Technische Universität Dresden

Fakultät Informatik

Institut für Systemarchitektur

Professur für Datenschutz und Datensicherheit

Realisierung eines Honeypot-Netzes

Diplomarbeit

Vorgelegt von:	Pascal Brückner geboren am 16.08.1988 in Zwickau Matrikelnummer 3390554
Betreuer:	Dr.-Ing. Stefan Köpsell

Beginn der Arbeit:	Dresden, den 17.02.2014
Tag der Einreichung:	Dresden, den 17.11.2014

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir eingereichte Diplomarbeit mit dem Thema **Realisierung eines Honeypot-Netzes** vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Dresden, den

Unterschrift:

(Pascal Brückner)

Abstract Honeypots haben sich als probates Mittel zur Ergänzung bestehender IT-Sicherheitsinfrastrukturen etabliert. Ihre große Flexibilität erlaubt sowohl den Einsatz mit einer öffentlichen IP, um Angriffe direkt aus dem Internet zu lokalisieren, sowie auch den effektiven Betrieb innerhalb des Intranets, um Bedrohungen „aus dem Inneren“ erkennen zu können. Darunter fällt beispielsweise sich selbstständig ausbreitende Malware oder aber auch gezielter unrechtmäßiger Zugriff durch unautorisierte Personen. Im Rahmen der vorliegenden Arbeit wird eine Honeypot-Architektur für das hochkomplexe Sächsische Verwaltungsnetz (SVN) konzipiert, implementiert und anschließend im praktischen Einsatz evaluiert. Der Fokus der Arbeit liegt hierbei insbesondere auf Skalierbarkeit, Benutzerfreundlichkeit und Kosteneffizienz der entwickelten Lösung.

Danksagung

An dieser Stelle möchte ich allen Personen danken, die durch ihre Unterstützung zum Gelingen dieser Arbeit beigetragen haben.

Mein Dank gilt insbesondere meinem Betreuer, Herrn Dr.-Ing. Stefan Köpsell, der jederzeit für Rückfragen zur Verfügung stand und diese interessante Arbeit an einer real existierenden IT-Infrastruktur überhaupt erst ermöglichte.

Weiterhin möchte ich meinen Ansprechpartnern innerhalb des Staatsministeriums der Justiz, Herrn Karl-Otto Feger und Herrn Christoph Damm, für die Zusammenarbeit und die Bereitstellung aller Mittel danken, die zur Integration von Testsystemen in das Sächsische Verwaltungsnetz benötigt wurden. Herzlich bedanken möchte ich mich auch bei den Mitarbeitern des Staatsbetriebs Sächsische Informatik Dienste, insbesondere bei Herrn Uwe Hoppenz und Herrn Jörg Schneider, für die individuelle Betreuung bei allen technischen und organisatorischen Fragen.

Mein Dank gebührt außerdem Herrn Uwe Hübler und insbesondere Herrn Andreas Fischer von T-Systems für die Unterstützung während der technischen Umsetzung dieser Diplomarbeit.

Für das kritische Korrekturlesen der Arbeit und ihre konstruktiven Vorschläge möchte ich außerdem noch Richard Stosch, Bodo von Sobiesky, Christoph Damm und meinem Betreuer Herrn Stefan Köpsell danken.

Abschließend gilt mein ganz besonderer Dank meinen Eltern, die mir meinen Berufswunsch überhaupt erst ermöglicht haben und stets helfend zur Seite standen sowie allen Freunden, die mich während der Entstehung der Arbeit mit Ausdauer und Geduld unterstützt haben.

Inhaltsverzeichnis

I Einleitung 9

II Grundlagen 10

1 Einsatz von Honeypots 12

1.1 Klassifikation 13

1.1.1 Interaktivität 13

1.1.2 Endpunkt 16

1.1.3 Netzwerk-Teleskop 17

1.1.4 Honeytoken 18

1.1.5 Sonstige 18

1.2 Netzwerkintegration 19

2 Die IT-Infrastruktur der sächsischen Landesverwaltung 23

2.1 Das Sächsische Verwaltungsnetz 23

III Anforderungsanalyse 26

3 Technische Anforderungen 27

4 Organisatorische Anforderungen 29

IV Konzeption 30

5 Trafficanalyse 30

5.0.1 Auswertung 32

5.0.2 Schlussfolgerungen 33

6 Architektur **34**

6.1 Verwandte Arbeiten 34

6.2 Netzwerktopologie 37

6.3 Management und Analyse 40

6.4 Sensoren 42

V Implementierung: HoneySens **46**

7 Server **46**

7.1 REST-API 47

7.1.1 Schnittstellendefinition 48

7.1.2 Domänenmodell 53

7.1.3 Implementierung 59

7.2 Web-Frontend 68

7.2.1 Architektur 68

7.2.2 Implementierung 71

7.2.3 Funktionsumfang 77

7.3 Deployment 84

8 Sensoren **88**

8.1 Hardwareplattform und Betriebssystem 88

8.2 Implementierung 89

8.2.1 Komponenten 89

8.2.2 Management 90

8.2.3 Passive Scan Mode 92

8.2.4 Honeypots 94

9 Problemstellungen **96**

9.1 Firmwareupdate 96

9.2 Sensorinstallation 99

VI Testbetrieb **100**

10 Labortests **101**

11 Produktive Testumgebungen **103**

12 Auswertung **104**

VII Schlussbetrachtung **105**

Abbildungsverzeichnis

1 Die High-Level-Architektur des SVNs 25

2 Die hybriden Honeynet-Architekturen Collapsar und Potemkin [32] 35

3 Die abstrakte Topologie des Honey-Netzes 39

4 JSON-Objekt zum Anlegen einer Sensor-Konfiguration 52

5 Antwort des Servers nach erfolgreicher POST-Operation 52

6 Das HoneySens-Domänenmodell (Teil 1) 54

7 Das HoneySens-Domänenmodell (Teil 2) 58

8 Serverseitige Verzeichnisstruktur 63

9 Interaktion mit dem EntityManager $em des doctrine-Frameworks 64

10 Auszug aus der User-Klasse 65

11 Verzeichnisstruktur der Webanwendung 72

12 HTML-Grundgerüst mit Templates 73

13 Model-Implementierung im Frontend 74

14 Model-View-Integration im Web-Frontend 75

15 Dashboard der Webanwendung 78

16 Ereignisliste 80

17 Sensorliste 82

18 Dialog zum Hinzufügen eines Sensors 82

19 HoneySens-Dockerfile 86

20 Sensor-Verzeichnisstruktur 90

21 Funktion zur Ermittlung des sensorspezifischen Polling-Delays 91

22 Arbeitsweise des Passive Scan Modes (Pseudocode) 93

Teil I

Einleitung

Der *Internet Crime Report* des *Internet Crime Complaint Centers*, welches weltweit Daten über die durch Cyberkriminalität verursachten Schäden zu erheben versucht, verzeichnete für das Jahr 2013 mehr als 260.000 Vorfälle und einen Gesamtverlust von über 780 Millionen Dollar [1]. Für den gleichen Zeitraum nennt der *Norton Cybercrime Report* hingegen angerichtete Schäden im Wert von 113 Milliarden Dollar und 378 Millionen direkt Betroffene [2]. Der Bericht besagt weiterhin, dass 41 % aller sich im Netz bewegenden Erwachsenen bereits einmal Opfer von Malware, Phishing oder vergleichbaren Angriffen geworden sind. Es wird schnell deutlich, dass es aufgrund des schieren Ausmaßes dieser Form von Kriminalität sehr schwierig geworden ist, darüber allgemeingültige Aussagen zu treffen; insbesondere unter Berücksichtigung der Tatsache, dass die Entwicklung von auf spezifische IT-Infrastrukturen zugeschnittener Malware in den letzten Jahren zugenommen hat, wie es das Beispiel *Stuxnet* bereits im Jahr 2010 demonstrierte [20]. Im *Verizon Data Breach Investigations Report* für das Jahr 2014 wurde weiterhin deutlich, dass Angriffe der Kategorie *Insider Misuse* und *Cyber-espionage* konstante Bedrohungen darstellen [3], denen nur schwer mit herkömmlichen Sicherheitsbestrebungen wie beispielsweise restriktiven Firewall-Policies begegnet werden kann. Ein Angreifer, der physischen Zugang zum internen Netzwerk besitzt oder womöglich über Techniken des *Social Engineerings* privilegierten Zugang erhalten hat, aber auch Mitarbeiter, die ihre Privilegien aus welchen Gründen auch immer überschreiten, stellen somit eine ernst zu nehmende Gefahr für die Integrität einer jeden komplexen IT-Infrastruktur dar. Die möglichen Wege der Infektion von Produktivsystemen mit Malware sind durch in manchen Unternehmen zulässigen Prinzipien wie *Bring your own device* inzwischen vielfältiger Natur, ebenso kann aber auch schon der Besuch einer attackierenden Website oder das unbedachte Öffnen eines scheinbar harmlosen E-Mail-Anhangs einen Einbruch in das Netzwerk nach sich ziehen. All diesen Fällen ist gemein, dass die das interne Netzwerk vom Internet trennenden Schutzmaßnahmen versagt haben oder aber bedingt durch die Art des Angriffes gar keine Wirkung zeigen konnten. *Honeypots* stellen ein erprobtes Mittel dar, das in Zusammenarbeit mit anderen Technologien wie *Intrusion Detection Systems* und *Anti-Viren-Software* die Sicherheit innerhalb eines Netzwerks verbessern und auch neue, während eines Vorfalls noch unbekannte Angriffsmuster erkennen kann [16].

Im Rahmen dieser Arbeit soll ein Honey-Netz entworfen werden, das speziell auf derartige Angriffe *aus dem Inneren* hin optimiert ist und dabei helfen soll, die Bedrohungslage innerhalb einer komplexen Netzwerkarchitektur in kurzer Zeit und mit nur wenigen zusätzlichen Ressourcen zu analysieren. Grundlage ist dabei die IT-Infrastruktur der sächsischen Landesverwaltung, auf deren spezifische Anforderungen sowohl der Entwurf als auch die Entwicklung eines funktionsfähigen Prototypen ausgerichtet sind. Noch vor der Anforderungsanalyse und Konzeption des Gesamtsystems werden zunächst die für den Betrieb von Honeypots benötigten Grundlagen beschrieben, gefolgt von einem Überblick über die Architektur des *Sächsischen Verwaltungsnetzes* (SVN). Den umfangreichsten Teil der Arbeit nimmt schließlich die Beschreibung der Implementierung des *HoneySens* genannten Prototypen ein, der anschließend im Testbetrieb sowohl unter kontrollierten Laborbedingungen als auch im Rahmen von zwei Testinstallationen innerhalb von produktiven Umgebungen evaluiert wird. Die Arbeit schließt mit einer Auswertung der gewonnenen Erkenntnisse und einem Ausblick auf mögliche zukünftige Entwicklungen ab.

Teil II

Grundlagen

Das Zeitalter der modernen Informationstechnik ist geprägt von einer kontinuierlichen und sich ständig im Wandel befindlichen Bedrohungslage. Ziel der Angreifer sind zumeist digitale Güter: Der Diebstahl sensibler Informationen wie Kreditkarten- oder Accountdaten, aber auch der Missbrauch fremder Systeme als Proxy sind an der Tagesordnung.

Als Reaktion auf diese Gefahren wurden im Laufe der Zeit verschiedene Verteidigungstechniken entwickelt. Zu den populärsten gehört **Anti-Viren-Software** (AV-Software), die Schadsoftware zu erkennen versucht, indem sie Dateien auf bekannte Signaturen von Schadcode überprüft. Frühe AV-Software entstand in den 80er Jahren [40] und wurde seitdem beständig weiterentwickelt. Moderne kommerzielle Produkte dieser Kategorie sind sowohl für Desktops als auch Server erhältlich und auf Benutzerfreundlichkeit hin optimiert. Da jedoch immer neue Bedrohungen auftauchen, erfordert diese Software auch ein beständiges Aktualisieren der Signaturdaten. Trotz der Tatsache, dass AV-Software sich speziell im Bereich der Windows-Desktops, aber auch auf anderen Plattformen zur Standardausstattung herausgebildet hat, erklärte der Vizepräsident der Firma *Symantec* im Mai 2014 kommerzielle AV-Software für „tot“ [41]. Als Grund führte er an, dass nur noch ca. 45 Prozent aller Angriffe von dieser erkannt werden würden. Aufgrund der wachsenden Malwarevielfalt ließen sich Einbrüche folglich immer schwerer verhindern, was eine zunehmende Fokussierung auf die *Analyse* von Vorfällen erfordere. Bei dieser erweisen sich andere Tools als nützlich, wie beispielsweise *Intrusion Detection Systeme* (IDS) oder eben Honeypots.

Eine weitere weit verbreitete Verteidigungstechnik ist der Einsatz von **Firewalls**. Diese existieren in zahlreichen Varianten, weshalb das Konzept von Cheswick und Bellovin eher abstrakt definiert wurde:

> „A firewall is a collection of components, interposed between two networks, that filters traffic between them according to some security policy.“ [18]

Firewalls operieren auf einer oder auch mehreren Ebenen des OSI-Schichtenmodells zugleich. Sie inspizieren live die einzelnen Pakete, die am System ankommen oder es verlassen, und wenden dabei einen Regelsatz an, der festlegt, wie mit bestimmten Paketen verfahren werden soll. Typischerweise werden sie entweder akzeptiert, verworfen oder zurückgewiesen – die Konfiguration der Firewall ist hierbei speziell vom Administrator an die Anforderungen des jeweiligen Netzwerks anzupassen, in dem sie zum Einsatz kommt. Firewalls sind ein essentieller Bestandteil jeglicher IT-Sicherheitskonzepte und können sowohl als separate Geräte als auch in Form einer reinen Softwarelösung vorkommen.

Im Kontrast zu Firewalls erlauben **Network Intrusion Detection Systems** (NIDS) das Untersuchen des Datenverkehrs im Netzwerk hinsichtlich Signaturen bekannter Schadsoftware oder Angriffsmustern. Analog zu AV-Software muss die lokale Signaturdatenbank regelmäßig mit Patches vom Hersteller des NIDS aktualisiert werden, damit das System in der Lage bleibt, auch neue Gefahren zu erkennen. Während NIDS rein passive Systeme sind, die lediglich Daten aufzeichnen und im Falle eines erfolgreichen Signaturabgleichs einen Alarm generieren, können **Network Intrusion Prevention Systems** (NIPS) in den Datenverkehr eingreifen und auch selbst Pakete generieren oder verwerfen – sie sind folglich das

aktive Gegenstück zu NIDS. Sie können ähnlich wie Firewalls verdächtige Verbindungen gezielt unterbrechen und somit im Idealfall einen Angriff, wie beispielsweise die Infektion eines Systems im zu schützenden Netzwerk mit Schadsoftware, verhindern. Zudem existieren noch sogenannte **Host Intrusion Detection Systems** (HIDS), deren Ziel das Überwachen aller Aktivitäten eines einzelnen PC-Systems ist. Solche Software ist beispielsweise in der Lage, Angreifer daran zu hindern, lokale Systemdateien zu modifizieren, was häufig Bestandteil der Installation eines Rootkits ist.

Allen zuvor genannten Sicherheitskonzepten ist gemein, dass sie lediglich auf bereits bekannte Gefahren reagieren können. Sie sind reaktive Systeme, die neuartige Malware (die wiederum auf neu entdeckten Sicherheitslücken und vielleicht auch *Zero-Day-Exploits*[1] basiert) erst erkennen können, nachdem die Signaturdatenbank entsprechend aktualisiert wurde. Um jedoch diesen Gefahren nicht völlig schutzlos ausgeliefert zu sein, haben beispielsweise die Entwickler von AV-Software und IDS/IPS *Heuristiken* implementiert, die Datenströme oder die Systemaufrufe einzelner Prozesse auf Auffälligkeiten hin überwachen, welche häufig Anzeichen für einen versuchten Angriff sind, aber nicht zwingend sein müssen. Somit ist es unter Umständen möglich, auch neuartige oder modifizierte Malware zu identifizieren. Zugleich steigt jedoch die Gefahr von *False Positives* deutlich an, was im schlimmsten Fall verursachen kann, dass für einen Nutzer das Weiterarbeiten unmöglich wird.

Ein weiteres, proaktives Instrument zur Absicherung von IT-Landschaften sind **Honeypots**. Dabei handelt es sich um gut überwachte Systeme, die für potentielle Angreifer nach Möglichkeit nicht von einem Produktivsystem im gleichen Netzsegment zu unterscheiden sind. Die Idee beruht dabei auf dem Konzept der Täuschung: Ein Angreifer soll nach Möglichkeit von den Produktivsystemen, auf denen die „echten" Daten liegen, abgelenkt werden. Durch das umfangreiche Monitoring eines Honeypots ist es zudem möglich, vielfältige Informationen über den Eindringling zu gewinnen – Art und Umfang dieser Daten hängt hierbei vom Typ des eingesetzten Honeypots ab [16]. Für den Begriff des „Honeypots" existieren in der Fachliteratur zahlreiche Definitionen, üblicherweise findet jedoch jene von Lance Spitzner Anwendung:

> „A honeypot is a closely monitored computing resource that we want to be probed, attacked or compromised." [37]

Besondere Aufmerksamkeit liegt hierbei immer auf dem Begriff *Monitoring*, also der umfangreichen Überwachung des Honeypot-Systems. Dies dient zum einen dem offensichtlichen Zweck der Informationsgewinnung über Angreifer oder anderweitige verdächtige Vorgänge auf dem System, andererseits aber auch der Eindämmung von Gefahren, die vom Honeypot selbst ausgehen können. Bei einer gewünschten und im Sinne der obigen Begriffsdefinition unvermeidlichen Manipulation einer Honeypot-Installation durch einen Angreifer muss sichergestellt werden, dass diese Plattform nicht als Ausgangspunkt für weitere Angriffe auf Produktivsysteme genutzt werden kann. Die Ausbreitung auf andere Honeypots kann hingegen durchaus erwünscht sein, um beispielsweise die Verbreitung von Viren in einem Netzwerk zu untersuchen. Auf diesen Zweck ausgerichtete Maßnahmen werden unter dem Begriff *Containment* zusammengefasst.

Einer der größten Vorteile des Einsatzes von Honeypots im Vergleich zu den zuvor genannten Verteidigungstechniken ist die Möglichkeit der Erkennung von *Zero-Day-Exploits*. Honeypots erfüllen in einem Netzwerk keine andere Aufgabe als das Warten auf verdächtige Aktivitäten und das Aufzeichnen derselben. Somit können nach Definition jegliche Kommunikation oder sonstigen Zustandsänderungen (z.B. im

[1] „A zero-day attack is a cyber attack exploiting a vulnerability that has not been disclosed publicly" [15]

Dateisystem oder den aktiven Prozessen), die nicht zum stillen erwarteten Systemverhalten gehören, als verdächtig klassifiziert und aufgezeichnet werden. Es ist folglich auch möglich, bisher noch unbekannte Angriffe aufzuzeichnen und zu analysieren.

Die zuvor genannten Eigenschaften sind je nach Honeypot-Typ verschieden stark ausgeprägt und wiederum mit Vor- und Nachteilen verbunden, die im Anschluss an dieses Kapitel näher beleuchtet werden. Essentiell für den Einsatz von Honeypots zur Absicherung von IT-Infrastrukturen ist jedoch die Berücksichtigung der Tatsache, dass es sich lediglich um Ergänzungslösungen handelt, die andere Maßnahmen wie Firewalls oder NIDS keineswegs ersetzen können. Die Erweiterung von regel- und signaturbasierten Sicherheitsverfahren durch Honeypots erhöht allerdings die Effizienz der Gesamtlösung, insbesondere durch die Qualität und Quantität der gesammelten Daten. So können beispielsweise mit Hilfe der Informationen über Zero-Day-Exploits, die mit Honeypots gewonnen wurden, neue Signaturen für bestehende HIDS/NIDS erzeugt werden.

Im Rahmen dieser Arbeit soll die bestehende umfangreiche IT-Infrastruktur des Landes Sachsen durch den Einsatz von Honeypots ergänzt werden. Es ist zu analysieren, welche Rahmenbedingungen hierbei zu beachten sind und welche Hard- und Software unter Berücksichtigung der spezifischen Anforderungen dieses Netzes zum Einsatz kommen soll.

1 Einsatz von Honeypots

Honeypots sind lediglich eine von vielen möglichen Implementierungen des Konzeptes der „Täuschung“ und existieren ihrerseits wiederum in vielfältigen Varianten, die sich unter anderem durch die mit dem Einsatz verbundenen Risiken, die Menge und Qualität der gewonnenen Informationen, ihren Installationsaufwand und ihre Wartbarkeit massiv voneinander unterscheiden [23]. Ziel dieses Kapitels ist es, die wichtigsten dieser Varianten mit ihren jeweiligen Vor- und Nachteilen vorzustellen.

Allen Honeypotlösungen gemein ist das Wechselspiel zwischen den drei übergeordneten Zielen **Sicherheit**, **Performance** und **Genauigkeit** [32]. Sicherheit bezeichnet hierbei die im Rahmen dieser Arbeit oft unter dem Begriff *Containment* zusammengefassten Maßnahmen, mit denen verhindert werden soll, dass ein Angreifer ernsthaften Schaden außerhalb eines Honeypots anrichten kann. Performance ist hingegen ein Indikator für die Skalierbarkeit eines Honeypots mit wachsendem Datenverkehr, wohingegen die Genauigkeit ein Maß für die Exaktheit der Emulation eines Systems durch eine Honeypot-Infrastruktur ist. Der Wettstreit zwischen den Indikatoren besagt nun, dass es erfahrungsgemäß nicht möglich ist, alle drei Ziele gleichermaßen zu erfüllen. Je mehr Fokus eine Honeypot-Architektur auf die Genauigkeit der Emulation legt, desto mehr wachsen beispielsweise auch die damit verbundenen Anforderungen an die Hardware, was wiederum die erreichbare Performance vermindert. Ebenso verhält es sich exemplarisch mit der Sicherheit: Je umfassender die getroffenen Containment-Maßnahmen zum Schutz der den Honeypot umgebenden IT-Landschaft sind, desto weniger Interaktionsmöglichkeiten eröffnen sich einem potentiellen Angreifer bei der Übernahme eines Honeypot-Systems.

1.1 Klassifikation

Zur Einordnung der verschiedenen Konzepte, die zur Umsetzung des Honypot-Täuschungsprinzips entstanden sind, wurden verschiedene Taxonomien vorgeschlagen, darunter die „Taxonomy of Honeypots“ von Seifert et al. [34], an der sich die nachfolgenden Ausführungen grob orientieren.

1.1.1 Interaktivität

Um die Idee der Täuschung eines Angreifers möglichst effektiv umzusetzen, versuchen Honeypots bestehende Produktivsysteme im selben Netzsegment nachzubilden, indem sie beispielsweise angebotene Netzwerkdienste replizieren. Der Honeypot-Einsatz gilt dann als gelungen, wenn es dem Angreifer nicht mehr möglich ist, die Fallen von den realen Systemen zu unterscheiden. Über einen Angreifer können letztendlich umso mehr Daten gesammelt werden, je umfangreicher die Interaktion mit dem Honeypot-System ausfällt. Wie bereits angesprochen, wird in der Praxis oft versucht, bereits bestehende Systeme und die von diesen angebotenen Dienste innerhalb einer IT-Infrastruktur mit Hilfe von Honeypots nachzubilden. Folglich bieten besagte Produktivsysteme die höchste theoretisch erreichbare Interaktivität und können als Maßstab zur Bewertung nach diesem Kriterium herangezogen werden.

Sogenannte **High-Interaction-Honeypots** sind vollwertige Systeme, die dieselbe Software einsetzen, die auch auf den Produktivsystemen zum Einsatz kommt. Dies könnte beispielsweise ein handelsüblicher Desktop-PC mit Windows-Betriebssystem und für diese Klasse an Rechnern typischer Anwendungssoftware sein. Durch Modifikationen des Betriebssystems kann dafür gesorgt werden, dass jegliche Systemaufrufe und die gesamte Netzwerkkommunikation des Rechners live an ein externes Überwachungssystem weitergeleitet werden. Zusätzliche Containment-Maßnahmen, wie z.B. eine transparente Netzwerkbrücke, können zum Einsatz kommen, um den Netzwerkverkehr vom und zum Honeypot zu überwachen und bei Bedarf zu regulieren, damit ausgehend vom infizierten System nicht weitere Angriffe gestartet werden. Diese Kategorie von Software wurde von Niels Provos und Thorsten Holz folgendermaßen spezifiziert [32]:

> „High-Interaction honeypots offer the adversary a full system to interact with. This means that the honeypot does not emulate any services, functionality, or base operating systems. Instead, it provides real systems and services, the same used in organizations today. Thus, the attacker can completely compromise the machine and take control of it.“

Die hohe Interaktivität erlaubt es einem Eindringling, das Honeypot-System vollständig zu übernehmen und nach Belieben zu modifizieren. Je länger dieser auf dem System verbleibt und Spuren hinterlässt, desto umfangreicher und wertvoller ist die Menge der gewonnenen Informationen durch das Monitoring des Honeypots. Durch das geschickte Verteilen von Dummydaten auf dem Locksystem kann der Angreifer zusätzlich motiviert werden, weiter in das System vorzudringen. Falls es sich beim Eindringling um einen Menschen und nicht ein automatisches System (*Bot*) handelt, wird dieser im Idealfall auch von den Produktivsystemen abgelenkt, die sein eigentliches Ziel darstellen. Die hierdurch gewonnene Zeit kann dann zusätzlich zur Absicherung der bestehenden Systeme und zum Aussperren des Angreifers genutzt werden, falls der Vorfall live beobachtet wird. Die dank dieser Eigenschaft gewonnenen umfangreichen Informationen über den Angreifer und seine Vorgehensweise stehen jedoch einem erheblichen

Sicherheitsrisiko gegenüber, dem mit geeigneten Containment-Maßnahmen begegnet werden muss: Da der Angreifer einen High-Interaction-Honeypot vollständig unter seine Kontrolle bringen kann, besteht schließlich die Gefahr, dass dieser zur Quelle weiterer bösartiger Aktivitäten wird. High-Interaction-Honeypots eignen sich zusätzlich noch hervorragend zum Erkennen von Zero-Day-Exploits und werden genau für diesen Einsatzzweck auch von großen Unternehmen wie beispielsweise Microsoft (in Form des *Honeymonkey*-Projekts [44]) eingesetzt. Zusätzlich können sie auch abseits ihres primären Einsatzzweckes als Sandbox zur Malware-Analyse [16] verwendet werden.

