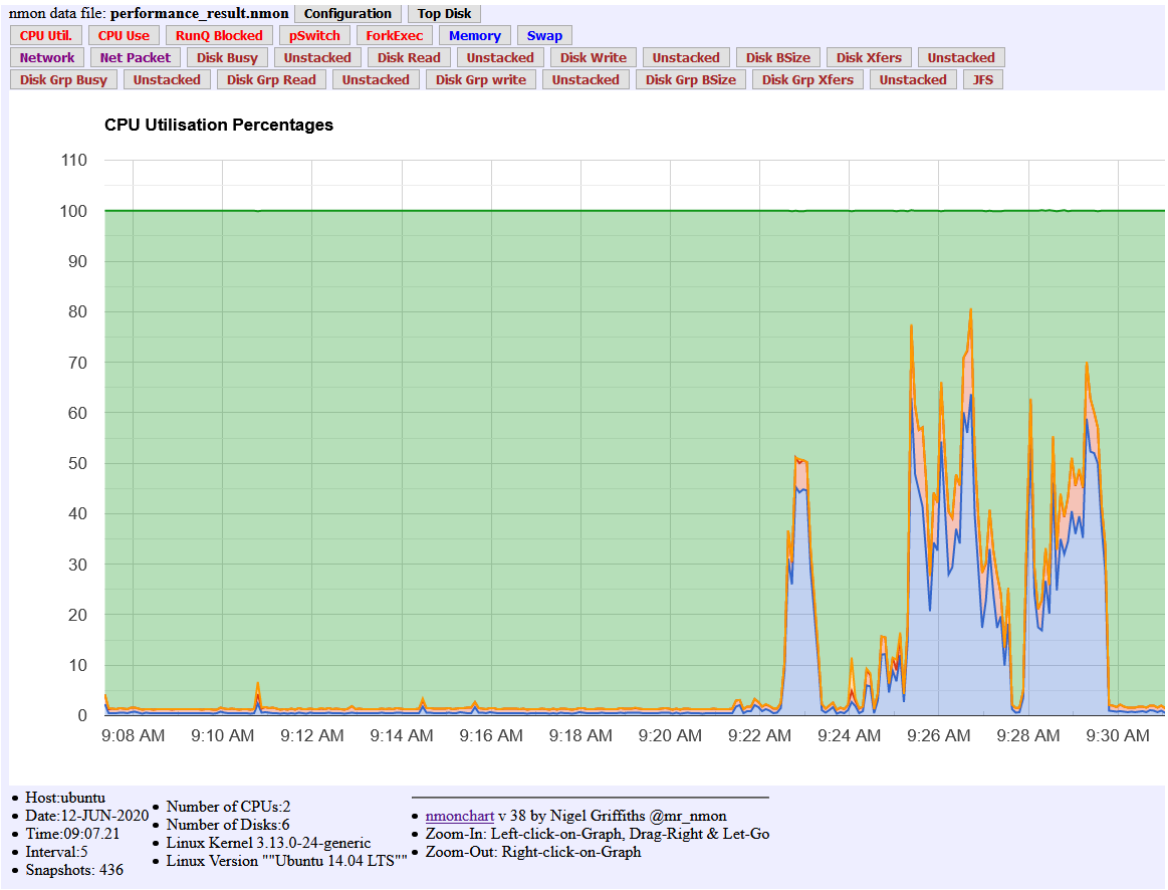

Abbildung 5: Ein Ausschnitt aus der Performance Analyse eines OpenVAS Basic Scans zu finden in der Übersichtsdatei unter ‘performance_result.html’