Ein weiterer wichtiger Aspekt der High-Interaction-Honeypots ist die Reaktion auf einen Einbruch. Falls durch einen Angreifer Änderungen am System vorgenommen wurden, beispielsweise durch die Installation eines Rootkits, das Systemdateien verändert, müssen diese zur späteren Analyse untersucht und das Honeypot-System wieder auf seinen Ursprungszustand zurückgesetzt werden. Bei Verwendung eines separaten PC-Systems als Honeypot (*„bare-metal“*) gestaltet sich dieser Schritt als umständlich: Ein Festplattenabbild des frisch installierten Systems muss nach jedem erfolgreichen Einbruch in den Honeypot wiederhergestellt werden. Dies erfordert das manuelle Booten des Rechners von einem Rettungssystem, das Wiederherstellen des sauberen Festplattenabbildes und einen erneuten Start des Honeypots selbst. Dieses umfangreiche Prozedere lässt sich unter Ausnutzung moderner *Virtualisierungstechniken* deutlich vereinfachen: High-Interaction-Honeypots können in virtuellen Maschinen betrieben werden und erlauben somit nebenbei eine effizientere Auslastung der Ressourcen des Hostsystems, die beim Bare-Metal-Betrieb sonst nicht genutzt werden würden [42]. Weiterhin ist es ohne zusätzlichen Aufwand möglich, vom sauberen, frisch installierten Honeypotsystem ein Abbild (*Snapshot*) zu erstellen, das nach erfolgter Infektion zur Wiederherstellung genutzt werden kann. Der Betrieb von mehreren virtuellen Honeypots auf einem einzigen physischen Host bietet zudem den Vorteil der Abschirmung der Honeypot-Instanzen untereinander, die implizit durch den Hypervisor gewährleistet wird. Virtualisierung fügt jedoch auch eine weitere Ebene an Komplexität in eine Honeypot-Infrastruktur ein, was sich in neuen die Sicherheit gefährdenden Risiken äußern kann. So besteht beispielsweise die Gefahr, dass durch Fehler im Hypervisor virtuelle Maschinen zum Absturz gebracht werden können oder – schlimmer – es Möglichkeiten gibt, aus dem Honeypot „auszubrechen“ und Zugriff auf das Hostsystem zu erlangen [30]. Um diesen Gefahren zu begegnen, sind umfassende Maßnahmen zum Containment der virtuellen Maschinen, bzw. des Hostsystems zu treffen. Ein gemeinsamer Betrieb von Honeypot- und produktiven Instanzen auf ein- und demselben Hostsystem sollte vermieden werden. Da sich der Betrieb von Honeypots mit Hilfe von Virtualisierungssoftware deutlich vom Bare-Metal-Einsatz abhebt, unterscheiden Niels Provos und Thorsten Holz zwischen **Physical Honeypots** und **Virtual Honeypots** [32].

Der Einsatz von hochgradig interaktiven Honeypots ist speziell in großen Unternehmen und anderen umfangreichen IT-Landschaften mit einigen schwerwiegenden Nachteilen verbunden: Obwohl sie herkömmliche Securitylösungen wie Firewalls und NIDS lediglich ergänzen, aber keineswegs ersetzen können, beschert ihr Betrieb einen hohen zusätzlichen Installations- und Wartungsaufwand. Sowohl das Einrichten der Systeme als auch ihr Betrieb und die ständige Überwachung sind ausgesprochen komplex und erfordern zu diesem Zweck ausgebildetes Personal. Aus den von aktuellen High-Interaction-Honeypots generierten Protokolldaten können nur erfahrene Anwender genügend Informationen gewinnen, um den Angriffshergang vollständig nachvollziehen zu können. Gebündelt mit den Sicherheitsrisiken während des Betriebs scheint es verständlich, dass High-Interaction-Honeypots von Unternehmen bisher eher selten eingesetzt werden[23]. In solch einer Situation sollte jedoch der Einsatz von **Low-Interaction-Honeypots** in Erwägung gezogen werden.

Hierbei handelt es sich um Software, die real existierende Betriebssysteme, Netzwerkdienste und auch Sicherheitslücken selektiv *emuliert*. Provos und Holz definieren [32]:

> „A low-interaction honeypot often simulates a limited number of network services and implements just enough of the Internet protocols, usually TCP and IP, to allow interaction with the adversary and make him believe he is connecting to a real system.“

Es existiert eine große Vielfalt an Software, die als Low-Interaction-Honeypot genutzt werden kann – der bezeichnende Unterschied zwischen diesen Lösungen ist immer die emulierte Systemtiefe. Die Auswahl reicht von einfachen Skripten, die lediglich eine TCP-Verbindung aufbauen und alle ankommenden Pakete aufzeichnen, bis hin zu komplexen Implementierungen ganzer Netzwerkdienste wie *SSH* oder dem *SMB*-Protokoll, das für die Windows-Dateifreigabe genutzt wird. Je umfangreicher die Emulation ausfällt, desto größer ist die Interaktivität des Honeypots und desto mehr Informationen können idealerweise über einen Angreifer gewonnen werden. Die Bandbreite an potentiell einsetzbarer Software spiegelt das theoretische Potential wieder: Einfache historische Honeypots wie das *Deception Toolkit* [6] öffnen lediglich Ports und zeichnen ankommenden Datenverkehr auf, während moderne Projekte wie beispielsweise *dionaea* oder *amun* eine ganze Reihe von funktionsfähigen Diensten inklusive bekannter Sicherheitslücken nachbilden [14].

Den Low-Interaction-Honeypots gemein ist die im Vergleich zu ihren hochgradig interaktiven Pendants geringere Systemtiefe und Interaktivität und folglich geringere Menge an gewinnbaren Informationen über einen potentiellen Angreifer, der sich typischerweise auch kürzer mit dem System selbst befasst – zu oft sind die Interaktionsmöglichkeiten zu gering, um ihn längerfristig aufzuhalten. Eine Ausnahme ist in diesem Bezug der Honeypot *kippo*[2], der das SSH-Protokoll emuliert und nach einem erfolgten Einbruch dem Angreifer auch eine interaktive Shell zur Verfügung stellt (die allerdings nur simuliert ist und nur wenige Befehle kennt). An den von den kippo-Entwicklern aufgezeichneten Beispiel-Sessions wird ersichtlich, dass dieser Honeypot menschliche Angreifer durchaus längere Zeit beschäftigen und gleichzeitig noch viele Informationen über die Vorgehensweise und verwendete Hilfsmittel gewinnen kann. Die geringere Systemtiefe der Low-Interaction-Honeypots hat im praktischen Einsatz aber auch positive Auswirkungen: Das von solcher Software ausgehende Risiko für die umgebende IT-Infrastruktur fällt deutlich geringer aus, da es einem Eindringling per Definition nicht möglich ist, das System vollständig zu übernehmen. Es ist folglich auch möglich, solche Honeypots längere Zeit unbeaufsichtigt und – ein geeignetes Frontend vorausgesetzt – von Personal auswerten zu lassen, das nur rudimentäre Erfahrungen auf diesem Gebiet aufweist. Typischerweise verfolgen Low-Interaction-Honeypots einen spezifischen Einsatzzweck – ein typisches Beispiel hierfür ist das Sammeln von Malware – und erlauben keine darüber hinausgehende Interaktion. Exemplarisch sei an dieser Stelle der Honeypot *nepenthes* erwähnt, der ausgewählte Sicherheitslücken einer Reihe von Netzwerkdiensten wie z.B. SMB/CIFS simuliert, ohne die Funktionalität der Dienste selbst nachzubilden [9]. Eine Verbindung wird abgebrochen, sobald der Service für ihn unbekannte Pakete empfängt, die nicht auf eines der erwarteten Kommunikationsmuster passen. Ziel der Software ist es, die nach Ausnutzung eines Exploits typischerweise vom Angreifer verwendete Malware zur Infektion des Systems zu empfangen, abzuspeichern und anschließend die Verbindung zu unterbrechen. Da nepenthes primär Sicherheitslücken der Windows-Plattformen simuliert, wäre ein direktes Ausführen der Malware ohnehin nicht möglich [32]. Die bereits erwähnten Projekte nepenthes und kippo kombinieren Konzepte von Low- und High-Interaction-Honeypots gleichermaßen

[2] `https://github.com/desaster/kippo` (abgerufen im November 2014)

und lassen sich außerdem nur schwer einer eindeutigen Kategorie zuordnen. Aus diesem Grund werden sie von ihren Autoren selbst als **Medium-Interaction-Honeypots** bezeichnet [16].

Zuletzt sei noch darauf hingewiesen, dass auch Low- und Medium-Interaction-Honeypots trotz ihrer geringen Sicherheitsimplikationen umfassender Containment-Maßnahmen bedürfen. Ursache ist einerseits, dass es unter Umständen auch mit Hilfe der geringen Interaktionsmöglichkeiten dieser Honeypots theoretisch möglich ist, in beschränktem Rahmen weitere Systeme anzugreifen. So erlaubt die simulierte Shell in kippo das Absetzen von HTTP[3]-Requests an beliebige URLs, was theoretisch für SQL-Injection-Angriffe genutzt werden könnte. Andererseits besteht immer auch die Gefahr von noch unentdeckten Sicherheitslücken in der Honeypot-Software selbst, die im schlimmsten Fall Zugriff auf das Hostsystem erlauben, auf dem der Honeypot läuft. Zur Absicherung empfiehlt sich das Betreiben der betreffenden Software innerhalb von Containerumgebungen wie *LinuX Containers* (LXC), *User Mode Linux* (UML) oder auch mit Hilfe eines Hypervisors (*VirtualBox*, *KVM*, etc.) – analog zu High-Interaction-Honeypots [16].

1.1.2 Endpunkt

Zusätzliche Diversifizierung kann anhand der Rolle erfolgen, die ein Honeypot in einer „Konversation“ einnimmt. Viele Netzwerkprotokolle – speziell die von Honeypots implementierten – arbeiten nach dem traditionellen Client/Server-Modell und sind somit Vorbild für den Entwurf entsprechender Honeypots, die diese Dienste repräsentieren. Unter dem Begriff der **Server-Side-Honeypots** ist Software zu verstehen, die selbst Netzwerkdienste anbietet und auf sich mit diesen verbindende Clients wartet. Es sind Honeypots im herkömmlichen Sinne, wie sie in den Anfangsjahren der „Hackerfallen“ ausschließlich existierten: Systeme, die nach Möglichkeit von den produktiven Servern und Workstations innerhalb einer IT-Infrastruktur nicht zu unterscheiden sind und Angriffe von extern oder auch intern auf sich ziehen sollen. Auch hier gilt wieder die zuvor ausgeführte Unterscheidung nach Interaktivität: Serverseitige High-Interaction-Honeypots sind vollwertige Systeme mit „echten“ Betriebssystemen und Anwendungssoftware, die Netzwerkdienste anbieten und genauestens auf Zustandsänderungen überwacht werden, während serverseitige Low-Interaction-Honeypots eine bereits beschriebene Dienste-Emulation durchführen. Ein wichtiger Aspekt, der bei der Installation von serverseitigen Honeypots berücksichtigt werden muss, ist deren Platzierung innerhalb des Netzwerkes. Ihre Konfiguration variiert stark und sollte sich am Netzsegment orientieren, in dem sie aufgestellt werden. So erscheint es ratsam, dass von Honeypots innerhalb einer *demilitarisierten Zone* (DMZ) angebotene Dienste identisch zu denen der Server im gleichen Netzwerk sind. Je besser sich Honeypots in die Servicelandschaft integrieren und je mehr IP-Adressen für einzelne oder mehrere Honeypot-Instanzen eingesetzt werden, desto größer ist die Chance, auf Angriffe aufmerksam zu werden. Analog dazu sollten sich Honeypots, die beispielsweise in einem Büronetzwerk aufgestellt werden, an deren typischen Services orientieren und im Falle von Windows-Systemen ebenfalls SMB/CIFS-Freigaben anbieten.

Analog zu serverseitigen Honeypots existiert noch die Gruppe der sogenannten **Client-Side-Honeypots**, die als Reaktion auf die Beobachtung hin entstanden ist, dass Angriffe auf IT-Systeme in den letzten Jahren gehäuft nicht mehr auf die Server-, sondern die Clientseite ausgeübt wurden [16]. So sind inzwischen beispielsweise sogenannte *Drive-By-Downloads* populär, bei denen Sicherheitslücken in Web-Browsern ausgenutzt werden, um solche Clients mit Malware zu infizieren, die mit anfälligen Browser-Versionen

[3]Hypertext Transfer Protocol

schadhafte Websites besuchen. Um diesem Trend zu begegnen, wurde Honeypot-Software entwickelt, die Browser- und Nutzerverhalten simuliert und automatisiert fragwürdige Websites besucht, um von diesen genutzte Exploits anhand von auffälligem Systemverhalten zu identifizieren. Low-Interaction-Honeypots dieses Typs sind hierbei auf bekannte Sicherheitslücken getrimmt und benötigen ähnlich wie AV-Software regelmäßige Signaturupdates, können allerdings auch von Angreifern eingeschleusten Shellcode emulieren und auf diese Weise noch unbekannte Exploits aufdecken. High-Interaction-Honeypots nutzen hingegen unterschiedliche „echte“ Browser verschiedener Versionen und überwachen das darunterliegende Betriebssystem auf untypische Veränderungen in der Prozessliste, im Dateisystem oder im Falle von Windows-Systemen in der Registry. Es existieren hierzu eine Reihe von Techniken, die in der Regel von diversen Honeypots implementiert werden und auf verschiedene Typen von Exploits ausgelegt sind. Eine Kombination ungleichartiger Honeypot-Software ist daher sinnvoll [14].

Client-Side-Honeypots existieren nicht nur in Form simulierter Web-Browser. Selbstverständlich kann ein solcher Honeypot auch für andere Protokolle oder auch spezifische Datenformate geschrieben werden, die über das Netzwerk empfangbar sind.

Wie bereits im vorherigen Abschnitt diskutiert, spielt die Platzierung von Server-Side-Honeypots eine entscheidende Rolle für ihre Erreichbarkeit und die Art der zu erwartenden Daten. Clientseitige Honeypots unterliegen hingegen weniger Beschränkungen, da für Server die Quell-IP-Adresse einer Anfrage in der Regel nicht von Bedeutung ist [32]. Die wichtigste Voraussetzung bei der Integration solcher Honeypots ist, dass sie den auf Exploits zu überprüfenden Server überhaupt erreichen können. Unter Umständen kann es vorkommen, dass Server wiederholte Anfragen von einer einzelnen IP-Adresse oder einem Subnetz ab einem gewissen Limit blockieren, um z.B. *DoS/DDoS*-Angriffe[4] abzuwehren. In solch einem Fall ist es ratsam, für ein Client-Honeynet mehrere ausgehende IP-Adressen zur Verfügung zu stellen oder längere Wartezeiten zwischen den Anfragen einzufügen.

Clientseitige Honeypots werden primär von großen Unternehmen wie *Microsoft* oder *Symantec* eingesetzt, die selbst AV-Software entwickeln. Der Einsatz in einer herkömmlichen IT-Infrastruktur ist abgesehen vom akademischen Wert der gewonnenen Informationen fragwürdig, da es praktisch unmöglich ist, alle von Nutzern einer IT-Infrastruktur theoretisch mit Clients kontaktierbaren Server zu überprüfen. Somit könnten diese Honeypots lediglich reaktiv eingesetzt werden, wenn eine Infektion schon erfolgt ist. Aus den dadurch gewonnenen Daten könnten zwar Rückschlüsse auf den Infektionsweg gezogen werden, das Aufwand-Nutzen-Verhältnis dieser Vorgehensweise ist allerdings fragwürdig.

1.1.3 Netzwerk-Teleskop

Ein im Rahmen dieser Arbeit wichtiges Konzept ist zudem das des Netzwerk-Teleskops, das als rein passives System einen fixen IP-Adressbereich überwacht und dabei jeglichen ankommenden Datenverkehr aufzeichnet. Ein Teleskop besitzt hierzu keine fixe(n) IP-Adresse(n) – stattdessen wird über eine entsprechende Routing-Konfiguration sichergestellt, dass Daten mit den zugehörigen Zieladressen an es weitergeleitet werden [16]. Zu beachten ist bei einer solchen Infrastruktur natürlich, dass das Teleskop nur bisher noch nicht im Netzwerk vorhandene oder vergebene IP-Adressen abdeckt. Besondere Vorsicht ist in Netzwerken geboten, die das *DHCP*-Protokoll zur automatischen Vergabe von IP-Adressen einsetzen. Hier muss sichergestellt werden, dass Traffic an den korrekten Host umgeleitet wird, sobald dieser

[4] Distributed Denial of Service

eine IP-Adresse bezieht.

Aufgrund ihrer rein passiven Natur können Netzwerk-Teleskope nicht auf ankommenden Datenverkehr antworten, was einerseits das Risiko ihres Einsatzes drastisch reduziert, andererseits aber auch wenig Informationen über einen potentiellen Angreifer preisgibt. In der Praxis lassen sich solche Sensoren sinnvoll mit anderen Low- oder High-Interaction-Honeypots kombinieren, um einen möglichst großen oder wahlweise auch den gesamten Adressbereich eines Netzwerks überwachen zu können.

1.1.4 Honeytoken

An dieser Stelle soll aus Gründen der Vollständigkeit noch auf das Konzept der *Honeytoken* eingegangen werden. Dabei handelt es sich um digitale Entitäten, deren Wert im Erkennen des unrechtmäßigen Zugriffs liegt [16]. Beispielsweise könnte dies eine Patientenakte im Krankenhaus sein, die einen fiktiven Patienten beschreibt und folglich niemals im regulären Klinikbetrieb geöffnet werden sollte. Falls doch ein Zugriff erfolgt, ist dieser per Definition verdächtig. Der Einsatz von Honeytoken erfolgt zumeist, um Angriffe *aus dem Inneren* zu identifizieren – also beispielsweise eine Person, die sich unrechtmäßig Zugriff auf das interne Netzwerk verschafft hat oder aber auch Angestellte, die die aufgestellten Richtlinien verletzen. Weitere typische Beispiele für den Einsatz von Honeytoken sind gezielt platzierte E-Mails, Accountdaten, Bankkontodaten oder auch URLs/Verweise.

1.1.5 Sonstige

Bei der soeben beschriebenen Taxonomie handelt es sich lediglich um eine Sammlung von Richtlinien zur Klassifikation von Honeypot-Software. Selbstverständlich existieren einerseits Programme, die verschiedene dieser Ansätze miteinander kombinieren und andererseits Honeypots, die sich nur schwer in die genannten Kategorien einordnen lassen. Ein Beispiel für letztere ist der *Ghost USB Honeypot*, der USB-Sticks simuliert, um sich über dieses Medium ausbreitende Malware zu identifizieren [31]. Als verwundbares virtuelles Bluetooth-Gerät soll hingegen der *Bluepot* Angreifer anlocken [36]. Beide Anwendungen stellen keine Honeypots im klassischen Sinne dar, da sie nicht den TCP/IP-Protokollstack nutzen, sondern auf andere Protokolle setzen und – zumindest in den genannten Fällen – besondere Hardware simulieren.

Der kombinierte Einsatz von Low- und High-Interaction-Honeypots resultiert in **Hybriden Honeypots**. Provos und Holz klassifizieren derartige Systeme folgendermaßen [32]:

> „When low-interaction systems are not powerful enough and high-interaction systems are too expensive, hybrid solutions offer the benefits of both worlds.“

Im praktischen Einsatz nutzen serverseitige hybride Systeme eine Reihe von Low-Interaction-Honeypots, um die Kommunikation mit zum Zeitpunkt des ersten Kontakts noch unbekannten Clients abzuwickeln. Falls sich ein solcher Datenstrom als interessant (im Sinne von möglicherweise bedrohlich) herausstellt oder die Limits der Low-Interaction-Honeypots im Sinne der simulierten Systemtiefe erreicht sind, übernehmen die High-Interaction-Honeypots. Ziel derartiger Architekturen ist primär das Versorgen von tausenden von gleichzeitigen Verbindungen, die beispielsweise anfallen, wenn ganze Netzwerke (beispiels-

weise ein Klasse-B-Netz mit bis zu 65534 IP-Adressen) mit Honeypots ausgestattet werden sollen, um ein einzelnes großes *Honeynet* zu bilden – ein Netzwerk von Honeypots. Analog zu serverseitigen Systemen existieren auch clientseitige hybride Honeypots, die beispielsweise zum massenhaften Überprüfen von Websites auf Drive-By-Downloads eingesetzt werden können [16], indem verdächtige URLs sowohl mit Low- als auch High-Interaction-Honeypots besucht und getestet werden.

Zum Zeitpunkt dieser Arbeit aktuelle, einsatzbereite hybride Systeme sind das clientseitige *HoneySpider Network 2*[5], welches Websites und deren Komponenten (u.a. Flash-Applets) mit Fokus auf Skalierbarkeit untersucht und das serverseitige *SurfCERT IDS*[6], das eine Reihe von Sensoren in verschiedenen Netzwerken einsetzt, um eingehenden Traffic an eine zentrale Instanz weiterzuleiten, auf der die Honeypot-Software und ein umfangreiches Webinterface zur Auswertung der gesammelten Daten betrieben werden [16]. Hybride Honeypots sind ein aktives Forschungsthema, weshalb in einschlägiger Literatur von einer ganzen Reihe von Prototypen und Versuchsprojekten berichtet wird, die jedoch nie veröffentlicht wurden und deren Quellcode somit nicht zugänglich ist. Beispiele für derartige Projekte sind *Collapsar* [24] oder *Potemkin* [42].

1.2 Netzwerkintegration

Oberste Priorität beim Einsatz von Honeypots gebührt der sorgsamen Überwachung und Abschirmung der Systeme, um zu verhindern, dass ein erfolgreicher Angreifer Honeypots selbst für weitere gegen andere Hosts gerichtete Aktivitäten missbraucht. Eine Ausnahme hiervon stellen Honeynets dar, in denen beispielsweise die Ausbreitung von Malware-Würmern untersucht wird und ein solches Verhalten folglich gewünscht ist – allerdings auch hier nur innerhalb des abgeschotteten und überwachten Netzwerks. Da es aufgrund der spezifischen Anforderungen bei der Installation von Honeypots häufig gewünscht ist, dass diese mit über das Internet erreichbaren Hosts kommunizieren, bedarf dieser Übertragungsweg besonderer Aufmerksamkeit. So empfiehlt beispielsweise das *Honeynet Project*[7], das sich mit der Entwicklung und Erforschung von IT-sicherheitsrelevanter Software beschäftigt, den Einsatz einer **Honeywall** [38]. Dabei handelt es sich um eine auf *CentOS* basierende Linux-Distribution, die als transparente Netzwerkbrücke zwischen einem Honeypot oder Honeynet und dem zu schützenden Produktivnetzwerk oder auch dem Internet agiert. Jeglicher Datenverkehr, der die Honeywall passiert, wird analysiert, protokolliert und ist über ein Webinterface auswertbar. Zusätzlich wird der Datenverkehr live mit Hilfe des NIDS *snort*[8] auf Auffälligkeiten hin untersucht. Weiterhin bietet die Distribution noch die Möglichkeit, den Paketfluss quantitativ zu begrenzen – beispielsweise auf 20 ausgehende Pakete pro Honeypot. Somit erhalten Angreifer mehr Freiheiten und können beispielsweise diese limitierte Trafficbeschränkung nutzen, um zusätzliche Malware herunterzuladen, eine Beteiligung der übernommenen Honeypots an beispielsweise *DDoS*-Angriffen wird jedoch verhindert. Wahlweise eingehende und/oder ausgehende Pakete, die einen Alarm auslösen, werden automatisch blockiert. Somit ist sichergestellt, dass von übernommenen Systemen im Honeynet keine weiteren Angriffe auf Hosts in anderen Netzwerken ausgeführt werden. Falls Angriffe auf die Honeypots von außerhalb erwünscht sind, kann eingehender Verkehr allerdings auch von dieser Regelung ausgeschlossen werden.

[5] `http://www.honeyspider.org` (abgerufen im September 2014)
[6] `http://ids.surfnet.nl` (abgerufen im September 2014)
[7] `http://honeynet.org` (abgerufen im September 2014)
[8] `http://snorg.org` (abgerufen im September 2014)

Der Einsatz einer Honeywall bringt zusätzlichen Installations- und Konfigurationsaufwand mit sich, speziell unter Berücksichtigung der Tatsache, dass die letzte Veröffentlichung dieser Distribution ins Jahr 2009 zurückreicht und somit eine Installation auf aktueller Hardware Probleme bereitet. Der Einsatz von Virtualisierungssoftware ist hier empfehlenswert. Selbstverständlich ist es auch möglich, auf Basis einer aktuellen Distribution die Funktionalität der Honeywall-Distribution zu replizieren, da diese ausschließlich Open-Source-Software einsetzt; der damit verbundene Aufwand ist jedoch enorm. Das Honeynet Project selbst empfiehlt die Honeywall hauptsächlich zur Absicherung von High-Interaction-Honeypots, da von diesen das größte Risiko ausgeht. Eine Übernahme des Systems durch einen Angreifer ist möglich und in der Regel auch erwünscht. Ein solches Szenario ist hingegen bei Low-Interaction-Honeypots nicht vorgesehen und nur denkbar, wenn Fehler in der Honeypot-Software selbst einen Ausbruch aus der simulierten Umgebung ermöglichen.

Um für diesen Fall gerüstet zu sein und den Einsatz einer hochkomplexen transparenten Netzwerkbrücke zu ersparen, was insbesondere beim Aufstellen mehrerer Honeypots innerhalb verschiedener interner Netze nötig wird, existieren noch eine Reihe weiterer Techniken zur Absicherung gegen potentielle Exploits der Honeypot-Software selbst:

- Grundsätzlich gilt es die **Abschirmungsmaßnahmen von Prozessen**, die das jeweilige Betriebssystem – im Falle von Honeypots in der Regel ein unixoides – mitbringt, korrekt zu konfigurieren. Im Falle eines Linux-Systems könnte das bedeuten, den Honeypot-Prozess nicht mit `root`-Rechten, sondern als gesonderten, nur für diese Software eingerichteten Benutzer laufen zu lassen. Falls der Prozess kompromittiert wird, besitzt der Angreifer zunächst nur die Rechte des separat eingerichteten Benutzers. Diese sollten dementsprechend auch ausschließlich auf die nötige Funktionalität zum Betrieb des Honeypots beschränkt sein, um die Angriffsfläche zu verringern. Modernere Linux-Kernel bieten zudem die Möglichkeit, über *Control Groups* (cgroups) und *Namespaces* die für eine Anwendung bereitgestellten Systemressourcen und deren Interaktionsmöglichkeiten mit anderen Prozessen zu regulieren, was ebenfalls zur Risikominimierung genutzt werden kann.
- Das „Einsperren“ des risikobehafteten Prozesses in ein eigenes virtuelles Dateisystem wird unter Linux-Systemen üblicherweise mit Hilfe des Tools **chroot** bewerkstelligt. Somit ist sichergestellt, dass ein Angreifer, der den Low-Interaction-Honeypot-Prozess selbst kompromittiert, lediglich Zugriff auf ein minimales virtuelles Dateisystem hat, das dem Prozess exklusiv zur Verfügung steht. Ein Ausbruch aus einer solchen Umgebung ist nicht möglich, ohne weitere potentielle Systemfehler auszunutzen. Eine Alternative zu `chroot` ist beispielsweise *Jails* unter *FreeBSD*, das einen ähnlichen Ansatz verfolgt.
- Das Tool **systrace** [32] überwacht die von Prozessen getätigten Systemaufrufe mit Hilfe einer zuvor definierten oder angelernten Whitelist. Systemaufrufe außerhalb der Prozessspezifikationen werden blockiert. Dieser Schutzmechanismus ist speziell auf den unvorhergesehenen Missbrauch von Honeypots abgestimmt und kann Indizien auf Exploitversuche der Honeypot-Software liefern und diese zugleich unterbinden.
- Ein ressourcenintensiverer Ansatz ist wiederum der Einsatz von Virtualisierungssoftware. Diese schirmt implizit die virtuellen Honeypot-Instanzen von anderen virtuellen Maschinen ab, kann jedoch auch selbst Ziel eines Angriffs sein [10]. Eine Diskussion der verfügbaren Virtualisierungslösungen würde den Rahmen dieser Arbeit jedoch sprengen.
- Es ist unabdingbar, sowohl die Honeypot-Software selbst als auch das darunterliegende Betriebssystem immer aktuell zu halten und regelmäßig mit Sicherheitsaktualisierungen zu versorgen, sobald diese verfügbar sind.

Abgesehen von den bereits behandelten Sicherheitsaspekten bedarf auch die Integration von Honeypots in eine Netzwerkinfrastruktur einer genaueren Betrachtung. Diese unterscheidet sich zwischen serverseitigen und clientseitigen Honeypots. Die Installation letzterer stellt nur geringe Anforderungen: Da in der Regel mit clientseitigen Honeypots Ressourcen im Internet abgerufen und auf von ihnen ausgehende Gefahren (wie z.B. Drive-By-Downloads) hin untersucht werden sollen, muss ein solcher lediglich die gewünschten Hosting-Server erreichen können. Die Platzierung derartiger Honeypots im Netzwerk erlaubt somit unter Berücksichtigung der bereits besprochenen Containment-Maßnahmen vielerlei Freiheiten. Der Fokus bei solchen Architekturen und speziell clientseitigen hybriden Honeypots liegt schließlich verstärkt auf der Skalierbarkeit in Form der Parallelisierung des Betriebs, wie Microsoft schon 2005 mit dem **HoneyMonkey** getauften Honeynet, das Websites nach Exploits für den *Internet Explorer* durchsuchte, demonstrierte [44].

Die Integration von serverseitigen Honeypots bringt komplexere Anforderungen mit sich. Von zentraler Bedeutung ist die Frage nach der Erreichbarkeit der Systeme. Je mehr IP-Adressen und Adressräume durch einen oder mehrere Honeypots abgedeckt werden, desto größer ist die Wahrscheinlichkeit, dass ein Angreifer darauf aufmerksam wird. Neben dieser rein quantitativen Erreichbarkeit ist jedoch auch die Qualität der Täuschung entscheidend: Zumindest ein Eindringling, der gezielt gegen ein bestimmtes Netzwerk vorgeht, wird vor dem eigentlichen Angriff Scanner wie *nmap*[9] verwenden, um einen Überblick über alle Hosts und die angebotenen Dienste zu erhalten. Damit die Honeypots sich gut in diese Architektur integrieren, sollten sie sich nicht durch die Scanergebnisse bereits von den restlichen Systemen abheben, sondern hinsichtlich ihrer Architektur (Betriebssystem, Dienste) nahtlos in das Netzwerk integriert sein. Es ist daher sinnvoll, eine Honeypot-Infrastruktur – soweit möglich – an die Gegebenheiten der zu schützenden IT-Landschaft anzupassen. Ein Großteil der beobachteten Angriffe auf Netzwerke sind allerdings zum Zeitpunkt dieser Arbeit automatisierter Natur [16], werden also von Bots oder Malware ohne menschliche Intervention durchgeführt. Eine Eigenheit solcher Angriffsversuche ist die scheinbar zufällige, häufig aber algorithmische Opferauswahl ohne vorherigen Scan, ob ein verwundbarer Dienst überhaupt angeboten wird. Das Zuschneiden der Honeypots auf die sie umgebende Architektur ist folglich eher gegen nicht automatisierte Angriffe hilfreich. Gleichzeitig kann aber angenommen werden, dass auch nicht speziell für das betreffende Netzwerk angepasste Honeypots Traffic erhalten werden. Es ist daher im Einzelfall abzuwägen, ob es sinnvoll ist, bestehende Systeme mit Hilfe von Honeypots nachzubilden.

Für die Platzierung der Honeypots selbst existieren verschiedene Möglichkeiten:

- In einem **externen Setup** befindet sich der Honeypot direkt hinter einem Gateway, das vom Internet ins interne Netzwerk führt. Ausgewählter Traffic, beispielsweise gefiltert nach einem bestimmten Port oder nach ausgewählten Ziel-IPs, die der Honeypot repräsentiert, wird direkt an diesen weitergeleitet. Das Gateway übernimmt in einem solchen Fall auch die Aufgabe des Firewallings und ist somit für das Containment des Honeypots mitverantwortlich – es kann entsprechend den vom Honeypot ausgehenden Traffic ins Internet limitieren. Der Einsatz einer *Honeywall* als Gateway ist denkbar. Um die Gefährdung des internen Netzes durch eine Übernahme des Honeypots jedoch weiter zu minimieren, werden Honeypots üblicherweise hinter einer weiteren, separaten Firewall in einem eigenen Netzsegment platziert [16]. Die Integration mehrerer – häufig auch virtueller – zusätzlicher Honeypot-Systeme in das Netzsegment des Honeypots kreiert ein abgeschlossenes Honeynet, in dem nach Infektion eines Hosts auch die Ausbreitung von Malware auf weitere Hosts beobachtet werden kann.

[9] `http://nmap.org` (abgerufen im September 2014)

- In **internen Setups** werden Honeypots so platziert, dass sie ausschließlich aus dem internen Netzwerk einer Organisation heraus kontaktiert werden können. Die Containment-Maßnahmen bedürfen speziell im Fall des Einsatzes von serverseitigen High-Interaction-Honeypots besonderer Aufmerksamkeit, da Angreifer sich nach Übernahme eines solchen Systems innerhalb des Produktivnetzwerks befinden, das auch gefährdete Maschinen enthält. Diesem Risiko steht die Fülle der gewinnbaren Informationen gegenüber: das Netzwerk sieht per Definition für den Angreifer realistisch aus, was die Chance erhöht, dass mehr und detailliertere Informationen über dessen Vorgehensweise und Absichten gewonnen werden können. Weiterhin können derartige intern platzierte Honeypots als Sensoren für mit Malware infizierte Hosts verwendet werden. Da der Zugriff auf interne Honeypots nur aus dem internen Netz erfolgt, da das Gateway keinen Traffic aus dem Internet direkt an den Honeypot weiterleitet, ist jeder Verbindungsversuch mit derartigen Honeypots bereits als verdächtig einzustufen [16]. Falls das Risiko als zu groß eingestuft wird, können natürlich auch interne Setups ein eigenes Subnetz nutzen, das ein gesondertes Honeynet darstellt und die Kommunikation zwischen Produktivsystemen und den Honeypots limitiert. In allen Fällen ist bei der IP-Adressvergabe zu berücksichtigen, dass speziell im Falle des Einsatzes von *DHCP* die Honeypots keine Adressen erhalten, die zuvor von anderen Hosts genutzt wurden. Dies vermeidet *False Positives*, die durch Pakete von Hosts verursacht werden könnten, die mit dem „alten“ Host zu kommunizieren versuchen, der zuvor die IP-Adresse eines Honeypots besaß.

Sowohl interne als auch externe Setups beschreiben im Wesentlichen autonome Systeme, da jeder Honeypot separat für sich Daten sammelt. Für kleinere Architekturen mit nur einigen wenigen Honeypots ist dieses Organisationsprinzip ausreichend. Die Daten eines jeden Hosts gesondert auszuwerten, wird jedoch mit steigender Anzahl an Honeypots immer ineffizienter und zeitaufwändiger. Hinzu kommt der Wunsch, Korrelationen zwischen den gesammelten Daten der einzelnen Knoten herzustellen. Um beispielsweise die Ausbreitung eines Malware-Wurms untersuchen zu können, müssten all diese unabhängig voneinander gesammelten Daten manuell zueinander in Bezug gesetzt werden.

Abhilfe bei diesem Problem schaffen die bereits genannten hybriden Honeypot-Architekturen, die speziell mit Fokus auf Skalierbarkeit entwickelt wurden. Ein von diesen abgeleitetes Prinzip ist das sogenannte **Network of Sensors** [23]. Während Hybridsysteme sich durch den gleichzeitigen Einsatz von Low- und High-Interaction-Honeypots auszeichnen, beschreibt ein solches Sensornetzwerk viel mehr eine Architektur, mit der eine große Anzahl an Sensoren beliebigen Typs effizient implementiert und verwaltet werden kann. Zu diesem Zweck wird eine Reihe von Rechnern in einem oder auch mehreren Netzwerken installiert, die alle gesammelten Daten an eine zentrale Serverinstanz zur Auswertung weiterleiten – häufig unterstützt durch ein Webinterface. In einem weiteren Schritt wäre es auch denkbar, über die zentrale Instanz eine Management- und Konfigurationsschnittstelle anzubieten. Auf den Sensoren läuft entweder direkt die Honeypot-Software, oder aber jeder Sensor ist über einen Tunnel, beispielsweise via *VPN*[10], mit der Serverseite verbunden und leitet jeglichen eingehenden Datenverkehr direkt an den Server weiter, wo die Honeypots an zentraler Stelle laufen. Dies verbessert die Wartbarkeit des Systems, da nur an einem Punkt die Honeypots überwacht werden müssen, etabliert jedoch auch einen *Single Point of Failure* und stellt zudem höhere Bandbreiten- und Hardwareanforderungen an das Serversystem. Eine große Stärke dieser Technik ist allerdings die Möglichkeit, alle gesammelten Daten einer großen Menge von Honeypots zentral auswerten und leicht miteinander in Korrelation setzen zu können.

[10] Virtual Private Network

2 Die IT-Infrastruktur der sächsischen Landesverwaltung

Im Rahmen der Diplomarbeit fungierte der *Staatsbetrieb Sächsische Informatik Dienste* (SID) als Praxispartner und Auftraggeber zugleich. Dabei handelt es sich um eine dem *Staatsministerium der Justiz und für Europa* unterstellten Organisationseinheit, die als IT-Dienstleister für die sächsische Landesverwaltung auftritt [11]. Seine Zielstellung beschreibt der Staatsbetrieb folgendermaßen:

> „Der SID ist zentraler Ansprechpartner für alle IT-Belange seiner Kunden. Ziel des Staatsbetriebes ist es, in allen Ressorts der Landesregierung einen hohen IT-Standard zu halten.“

Hierzu unterstützt der SID die Landesverwaltung und die zugehörigen Verwaltungseinheiten in den Bereichen der IT-Verfahren, -Vorhaben, -Infrastruktur und -Beschaffung, sowie beim Ausarbeiten und der Implementierung von Sicherheitskonzepten.

2.1 Das Sächsische Verwaltungsnetz

Eine der vielen Aufgaben des SIDs ist die Verwaltung des *Sächsischen Verwaltungsnetzes* (SVN). Dabei handelt es sich um ein leistungsfähiges, weiträumiges Kommunikationsnetz für den zügigen Informationsaustausch innerhalb der sächsischen Landesverwaltung. Die verschiedenen Ressorts und Behörden des Landes sind dabei im SVN in Form von logischen Netzsegmenten vertreten [26]. Das Netzwerk selbst untergliedert sich wiederum in vier große und eigenständige Teilbereiche: das interne Netzwerk, den Internetzugang, die Mobilkommunikation und die Festnetztelefonie. Zudem wird das SVN auch vom *Kommunalen Datennetz Sachsen* (KDN) beansprucht, das die Gemeindeverwaltungen in Sachsen zur Kommunikation nutzen können. Aktuell wird diese Einrichtung von Landratsämtern, Stadtverwaltungen und Gemeindeverwaltungen als Sprach- und Datennetz eingesetzt [7].

Das SVN stellt sämtliche für die Landesverwaltung benötigten Informations- und Kommunikationsdienste und die zugehörige Infrastruktur bereit. Es besitzt hierzu eine hochverfügbare Backbone-Architektur, an die alle Netzwerkteilnehmer des Landes Sachsen angeschlossen sind. Diese zentrale Architektur untergliedert sich wiederum in drei als *Cluster* bezeichnete separate Bereiche:

- Das **Core Cluster** bezeichnet ein auf der *RPR*-Technologie[11] basierendes Netzwerk zwischen Standorten in Leipzig, Dresden und Chemnitz.
- Der **Ost-Cluster** verbindet die Städte Bautzen, Kamenz und Dresden mit Hilfe von *Ethernet Connect 1G*, einem von der Firma *Telekom* bereitgestellten Produkt auf Gigabit-Ethernet-Basis.
- Ebenfalls auf Ethernet-Basis sind die Städte Chemnitz, Plauen und Zwickau miteinander verbunden und bilden den sogenannten **West-Cluster**.

Im zuvor beschriebenen Backbone-Netz existieren eine Reihe von Schnittstellen und Übergänge in andere Netzwerke und zu Plattformen, die ebenfalls über das SVN erreichbar sind oder dieses als Grundlage nutzen. Hierzu zählen unter anderem die Mobilfunknetze für Sprache und Daten, der Anschluss an das öffentliche Sprachnetz für sogenannten *Breakin-* und *Breakout*-Verkehr, der zentrale Internetzugang, die

[11] Resilient Packet Ring

eGovernment-Plattform, das KDN und das Teilnetz mit den *Zentralen Diensten* (ZSS). Über die physischen Netzsegmente hinweg werden weiterhin Technologien wie *Virtual Private Networking* (VPN), *Virtual Routing and Forwarding* (VRF) und *Virtual Local Area Network* (VLAN) eingesetzt, um logische Netzsegmente zu bilden, die wiederum durch Firewalls voneinander abgetrennt sind. Einen Überblick über diese Architektur gibt die Abbildung 1.

Die **Zentrale Dienste Plattform** (ZSS) soll an dieser Stelle noch kurz näher erläutert werden, da diese im weiteren Verlauf der Arbeit als Integrationsnetzwerk für Honeypots relevant sein wird. Es handelt sich hierbei um eine Plattform, die von den SVN-Teilnehmern zentral genutzte, interne – also nicht von außen über das Internet erreichbare – Dienste anbietet. Hierzu zählen unter anderem Web-Server, Verzeichnisdienste wie *DNS*, *ADS* und *RADIUS*, E-Mail-Infrastrukturen mit Spamschutz, Webzugang, Virenscans usw., Zeit- (*NTP*[12]) und Videokonferenzdienste [26]. Zur Absicherung der angebotenen Services ist zudem entsprechende Schutzinfrastruktur vorhanden, darunter Firewalls, IDS/IPS und AV-Scanner. Das Teilnetz stellt eine zentrale Komponente des SVNs dar und ist somit ein interessanter Ansatzpunkt zur Integration von Honeypots.

Es existiert weiterhin ein autarkes, über alle Netzsegmente hinweg betriebenes logisches **Sprachnetz**, das auf *VoIP*-Basis[13] wiederum in ein Produktiv- und ein davon abgetrenntes Testnetzwerk untergliedert ist. Es wird für die interne VoIP-Kommunikation zwischen den an das SVN angeschlossenen Stellen eingesetzt und ist aufgrund der administrativen Freiheiten für die partizipierenden Standorte ein weiterer interessanter Standort für Honeypots.

Da das ZSS zentral verwaltet wird, sind an das Netz auch noch einige *demilitarisierte Zonen* angebunden, die u.a. für das Beherbergen von zusätzlichen Diensten genutzt werden, deren Administration jedoch vom jeweiligen Dienstebetreiber durchgeführt wird (sogenanntes *Housing*). Aufgrund der Fremdadministration etablieren sich auch diese Subnetze als interessante Aufstellungsorte für Honeypots, um frühzeitig Gefahren von möglicherweise übernommenen Servern begegnen zu können – speziell weil sowohl interne als auch externe Housing-DMZs existieren, die über das Internet direkt adressierbar sind.

In den Netzen der Ressorts sind weiterhin auch mehr als zwanzig *Active-Directory*-Domänen (AD) vorhanden, die zur Verwaltung der überwiegend mit *Microsoft Windows* ausgestatteten Clientsysteme herangezogen werden. Während serverseitig eine heterogene Betriebssystemauswahl vorliegt, kommen in den zahlreichen Behörden und Ämtern überwiegend Windows-Clients zum Einsatz. Da diese immer akut durch Malware gefährdet sind, die beispielsweise über Drive-By-Downloads oder E-Mail-Anhänge ins Netzwerk gelangt, ist auch in diesen Netzsegmenten der Einsatz von Honeypots denkbar.

[12] Network Time Protocol
[13] Voice over IP

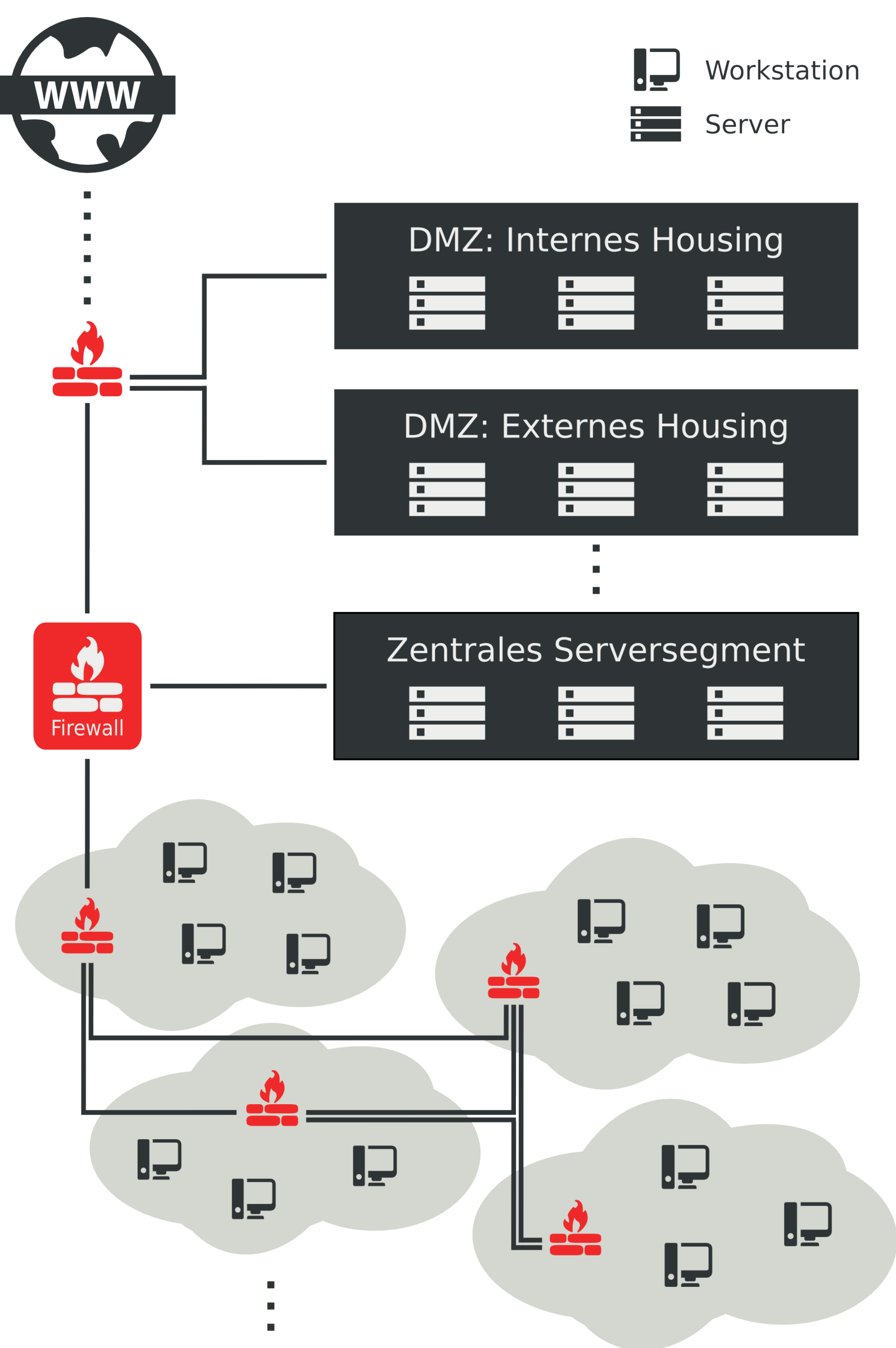


Abbildung 1: Die High-Level-Architektur des SVNs

Teil III

Anforderungsanalyse

Im Rahmen dieser Arbeit soll ein Honeypot-Netzwerk für den Einsatz im sächsischen Verwaltungsnetz (SVN) konzipiert und prototypisch implementiert werden, das den spezifischen Anforderungen der IT-Sicherheit und den infrastrukturellen Gegebenheiten des zu schützenden Netzwerkes gerecht wird. Die Evaluation erfolgt im Laufe der Arbeit anhand der gesammelten Daten der im Produktivnetzwerk installierten Honeypots innerhalb eines klar definierten Testzeitraumes.

Dieses hochkomplexe, weiträumige und heterogene Netzwerk mit seinen tausenden von Hosts ist allein schon aufgrund seiner Größe und der Tatsache, dass eine große Zahl von Menschen mit unterschiedlichem Sicherheitsbewusstsein damit arbeitet und auf die im Netz angebotenen Ressourcen zugreift, ständigen Gefahren von sowohl außerhalb als auch innerhalb ausgesetzt. *Äußere Gefahren* umfassen primär Angriffe aus dem Internet auf die direkt erreichbaren Teile der Serverinfrastruktur. Hierzu zählen primär Web- und Mail-Server sowie alle sonstigen für die Öffentlichkeit oder externe Mitarbeiter angebotenen Dienste. Diese Art von Vorfällen soll im Rahmen dieser Arbeit jedoch nicht betrachtet werden, da dieses Thema schon Gegenstand einer Reihe von vorhergehenden Arbeiten war. Im Gegensatz dazu liegt der Fokus nun auf den Risiken, die aus dem *Inneren* des Netzwerks drohen: Sei es durch sich automatisch ausbreitende Malware, die beispielsweise über E-Mail oder Drive-By-Downloads ins Netzwerk gelangt ist, durch Rechner, die Teil eines Botnetzes geworden sind und beispielsweise an rechtswidrigen DDoS-Angriffen teilnehmen oder aber auch durch Personen, die sich vor Ort unerlaubterweise Zugang zum Netzwerk oder zu spezifischen Diensten verschafft haben (der sogenannte **Insider Threat**).

Das SVN besteht, wie bereits im vorherigen Kapitel erläutert, aus mannigfaltigen physischen und logischen Teilnetzen, deren Übergänge in andere Netzsegmente durch separate Firewalls und in vielen Fällen auch NIDS abgesichert sind. Es wird prinzipiell ein *Whitelist*-Ansatz verfolgt, bei dem die für den Betrieb des Netzwerkes oder den Zugriff auf erlaubte Dienste notwendigen Zugriffe erlaubt sind und aller sonstiger Datenverkehr verboten ist. Die enorme Größe des SVNs, sowie administrative Freiräume schaffen jedoch auch weitgefasste Whitelist-Einträge, die unter Umständen trotzdem zu unbemerkten unrechtmäßigen Zugriffen führen, die somit vom Firewall-Ansatz nicht mehr abgedeckt werden. In dieser Situation kann der Einsatz von Honeypots mit allen im Grundlagen-Kapitel beschriebenen Vorteilen und Risiken ausgesprochen hilfreich sein, beispielsweise um sich innerhalb eines Teilnetzes ausbreitende Malware zu entdecken, die erst beim Versuch, das Subnetz selbst zu verlassen, von Firewalls aufgespürt worden wäre.

Es ist somit das Ziel dieser Arbeit, eine Honeypot-Architektur zum Einsatz innerhalb des SVNs zu entwickeln, mit der interne Bedrohungen – also Zu- und Angriffe aus dem internen Netz heraus – erkannt, lokalisiert und analysiert werden können. Aus den Gegebenheiten des SVNs resultieren zudem eine Reihe von spezifischen Anforderungen, die im Folgenden katalogisiert und nach technischen und organisatorischen Anforderungen getrennt werden.

3 Technische Anforderungen

Funktionalität Das zu entwerfende System soll der primären Aufgabe gerecht werden, Angriffe – sowohl erfolgreiche als auch gescheiterte – auf oder auch von Rechnersystemen innerhalb des sächsischen Verwaltungsnetzes (SVN) durch die Integration von Honeypots erkennen zu können. Der Fokus liegt dabei explizit auf Übergriffen „aus dem Inneren" und nicht etwa aus dem Internet.

Skalierbarkeit Die komplexe Struktur des SVNs besteht aus vielen, teilweise voneinander unabhängigen Teilnetzen, an deren Übergängen wiederum mit Firewalls strenge Regeln forciert werden. Dies erschwert die Erreichbarkeit von an zentralen Stellen platzierten Honeypot-Systemen. Um außerdem sich primär lokal ausbreitende Malware oder anderweitige Angreifer, die sich nur innerhalb eines Netzsegmentes bewegen, aufzuspüren, stellt eine Integration von Honeypots in jedes einzelne physische und logische Subnetz den Idealfall dar. Diese große Anzahl an Fallen verlangt nach einem gut skalierenden System, das all diese Honeypots gleichzeitig verwalten und die von ihnen gesammelten Daten miteinander in Beziehung setzen kann.

Benachrichtigung bei Vorfällen Die für die IT-Sicherheit zuständigen Ansprechpartner müssen über Vorfälle, die auf den einzelnen Honeypots registriert werden, *zeitnah* informiert werden. Eine Benachrichtigung per E-Mail ist wünschenswert, sollte jedoch nicht die einzige Informationsmöglichkeit sein. Es ist denkbar, zusätzliche Medien mit einzubinden oder an zentraler Stelle eine Auflistung der gesammelten Ereignisse zur genaueren Inspektion zur Verfügung zu stellen.

Containment Es muss sichergestellt werden, dass Angriffe auf die Honeypots keine weiteren Angriffe auf andere Hosts im Netzwerk nach sich ziehen – unabhängig davon, ob diese bereits infiziert sind oder nicht. Unter Berücksichtigung der Tatsache, dass kein separates Honeynet geschaffen werden soll, sondern stattdessen die Honeypot-Systeme direkt in die Produktivnetzwerke integriert werden, verlangt diese Anforderung autonome, hinreichend abgesicherte Honeypot-Systeme.

Datenauswertung Die gesammelten Daten von einer großen Zahl von Honeypots sollen miteinander in Korrelation gesetzt werden können, um beispielsweise die Ausbreitung von Malware innerhalb des SVNs oder die Bewegungen von Angreifern innerhalb des Gesamtnetzes nachvollziehen zu können. Entsprechende automatisch generierte Statistiken auf Basis der gesammelten Daten helfen bei diesem Vorgehen. Zusätzlich erleichtert eine graphische Aufbereitung der Statistiken einen schnellen Überblick über vergangene Ereignisse und den aktuellen Zustand des Honeynets.

Verschlüsselung Die Kommunikation zwischen den Honeynet-Komponenten soll verschlüsselt erfolgen, um zu verhindern, dass im Falle eines erfolgreichen Einbruchs in das Netzwerk Informationen über den Honeypot-Betrieb oder die vom Angreifer gesammelten Daten gestohlen werden. Dies betrifft insbesondere die Berichterstattung und das Abfragen der Honeypots nach aufgezeichneten Ereignisdaten.

Günstige Beschaffung Die Beschaffung und der Betrieb des Honeynets sollen nur einen geringen finanziellen Aufwand mit sich bringen. Dies beinhaltet preiswerte, leicht ersetzbare Hardware – schwer zu beschaffende Spezialkomponenten gilt es zu vermeiden. Wenn bereits vorhandene Infrastruktur von den Honeypots mitgenutzt werden kann, ist dies einem Neukauf vorzuziehen – so soll die zusätzliche Anschaffung von beispielsweise nur für die Honeypots gedachter Switching-Hardware vermieden werden. Ebenso soll die Software nach Möglichkeit keine zusätzlichen Kosten verursachen: Open-Source-Software ist folglich proprietärer vorzuziehen.

Standardkonformität Eine Integration in das bestehende Netzwerk soll so einfach wie möglich gelingen, weshalb bereits etablierte Standards sowohl auf Hardware-, als auch Softwareseite berücksichtigt werden sollen. Beispielsweise kommt innerhalb des SVNs *Ethernet* als Standard-Netzwerktechnologie zum Einsatz. Gleiche Anforderungen gelten auch für die Stromversorgung der Geräte, die optional auch via *Power over Ethernet* (PoE) erfolgen kann. Softwareseitig sollen ebenfalls die bereits unterstützten Dienste und Protokolle mitbenutzt werden, um Services nicht unnötig replizieren oder ergänzen zu müssen. Dies betrifft beispielsweise die Verwendung von *NTP*, *HTTP(S)* und *SSH*.

Transparenz Sowohl die Integration als auch der Betrieb der Honeypots sollen keinerlei spürbare Beeinträchtigungen auf den normalen Betrieb des SVNs haben. Alle Dienste müssen weiterhin ohne Einschränkungen erreichbar sein. Ein Ausfall von einem oder mehreren Honeypots soll ebenfalls lediglich für das zuständige IT-Sicherheitspersonal sichtbar und relevant sein und keine Auswirkungen auf das restliche Netzwerk haben. Ebenfalls ist gefordert, dass die Honeypots selbst nur absolut notwendigen Datenverkehr erzeugen, um die Netzwerk-Infrastruktur nicht zusätzlich zu belasten.

Aktualisierbarkeit Honeypot-Software wird stetig weiterentwickelt, um auf neue Bedrohungen reagieren zu können. Analog dazu soll auch die im Rahmen dieser Arbeit etablierte Honeypot-Infrastruktur eine Möglichkeit vorsehen, die einzelnen Bestandteile mit möglichst geringem Aufwand aktualisieren zu können. Hierzu zählt die eingesetzte Honeypot-Software selbst, aber auch das darunterliegende Betriebssystem und alle weiteren für den Betrieb des Honeypots erforderlichen Komponenten.

Umgang mit *False Positives* Je nach Konfiguration weist nicht jede von Honeypots generierte Meldung zwangsweise auf einen tatsächlichen Angriff hin. Bei genauerer Betrachtung der gesammelten Daten können sich derartige Ereignisse als *False Positives* herausstellen. Das zu etablierende System sollte bei der Klassifizierung der Daten durch eine übersichtliche Repräsentation der gesammelten Informationen helfen und ebenso die Möglichkeit bieten, uninteressante Ereignisse herauszufiltern oder die betreffenden, derartige Meldungen erzeugenden Mechanismen zu deaktivieren.

4 Organisatorische Anforderungen

Benutzerfreundlichkeit Eine Hürde beim Betrieb von Honeypots in Unternehmen ist die notwendige Expertise zu Installation, Betrieb und Interpretation der gesammelten Daten. Um das im Rahmen dieser Arbeit entwickelte Honeynet auch für hinsichtlich IT-Security ungeschultes Personal nutzbar zu machen, sollen alle Funktionen der Implementierung möglichst leicht verständlich oder sogar selbsterklärend sein. Die (webbasierte – Details hierzu im nachfolgenden Kapitel) Benutzeroberfläche zur Installation und Wartung der Systeme soll sich an etablierte *W3C*-Standards[14] halten und soweit realisierbar Technologien zur Unterstützung der Benutzerfreundlichkeit durch Animationen, Hilfstexte und graphische Visualisierungen unterstützen. Die Anwendung von Techniken des *Responsive Web Design* zur Verbesserung der UI-Bedienung auf mobilen Endgeräten ist erstrebenswert.

Leichte Installation Das Hinzufügen weiterer Honeypots zum Netzwerk soll dem zuständigen Administrator möglichst leicht von der Hand gehen. Nach Beschaffung der entsprechenden Hardware hat die bestehende Management-Infrastruktur bei der Installation und Konfiguration der Honeypot-Software sowie der Anpassung des Betriebssystems zu assistieren.

Einfache Wartbarkeit Analog zur Installation soll das System auch bei der Wartung der bestehenden Honeypots behilflich sein. Dies umfasst die Remote-Konfiguration sowie das Hinzufügen, Entfernen und Aktualisieren der Honeypots. Diese Operationen sollen auf komfortable Weise über eine zentrale Benutzeroberfläche durchgeführt werden können.

Integration Die Integration der Honeypots in das Netzwerk sollte möglichst geringe Anforderungen nach sich ziehen und beispielsweise nur ein Minimum an zusätzlichem Konfigurationsaufwand der bestehenden Infrastruktur erfordern. Dies umfasst u.a. die Re-Konfiguration von Firewalls, Switches oder DHCP-Servern.

Vertraulichkeit Da im sächsischen Verwaltungsnetzwerk viele vertrauliche Daten bewegt werden, sollen die Honeypots ausschließlich direkt an sie selbst gerichtete Pakete aufzeichnen und auswerten. Multicasts und Broadcasts können unter Umständen vertrauliche Daten beinhalten oder zumindest Rückschlüsse auf die Netzstruktur erlauben und sind folglich zu ignorieren.

[14] World Wide Web Consortium

Teil IV

Konzeption

Der Konzeption eines auf das SVN zugeschnittenen Honey-Netzes ging die wichtige Frage voraus, welcher Natur der ankommende Datenverkehr an einzelnen Hosts in verschiedenen Teilnetzen des SVNs ist. Die Fragestellung umfasste sowohl die Menge als auch die Art der ankommenden Daten. Die Beantwortung der Problemstellung war für die spätere Auswahl einer passenden Hardwareplattform und verschiedene softwaretechnische Fragen, wie beispielsweise die zentrale oder dezentrale Struktur des zu entwerfenden Netzes, entscheidend. Im Folgenden werden diese Trafficanalyse, die aus dieser resultierenden Schlussfolgerungen und der Architekturentwurf der zu etablierenden Infrastruktur vorgestellt.

5 Trafficanalyse

Dem Entwurf einer auf die Gegebenheiten der vorliegenden IT-Infrastruktur zugeschnittenen Architektur ging eine Analyse des innerhalb der Netzsegmente zu erwartenden Datenverkehrs voraus. Dies war insbesondere für die Auswahl der innerhalb des späteren Honey-Netzes anzubietenden simulierten Dienste und geeigneter Honeypot-Software von Bedeutung. Um also ein möglichst umfassendes Gesamtbild zu erhalten, fand die Analyse des Datenverkehrs direkt in ausgewählten Teilnetzen des SVNs statt. Es wurden Sensoren integriert, die jeglichen direkt an sie adressierten Datenverkehr aufzeichneten, ohne auf diesen zu antworten. Multi- und Broadcasts wurden hingegen gezielt ignoriert, da derartige Pakete potentiell vertrauliche Daten enthalten oder zumindest Rückschlüsse auf Details der Netzinfrastruktur erlauben, ohne einen Mehrwert im Sinne der Aufgabenstellung dieser Arbeit zu bieten. Nach einem mehrwöchigen Testzeitraum wurden die von den Sensoren aufgezeichneten Daten analysiert. Auf Basis dieser Informationen und unter Berücksichtigung der zuvor genannten Anforderungen resultierte schließlich ein erster Entwurf des späteren Honey-Netzes.

Der Entwurf des Systems für die Trafficanalyse begann zunächst mit der Auswahl und Beschaffung geeigneter Hardware, gefolgt von einer minimalen, speziell auf die Aufgabe zugeschnittenen Softwareplattform. Nach deren Konfiguration und einer Reihe spezifischer Tests schloss sich zuletzt die Integration in das Produktivnetzwerk an. Als Hardwarebasis für die Sensoren wurden Boards vom Typ **BeagleBone Black**[15] ausgewählt, da diese an der Fakultät Informatik in ausreichender Anzahl vorhanden waren und sich durch ihre geringe Größe, den niedrigen Stromverbrauch und der großen Softwareauswahl als preiswerte Testplattform anboten. Die von der Firma *Texas Instruments* produzierten „Ein-Platinen-Computer“ besitzen einen auf der ARM-Architektur basierenden *Cortex-A8*-Prozessor mit einem Takt von 1 GHz, 512 MB DDR3-Arbeitsspeicher und sind unter anderem mit Anschlüssen für USB 2.0 und Ethernet (10/100 MBit) ausgestattet. Als Massenspeicher können sowohl ein auf dem Board integrierter eMMC-Chip mit einer Größe von 2 GB oder eine über einen entsprechenden Slot anschließbare microSD-Karte genutzt werden. Die Leistungsfähigkeit erschien für den Einsatz innerhalb des SVNs mehr als ausreichend, lediglich eine Gigabit-Ethernet-Schnittstelle wäre wünschenswert gewesen und wurde eingangs als potentieller Flaschenhals des Projektes gehandelt, was sich in der Folge jedoch nicht bestätigte. Der

[15] `http://beagleboard.org/black` (abgerufen im September 2014)

kleine Formfaktor von 86x53 Millimetern erleichterte die Integration der Sensoren innerhalb der Serverräume erheblich. Lediglich die Stromversorgung der Systeme bereitete Probleme: Die Boards tolerierten eine flexible Einspeisung wahlweise über USB oder ein separates Netzteil, in den Serverschränken waren jedoch standardmäßig nur Kaltgerätestecker vorhanden. Mit separat eingekauften Adaptersteckern konnte dieses Problem jedoch zeitnah gelöst werden. Alternativ existieren für das BeagleBone-Board spezielle Adapterkabel, um die Versorgung des Rechners via *Power over Ethernet* (PoE) zu ermöglichen. Dies erfordert jedoch auch kompatible PoE-Switches, die nicht in allen Netzsegmenten vorhanden waren.

Auf den BeagleBones war die freie Linux-Distribution *Angström Linux*[16] vorinstalliert, die auf Einsteigerfreundlichkeit getrimmt war und somit bereits eine große Anzahl von Diensten – darunter Web- und SSH-Server – vorinstalliert hatte. Da diese Distribution zum Zeitpunkt dieser Arbeit bereits nicht mehr offiziell vom BeagleBone-Projekt unterstützt wurde und ein „sauberes“ minimales System eine bessere Grundlage für das Honeynet-Projekt ist, wurden alle Systeme mit einem speziell für das BeagleBone Black angepassten *Debian GNU/Linux*[17] ausgestattet. Die boardspezifischen Modifikationen umfassen im Wesentlichen eine Reihe von Bootskripten, die das Vervielfältigen des Systems auf den internen Speicher oder eine SD-Karte ermöglichen und eine Reihe USB-spezifischer Dienste starten. Somit ist es möglich, über den USB-Port eine zweite Netzwerkverbindung mit dem BeagleBone herzustellen. Der Softwarestack wurde analog zur Idee der Installation leichtgewichtiger Sensoren minimal gehalten. Die bereits vorinstallierten Dienste wurden vom Ethernet-Interface auf das USB-Interface verschoben, um eine Wartung des Systems nur mit physischem Zugang zum Board zu ermöglichen. Nachdem sichergestellt war, dass auf dem Ethernet-Interface keinerlei Dienste offeriert werden, wurde abschließend die Software *daemonlogger*[18] als Monitoringlösung installiert. Es handelt sich hierbei um einen Daemon, der ein Netzwerkinterface abhört und sämtlichen ankommenden Datenverkehr im *pcap*-Format abspeichert. Die Software agiert rein passiv und erlaubt es zudem, *Berkeley Packet Filter* (BPF) [4] zur genaueren Spezifikation der zu sammelnden Daten einzusetzen. Diese Möglichkeit wurde genutzt, um die Beschränkung des Monitorings auf Unicast-Traffic zu realisieren. Zuletzt musste noch sichergestellt werden, dass im Falle einer großen Trafficflut ausreichend Speicherplatz für die Aktivitäten des Betriebssystems selbst verfügbar bleibt – die zu erwartenden Datenmengen waren eine unbekannte Größe. Auch hierfür bietet daemonlogger einen entsprechenden Schalter, der die Maximalgröße der `pcap`-Datei spezifiziert.

Der resultierende Befehl zum Start des Prozesses sah schlussendlich folgendermaßen aus:

```
daemonlogger -i eth0 -l /var/log/daemonlogger -n local -m 3
             -s 1073741824 -d not multicast and not broadcast
             and host $IP and not arp
```

Insgesamt wurden über einen Zeitraum von sieben Wochen sechs Sensoren in verschiedenen Teilnetzen platziert. Eine Übersicht gibt Tabelle 1. Da die BeagleBone-Boards keine integrierte Hardwareuhr besaßen, wurden auf den Sensoren jeweils individuelle NTP-Clients für die Zeitsynchronisation konfiguriert. Die entsprechenden Server waren bereits innerhalb des SVNs vorhanden. Ohne korrekt konfiguriertes NTP wären die Timestamps der gesammelten Daten unbrauchbar und eine exakte Zuordnung von eventuellen Vorfällen nur schwer möglich gewesen.

Die Platzierung der Sensoren verlief abgesehen von der Gewährleistung der Stromversorgung durch entsprechende Adapterkabel reibungslos. Vom zuvor beschriebenen auf Debian-Linux basierenden System

[16] `http://angstrom-distribution.org` (abgerufen im September 2014)
[17] `http://debian.org` (abgerufen im September 2014)
[18] `http://sourceforge.net/projects/daemonlogger/` (abgerufen im September 2014)

Sensor	Teilnetz
bb2	Housing-DMZ, privat
bb3	Housing-DMZ, öffentlich
bb4	Zentrales Serversegment
bb5	VOID-Testlokation
bb6	Internet-DMZ, privates Adresssegment
bb7	Büronetzwerk im Verwaltungsnetz

Tabelle 1: Aufstellungsorte der Sensoren

wurde ein Abbild erstellt, das auf die SD-Karten aller Sensoren geklont wurde. Anschließend waren nur noch minimale Anpassungen an die Gegebenheiten des jeweiligen Subnetzes erforderlich: Netzwerkkonfiguration (DHCP/statisch), Zeitsynchronisation (NTP) und die Auswahl eines eindeutigen Hostnamens.

Sensor	Pakete	ICMP	TCP
bb2	9	9	0
bb3	358	358	0
bb4	28	10	18
bb5	57	45	12
bb6	43	43	0
bb7	347	288	59

Tabelle 2: Akkumulierte Sensordaten

5.0.1 Auswertung

Am Ende des Testzeitraums wurden die vom daemonlogger auf den Sensoren erzeugten `pcap`-Dateien der Reihe nach mit dem GUI-basierten Tool **Wireshark**[19] ausgewertet. Eine Zusammenfassung der aufgezeichneten Netzwerkpakete listet Tabelle 2 auf. Es resultierten eine Reihe von Beobachtungen:

- Der in einem öffentlichen Netz platzierte Sensor `bb3` besaß eine über das Internet erreichbare IP-Adresse, war jedoch aufgrund der vorgeschalteten Firewalls ausschließlich via ICMP zu erreichen. Folglich empfing dieses Gerät im Vergleich die meisten Daten – allerdings ausschließlich ICMP-Ping-Requests (Echo) von Hosts aus aller Welt, jedoch nicht von benachbarten Knoten innerhalb des SVNs. Diese Pakete sind nicht als Angriff zu werten, somit hatte der Sensor im Wesentlichen eine Kontrollfunktion: Er zeigte auf, dass die eingesetzte Hard- und Softwarekombination dazu geeignet war, über einen längeren Zeitraum hinweg ohne menschliche Intervention zuverlässig Daten aufzuzeichnen.
- Die Sensoren `bb2`, `bb4` und `bb5` verzeichneten keinerlei verdächtigen Datenverkehr. In den Logdateien fanden sich lediglich die kurz nach der Installation zur Kontrolle der Funktionalität durchgeführten Ping- und SSH-Anfragen sowie die (erwünschte) Kommunikation mit dem NTP-Server. Die Aufzeichnung der letzteren hätte mit einer Anpassung des daemonlogger-Filters vermieden werden können, was angesichts der geringen Menge der insgesamt registrierten Ereignisse jedoch keinen Mehrwert geboten hätte.

[19] `http://wireshark.org` (abgerufen im September 2014)

- Die Sensoren bb5 und bb7 registrierten als verdächtig einzustufende Ereignisse. Gerät 5 empfing sechs aufeinanderfolgende TCP-Anfragen an Port 80, auf dem normalerweise Web-Server laufen. Die Pakete wurden in einem Zeitfenster von unter einer Minute morgens um 05:44 Uhr empfangen. Diese Informationen sowie die Quell-IP-Adresse des „Angriffs“ wurden an die zuständige Administration zur Untersuchung weitergegeben. Sensor 7 hingegen hatte innerhalb des Testzeitraumes Aktivität auf Port 22 (SSH) aufgezeichnet, was ebenfalls zu einem späteren Zeitpunkt weiter analysiert wurde.

5.0.2 Schlussfolgerungen

Die Trafficanalyse hat gezeigt, dass die grundsätzliche Idee, Sensoren mit Hilfe einer preiswerten Hardwareplattform und unter dem Einsatz von frei verfügbarer Open-Source-Software zu erstellen und an interessanten Stellen im Netzwerk zu platzieren, zu den erwarteten Ergebnissen führt: Jeglicher ankommender Datenverkehr kann aufgezeichnet und zu einem späteren Zeitpunkt ausgewertet werden. Die im pcap-Format gespeicherten Daten enthalten alle nötigen Informationen, um auf möglicherweise als Angriff eingestufte Ereignisse reagieren zu können: Darunter Quell-IP- und -MAC-Adresse der Pakete, ein Zeitstempel sowie der Zielport des Datenstroms, was zur Identifikation der von einem Angreifer erwarteten Netzwerkdienste und -protokolle herangezogen werden kann. Da jedes Paket vollständig und unverändert gespeichert wird, ist außer den Headerinformationen auch der Datenbereich vorhanden und kann potentiell genutzt werden, um Details über einen geplanten Angriff zu gewinnen – beispielsweise, welchen Exploit ein Eindringling auszunutzen versucht. Bei der Analyse hat sich zudem herausgestellt, dass Datenverkehr primär auf den „well known“ Ports zu erwarten ist, darunter die für die Dienste SSH (Port 22/TCP) und HTTP (Port 80/TCP).

Weiterhin hat das Experiment verdeutlicht, dass das zu erwartende Trafficaufkommen mit nur 492 Paketen binnen des siebenwöchigen Testzeitraumes sehr gering ausfällt und sich die Befürchtung, dass die 100-MBit-Ethernet-Schnittstellen der Sensoren innerhalb eines ausgelasteten Netzabschnittes nicht ausreichend seien, nicht bestätigt. Die Anschaffung von kostenintensiverer Hardware erschien folglich nicht sinnvoll. Einzig die Gewährleistung der Stromversorgung verursachte Komplikationen, da innerhalb der Serverschränke lediglich mit den BeagleBone-Boards inkompatible Anschlüsse für Kaltgerätestecker vorhanden waren. Bei Beibehaltung dieser Hardwareplattform ist die Beschaffung kompatibler Adapterstecker in Erwägung zu ziehen.

6 Architektur

Aus den gewonnenen Erkenntnissen der zuvor beschriebenen Trafficanalyse galt es nun, ein unter Berücksichtigung der in Teil III genannten Anforderungen möglichst endanwenderfreundliches und skalierbares Honey-Netz zu entwerfen und prototypisch zu implementieren. Es erschien zunächst sinnvoll, den Blick auf vorhandene Literatur in diesem Bereich zu richten und bereits getestete Ansätze als potentielle Grundlage des zu entwickelnden Systems in Betracht zu ziehen.

6.1 Verwandte Arbeiten

Das Sächsische Verwaltungsnetz unterscheidet sich als Basisinfrastruktur insofern von „klassischen“ Honey-Netzen, als dass hierfür kein separates Subnetz vorgesehen ist, das sich ausschließlich aus Honeypot-Knoten zusammensetzt und in welches fragwürdiger Datenverkehr zur Analyse weitergeleitet wird. Dieser Ansatz erlangte insbesondere durch die von Niels Provos entwickelte Software **honeyd** Verbreitung [32], die die Realisierung eines solchen abgeschlossenen Netzes wahlweise physischer oder virtueller Natur im Vergleich zu früheren Tools deutlich vereinfacht hat. Die Herausforderung eines derartigen Setups liegt jedoch in der Einrichtung der dafür notwendigen Routing-Infrastruktur. Fragwürdiger Datenverkehr muss entweder bereits beim Routing von „gutmütigem“ unterschieden werden – was beispielsweise mit dem Einsatz eines zusätzlichen IDS möglich wäre – oder aber es werden alle ankommenden Pakete geklont und zusätzlich zu ihrem designierten Ziel an Hosts im Honey-Netz verschickt, um im Falle eines versuchten Angriffes die Kommunikation automatisch mit entsprechenden Filterregeln zu unterbinden.

Der Haupteinsatzzweck von honeyd ist die Einrichtung einer großen Zahl vordefinierter virtueller Honeypot-Knoten, die selbst auf nur wenigen physischen Hosts laufen. Die Software selbst kann mit Hilfe einfacher Skripte als Low-Interaction-Honeypot eingesetzt werden und so beispielsweise für mehrere hundert IP-Adressen gleichzeitig einen SSH-Service simulieren, unterstützt jedoch auch die Integration von separaten High-Interaction-Honeypots. Somit ist es beispielsweise möglich, Honeypot-Hosts zu simulieren, deren Dienste gleichzeitig von Low- und High-Interaction-Honeypots bereitgestellt werden.

Ein mit honeyd betriebenes Honey-Netz besitzt den Nachteil, dass alle Knoten vor dem eigentlichen Betrieb definiert werden müssen und anschließend die physischen Systeme, auf denen die Low- und insbesondere die High-Interaction-Honeypots laufen solange „idle“ sind, wie keine verdächtigen Verbindungen zu ihnen aufgebaut werden. Speziell die für den Betrieb notwendigen virtuellen Maschinen für jeden einzelnen High-Interaction-Honeypot belegen in dieser Zeit Ressourcen, die für andere Aufgaben vor allem aus Sicherheitsgründen nicht mehr zur Verfügung stehen. Es ist somit wünschenswert, die benötigten Ressourcen zu minimieren. Einen solchen Ansatz verfolgt das Projekt **Honeydoop** von Kulkarni et al. [25]: Honeypots werden dynamisch generiert und der zum Zeitpunkt eines Angriffs vorliegenden Bedrohungslage automatisch angepasst. Im Falle eines verdächtigen eintreffenden Datenstromes wird dessen Zielknoten im regulären Produktivnetzwerk ausfindig gemacht und eine neue virtuelle Honeypot-Instanz gestartet, deren Konfiguration so angepasst wird, dass sie dem angegriffenen Host möglichst ähnlich sieht. Somit sind vergleichsweise weniger physische Ressourcen zum Betrieb des Honeypot-Netzwerkes nötig. Die Honeypot-Instanzen laufen zudem an einer zentralen Stelle innerhalb des Netzwerks – typischerweise innerhalb eines abgeschotteten Subnetzes, dessen ausgehender Datenverkehr Containment-Maßnahmen unterliegt. Diese Architektur erleichtert zwar die Wartbarkeit und das

Management des Gesamtsystems, ist jedoch auch mit organisatorischen Nachteilen verbunden: Honeydoop benötigt als *Intrusion Detector* mehrere im Netzwerk verteilte IDS und führt nach dem Starten einer neuen Honeypot-Instanz eine Rekonfiguration der entsprechenden Netzwerkkomponenten durch um sicherzustellen, dass der gewünschte Datenstrom korrekt umgeleitet wird. Ein derartiges Vorgehen wäre innerhalb des SVNs jedoch nicht realisierbar, da es erhebliche Eingriffe in die Struktur und eine Umkonfiguration einer großen Zahl von Komponenten im Netzwerk nach sich ziehen würde. Die in der Anforderungsanalyse geforderte Integrierbarkeit wäre mit einer solchen Lösung somit nicht gegeben.

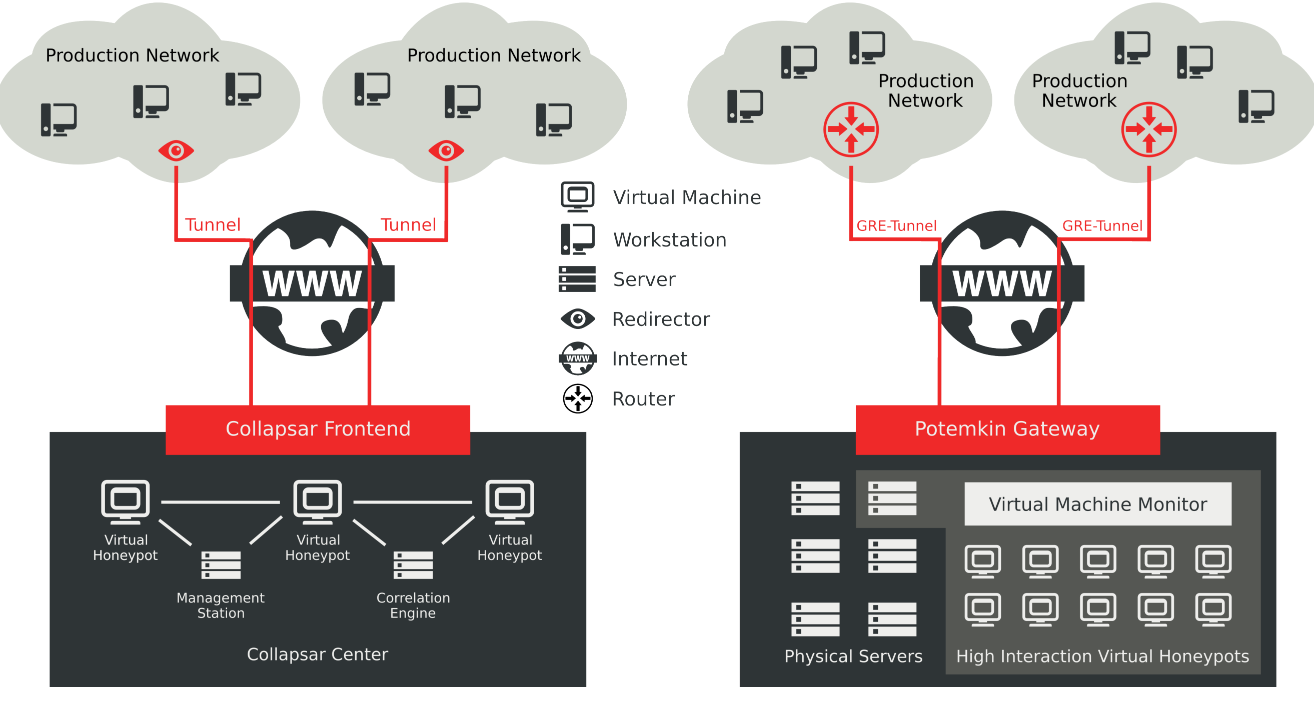


Abbildung 2: Die hybriden Honeynet-Architekturen Collapsar und Potemkin [32]

Einen alternativen Ansatz zur Realisierung von Honey-Netzen stellen die Forschungsprojekte **Collapsar** [24] und **Potemkin** [42] vor, deren graphische Gegenüberstellung in Abbildung 2 gezeigt wird. Ersteres ist speziell auf den verteilten Einsatz in mehreren unterschiedlichen Netzsegmenten zugeschnitten und wird von den Entwicklern Xuxian Jiang und Dongyan Xu als *„a virtual-machine-based architecture for network attack detection"* beschrieben. Collapsar sieht hierfür sogenannte *Redirectors* vor, die in verschiedenen Netzwerken platziert werden und über Tunneltechnologien wie *GRE*[20] oder *VPN* mit einem als Frontend agierenden *Collapsar Center* verbunden sind. Dieses setzt sich wiederum aus verschiedenen Modulen unter anderem zur Verwaltung der als High-Interaction-Honeypots dienenden virtuellen Maschinen, zur Analyse und Korrelation der gewonnenen Daten und zur Gewährleistung von Containment-Maßnahmen zusammen. Zusätzlich beinhaltet dieses Frontend auch den Endpunkt für die gewählte Tunneltechnologie und muss somit für die Redirectors erreichbar sein. Bei den Redirectors handelt es sich um Rechnersysteme, die mit einem vom Collapsar-Projekt zurechtgeschnittenen minimalen Betriebssystem laufen und deren einzige Aufgabe es ist, einen Tunnel zum *Collapsar Center* aufzubauen und allen ankommenden Datenverkehr dahin weiterzuleiten sowie analog die vom Frontend empfangenen Pakete zum potentiellen Angreifer zurückzusenden. Die Honeypots laufen folglich alle zentralisiert und virtualisiert, was wiederum Management und die Auswertung der gesammelten Daten vereinfacht.

[20]Generic Routing Encapsulation

Die Collapsar-Architektur hat im Vergleich zum zuvor beschriebenen Honeydoop-System den Nachteil, dass der Datenverkehr der Produktivsysteme nicht durch ein IDS analysiert und im Bedarfsfall an Honeypots weitergeleitet wird. Angriffe auf diese Hosts bleiben daher für die Collapsar-Honeypots unsichtbar. Vielmehr erhält jeder Redirector eine eigenständige IP-Adresse im Teilnetz und soll mit in diesem Fall statisch konfigurierten Diensten für Angreifer von den benachbarten Systemen nicht zu unterscheiden sein. Gleichzeitig erwächst daraus allerdings der Vorteil, dass Collapsar in eine bestehende IT-Infrastruktur verhältnismäßig transparent integriert werden kann: Die Redirectors besitzen nur geringe Hardwareanforderungen und es muss seitens der Routing- und NAT-Konfiguration lediglich sichergestellt werden, dass das Collapsar-Frontend erreichbar ist. Nachdem ein VPN- oder GRE-Tunnel aufgebaut wurde, erfolgt jeglicher weiterer Datenaustausch mit dem Frontend nur noch darüber. Es ist sogar möglich und von den Entwicklern auch gewollt, die Redirectors weltweit verteilt in verschiedensten Netzen zu betreiben und über das Internet mit dem Collapsar Center zu verbinden. Dies kann beispielsweise für ein internationales Unternehmen mit vielen Standorten von Vorteil sein. Es muss jedoch angemerkt werden, dass in einem solchen Fall die Honeypots von aufmerksamen Angreifern bereits bei einem Scan identifiziert werden könnten, indem die Antwortzeiten der verschiedenen Hosts miteinander verglichen werden. Da an den Redirectors eintreffende Pakete zunächst zum Frontend geschickt, verarbeitet und wieder zurückgesendet werden müssen, ist bei diesen Knoten eine deutliche Verzögerung erkennbar.

Das an der University of California von Vrable et al. entwickelte hybride Honey-Netz **Potemkin** [42] erweitert den eben beschriebenen Ansatz analog zu Honeydoop, um ein hochskalierbares Gesamtsystem zum parallelen Betrieb von tausenden von Honeypots zu erhalten. Dieses Ziel wird durch das dynamische, vollautomatische Anlegen von virtuellen Honeypot-Instanzen erreicht, sobald Datenverkehr eintrifft. Das System ist auf die möglichst effiziente Ausnutzung der Ressourcen optimiert, indem beispielsweise inaktive virtuelle Hosts so spät wie möglich erstellt und schon nach kurzer Zeit wieder beendet werden („*late binding*“). Die Kommunikation zwischen dem Gateway und den Honeypots findet außerdem direkt via MAC-Adressen statt, um die IP-Adressumsetzung einzusparen. Jenes Gateway ist wiederum auch für die Einhaltung einer *Containment Policy* verantwortlich, um Angriffe auf weitere Hosts von übernommenen Honeypots aus zu vermeiden. Potemkin hat hierzu mehrere Vorgehensweisen zur Auswahl, darunter auch das *Internal Reflect* genannte Verfahren, bei dem Pakete, die für andere Knoten (beispielsweise im Internet) bestimmt sind, wieder zurück in die sogenannte *Honeyfarm* an wiederum nur zu diesem Zweck erstellte weitere Honeypots umgeleitet werden. Somit ist es möglich, die Verbreitungscharakteristika von Malware nachzuvollziehen. Strukturell kommen im Gegensatz zu Collapsar keine separaten Redirector-Knoten in den verschiedenen Netzen zum Einsatz, sondern es ist eine Umkonfiguration der zuständigen Router erforderlich, um für die von den Honeypots belegten IP-Adressbereiche GRE-Tunnel zum Gateway aufzubauen.

Sowohl Collapsar als auch Potemkin sind lediglich Forschungsprototypen und wurden nie für die Allgemeinheit veröffentlicht [16], können also für das im Rahmen dieser Arbeit zu entwerfende Honey-Netz lediglich als architekturelle Referenz dienen. Erwähnt werden sollte an dieser Stelle jedoch noch das von Kees Trippelvitz entwickelte **SURFcert IDS** [39], welches vom Projektbetreuer selbst als *„an open source Distributed Intrusion Detection System based on passive sensors“* beschrieben wird. Es handelt sich um eine dem Collapsar-System sehr ähnliche Implementierung, bei der an verschiedenen Stellen im Netzwerk platzierte Sensoren über *OpenVPN* mit einem *Tunnel Server* verbunden sind, auf dem mehrere Low-Interaction-Honeypots für alle Sensoren gemeinsame Honeypot-Dienste anbieten. Die gewonnenen Daten werden anschließend an einen *Logging Server* gesendet, der die Informationen in einer Datenbank speichert und über ein umfangreiches Webinterface zugänglich macht. Im Vergleich zu den zuvor beschriebenen Projekten skaliert dieses System deutlich schlechter, da lediglich ein einziger zentraler Host

für die Beantwortung der von den Sensoren gesammelten Anfragen zur Verfügung steht. Leider wurde das Projekt seit 2011 nicht mehr weiterentwickelt, bietet jedoch eine interessante potentielle Grundlage für ein Honey-Netz innerhalb des SVNs, da eine transparente Integration leicht möglich ist. Eine Evaluation des SURFcert-Systems wurde zudem schon in einer dieser Arbeit vorausgehenden Belegarbeit durchgeführt, bei der der Datenbankserver und die GUI zur Auswertung von gesammelten Daten von 50 weltweit verteilten Sensoren eingesetzt wurden. Als Ersatz für den aus technischen Gründen nicht nutzbaren VPN-Tunnel wurde für die Sensoren eine eigens entwickelte Lösung auf Basis von *User Mode Linux* eingesetzt [16]. Bei diesem Experiment hat sich in Bezug auf den Architekturentwurf eines Honey-Netzes herausgestellt, dass das graphische Webinterface das Interpretieren der Informationen deutlich erleichtert, im Falle des SURFcert IDS jedoch viele darüber hinausgehenden Funktionen zur Verwaltung des Sensornetzwerks nicht zur Verfügung stellt und somit hinsichtlich der Wartbarkeit, Benutzerfreundlichkeit und Aktualisierbarkeit nicht im Sinne der im SVN gestellten Anforderungen arbeitet.

6.2 Netzwerktopologie

Nach ausführlicher Evaluation der existierenden Ansätze aus dem vorherigen Abschnitt und unter Berücksichtigung der in Teil III beschriebenen spezifischen Anforderungen wurde ein Collapsar-ähnliches *Sensornetzwerk* als die geeignetste architektonische Lösung ausgewählt. Das sächsische Verwaltungsnetz besteht wie bereits beschrieben aus vielen Teilnetzen, an deren Übergängen ausgesprochen restriktive Firewalls die Kommunikationsmöglichkeiten auf das Wesentliche beschränken. Es galt somit, eine im SVN nutzbare Gesamtlösung zu finden, die auch mit den Firewall-Policies harmoniert, um die Integration des Systems im laufenden Betrieb so transparent wie möglich zu gestalten. Aus diesem Grund wurde auch die Errichtung eines weiteren separaten Honey-Subnetzes, in das verdächtige Anfragen umgeleitet werden, ausgeschlossen: Dieser Schritt wäre erneut mit organisatorischem Aufwand und dem zusätzlichen Einsatz von IDS zur Detektion verdächtiger Datenströme verbunden. Vielmehr lag der Fokus auf der Idee, möglichst autonome Honeypots innerhalb der jeweiligen Subnetze zu platzieren, die IP-Adressen im Bereich der benachbarten Hosts erhalten und – soweit softwaretechnisch möglich – sich hinsichtlich der angebotenen Dienste an den umliegenden Produktivsystemen orientieren. Somit ist es schließlich nicht möglich, den Netzwerkverkehr der bereits existierenden Hosts zu analysieren und im Verdachtsfalle einzelne Verbindungen zu Honeypots umzuleiten. Dies ist allerdings angesichts der Intention dieser Arbeit nicht zwingend ein Nachteil: Das für das SVN zu entwickelnde Honey-Netz ist nicht zur Detektion von Gefahren von „außen“ zu konzipieren, sondern für **Bedrohungen aus dem Inneren**. Dies bedeutet, dass die Honeypots in der Regel nicht oder nur massiv durch Firewalls reguliert aus dem Internet zu erreichen sind und hingegen versucht werden soll, sich bereits im Netzwerk befindliche Angreifer oder sich autonom ausbreitende Malware zu detektieren. In diesem Punkt unterscheidet sich die vorliegende Architektur von vielen bereits existierenden Ansätzen: Der Fokus des Systems liegt aufgrund der geringen zu erwartenden Paketmenge (wie die Trafficanalyse gezeigt hat) stärker auf der Verwaltung der individuellen Honeypots und der Benutzerfreundlichkeit des Gesamtsystems auch für nicht speziell im Bereich der IT-Sicherheit geschultes Personal. Theoretisch muss jedes an einem der Sensoren auftretende, direkt an den Host selbst gerichtete Paket als Angriff gewertet werden und bedarf einer genaueren Untersuchung. Es gilt, für zuständige Stellen die aufgetretenen Ereignisse in übersichtlicher Form zu *visualisieren* und bei Bedarf zusätzlich automatisch über Kommunikationswege wie beispielsweise E-Mail zu *informieren*.

Die nachfolgende Graphik 3 veranschaulicht die Topologie des geplanten Honey-Netzwerks. Die in den einzelnen Netzwerken platzierten Sensoren – um die Angriffsfläche für Angreifer zu erhöhen, wäre es zudem denkbar, diesen Geräten mehrere IP-Adressen gleichzeitig zuzuweisen – sind mit einem zentralen Management-System verbunden, an das sie alle Informationen über aufgetretene Ereignisse weiterleiten und welches zudem die Verwaltung der Knoten übernimmt. Um die bestehende Firewall-Infrastruktur nicht umkonfigurieren zu müssen, orientiert sich die Architektur zudem an der bestehenden Policy. Da in den meisten Netzwerken Systeme stehen, die mit Rechnern im Internet kommunizieren, ist das Protokoll HTTP, bzw. dessen verschlüsselte Variante *HTTPS*, normalerweise für ausgehende Verbindungen erlaubt. Diesen Umstand macht sich die Honey-Netz-Architektur zunutze, indem der Kommunikationskanal zwischen Sensoren und Managementserver **gerichtet** ist, sodass lediglich sichergestellt werden muss, dass alle Honeypots den Server erreichen können. Das Resultat ist eine *Push-Architektur*: Es liegt im Aufgabenbereich der Sensoren, alle über vorgefallene Ereignisse gewonnenen Informationen an die zentrale Komponente weiterzuleiten. Dieser kann die Sensoren folglich nicht selbstständig kontaktieren. Eine Ausnahme ist aber insbesondere der **Polling** genannte Vorgang, bei dem die Sensoren zusätzlich ein *Pull-Verfahren* nutzen, um in regelmäßigen Abständen sowohl ihren Systemzustand auszusenden, als auch Details über die eigene Konfiguration in Erfahrung zu bringen.

Ein weiterer wichtiger Aspekt ist der Ort, an dem die Honeypot-Software selbst vorgehalten wird. Im Einklang mit den vorgestellten bereits existierenden Arbeiten könnten die Honeypots ebenfalls zentralisiert auf einem einzigen Host (oder einer Gruppe von wenigen Servern) betrieben werden, was Management und Software-Updates vereinfachen würde. Hierfür wäre jedoch zusätzliche Infrastruktur beispielsweise in Form einer VPN-Lösung notwendig, wie es von Collapsar und Potemkin praktiziert wird. Die restriktiven Firewall-Regeln erschweren jedoch die Kommunikation mit einem VPN-Endpunkt stark: Standardports wie 1194/UDP für *OpenVPN* oder 1723/TCP für *PPTP* (GRE-Tunnel) sind in der Regel gesperrt. Ein Ausweg hierfür wäre, die Dienste auf andere, erlaubte Ports wie HTTP(S) umzuleiten, was jedoch unter Umständen von Firewalls, die *Deep Packet Inspection* nutzen, oder internen IDS blockiert werden könnte. Zusätzlich besteht in einem solchen Fall noch das bereits besprochene Risiko, dass Honeypots aufgrund der Latenzunterschiede zwischen den Produktivsystemen und den Sensoren, die alle Daten erst zum Honey-Netz-Server routen müssen, für aufmerksame Eindringlinge identifizierbar wären. Im Falle von automatisierten Angriffen oder sich selbstständig ausbreitender Malware würde dieser Aspekt allerdings keine Rolle spielen. Beide Punkte sprechen jedoch vielmehr für die Alternativlösung, Honeypot-Software direkt **lokal** auf den einzelnen Sensoren zu betreiben. Somit sind die Honeypots hinsichtlich der Latenz nicht von den umgebenden Rechnern zu unterscheiden und es ist möglich, HTTP(S) als Kommunikationsprotokoll zum Senden von Ereignissen und Abfragen von Konfigurationsdaten einzusetzen. Aus dieser Entscheidung resultiert jedoch auch ein starker Fokus auf dem Management-System des Honey-Netzes, das zum Ausgleich entsprechend umfangreichere Funktionen wie beispielsweise die Softwareaktualisierung von Sensoren und die Konfiguration der Honeypots selbst anbieten sollte.

Wenn neue Teilnetze zum Honey-Netz hinzugefügt werden sollen, kann dies durch Installation eines zusätzlichen Sensors innerhalb des Netzes und der Registrierung desselben beim Server erfolgen. **Skalierbarkeit** innerhalb eines einzelnen Netzes ist wiederum entweder durch das Einfügen zusätzlicher Sensoren oder aber das Zuweisen weiterer IP-Adressen zu einem bereits existierenden Honeypot möglich. Da alle Informationen über eventuelle Vorfälle an zentraler Stelle gesammelt werden, übernimmt diese auch die damit verbundenen Verantwortlichkeiten der Datenauswertung und der automatischen Benachrichtigung. All diese Funktionen unterliegen zudem der Bedingung, möglichst benutzerfreundlich und zentral erreichbar zu sein. Geeignete **Containment-Maßnahmen** sind entsprechend der eben beschriebenen Architektur ebenfalls lokal auf den jeweiligen Sensoren zu treffen.

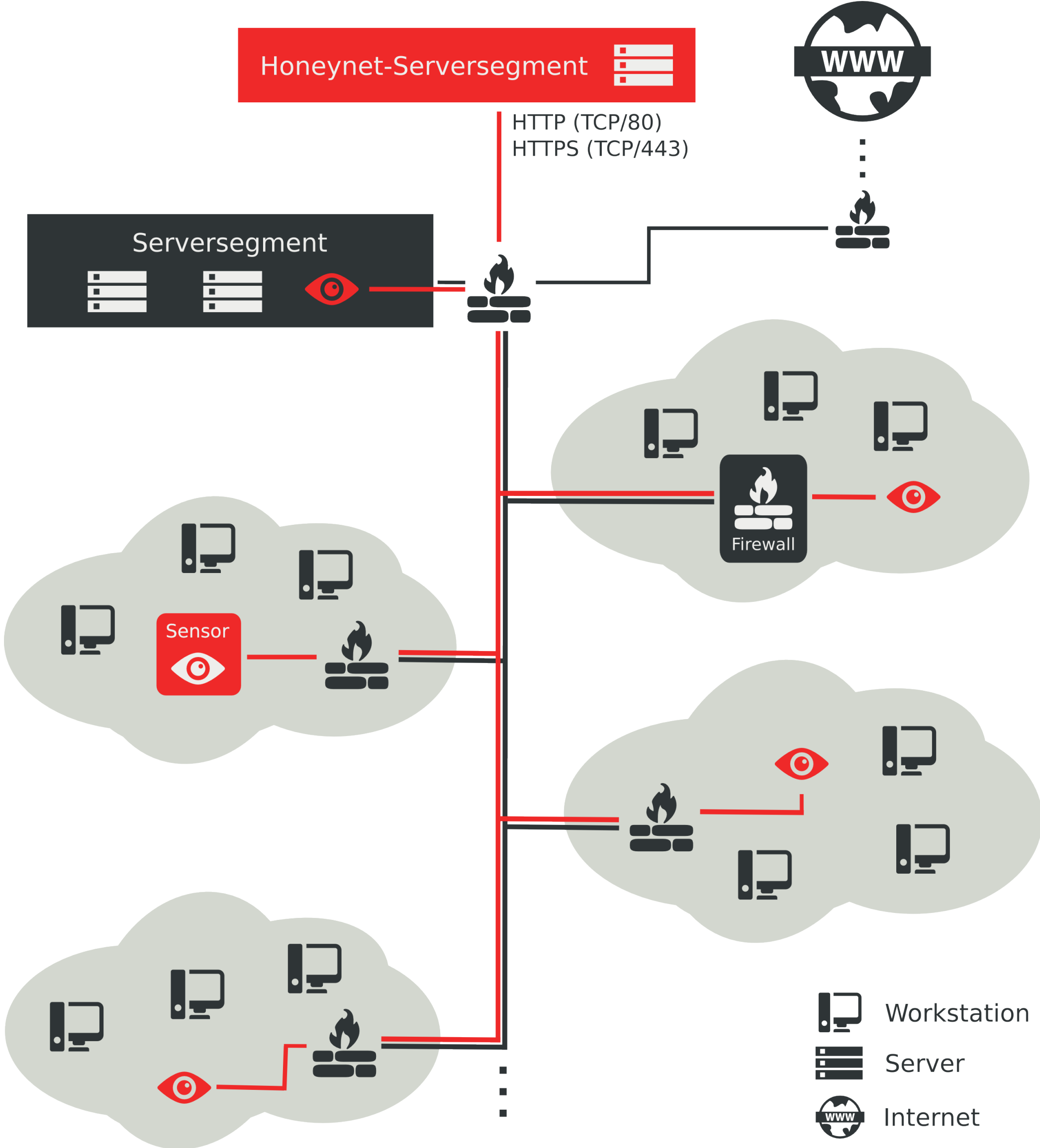


Abbildung 3: Die abstrakte Topologie des Honey-Netzes

Die Anforderung der **Verschlüsselung** bezieht sich insbesondere auf den Kommunikationskanal zwischen dem Honeynet-Server und den Sensoren. Da die Kommunikation wie bereits angesprochen HTTP-basiert stattfinden sollte, ist hier der Einsatz der mit TLS[21] abgesicherten, ansonsten aber äquivalenten Variante HTTPS angedacht. Dies hat den Vorteil, dass ein möglicherweise von einem Angreifer übernommener Sensor (oder irgendein anderes System im Netzwerk) sich einerseits nicht fälschlicherweise als Honeynet-Server ausgeben kann und andererseits auch ein Abfangen und Mitlesen der Kommunikation, die potentiell vertrauliche Daten enthält, verhindert bzw. zumindest deutlich erschwert wird.

[21] Transport Layer Security

6.3 Management und Analyse

Die zentrale Anlaufstelle für alle Sensoren des Honey-Netzes stellt der an einer von allen Knoten erreichbaren Stelle platzierte Server dar. Um die Hardware- und Integrationskosten des Gesamtsystems möglichst gering zu halten, werden alle Verwaltungs- und Analysefunktionen ebenfalls von diesem Serversystem übernommen. Dies umfasst prinzipiell die folgenden Funktionen:

- Speichern und Auflisten aller an den Sensoren aufgetretenen Ereignisse und die Möglichkeit, relevante Details über diese (wie beispielsweise fehlgeschlagene Login-Versuche an einem SSH-Honeypot) abzurufen. Funktionen zum Sortieren dieser Liste gehören zusätzlich zum geplanten Funktionsumfang.
- Bereitstellung einer Übersicht über alle registrierten Sensoren und deren aktuellen Status (online/offline/im Updateprozess etc.)
- Speichern und Auflisten von **Statusinformationen** der Sensoren, wie die aktuell verwendete Software-Revision oder die momentane Auslastung.
- Bereitstellung von Infrastruktur zum **Aktualisieren der Sensorsoftware**: Verwaltung und automatisierte Verteilung sowie Installation von Firmware-Images.
- **Server für die Zeitsynchronisation**: Da HTTP(S) als alleiniger Kommunikationskanal zwischen Sensoren und Server genutzt werden soll, ist hier eine Synchronisation der Uhrzeit über dieses Protokoll zu implementieren.
- Sicherstellen der **verschlüsselten Kommunikation zwischen allen Entitäten**: Sowohl die Kommunikation zwischen Benutzer und Server als auch zwischen Sensoren und Server soll verschlüsselt erfolgen. Hierfür soll der Server als eigene *Certificate Authority* (CA) auftreten und Zertifikate für die Sensoren generieren, mit denen sie sich später für alle Anfragen ausweisen müssen.
- **Infrastruktur zur benutzerfreundlichen Sensorverwaltung**: Dies umfasst primär das Installieren neuer Sensoren, speziell die Registrierung dieser beim Server, die Generierung von nutzbaren Installationsmedien aus den jeweiligen Firmware-Images und die Integration der spezifischen Konfigurationsoptionen für jeden Sensor. Beim Entfernen von Sensoren auf dem Server soll sichergestellt sein, dass jegliche mit dem Sensor verbundenen Einstellungen und Ereignisse ebenfalls vernichtet werden.
- Verwaltung verschiedener Versionen von Firmware-Images sowie die Möglichkeit, einzelne Sensoren selektiv mit bestimmter Firmware zu versorgen, um beispielsweise die Funktionalität einer neuen Softwarerevision zuerst mit einigen wenigen Sensoren zu testen.
- **Notifikation via E-Mail**: Um interessierte Stellen in regelmäßigen Intervallen über die Arbeit des Systems informieren zu können, soll der Server wöchentlich Zusammenfassungen über die aufgetretenen Ereignisse und den Status der Sensoren (online/offline) via E-Mail verschicken. Falls mehrere Personen an dieser interessiert sind, ist es essentiell, hierfür mehrere Adressen zugleich registrieren zu können. Zusätzlich soll eine *sofortige Notifikation* von zuständigen Stellen über E-Mail im Falle eines kritischen aufgetretenen Ereignisses erfolgen.
- Graphische Aufbereitung und Anzeige von Statistiken über die gesammelte Datenmenge.
- **Unterstützung mehrerer Benutzer** mit unterschiedlichen Rechten: Das System und alle zuvor genannten Funktionen sollen nicht allen Nutzern des Serversystems zugleich zur Verfügung stehen und somit hinter einem Login verborgen sein. Eine Benutzerverwaltung, in der verschiedene „Rollen“ für die Accounts vergeben werden, soll sicherstellen, dass einem angemeldeten Benutzer nur die ihm erlaubten Optionen präsentiert werden. Durch die Benutzerverwaltung soll zudem sichergestellt werden, dass nicht-autorisierten Personen der Zugang zum System verwehrt bleibt.

Unter Berücksichtigung der im vorigen Kapitel besprochenen Voraussetzung, die Kommunikation zwischen Sensor und Server ausschließlich auf die TCP-Ports 80 (HTTP) und 443 (HTTPS) zu beschränken, und aufgrund der Möglichkeit, dass Firewalls mit Hilfe von *Deep Packet Inspection* Pakete auf diesen Ports, die sich nicht an die HTTP-Protokolldefinition halten, einfach verwerfen, wurden für die Serverschnittstelle ein webbasiertes Front- und Backend konzipiert. Kernkomponente der Architektur ist eine **API**[22], über die alle Sensoren mit dem Server Daten austauschen. Da eine webbasierte API aus diesem Grund ohnehin über das Netzwerk erreichbar sein muss, können Clients als „Honeynet-Frontend“ in beliebigen Programmiersprachen und für beliebige Architekturen entwickelt werden. Der im Rahmen dieser Arbeit entwickelte Prototyp sieht zu diesem Zweck ebenfalls ein webbasiertes Frontend vor, das alle zuvor genannten Funktionalitäten vereint.

Im Bereich der serviceorientierten Architekturen existiert eine Reihe von Protokollen, um Dienste in verteilten Netzwerken zur Verfügung zu stellen. Bekannte Vertreter sind *SOAP*[23] und *XML-RPC*, beides XML-basierte Standards, sowie der *Representational State Transfer* (REST). Letzterer nutzt HTTP-Methoden wie `GET`, `POST`, `PUT` und `DELETE` zur semantischen Abbildung von **CRUD**-Operationen, die die Grundfunktionen zum persistenten Speichern von Daten repräsentieren (**C**reate, **R**ead, **U**pdate, **D**elete). Da das *REST*-Prinzip direkt auf dem HTTP-Protokoll basiert, wurde es für die diskutierte Honey-Netz-Architektur als Kommunikationsprotokoll ausgewählt. Aus dieser Entscheidung resultiert eine Reihe von Implikationen:

- Die modellierten Domänenobjekte bzw. Ressourcen des Honey-Netzes (dies können beispielsweise *Ereignisse*, *Sensoren* und *Benutzer* sein) sind über eindeutige URIs auf dem Server erreichbar und können mit folgenden HTTP-Nachrichtentypen modifiziert werden:

  GET : Lesen aller Ressourcen eines bestimmten Typs (z.B. Auflistung aller registrierten Sensoren). Optional kann eine ID mitgegeben werden, um nur ein bestimmtes Objekt abzufragen.

  POST : Speichern eines neuen Objektes des gewählten Typs.

  PUT : Modifizieren eines bereits existierenden Objekts (*Update*) mit einer gewählten ID.

  DELETE : Entfernen eines gewählten Objektes mit einer gewählten ID.

- Sensoren senden mit jeder Nachricht an die API zur Authentifikation ihre ID und eine mit Hilfe ihres lokalen Zertifikates generierte Signatur aller Daten mit. Auf Serverseite sind für alle registrierten Sensoren die öffentlichen Zertifikate hinterlegt, sodass überprüft werden kann, ob eine ankommende Nachricht gültig ist.
- Laut dem REST-Paradigma soll die Kommunikation mit dem Server *zustandslos* sein. Dies bedeutet, dass es keinen Kontext gibt und zwei Anfragen mit den gleichen Parametern zu jedem Zeitpunkt (mit gleichem Datenstand) identische Ergebnisse liefern müssen. Dieses Paradigma wird jedoch von der REST-API des Honey-Netzes an einer Stelle des geplanten API-Interfaces durchbrochen: Benutzer des Web-Frontends können sich über eine spezielle *Session*-Ressource registrieren und müssen somit nicht bei jeder Anfrage ein Authentifizierungstoken mitschicken. Hierbei handelt es sich um eine Vereinfachung des REST-Paradigmas, die es erlaubt, auf die in Browsern übliche Sessionverwaltung mit Cookies zurückzugreifen.
- Die Absicherung der gesamten Kommunikation erfolgt mit Hilfe von *TLS*[24]. Die Gewährleistung des verschlüsselten Datenverkehrs ist folglich Aufgabe des Web-Servers und nicht Bestandteil der API-Implementierung.

[22] Application Programming Interface
[23] ursprünglich „Simple Object Access Protocol“
[24] Transport Layer Security

- Als Datenformat zur Kommunikation mit der REST-API wird *JSON*[25] verwendet, eine an die Objektsyntax von JavaScript angelehnte, als Zeichenkette serialisierbare Struktur mit nur geringem Overhead. Alle Informationen, ob vom Server empfangene oder zu diesem gesendete, werden vor dem Absenden in das JSON-Format konvertiert. Dieser Schritt erleichtert den Datenaustausch insofern, dass JSON zumeist mit Bibliotheken in Objekte aller gängigen Programmiersprachen konvertiert werden kann und die Unabhängigkeit des Projektes von bestimmten Technologien erhöht.

Alle domänenspezifischen Daten werden auf Serverseite in einer relationalen Datenbank gespeichert. Auf das verwendete Datenmodell wird im Kapitel über die Implementierung detaillierter eingegangen. Die REST-API wird von einem Web-Server angeboten, der zugleich auch Host für das Web-Frontend ist. Die Serversoftware und jegliche zusätzlich nötige Infrastruktur werden gemeinsam auf dem zentralen Server des Honey-Netzes betrieben.

6.4 Sensoren

Nach der Beschreibung der allgemeinen Netzwerktopologie und der Management-Seite gilt es nun die Clients des Systems, die sogenannten „*Sensoren*“, architektonisch näher zu beleuchten. Bei diesen handelt es sich um vollwertige Hosts, die IP-Adressen des jeweils lokalen Netzwerks erhalten und deren Zweck es ist, jegliche Verbindungsversuche von anderen Netzwerkknoten mindestens aufzuzeichnen und optional auch zu beantworten, wofür allerdings zur Anfrage passende Honeypot-Software notwendig ist. Der Begriff „Sensor“ wurde einem reinen „Honeypot“ vorgezogen, da diese Systeme auch ohne Honeypot-Software funktionieren können, indem sie allen an sie gerichteten Verkehr aufzeichnen. In einer solchen Konfiguration erfüllen sie die Aufgabe eines *Netzwerkteleskops* [16], was sich deutlich von der Arbeitsweise eines Honeypots unterscheidet.

Um den geforderten geringen Hardwareanforderungen und Software-Beschaffungskosten gerecht zu werden, ist auf allen Sensoren der Einsatz eines linuxbasierten Betriebssystems vorgesehen. Zusätzlich spricht für diese Entscheidung, dass frei verfügbare Honeypot-Software immer auch auf diesem Betriebssystem lauffähig ist, unabhängig von der verwendeten Distribution [16]. Da eine große Anzahl parallel betriebener Sensoren vorgesehen ist, muss die komfortable Verwaltung derselben mit Hilfe des Honeynet-Servers sichergestellt werden. Dieser bietet eine zuvor beschriebene REST-API als einzigen erreichbaren Dienst an. Die Push-Architektur verbietet zudem, dass Sensoren vom Server direkt kontaktiert werden können. Somit ist es Aufgabe der Sensoren, sicherzustellen, dass der Server in regelmäßigen Intervallen kontaktiert und mit aktuellen Statusinformationen versorgt wird. Um zudem zu verhindern, dass alle beim Server registrierten Sensoren dies gleichzeitig tun und damit eigentlich unnötige Lastspitzen im Netzwerk erzeugen, sollten die Abfrageintervalle in irgendeiner Form randomisiert sein.

Die folgenden Absätze behandeln verschiedene besondere Teilaspekte des Sensor-Betriebssystems.

[25] JavaScript Object Notation

Polling Das regelmäßige Kontaktieren des Servers („*Polling*“) besteht aus zwei Phasen und läuft abstrakt folgendermaßen ab:

1. **Request**
   (a) Der Sensor sammelt Informationen über den lokalen Systemzustand, wie beispielsweise die Auslastung, die Belegung des Arbeitsspeichers oder die gerade installierte Firmware-Revision.
   (b) Die gesammelten Daten werden ins JSON-Format *konvertiert*.
   (c) Das resultierende JSON-Objekt wird mit Hilfe des auf jedem Sensor vorliegenden, vom Server ausgestellten Zertifikates *signiert*.
   (d) Aus der Signatur und den Daten wird erneut ein finales JSON-Objekt generiert und zum Server an die *Sensorstatus*-Ressource der API gesendet.
   (e) Auf Serverseite wird validiert, ob der Sensor in der Datenbank existiert, dessen Zertifikat noch gültig ist und ob die Signatur des Datenblocks korrekt ist.
2. **Response**
   (a) Der Server liest aus der Datenbank die aktuelle Konfiguration des betreffenden Sensors aus, konvertiert diese zu einem JSON-Objekt und sendet sie zurück zum Sensor. Dies umfasst Parameter wie das Polling-Intervall, die für den Sensor angedachte Firmware-Revision und Details bezüglich der Honeypot-Software.
   (b) Nach Empfang liest der Sensor seine Konfigurationsdaten aus und nimmt, falls nötig, Änderungen am lokalen Systemzustand vor. Zu diesem Zweck hält er die ihm zuletzt mitgeteilte Konfiguration lokal vor, um für den Fall der Nichterreichbarkeit der Serverseite noch arbeiten zu können. Um die Integrität der Servernachricht zu gewährleisten, sollte sie ebenfalls signiert sein und vom Sensor eine entsprechende Überprüfung durchgeführt werden.
   (c) Die empfangene für den Sensor serverseitig vorgesehene Firmware-Revision wird mit der lokal vorhandenen verglichen. Falls Unterschiede bestehen, wird der Update-Prozess angestoßen. In Vorbereitung dessen werden alle Honeypot-Dienste zuvor beendet und die Kontrolle an das Skript zur Aktualisierung übergeben.
   (d) Dienste, deren Status sich geändert hat, werden entsprechend der neuen Konfigurationsdaten gestartet, angehalten oder umkonfiguriert.

Die Intervalle, in denen obige Prozedur ausgeführt wird, sind serverseitig konfigurierbar und Teil der lokalen Sensorkonfiguration. Eine randomisierte Variation von diesem Wert, um zu verhindern, dass alle Sensoren gleichzeitig das Polling durchführen, wird sensorseitig implementiert.

Firmware Die Gesamtheit aller Software, die auf den Sensoren betrieben wird, erhält im Rahmen dieser Arbeit den Namen *Firmware*. Dies umfasst das linuxbasierte Betriebssystem, bestehend aus dem Kernel und der zugehörigen Userland-Basissoftware, sowie alle zusätzlich für den Zweck des Honey-Netzes installierte Software: primär Skripte zur Verwaltung des jeweiligen Sensors sowie die Honeypot-Software selbst. Firmware wird auf einem separaten Entwicklersystem modifiziert, aktualisiert und letztendlich zur Weiterverteilung gepackt. Ein Firmware-Archiv beinhaltet somit die Software selbst in Form des Sensor-Dateisystems und eine Reihe von Metadaten, nämlich

- den **Namen** der Firmware,
- die Firmware-**Revision** (Versions-String),
- ein **Changelog**, bestehend aus Einträgen, die die Änderungen der vergangenen Revisionen zusammenfassen.

Update Falls im zuvor beschriebenen Polling-Verfahren abweichende Firmware-Revisionen aufgetreten sind, stößt der Sensor den Update-Prozess an. Um den Administratoren des Systems freie Hand bei der Firmwareverteilung zu lassen, findet keine numerische Überprüfung der Firmware-Revision statt (um beispielsweise nur auf aktuellere Versionen zu aktualisieren). Vielmehr ist es gewollt, dass auch Downgrades für den Fall einer neueren, nicht korrekt funktionierenden Firmware möglich sind, weshalb das Firmware-Update sofort bei jeder Abweichung der Metadaten durchgeführt wird.

Der Aktualisierungsprozess selbst läuft in folgenden Schritten ab:

1. Beenden aller nicht für den Aktualisierungsprozess erforderlichen Dienste. Der Sensor ist währenddessen nicht als Teil des Honey-Netzes funktionsfähig.
2. Reduzieren des Polling-Intervalls auf eine Minute. Somit ist sichergestellt, dass sich im Update-Prozess befindliche Knoten regelmäßig beim Server melden und einen Statusbericht abliefern können.
3. Download der neuen Firmware vom Server. Hierzu muss der Sensor lediglich eine `GET`-Anfrage an die *Sensorimage*-Ressource der API stellen und die ID des gewünschten Firmware-Images mitliefern. Die Serverseite schickt daraufhin das Firmware-Archiv, das der Sensor lokal in einem temporären Verzeichnis ablegt.
4. Die lokale Konfiguration des Sensors wird extrahiert, gepackt und zur späteren Rekonfiguration gespeichert.
5. Nach dem Entpacken der Firmware- und Metadaten und einer ausführlichen Integritätsprüfung werden diese genutzt, um das bestehende System zu überschreiben. Wie dies im Detail funktioniert, ist hardwareabhängig und wird für den im Rahmen dieser Arbeit entwickelten Prototypen im Kapitel zur Implementierung detailliert beschrieben.
6. Für den Neustart des Sensorsystems werden zuletzt Änderungen am Boot-Manager und an der Partitionstabelle vorgenommen sowie die zuvor gesicherte Konfiguration zurückgeschrieben.
7. Während das System mit der neuen Firmware bootet, wird die gesicherte Konfiguration wieder übernommen. Die Separation der Konfiguration erlaubt es, generische Firmware-Images für alle Systeme gleichzeitig einzusetzen.

Installation Für das Hinzufügen von neuen Sensoren zum Honey-Netz existiert ein einheitlicher Prozess, bei dem der Administrator vom Software-Protoypen seitens des Web-Frontends unterstützt wird. Voraussetzung zur Registrierung neuer Sensoren ist die Existenz eines gültigen, als systemweiter Standard deklarierten Firmware-Images. Das im Rahmen dieser Arbeit individuell entwickelte und zuvor beschriebene Image-Format kann nicht einfach auf einen Datenträger geschrieben und zur Installation eines neuen Systems verwendet werden, da es speziell auf die Bereitstellung vollautomatischer Updates ausgelegt ist. Folglich muss jede Firmware, die zur Installation neuer Sensoren verwendet werden soll, zuerst separat in ein bootbares Medienformat konvertiert werden. Anschließend ist es beispielsweise möglich, dieses auf eine SD-Karte zu schreiben, von der ein Sensorsystem anschließend booten und die Software installieren kann. Die Details dieses Prozesses hängen stark von den Gegebenheiten der verwendeten Hardwareplattform für die Sensoren ab und werden für die prototypische Implementierung dieser Arbeit im zuständigen Kapitel näher beleuchtet. Das Prinzip orientiert sich jedoch am Update-Verfahren:

1. Über eine HTTP-`POST`-Nachricht an die *Sensor*-Ressource der REST-API wird ein neuer Sensor beim Server registriert. Die Anfrage beinhaltet elementare sensorspezifische Einstellungen, darunter die Netzwerkkonfiguration (DHCP/statisch), die URL, unter der der Honeynet-Server erreichbar

ist, sowie Basiseinstellungen zu den Honeypot-Diensten.
2. Der Server generiert ein Archiv, das alle initialen Konfigurationsdaten des jeweiligen Sensors beinhaltet und von diesem bei der Installation verarbeitet werden kann. Es wird über die *Sensorconfig*-Ressource der API zum Download angeboten.
3. Es ist nun Aufgabe des Benutzers, ein Installationsmedium des aktuellen Firmware-Images und das individuelle Konfigurationsarchiv für den neuen Sensor vom Server herunterzuladen und auf einen Datenträger zu schreiben.
4. Der Datenträger wird in den Sensor eingelegt und der Bootvorgang gestartet, die Installation erfolgt abschließend vollautomatisch. Über die API bzw. ein dafür entwickeltes Frontend kann der Sensorstatus überprüft werden.

Honeypots Neben den für das Management und den Betrieb nötigen Diensten und Skripten ist der Zweck der Sensoren, als Honeypot für potentielle Angreifer zu dienen. Für diese Aufgabe gilt es, auf Linux lauffähige Honeypots auszuwählen und für den Einsatz im Honey-Netz zu modifizieren. Da die gewonnenen Daten lokal auf den Sensoren nicht genutzt werden können, müssen diese stattdessen mit Hilfe von speziell für diesen Zweck entwickelter Plugins in das von der REST-API erwartete Format konvertiert und zum Server gesendet werden. Honeypots sollten prinzipbedingt die umliegenden Hosts im gleichen Netzsegment hinsichtlich der angebotenen Dienste reflektieren [16], was angesichts der Größe und Softwarevielfalt des SVNs nahezu unmöglich wird, da vielerorts Appliances mit proprietären, selten verwendeten Protokollen zum Einsatz kommen, für deren Emulation gesonderte Honeypot-Software geschrieben werden müsste. Stattdessen bieten die Sensoren – insbesondere unter Berücksichtigung der Trafficanalyse, bei der überhaupt nur sehr wenige Anfragen aus dem „Inneren“ gemessen wurden – nur einige generische, als Einfallstor bekannte Services an. Da diese aus dem Internet nicht erreichbaren Knoten ohnehin kaum Netzwerkverkehr aufweisen und daher jedes direkt an sie gerichtete Paket als verdächtig einstufen, bietet Honeypot-Software noch den Bonus, zusätzliche Informationen über potentielle Angreifer ermitteln zu können.

Grundbestandteil des Sensorsystems ist zudem ein „**Passive Scan Mode**“ getaufter Dienst, der alle direkt an das lokale System gerichteten Anfragen aufzeichnet und an den Server weiterleitet. Der Dienst operiert auf Basis der Sensor-Firewall und kann somit unabhängig von bestimmten Ports oder Protokollen agieren, liefert jedoch auch nur sehr elementare Informationen über einen Vorfall: Die Quell-IP-Adresse, den Zielport der Anfrage und einen Zeitstempel. Um zudem zu verhindern, dass im Falle eines Portscans tausende von verdächtigen Ereignissen vom Sensor zum Server geschickt werden, beinhaltet der Dienst eine elementare *Portscan-Erkennung*, indem er Netzwerkverkehr von einer Quell-Adresse sammelt, konsolidiert und auswertet. Es wird anschließend nur das Resultat dieser Analyse – Einzelvorfall oder Portscan – zum Server gesendet.

Die in obigem Abschnitt beschriebene Architektur wurde im Rahmen dieser Arbeit für das SVN als ein prototypisches Sensornetzwerk implementiert. Die Details und gewonnenen Erfahrungen aus dieser Entwicklung werden im nachfolgenden Kapitel ausführlich besprochen.

Teil V

Implementierung: HoneySens

Nach den konzeptionellen Ausführungen über das für das Sächsische Verwaltungsnetz entworfene Sensornetzwerk im vorigen Abschnitt beschreibt dieses Kapitel nun die Implementierung eines funktionsfähigen Prototypen im Detail. Der Projektname „*HoneySens*“ soll den Fokus auf die Tatsache lenken, dass traditionelle (Low- oder High-Interaction-) Honeypots nur eine – zweifelsohne wichtige – Komponente des Gesamtsystems sind und es sich vielmehr um ein Netzwerk von unabhängigen Sensoren und die für deren Betrieb zuständige Infrastruktur handelt.

Der Prototyp ist speziell für den Einsatz innerhalb des SVNs konzipiert worden und trifft deshalb an einigen Stellen Annahmen, die für andere große IT-Infrastrukturen nicht gelten. Mit Ausblick auf eine spätere Weiterentwicklung der Software wurde jedoch darauf geachtet, das Gesamtsystem modular zu gestalten und für ähnliche Netzwerke, in denen beispielsweise andere Betriebssysteme oder Hardwarearchitekturen zum Einsatz kommen, adaptierbar zu machen. Details hierfür werden im Rahmen dieses Kapitels und im Ausblick am Ende der Arbeit genannt.

Das HoneySens-System besteht im Wesentlichen aus zwei sehr verschiedenen Komponenten: dem zentralen Server und den verteilten Sensoren. Beide können sich in ihrer Hardware-Architektur voneinander unterscheiden. Aus diesem Grund ist es essentiell, die Kommunikation auf einem gemeinsamen Netzwerkprotokoll und einheitlichen Standards aufzubauen. Zunächst erfolgt deren Spezifikation im nachfolgenden Abschnitt, Implementierungsdetails zur jeweiligen Client- (Sensor-) und Serverseite schließen sich an.

7 Server

Die Serverseite des HoneySens-Projektes übernimmt die elementaren Aufgaben der Verwaltung und Datenhaltung aller Sensoren. Konkret empfängt sie hierfür alle an den Sensoren aufgetretenen netzwerkbezogenen Ereignisse wie Portscans, Verbindungs- oder Einbruchsversuche (die durch Honeypot-Software erkannt werden), teilt diesen allerdings auch auf Anfrage ihre jeweilige Konfiguration mit und überprüft die Gültigkeit aller empfangenen Sensordaten. Das Management des Netzwerks, darunter das Hinzufügen und Entfernen von Sensoren sowie das Anpassen der auf diesen befindlichen Software gehört ebenfalls zum Aufgabenbereich des Serversystems. Aufgrund von Limitierungen der Kommunikation zwischen den Teilnetzen des SVNs kommt beim Aufzeichnen von vorgefallenen Ereignissen, wie bereits in Kapitel 6.2 angesprochen, eine Push-Architektur zum Einsatz. Das bedeutet, dass lediglich die Sensoren den Server direkt kontaktieren können. Der umgekehrte Weg kann nicht sichergestellt und sollte deshalb vermieden werden. Es fällt zusätzlich in den Aufgabenbereich der Sensoren, in regelmäßigen Abständen die eigene Konfiguration beim Server abzufragen.

Die Serverkomponente ist unter Berücksichtigung der Konzeption wiederum zweigeteilt. Die wichtigste Schnittstelle stellt eine sogenannte **REST-API** dar, ein HTTP(S)-basiertes Interface, das alle sowohl für die Sensoren als auch die Administration relevanten Funktionen in einem einheitlichen Format zur

Verfügung stellt. Weiterhin kann ein **Web-Frontend** als exemplarische Implementierung eines die API nutzenden Clients ebenfalls direkt auf demselben Web-Server betrieben werden. Da die API vom Server öffentlich angeboten wird und auf offene, dokumentierte Standards setzt, könnte auch Client-Software für Desktops oder Mobilgeräte wie Smartphones oder Tablets angeboten werden. Aufgrund der Beschränkungen innerhalb des SVNs kommunizieren die Sensoren ausschließlich über diesen Kanal mit dem Server. Jegliche ausgehende Kommunikation der Clients richtet sich – abgesehen von solcher mit potentiellen Angreifern – an diese API, für eigentlich über andere Ports laufende Dienste wie die Zeitsynchronisation (NTP) mussten HTTP-basierte Ersatzlösungen gefunden werden.

Weder die API noch das Web-Frontend setzen spezielle Hardwarekomponenten oder eine bestimmte Architektur voraus. Es kommen ausschließlich offene, betriebssystemunabhängige Standards zum Einsatz. Das System wurde jedoch auf Linux-Systemen entwickelt und getestet und nutzt in einigen unterstützenden Skripten quelloffene, teilweise aber nur auf unixoiden Betriebssystemen vorhandene Werkzeuge. Eine Adaption dieser Skripte für alternative Plattformen ist jedoch möglich. Weiterhin ist der Deployment-Prozess zum Zeitpunkt der Entstehung dieser Arbeit nur unter Linux nativ nutzbar, kann jedoch auch unter anderen Betriebssystemen mit Hilfe von Virtualisierungstechniken angewendet werden.

7.1 REST-API

Bei REST handelt es sich um ein Programmierparadigma, bei dem Objekte einer Anwendungsdomäne als *Ressourcen* definiert werden. Jegliche Kommunikation mit der Schnittstelle erfolgt über Anfragen an die gewünschte Ressource, die Architektur ist dabei per Definition *zustandslos*: Alle für das Ergebnis einer jeden Operation nötigen Parameter müssen in der Nachricht des Clients enthalten sein. Ein Kontext existiert nicht, d.h. dass zwei Anfragen mit identischen Parametern unter der Voraussetzung, dass sich der serverseitige Datenstand inzwischen nicht verändert hat, immer zum gleichen Ergebnis führen. REST nutzt zudem direkt die Eigenschaften des HTTP-Protokolls, indem die angefragte Ressource mit Hilfe der URI im Header und die auszuführende Operation über die HTTP-Methode spezifiziert werden. Das empfohlene und für den HoneySens-Prototypen verwendete Mapping von HTTP-Methoden auf CRUD[26]-Operationen beschreibt Tabelle 3.

HTTP-Statuscode	CRUD-Operation
`POST`	Speichern einer neuen Ressource (CREATE)
`GET`	Lesen einer oder mehrerer Ressourcen (READ)
`PUT`	Aktualisieren einer bereits vorhandenen Ressource (UPDATE)
`DELETE`	Entfernen einer vorhandenen Ressource (DELETE)

Tabelle 3: Mapping von HTTP-Methoden auf CRUD-Operationen

Da REST selbst ein abstraktes, architektur- und plattformunabhängiges Konzept zum Informationsaustausch in verteilten Systemen ist, können Clients für eine derartige API in zahlreichen Programmiersprachen geschrieben und von beliebigen Betriebssystemen aus genutzt werden. Dies ist insbesondere in heterogenen verteilten Architekturen von Vorteil. Frameworks, die das Kommunikationsmodell direkt unterstützen, übersetzen implizit Operationen wie `save()` oder `delete()` von Model-Objekten in entsprechende REST-Anfragen. Die Abhängigkeit von HTTP muss, insofern das Protokoll in der jeweiligen Infrastruktur eingesetzt werden kann, außerdem kein Nachteil sein: Für viele Anwendungen

[26] Basisoperationen auf Entitäten: **C**reate, **R**ead, **U**pdate, **D**elete

essentielle Sicherheitsfunktionen wie Verschlüsselung oder Authentifizierung stehen beispielsweise direkt über HTTPS zur Verfügung, eine um TLS erweiterte Variante des HTTP-Protokolls. Zusätzlich ist es ein standardisiertes, offenes Protokoll und erlaubt somit großen Freiraum bei der Auswahl passender Serversoftware.

Das REST-Paradigma schreibt weiterhin kein Format für die im HTTP-Body übertragenen Daten vor, weshalb es Aufgabe des Programmierers ist, einen für die jeweilige Anwendung geeigneten Standard auszuwählen. Einige Frameworks abstrahieren zudem das Serialisieren von Anwendungsobjekten in innerhalb einer Nachricht eingebettete Zeichenketten und nehmen dem Entwickler diese Entscheidung somit ab. In der Praxis nutzen REST-APIs typischerweise *JSON* als Datenformat, es können aber auch *XML*, URL-kodierte HTML-Formularstrings (z.B. `var1=wert1&var2=wert2`) oder Binärdaten versendet werden. JSON hat sich zum Zeitpunkt dieser Arbeit allerdings als ein Web-Standard mit breiter Softwareunterstützung etabliert [19] und wurde deshalb auch für die HoneySens-API auserkoren. Die Syntax orientiert sich stark an der Notation von Objekten in JavaScript-Programmcode, wie das folgende Beispiel zeigt:

```
[{ name1: val1, name2: val2 }]
```

JSON-Daten werden als String gespeichert und übertragen, durch die minimalistische Syntax führen sie allerdings im Vergleich zu beispielsweise XML nur einen geringen Overhead mit sich. Die Objekte bestehen aus einer Menge von Attributen und zugehörigen Werten, wobei ein Wert wiederum ein String, ein weiteres Objekt oder ein Array sein kann. Arrays sind ebenfalls wieder aus Strings, Objekten oder weiteren Arrays zusammengesetzt. Objekte werden durch geschweifte und Arrays durch eckige Klammern begrenzt, Attributnamen und -werte durch einen Doppelpunkt voneinander und mit Kommata von anderen Attribut-Wert-Paaren getrennt. Als nativer Datentyp existieren lediglich Strings, das Umwandeln der einzelnen Werte in Boolean, Zahlen usw. ist dem JSON-Parser überlassen und abhängig von der gewählten Implementierung.

7.1.1 Schnittstellendefinition

Der folgende Abschnitt versucht einen Überblick über die im Rahmen des Prototypen entwickelte API-Schnittstelle und deren angebotene Funktionalität zu geben. Alle Ressourcen werden über die abstrakte URI

```
/api/<ressource type>/[ <id> | <operation> [/<parameter>] ... ]/
```

adressiert. Die Semantik der jeweiligen Operation hängt zudem gemäß dem REST-Paradigma von der in der Anfrage spezifizierten HTTP-Methode ab. Es gelten die folgenden Richtlinien:

- Wird lediglich eine Ressource ohne die optionalen Felder mit der HTTP-Methode `GET` angefragt, werden alle vorhandenen Objekte des spezifizierten Typs zurückgegeben. Bei einer `POST`-Anfrage wird hingegen eine neue Ressource mit den im HTTP-Body als JSON-String hinterlegten Daten angelegt.
- Den Methoden `PUT` und `DELETE` muss eine ID mitgegeben werden, die das zu bearbeitende Objekt auf dem Server eindeutig spezifiziert. Bei `POST`-Operationen ist dies nicht nötig, da beim Anlegen einer neuen Ressource die ID erst noch vom Server vergeben werden muss. Im Falle von `GET` ist

es optional.

- Einige Ressourcen unterstützen weitere, über CRUD hinausgehende Methoden. In einem solchen Fall wird die jeweilige Operation in der URI nach der Ressource spezifiziert, anschließend folgen weitere kontextabhängige Parameter (wie beispielsweise wieder eine ID).

Eine ausführliche Übersicht über alle durch die API bereitgestellten Ressourcen liefert Tabelle 4.

Ressource	Beschreibung
`sensors`	Erlaubt das Anlegen, Auflisten und Entfernen von Sensoren. *Zusätzliche Methoden* `config`: Nimmt eine `id` als Parameter entgegen und bietet ein vom Server zuvor konfiguriertes Konfigurationsarchiv des zugehörigen Sensors zum Download an, falls vorhanden. Dieses wird bei der Installation neuer Sensoren benötigt (siehe auch Kapitel 9).
`events`	Erlaubt das Verwalten von sensorspezifischen Ereignissen. Die Methode `POST` steht exklusiv für Sensoren zur Verfügung und verlangt zur Authentifikation zusätzlich zu den Nutzdaten eine mit einem dem Server bekannten Zertifikat generierte gültige Signatur. Via `PUT` können lediglich Attribute, die den Bearbeitungsstatus des jeweiligen Ereignisses betreffen, modifiziert werden. Hiermit wird garantiert, dass alle weiteren Ereignisparameter unveränderlich sind und somit vom Client nicht neu gelesen werden müssen. Dies resultiert im Rahmen dieser Arbeit schlussendlich in Performanceverbesserungen im prototypischen Web-Frontend.
`eventdetails`	Hält zusätzliche, ereignisspezifische Informationen vor. Es kann sich bei diesen entweder um Daten mit Zeitstempel, die beispielsweise die Interaktion eines Angreifers mit einem Honeypot detailliert beschreiben, oder um beliebige vom betreffenden Sensor gewonnene Zusatzinformationen (z.B. den Namen eines erkannten Exploits) handeln. *Zusätzliche Methoden* `by-event`: Gibt alle Details eines bestimmten Ereignisses zurück, spezifiziert durch eine `id`.
`certs`	Dient der Verwaltung von Sensor-Zertifikaten. Diese werden vom Server generiert und anschließend dem Client für die Installation auf dem Sensor übergeben. Ein nachträgliches Verändern der Zertifikate ist nicht möglich.
`sensorconfigs`	Ermöglicht das Lesen und Schreiben von Sensor-Konfigurationsdaten. Diese umfassen das Update-Intervall, in dem sich der Sensor beim Server meldet, sowie zusätzliche Laufzeitoptionen der Honeypot-Dienste. Die spezielle Ressource mit der ID 1 repräsentiert die globale Standardkonfiguration für alle Sensoren, denen keine individuelle Konfiguration zugeordnet wurde.

`sensorstatus`	Repräsentiert die Summe der Statusinformationen, die Sensoren in regelmäßigen Abständen dem Server mitteilen. Sie umfassen lokale Zustandsdaten wie beispielsweise die Auslastung des Arbeitsspeichers, die aktuelle IP-Adresse oder die Versionsbezeichnung der Firmware. Die Methode `PUT` zum Verändern bereits gespeicherter Datensätze steht nicht zur Verfügung. Neue Daten können zudem lediglich von Sensoren mit gültigem Zertifikat via `POST` erzeugt werden. *Zusätzliche Methoden* `by-sensor`: Gibt alle vorhandenen Statusdaten eines bestimmten, mit einer `id` spezifizierten Sensors aus.
`sensorimages`	Diese Ressource repräsentiert die Firmware der Sensoren. Die API verwaltet allerdings lediglich Metadaten, die Software selbst wird im Dateisystem des Servers abgelegt (siehe Kapitel 9). Weitere Firmware wird hinzugefügt, indem die Firmware-Datei via `POST` an die `sensorimages`-Ressource gesendet wird. `PUT` erlaubt lediglich das Verändern des Flags `conversionStatus`, welches den Fortschritt der Konvertierung der Firmware in ein bootbares Installationsmedium beschreibt. Die Metadaten sind außerdem statisch und können nicht modifiziert werden. Beim Entfernen eines Objektes via `DELETE` werden vom Server auch die zugehörigen Daten im Dateisystem mit aufgeräumt. *Zusätzliche Methoden* `download`: Initiiert den Download der Firmware-Datei mit einer gewünschten `id`. `download-sd`: Startet den Download des zur Firmware gehörigen Abbilds für Speicherkarten (mit spezifizierter `id`), das zur Installation eines neuen Sensors eingesetzt werden kann. Die Konvertierung des Firmware-Formats zum SD-Format erfolgt durch den HoneySens-Server.
`users`	Repräsentiert einen Benutzer des HoneySens-Systems – im Falle eines Web-Clients eine Person, die sich an- und abmelden kann sowie mit einer Reihe von Rechten und Interaktionsmöglichkeiten ausgestattet ist. Ein Passwort kann nicht wieder ausgelesen, sondern nur via `PUT` geändert werden. Für die Rechteverwaltung existiert keine separate Ressource, stattdessen gibt es im HoneySens-Prototypen für Benutzer drei statische Gruppen mit zugehörigen Rechten, die statisch im Programmcode definiert sind. Details zur Benutzerverwaltung folgen in Kapitel 7.
`contacts`	Zur Notifikation im Falle von kritischen Ereignissen oder für regelmäßige Zusammenfassungen kann der HoneySens-Server E-Mails an interessierte Stellen versenden. Die Empfänger werden seitens der API als *Kontakte* bezeichnet und in Form von E-Mail-Adressen mit zugehörigen Sende-Flags für kritische Ereignisse und Zusammenfassungen repräsentiert. Es stehen alle CRUD-Operationen zur Verfügung.

`settings`	Applikationsspezifische Einstellungen werden nicht in der Datenbank, sondern einer Konfigurationsdatei auf dem die API anbietenden Web-Server gespeichert, da Änderungen an diesen Daten nur selten notwendig sind. Das betrifft SMTP-Parameter zum Mailversand und die Spezifikation des Endpunkts, über den die Sensoren ihren Server erreichen können. Für die `settings`-Ressource existieren keine separaten Datensätze; jede Option kann ohne Angabe einer `id` gesetzt sowie gelesen werden und ist global gültig. Es stehen folglich lediglich die Methoden `GET` und `PUT` zur Verfügung.
`state`	Diese Ressource kann nur via `GET` gelesen werden und gibt alle von einem Client über die API abrufbaren Informationen in einer großen JSON-Struktur zurück, indem sie alle zuvor genannten Ressourcen abfragt und die Rückgabewerte konkateniert. Die Funktionalität wird beispielsweise zum Initialisieren des Web-Clients benötigt, nachdem sich ein Benutzer neu angemeldet hat. Falls bei der Anfrage der Parameter `ts` zusammen mit einem Zeitstempel übergeben wird, werden Daten inkrementell zurückgeliefert. Nur falls eine Ressource Änderungen seit dem übermittelten Zeitpunkt erfahren hat, erfolgt eine Ausgabe. Der Server speichert zu diesem Zweck intern die letzten Änderungszeitpunkte. Es existiert weiterhin der optionale Parameter `last_id`, über den ein Client die ID des letzten Ereignisses mitteilen kann, das ihm bekannt ist. Der Mechanismus dient ebenfalls der inkrementellen Aktualisierung der Ereignisliste. Da diese jedoch sehr umfangreich werden kann, ist ein erneutes Empfangen *aller* Datensätze nicht effizient. Es wird deshalb anstatt eines Zeitstempels die letzte bekannte ID übergeben, sodass der Server lediglich alle neueren Ereignisse ausgeben muss.
`sessions`	Die Session-Ressource bricht wie zuvor bereits beschrieben mit dem REST-Paradigma, das kontextfreie Kommunikation vorschreibt. Sie wird zur Authentifikation von Benutzern verwendet und stellt eine Vereinfachung der strikten REST-Methodik dar: Anstatt Zugangsdaten mit jeder Anfrage erneut zum Server zu senden, genügt ein einmaliges Senden derselben via `POST` an die Ressource `sessions`, um eine neue Session zu erzeugen. Wenn der überlieferte Benutzername und das zugehörige Passwort vom Server als gültig befunden wurden, wird serverseitig das Session-Objekt erzeugt und ein *HTTP-Cookie* zurückgeschickt. Es ist anschließend Aufgabe des Clients (bzw. Browsers im Falle der prototypischen Webanwendung), dieses als Authentifizierungstoken bei jeder nachfolgenden Anfrage mitzusenden. Analog hierzu kann eine Session mit `DELETE` wieder geschlossen werden, was gleichbedeutend mit dem Abmelden eines Benutzers vom System ist.

Tabelle 4: Von der API angebotene Ressourcen

Sowohl Anfragen an die API als auch deren Antworten sind via JSON formatiert. Methoden, die den Datenstand serverseitig verändern, halten sich zudem an die HoneySens-spezifische Richtlinie, den Erfolg der Modifikation über das Attribut `success` und eventuell aufgetretene Fehler via `error` mitzuteilen. Zusätzliche Informationen werden in weiteren, operationsspezifischen Eigenschaften definiert. Eine Übersicht und Beschreibung aller individuellen Anfrage- und Rückgabeattribute der API würde den Rahmen dieser Arbeit jedoch sprengen.

Ein kurzes Beispiel soll die Funktionsweise der REST-API verdeutlichen: Für einen einzelnen Sensor ist exemplarisch eine individuelle Konfiguration anzulegen, um dessen Update-Intervall auf fünf Minuten zu begrenzen. Hierzu wird eine entsprechende HTTP-Nachricht an den HoneySens-Server gesendet:

```
POST http(s)://server/api/sensorconfigs
```

Der Body der Nachricht enthält die Attribute der anzulegenden Konfiguration in Form eines JSON-Objektes, wie in Abbildung 4 zu sehen. Es folgt als kompakter String formatiert dem Header des HTTP-Pakets.

```
{
  "interval": "5",
  "passiveScan": true,
  "kippoHoneypot": true,
  "dionaeaHoneypot": false,
  "image": "0",
  "sensor": "44"
}
```

Abbildung 4: JSON-Objekt zum Anlegen einer Sensor-Konfiguration

Der Server überprüft nach Erhalt der Nachricht zunächst, ob der Client zu dieser Aktion überhaupt berechtigt ist, indem er das im Header mit übergebene HTTP-Cookie auf Gültigkeit überprüft und – falls serverseitig eine zugehörige Session existiert – testet, ob der betreffende Benutzer die nötigen Rechte besitzt. Wenn diese Tests erfolgreich waren, das JSON-Objekt selbst valide ist und die neue Konfiguration auf dem Server angelegt wurde, teilt dieser den Erfolg wiederum in Form eines JSON-Objektes mit, zu sehen in Abbildung 5.

```
{
  "success": true,
  "id": 5
}
```

Abbildung 5: Antwort des Servers nach erfolgreicher POST-Operation

Zusätzlich zum Erfolgsattribut sendet er noch die ID des neu angelegten Datensatzes, welche die Synchronisation des Datenstandes zwischen Client und Server erleichtert. Falls die Authentifikation des Cookies fehlschlägt, der Benutzer nicht berechtigt ist oder ein anderer serverseitiger Fehler auftritt, würde das `success`-Attribut `false` sein und ein zusätzliches Attribut `error` eine Fehlermeldung beinhalten.

7.1.2 Domänenmodell

Beim HoneySens-Server handelt es sich um eine typische CRUD-Anwendung, deren Hauptaufgabe das Speichern, Verändern und Ausgeben von Daten ist. Diese könnten in Form von Textdateien im Dateisystem gespeichert werden, was jedoch im Falle von Mehrbenutzersystemen – wie HoneySens eines ist – Probleme mit vielen parallelen Zugriffen verursachen kann. Speziell bei größeren Datenmengen und der Notwendigkeit von *Transaktionen* ist der Einsatz von Datenbanktechnologien sinnvoll. Im Rahmen dieser Arbeit wurde die Einbindung einer *relationalen Datenbank* angestrebt, die *ACID*-Eigenschaften sicherstellt (**A**tomicity, **C**onsistency, **I**solation, **D**urability) [17]. Die Verantwortung der sicheren, konsistenten Datenhaltung ist somit Aufgabe einer separaten Instanz und muss nicht von Grund auf neu entwickelt werden. In relationalen Datenbanken wird zumeist die Sprache *SQL* sowohl zur Abfrage und Modifikation von Daten als auch zur Definition der anwendungsspezifischen Datenstrukturen eingesetzt. Dieser Abschnitt soll nun das für den Prototypen auf Basis der API-Definition basierende Domänenmodell und die damit verbundenen Implikationen vorstellen. Es dient später als Vorbild sowohl für das relationale Datenmodell als auch die Klassenstruktur der Anwendung auf Server- und Clientseite.

Die Anwendungsdomäne, primär bestehend aus Sensoren, deren Konfiguration und von ihnen ausgewerteten Ereignissen wurde bereits während der Konzeption in Kapitel 6 beschrieben. An dieser Stelle sollen deshalb die Beziehungen dieser Entitäten zueinander und ihre jeweils spezifischen Eigenschaften näher beschrieben werden. Einen Überblick über das Gesamtmodell gibt Abbildung 6.

Auf den **Sensoren** liegt das Hauptaugenmerk der physischen Architektur, weshalb diese auch im theoretischen Domänenmodell eine zentrale Rolle einnehmen. Jeder Sensor besitzt einen *einzigartigen Namen* – es muss sichergestellt werden, dass dieser nicht mehrfach im System vorhanden ist. Grund dafür ist, dass den physischen Sensoren für die Authentifikation beim Server nur ihr jeweiliger Name bekannt ist, nicht ihre ID in der Datenbank. Weiterhin wird er als *Common Name* für die sensorspezifischen Zertifikate verwendet, die zum Signieren aller gesendeten Daten benutzt werden. Falls ein Name doppelt vergeben werden würde, wäre für den Server nicht mehr ersichtlich, welchem der beiden Sensoren empfangene Daten zuzuordnen wären. Das Attribut *location* gibt den Ort an, an dem ein Sensor platziert wurde und stellt lediglich eine Hilfe für die das System nutzenden Administratoren dar. Es hat keinen weiteren Einfluss auf die Funktionalität des Gesamtsystems, kann jedoch bei einer großen Menge von Sensoren zum Auffinden derselben bei Problemen hilfreich sein.

Jeder Sensor besitzt ein individuelles Schlüsselpaar bestehend aus privatem Schlüssel und öffentlichem **Zertifikat** (*Cert*). Die privaten Schlüssel sind geheim und werden deswegen ausnahmslos auf den Sensoren hinterlegt, serverseitig wird hingegen nur das öffentliche Zertifikat gespeichert, um die Signaturen in den Nachrichten der Sensoren auf Gültigkeit überprüfen zu können. Somit besitzt die Entität *Cert* nur ein einziges Attribut *content*, das den unverschlüsselten Inhalt des öffentlichen Schlüssels als Zeichenkette aufnimmt.

Sensorspezifische Einstellungen repräsentiert die Entität **Konfiguration** (*Config*). Sie umfasst vorrangig eine Reihe von booleschen Werten, die jeweils einen betriebsspezifischen Dienst und dessen Status repräsentieren. Das Flag *passiveScan* steuert den sogenannten **Passive Scan Mode**, in dem alle eintreffenden TCP- und UDP-Pakete aufgezeichnet und an den Server übermittelt werden. Das zuständige Skript erzeugt Ereignisse auf dem Server und nimmt auch deren Klassifikation vor. Ebenfalls Bestandteil dieses Dienstes ist zudem eine rudimentäre Portscan-Erkennungsroutine, die eine große Zahl von empfangenen Paketen eines einzelnen Quell-Hosts zu einem einzelnen *Scan*-Ereignis zusammenfasst und

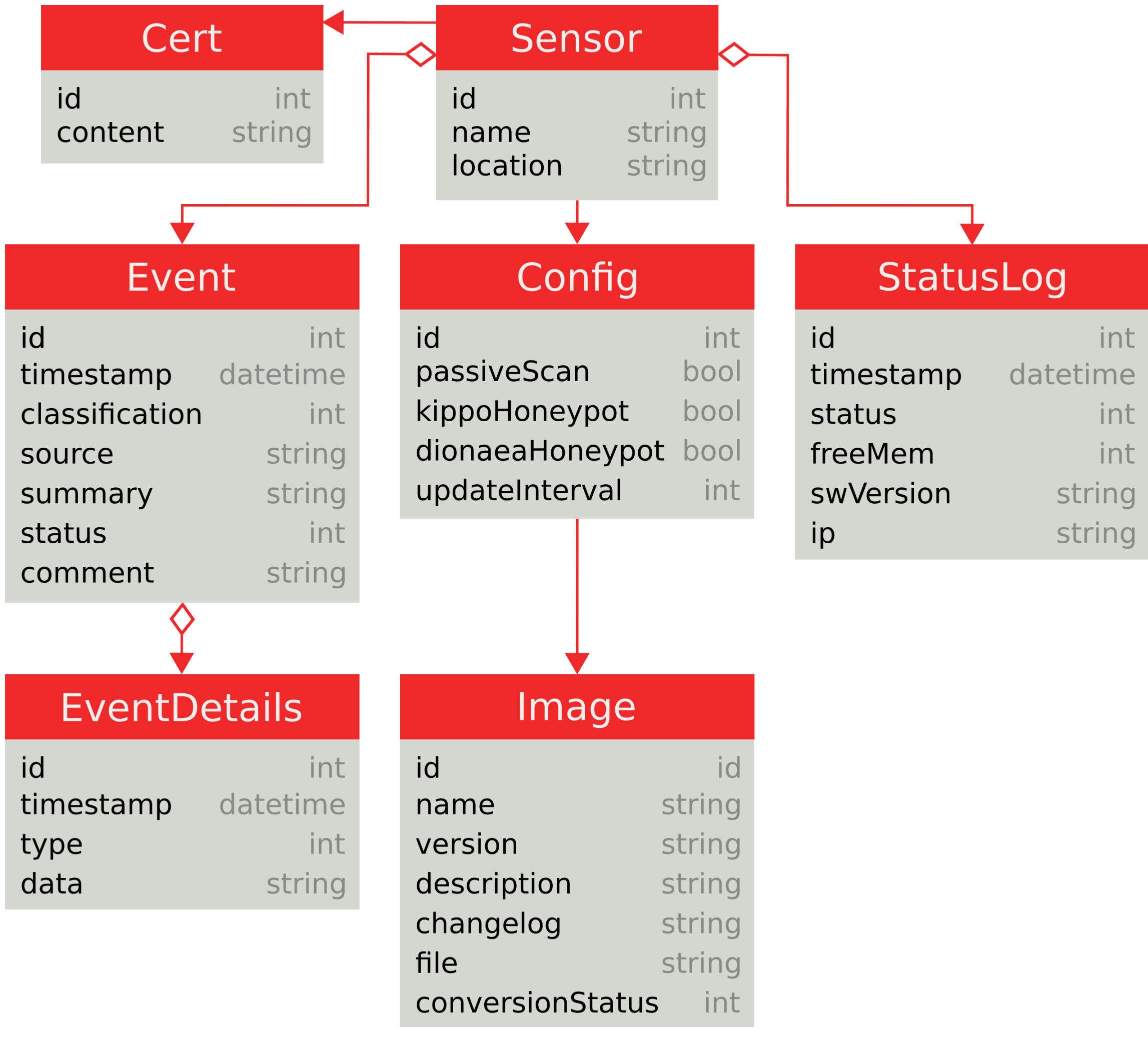


Abbildung 6: Das HoneySens-Domänenmodell (Teil 1)

verhindern soll, dass der Server mit Einzelereignissen überflutet wird. Die Flags *kippoHoneypot* und *dionaeaHoneypot* aktivieren bzw. deaktivieren Honeypot-Services, die konkrete Netzwerkdienste anbieten. Die Software *kippo* emuliert das SSH-Protokoll, *dionaea* ist hingegen im HoneySens-Prototypen für die Repräsentation der CIFS/SMB-Protokollfamilie samt typischer Sicherheitslücken zuständig. Weiterhin umfasst eine Sensorkonfiguration noch das Attribut *updateInterval*, das den Zeitraum zwischen zwei Update-Meldungen an den Server festlegt. Der Wert wird in Minuten spezifiziert – je größer er ausfällt, desto länger dauert es, bis zukünftige Konfigurationsänderungen übernommen werden. Gleichzeitig reduziert dies jedoch die Auslastung des zugrundeliegenden Netzwerks, was zur Vermeidung von Lastspitzen wünschenswert sein kann und im Zuge der Transparenz in der Anforderungsanalyse auch gefordert ist.

Jede Sensorkonfiguration verweist zudem noch auf ein **Firmware-Image**, im Modell repräsentiert durch die Entität *Image*. Sie spezifiziert für jeden Sensor die zu nutzende Firmware-Revision. Falls diese sich von der lokal installierten Sensorfirmware unterscheidet, nimmt er ein vollautomatisches Update vor. Jede Firmware besteht aus dem Image (einem Partitionsabbild mit Sensorsoftware) und einer Reihe zugehöriger Metadaten. All diese Daten werden dem Server in Form eines komprimierten Archivs zur Verfügung gestellt, die Extraktion der einzelnen Bestandteile und das Erzeugen einer Image-Ressource fällt daher

ebenfalls in dessen Aufgabenbereich. Ein *Image* im Kontext des Domänenmodells besitzt einen firmwarespezifischen Namen, einen Versionsstring sowie eine knappe Beschreibung. Das Attribut *changelog* beinhaltet einen umfangreichen String, der die Versionshistorie dieser und älterer Softwarerevisionen beschreibt. Von großer Wichtigkeit ist zudem noch das Attribut *file*, das den Namen angibt, unter dem die Firmware im Dateisystem des Servers abgelegt wurde – das Partitionsabbild wird nämlich nicht in der Datenbank gespeichert, da es typischerweise mehrere hundert Megabyte groß und deshalb für den Einsatz in einer relationalen Datenbank ungeeignet ist [35].

Für die Installation eines neuen Sensors ist ein Installationsmedium erforderlich. Das generische komprimierte Firmware-Archiv, wie es zuvor beschrieben wurde, ist hierfür allerdings ungeeignet, da es in dieser Form nicht direkt auf beispielsweise eine Speicherkarte geschrieben und ein System davon gestartet werden kann. Es wird allerdings für Sensor-Updates benötigt. Folglich ist es erforderlich, die Firmwaredaten auf dem Server in ein bootbares Format zu konvertieren. Für das Domänenmodell resultiert aus dieser Anforderung das Attribut *conversionStatus*, das für jedes Firmware-Image angibt, ob die Konvertierung bereits begonnen oder schon abgeschlossen wurde. Tabelle 5 gibt Aufschluss über die möglichen Zustände, die im Modell als Integer-Wert gespeichert werden.

Wert	**Symbol**	**Bedeutung**
0	`CONVERSION_UNDONE`	Konvertierung noch nicht in Auftrag gegeben
1	`CONVERSION_SCHEDULED`	Firmware in Warteschlange zur Konvertierung eingereiht
2	`CONVERSION_RUNNING`	Konvertierung läuft gerade
3	`CONVERSION_DONE`	Konvertierung in bootbares Medium ist abgeschlossen, resultierendes Image liegt vor

Tabelle 5: Übersicht der conversionStatus-Attributwerte

Die in gleichmäßigen Abständen von den Sensoren an den HoneySens-Server gesendeten **Statusnachrichten** werden im Domänenmodell durch die Entität *StatusLog* repräsentiert. Sie verweist auf den zugehörigen Sensor, der wiederum alle ihm eigenen Log-Objekte aggregiert. Ein StatusLog-Eintrag besitzt zunächst immer einen Zeitstempel im Attribut *timestamp*, der den Zeitpunkt des Eintreffens der entsprechenden Zustandsdaten am Server festhält. Dessen konkreter Datentyp bei der Umsetzung des Modells in einer Datenbank oder als objektorientiertes Klassenmodell ist flexibel und von den eingesetzten Technologien abhängig. In der einfachsten Form kann das Attribut als simpler UNIX-Zeitstempel[27] gespeichert werden. Das *status*-Attribut speichert in Form eines ganzzahligen Werts den aktuellen Sensor-Betriebszustand. Es wird genutzt, um einen sich gerade im Gange befindlichen Installations- oder Updateprozess, aber auch einen eventuell undefinierten Zustand anzuzeigen. Details zum Update und zur Neuinstallation werden im entsprechenden Abschnitt besprochen. Einen Überblick über alle möglichen Attributwerte gibt Tabelle 6.

Jede Statusnachricht informiert den Server weiterhin über aktuelle Systemeigenschaften. Im HoneySens-Prototypen beschränkt sich dies exemplarisch auf die Menge des noch freien Arbeitsspeichers in Megabyte (Attribut *freeMem*, ganzzahlig), was exemplarisch zur Detektion eventueller Speicherüberläufe der Sensorsoftware genutzt werden kann. Eine Erweiterung um zusätzliche Attribute – denkbar wären die Auslastung der CPU oder des Netzwerk-Interfaces – ist jedoch ohne weiteres möglich. Das Attribut *swVersion* gibt zudem Auskunft über die aktuell eingesetzte Firmware-Revision, bestehend aus Namen und Versionsstring, während *ip* dem Server die aktuelle IP-Adresse des primären Netzwerkinterfaces als Zeichenkette mitteilt – jene Adresse, über die der Honeypot im Netzwerk zu erreichen ist.

[27] Ganzzahliger Wert, der die seit dem 01.01.1970 vergangenen Sekunden angibt

Wert	Symbol	Bedeutung
0	STATUS_ERROR	Sensor befindet sich in undefiniertem Zustand, in der Regel Folge eines unerwarteten Fehlers
1	STATUS_RUNNING	Sensor ist einsatzbereit
2	STATUS_UPDATE_PHASE1	Erste Phase des zweistufigen Firmware-Aktualisierungsprozessses läuft
3	STATUS_INSTALL_PHASE1	Erste Phase des zweistufigen Installationsprozesses läuft
4	STATUS_UPDATEINSTALL_PHASE2	Zweite Update-/Installationsphase läuft. Diese ist in beiden Fällen identisch und wird somit in einem Wert zusammengefasst

Tabelle 6: Übersicht der conversionStatus-Attributwerte

Die letzte mit den Sensoren in Beziehung stehende Entität ist das **Ereignis** (*Event*), welches einen sicherheitsrelevanten Vorfall an einem Sensor beschreibt. Ereignisdaten werden von den Sensoren selbst gesammelt, interpretiert und in kompakter Form mittels der REST-API beim Server registriert. Durch diesen Mechanismus können eintreffende Pakete von unbekannten Quellen, die einen Vorfall darstellen, nicht erst vollständig an den Server weitergeleitet, sondern direkt vom Sensor bearbeitet werden. Dies resultiert in einer geringeren Antwortzeit der Sensoren, die sich im Idealfall anderen im gleichen Netzsegment betriebenen Rechnern annähert. Jedes Ereignis verweist auf den Sensor, von dem es aufgezeichnet wurde, und besitzt wieder einen *timestamp*, der den Zeitpunkt, an dem es eingetreten ist, eindeutig benennt. Falls mehrere Pakete zu einem einzelnen Ereignis zusammengefasst wurden (beispielsweise einem Portscan), gilt der Zeitpunkt des ersten eingetroffenen Pakets. Analog zu den Statusnachrichten hängt die konkrete Implementierung des Attributs von den die Daten sammelnden Diensten ab. Wie dies im Prototypen realisiert ist, wird in Kapitel 7.1.3, *Implementierung*, besprochen.

Ein Ereignis resultiert also aus einem oder mehreren empfangenen Netzwerkpaketen und besitzt deshalb ein Attribut *source*, das dessen bzw. deren Quell-IP-Adresse als Zeichenkette spezifiziert. Ereignisse von unterschiedlichen Quell-Hosts werden im Prototypen immer separat betrachtet und daher nicht zusammengefasst[28]. Das Feld *summary* beinhaltet eine informelle Zusammenfassung des betreffenden Ereignisses, die ebenfalls direkt vom Sensor aus den aufgezeichneten Paketdaten generiert wird. Für den String gibt es keine Formatvorgaben, er umfasst im Prototypen typischerweise Protokollinformationen (z.B. TCP/UDP) und den kontaktierten Port. Von essentieller Bedeutung in der Ereignisverwaltung ist zudem die Klassifikation durch das Feld *classification*. Sie wird ebenfalls sensorseitig vorgenommen und als ganzzahliger Wert in der Datenbank gespeichert. Alle möglichen Werte listet Tabelle 7 auf.

Ein jedes Ereignis besitzt weiterhin noch die Felder *status* und *comment*, die als Unterstützung für die Administratoren des HoneySens-Systems gedacht sind und keinen Einfluss auf die Funktionalität des Sensornetzes haben. Das Attribut *comment* enthält hierfür einen Kommentartext für das jeweilige Ereignis, der von den Benutzern des Systems beliebig ausgelesen oder geändert werden kann. Angedachter Einsatzzweck des Feldes sind Notizen über den Bearbeitungsprozess des Vorfalls, beispielsweise gewonnene Informationen über den Verursacher. Das status-Attribut nimmt wiederum einen ganzzahligen Wert entgegen, der einen von vier vordefinierten Statuswerten annehmen kann, die in Tabelle 8 beschrieben sind. Eine auf der API basierende Client-Anwendung könnte Ereignisse anhand des Feldes sortieren und somit beispielsweise „erledigte“ Ereignisse, deren Ursache geklärt und behoben ist, ausblenden.

[28]Dies könnte beispielsweise zur DDoS-Erkennung nützlich sein und stellt eine Möglichkeit für die Weiterentwicklung dar

Wert	Symbol	Bedeutung
0	`CLASSIFICATION_UNKNOWN`	Fallback, der zum Einsatz kommt, falls ein Ereignis nicht zugeordnet werden kann.
1	`CLASSIFICATION_ICMP`	ICMP-Traffic, in der Regel Echo-Requests. Wird im Prototypen nicht aufgezeichnet, da ICMP im SVN vorkommen kann und typischerweise keine Gefahr darstellt.
2	`CLASSIFICATION_CONN_ATTEMPT`	Verbindungsversuch an einem TCP- oder UDP-Port, auf dem kein Honeypot-Dienst läuft.
3	`CLASSIFICATION_LOW_HP`	Von einem Low-Interaction-Honeypot aufgezeichnetes Ereignis.
4	`CLASSIFICATION_PORTSCAN`	Portscan, hervorgerufen durch viele aufeinanderfolgende Anfragen von einer einzigen Quell-IP-Adresse. Die Parameter für diese Klassifikation sind implementierungsspezifisch.

Tabelle 7: Übersicht der classification-Attributwerte

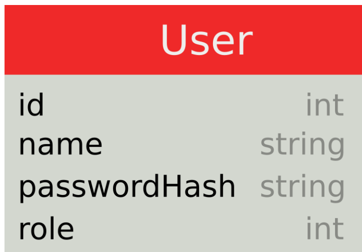
Wert	Symbol	Bedeutung
0	`STATUS_UNEDITED`	Jedes neu registrierte Ereignis besitzt zunächst diesen Status, der anzeigt, dass noch keine Nachforschung nach der Ursache des Vorfalls stattgefunden hat.
1	`STATUS_BUSY`	Zeigt an, dass die Ursache des Vorfalls im Moment untersucht wird.
2	`STATUS_RESOLVED`	Markiert erledigte Ereignisse, gewonnene Informationen über den Vorfall könnten im Kommentarfeld stehen.
3	`STATUS_IGNORED`	Bezeichnet Ereignisse, die keine Untersuchung erfordern. Darunter könnten Vorfälle fallen, die zum Testen einzelner Sensoren absichtlich erzeugt wurden.

Tabelle 8: Übersicht der status-Attributwerte eines Ereignisses

Die bereits beschriebenen Attribute reichen unter Umständen nicht aus, um ein Ereignis in allen nötigen Facetten zu beschreiben. Um diesem Umstand Abhilfe zu schaffen, existiert die Entität *Event-Details*, welche zusätzliche **Ereignisdetails** speichert. Für jedes Ereignis können hierbei beliebig viele Zusatzinformationen aggregiert werden. Es existieren grundsätzlich zwei verschiedene Arten derartiger Objekte, deren Unterscheidung mit Hilfe des ganzzahligen Attributs *type* erfolgt. Details vom Typ `TYPE_INTERACTION` beschreiben die Interaktion eines potentiellen Angreifers mit einem Sensor genauer, während die des Typs `TYPE_GENERIC` alle weiteren im Rahmen des Vorfalls gewonnenen Informationen beinhalten. Dazu könnten beispielsweise das vermutete Betriebssystem des Angreifers oder die Eigenschaften eines möglicherweise ausgenutzten Exploits zählen. Welche Daten letztendlich gespeichert werden, liegt jedoch im Verantwortungsbereich der Sensoren und der darauf betriebenen Software, da Ereignisdetails direkt zusammen mit Ereignissen über die API gespeichert werden.

EventDetails-Objekte sind schlank gehalten: Allen solchen Ressourcen gemein ist das Attribut *data*, welches beliebige Informationen als Zeichenkette speichert. Deren Format und Art sind von der Daten erzeugenden Software (Passive Scan, Honeypot etc.) abhängig. Falls die vorliegende Ressource Interaktionen des Angreifers beschreibt, benennt das Attribut *timestamp* zusätzlich noch den zugehörigen

exakten Zeitpunkt. Die Syntax des Attributs folgt den im Zusammenhang mit den Entitäten *StatusLog* und *Event* bereits beschriebenen Regeln. Die schrittweise Interaktion eines Angreifers mit einem Honeypot kann also mit Hilfe der Ereignisdetails genau nachvollzogen werden, ohne gesondert in die Protokolle der jeweiligen Honeypot-Software auf dem Sensor Einsicht nehmen zu müssen.



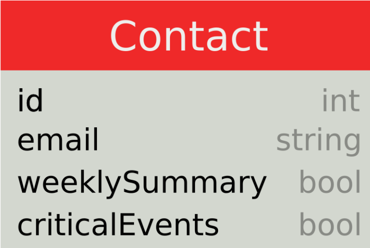


Abbildung 7: Das HoneySens-Domänenmodell (Teil 2)

Das Domänenmodell des HoneySens-Prototypen besitzt noch zwei weitere Entitäten, die nicht in direktem Zusammenhang mit den Sensoren stehen und in Abbildung 7 graphisch dokumentiert sind. Ein *User* beschreibt in diesem Zusammenhang einen **Benutzer** des HoneySens-Systems, der sich wie im vorherigen Abschnitt beschrieben über die *Session-Ressource* der API authentifizieren muss, um alle weitergehenden Funktionalitäten nutzen zu können. Zu diesem Zweck besitzt jeder Nutzer einen *Namen* und ein *Passwort*, dessen Hash als Zeichenkette in der Datenbank gespeichert wird. Zusätzlich gehört er einer von drei Gruppen an, die seine Zugriffsrechte in Bezug auf Ressourcen und den darauf ausführbaren Operationen regeln. Die aktive Gruppe eines Nutzers wird im Attribut *role* vermerkt, Details hierzu visualisiert Tabelle 9.

Wert	Symbol	Bedeutung
0	`ROLE_GUEST`	Gäste besitzen keinerlei Zugriffsrechte außer der Session-Ressource selbst, um sich zu authentifizieren. Die Gruppe wird als Standard für alle (noch) nicht authentifizierten Benutzer verwendet.
1	`ROLE_OBSERVER`	Darf bis auf die vorhandenen Benutzer alle Ressourcen im Backend einsehen, aber keine Veränderungen vornehmen.
2	`ROLE_MANAGER`	Kann abgesehen von Nutzeraccounts alle Ressourcen einsehen und verändern.
3	`ROLE_ADMIN`	Beinhaltet alle Rechte im System, insbesondere auch die Benutzerverwaltung.

Tabelle 9: Auflistung aller Benutzergruppen

Zuletzt soll der Blick noch auf die Entität *Contact* gerichtet werden, die einen **Kontakt** für E-Mail-Benachrichtigungen symbolisiert. Der HoneySens-Prototyp erlaubt es, einen SMTP-Server samt zugehörigem Account zu bestimmen, um das Senden von wöchentlichen Zusammenfassungen und Sofortbenachrichtigungen bei kritischen Ereignissen via E-Mail zu ermöglichen. HoneySens unterstützt weiterhin beliebig viele Zieladressen für derartige Nachrichten, jede davon durch ein entsprechendes Kontaktobjekt repräsentiert. Ein Kontakt hält eine E-Mail-Adresse als Zeichenkette vor, an die alle Nachrichten zu versenden sind. Zusätzlich kann für jede Adresse noch definiert werden, welche Arten von Benachrichtigungen gesendet werden sollen. Dies erfolgt mit den booleschen Attributen *weeklySummary* und *criticalEvents*, die beide optional sind. Sobald das System Nachrichten generiert, werden

alle Kontakte, die zum entsprechenden Typ passen – wöchentliche Zusammenfassung oder kritisches Ereignis – über E-Mail informiert. Die Einordnung von Ereignissen als „kritisch“ wird im Prototypen anhand des *classification*-Attributs vorgenommen: Jede Kontaktaufnahme zur laufenden Honeypot-Software (`CLASSIFICATION_LOW_HP`) und Portscans (`CLASSIFICATION_PORTSCAN`) fallen in diese Kategorie.

7.1.3 Implementierung

Der Entwicklung der prototypischen REST-API ging die Auswahl eines geeigneten Softwarestacks voraus. Dieser umfasst primär einen HTTP-Server, eine Datenbankimplementierung, eine serverseitige Programmiersprache, Frameworks zur Strukturierung der Anwendung und zusätzliche Bibliotheken für weitergehende Funktionalitäten wie Mailversand, Kryptographie oder Datenbankabstraktion. Die endgültige Entscheidung beeinflussten eine Vielzahl von Faktoren, darunter die Beschränkung auf quelloffene, freie Software (Open-Source-Software, kurz *OSS*), der Wunsch nach Standardkonformität in der Anforderungsanalyse und die bereits gesammelte Erfahrung des Entwicklers mit bestimmten Technologien. Der nachfolgende Abschnitt gibt einen Überblick über die ausgewählten Softwareprodukte und die Gründe für deren Einsatz.

Betriebssystem Open-Source-Software ist im Rahmen des Projektes erwünscht, um rechtliche Thematiken wie die Beschaffung von Lizenzen zu vereinfachen, gleichzeitig aber auch die Möglichkeit der Veränderung und Erweiterung der Software selbst als Option offen zu halten. Unter dieser Voraussetzung kommen bei der Auswahl eines geeigneten serverseitigen Betriebssystems hauptsächlich freie Varianten mit *Linux*- oder *BSD*-Kernel in Frage. Tatsächlich sind alle nachfolgend beschriebenen Softwarekomponenten mit diesen kompatibel. Einige für den Betrieb der API notwendigen Skripte setzen jedoch zwingend ein Linux-System voraus und sind auf die Funktionalität solcher Kernel zwingend angewiesen. Eine Portierung des betreffenden Quellcodes auf BSD-Systeme ist jedoch denkbar. HoneySens stellt keine konkreten Anforderungen hinsichtlich bestimmter Linux-Distributionen. Es ist deshalb im Rahmen dieser Arbeit ein Deployment-Verfahren entwickelt worden, das Installation und Betrieb des Systems distributionsunabhängig gestaltet. Details hierzu liefert Kapitel 7.3.

Web-Server Die REST-API ist per Definition webbasiert und soll über das HTTP-Protokoll und dessen mit TLS erweiterte, verschlüsselte Variante HTTPS nutzbar sein, was den Betrieb eines Web-Servers erfordert. Traditionell lieferten diese lediglich statische HTML-Seiten und CSS-Stylesheets aus, mit dem Aufkommen dynamischer Web-Technologien wie beispielsweise *AJAX*[29] etablierte sich allerdings auch die Übertragung anderer Formate zum Datenaustausch, darunter *XML* und das zuvor bereits beschriebene *JSON*. Da die Interpretation solcher Daten aber anwendungsspezifisch ist und keine zusätzlichen Anforderungen an die Serversoftware selbst stellt, da diese lediglich als Schnittstelle zwischen der Anwendung und den Clients durch die Bereitstellung des HTTP-Protokolls fungiert, gibt es für die Entwicklung des Prototypen nur wenige Einschränkungen bei der Softwareauswahl.

Um möglichst standardkonform zu bleiben, wurden im Rahmen dieser Arbeit lediglich die größten und bekanntesten Web-Server in Betracht gezogen. Diese besitzen große Entwicklergemeinschaften, wer-

[29] Asynchronous JavaScript and XML

den aktiv weiterentwickelt und sind durch ihren verbreiteten Einsatz hinreichend erprobt, was Sicherheit und Stabilität betrifft [27]. Die tatsächliche Verbreitung ist nur schwer zu messen und hängt von vielerlei Faktoren ab, beim quantitativen Ranking der verfügbaren Lösungen herrscht jedoch Einigkeit: Der von der *Apache Software Foundation*[30] entwickelte *Apache HTTP Server*, zum Zeitpunkt dieser Arbeit in Version 2.4.10 verfügbar, genießt weltweit die größte Verbreitung, trotz eines seit Jahren fallenden Marktanteils [8]. Die von Microsoft entwickelte Software *Internet Information Services* hat zuletzt an Popularität gewonnen, ist jedoch proprietärer Natur und auf das *Windows*-Betriebssystem angewiesen. In die engere Wahl für das Entwicklungssystem aufgenommen wurden schließlich noch die Programme *nginx*[31] und *Lighttpd*[32], welche genau wie Apache die Ansprüche an freie Software erfüllen. Sie sind für unixoide Betriebssysteme, wie sie für HoneySens serverseitig eingesetzt werden, verfügbar und vergleichsweise weit verbreitet [12], wobei nginx häufig auch als reiner Proxy und Load-Balancer vor anderen Web-Servern betrieben wird. Um die im Rahmen der Anforderungsanalyse geforderte Standardkonformität und Integrierbarkeit in bestehende Netze zu berücksichtigen, wurde bei der Entwicklung des HoneySens-Prototypen darauf geachtet, die Anwendung weitestgehend unabhängig vom eingesetzten Web-Server zu halten und somit einen eventuellen Austausch dieser Komponente zu einem späteren Zeitpunkt zu vereinfachen. Aufgrund der im Vorfeld bereits vorhandenen Erfahrungen des Entwicklers und der großen Verbreitung der Software wurde für das Entwicklungssystem und das später beschriebene Deployment-Verfahren letztendlich der Apache HTTP Server ausgewählt.

Datenbank Die Konzeption verlangt den Einsatz einer relationalen Datenbank, um alle serverseitigen Daten abzüglich der Binärdateien (Firmware, Installationsabbilder etc.) zu speichern. Typische freie Software für diesen Zweck sind *PostgreSQL*[33], *Firebird*[34] oder *MySQL*[35] bzw. dessen Fork *MariaDB*[36], der in einigen Linux-Distributionen das von *Oracle* übernommene MySQL ersetzt hat. Eine ausführliche Evaluation aller Lösungen ist an dieser Stelle aufgrund des gewaltigen Umfangs nicht möglich, aber auch nicht zwingend notwendig. Alle genannten Datenbanksysteme erfüllen die vergleichsweise geringen Anforderungen, die die HoneySens-Architektur stellt. Die typischerweise im Zusammenhang mit komplexen Geschäftsprozessen genutzten speziellen Features dieser Produkte werden im Rahmen einer simplen CRUD-Anwendung nicht benötigt. Auch die Skalierbarkeit ist kein entscheidendes Kriterium, da die erwartete durch HoneySens erzeugte Last deutlich geringer ausfällt als die einer frequentierten Website, für die derartige Datenbanken entworfen wurden. Aus diesen Gründen war bei der Auswahl schließlich entscheidend, dass der Autor dieser Arbeit mit MySQL bzw. dem fast identischen MariaDB bisher die meiste Erfahrung gesammelt hatte. Um jedoch zukünftigen Nutzern des Prototypen, die innerhalb ihrer bereits etablierten Infrastruktur möglicherweise auf andere Lösungen setzen, mehr Wahlfreiheit zu lassen, erfolgt jegliche Kommunikation mit der Datenbank über eine Abstraktionsschicht (*Object Relational Mapper*), die im Anschluss detailliert beschrieben wird. Somit ist die REST-API letztendlich unabhängig von der konkret eingesetzten Datenbank-Implementierung; MySQL bzw. MariaDB kommen ausschließlich bei der Entwicklung der Software und dem am Ende des Kapitels beschriebenen optionalen Deployment-Verfahren zum Einsatz.

[30] `http://apache.org` (abgerufen im Oktober 2014)
[31] `http://nginx.org` (abgerufen im Oktober 2014)
[32] `http://lighttpd.net` (abgerufen im Oktober 2014)
[33] `http://postgresql.org` (abgerufen im Oktober 2014)
[34] `http://firebirdsql.org` (abgerufen im Oktober 2014)
[35] `http://mysql.com` (abgerufen im Oktober 2014)
[36] `http://mariadb.org` (abgerufen im Oktober 2014)

Job-Server Der HoneySens-Server erfüllt neben der Bewältigung typischer CRUD-Tätigkeiten über die API noch einige weitergehende Aufgaben, darunter die Konvertierung der Firmware-Abbilder in bootbare Speicherkartenabbilder und die Generierung von Konfigurationsarchiven, die zur Initialisierung neuer Sensoren genutzt werden können. Derart komplexe Funktionen können mehrere Minuten bis – je nach Performance des Serversystems – Stunden dauern und übersteigen somit die typische Antwortzeit, die ein Client von einem Web-Server erwartet. Es ist daher sinnvoll, langanhaltende Aufgaben parallel im Hintergrund auszuführen und Nutzern des Prototypen somit auch die Möglichkeit zu geben, weiterhin mit dem Frontend zu interagieren. Dies hat zudem den Vorteil, dass die Ergebnisse eines solchen lang andauernden Tasks erhalten bleiben, selbst wenn ein Anwender zwischenzeitlich die Verbindung verliert.

Im Prototypen wird *Beanstalkd*[37] eingesetzt, ein sogenannter „Job-Server", der Warteschlangen für Aufgaben anbietet, die von Konsumenten (als separate Prozesse) ausgeführt werden. Die konkrete Implementierung dieser Aufgaben obliegt dabei dem Entwickler, der dabei eine große Auswahl an zugehörigen Bibliotheken für eine Vielzahl von Programmiersprachen hat. Die Kommunikation mit dem Beanstalkd-Prozess erfolgt ausschließlich über einen lokalen Netzwerksocket, der nach Möglichkeit nicht extern im Netzwerk ansprechbar sein sollte, um die Beeinflussung der Konsumentenprozesse durch potentielle Angreifer zu vermeiden. Zu diesem Zweck empfiehlt sich die Beschränkung der Kommunikation auf das lokale Loopback-Interface des Serversystems, das über das Netzwerk nicht erreichbar ist. Die mit dem Service interagierenden Komponenten, nämlich der Web-Server und die Konsumentenprozesse, laufen schließlich auf demselben Host, was diese Sicherheitsmaßnahme legitimiert. Weiterhin ist Beanstalkd sehr leichtgewichtig und hat nur wenige Abhängigkeiten, weshalb ein Kompilieren der Software auf unixoiden Systemen, die sie nicht über ihr Paketmanagement anbieten, keine Herausforderung darstellt. Das Programm ist außerdem sehr ressourcenschonend und bietet nur die absolut notwendigen Funktionen zur Verwaltung einer Task-Warteschlange an, was die Skalierbarkeit und Performance des HoneySens-Prototypen begünstigt.

Programmiersprachen Es existiert eine Vielzahl von Programmiersprachen zur Implementierung serverseitiger Anwendungen, weshalb die Auswahl einer solchen typischerweise weniger von der Sprache selbst, als vielmehr von den gesammelten Erfahrungen der Entwickler und dem vorhandenen „Ökosystem" im Sinne von Bibliotheken und hilfreichen Tools abhängt. HoneySens setzt aus den genannten Gründen serverseitig auf zwei Sprachen zugleich: Primär *PHP*[38] zur Implementierung der API selbst und an einigen Stellen außerdem noch *Python*[39], vorrangig in den systemnah arbeitenden Beanstalkd-Jobs. Die seit Version 5 vollständig objektorientierte Skriptsprache PHP ist zur Erstellung dynamischer Webauftritte entworfen worden und somit auch zur Entwicklung einer API geeignet. Sie besitzt bereits im Auslieferungszustand eine umfangreiche *Standard PHP Library (SPL)* genannte Klassenbibliothek und eine Reihe von Erweiterungen für häufig benötigte Funktionalität. Dazu gehört praktischerweise auch die Anbindung an populäre Datenbanksysteme, darunter das zur Entwicklung genutzte MySQL, über eine Abstraktionsschicht namens *PHP Data Objects* (PDO). Trotz des großen Standard-Funktionsumfanges ergaben sich im Laufe des Prototypings weitere Anforderungen, die nur mit zusätzlichen externen Bibliotheken zufriedenstellend erfüllt werden konnten. Diese werden am Ende des Kapitels vorgestellt.

Python ist eine universell einsetzbare Skriptsprache mit einer ebenfalls sehr umfangreichen Standardbibliothek. Sie wird vom HoneySens-Server für Skripte genutzt, die Shell-Befehle wie beispielsweise *GNU*

[37] `http://kr.github.io/beanstalkd/` (abgerufen im Oktober 2014)
[38] `http://php.net` (abgerufen im Oktober 2014)
[39] `http://python.org` (abgerufen im Oktober 2014)

`tar` zum Packen und Extrahieren von Archiven benötigen und direkt mit dem Dateisystem interagieren. Derartiges wäre ebenfalls in PHP mit Hilfe der Funtionen `exec()` oder `proc_open()` realisierbar, Python bietet in diesem Zusammenhang allerdings mehr Komfortfunktionen und *Syntactic Sugar*, wodurch beispielsweise der Umgang mit Arrays und Strings dank sogenannter *Slicing*-Operationen leichter von der Hand geht. Das `subprocess`-Modul erlaubt weiterhin die direkte Manipulation der Standard-Datenströme `stdin`, `stdout` und `stderr`, was im Zusammenspiel mit CLI-Anwendungen äußerst hilfreich ist. Die umfangreiche Standardbibliothek genügte fast allen gestellten Anforderungen, lediglich für die Verbindung zum Beanstalkd-Job-Server und zur MySQL-Datenbank waren externe Bibliotheken nötig: *beanstalkc*[40] und *PyMySQL*[41]. Im Vergleich zu PHP überzeugt Python insbesondere durch eine leichter lesbare Syntax, da Code-Blöcke nur durch ihre Einrückung gekennzeichnet werden. Die hierfür in PHP genutzten geschweiften Klammern erlauben aber wiederum eine übersichtlichere Integration von serverseitig ausführbarem Code in HTML-Templates, wovon bei der Entwicklung des Prototypen Gebrauch gemacht wurde.

Anwendungsstruktur Für die Implementierung der REST-API in PHP wurde das Framework *Slim*[42] eingesetzt, welches das Aufsetzen der für die Entwicklung einer neuen Anwendung nötigen Infrastruktur vereinfacht und ein schnelleres Prototyping ermöglicht. Slim übernimmt für die API im Wesentlichen das Mapping von Request-URIs wie `/api/sensors/config/3` auf die Anfragen beantwortenden Controller-Methoden, die Session-Verwaltung mit HTTP-Cookies und eine rudimentäre Fehlerbehandlung. Ein Blick auf die Verzeichnisstruktur der Serveranwendung gibt Aufschluss über alle beteiligten Komponenten, zu sehen in Abbildung 8. Sie orientiert sich stark an der empfohlenen Verzeichnisstruktur des *Zend Frameworks* [13], welches für das vorliegende Projekt zu umfangreich ist, mit dem der Autor im Vorfeld aber bereits Erfahrung gesammelt hatte.

Alle Anfragen an die REST-API werden zunächst an das PHP-Skript `index.php` im `public`-Verzeichnis weitergeleitet, welches die Initialisierung aller Anwendungskomponenten übernimmt und anschließend den *Dispatch*-Vorgang startet, bei dem die Bearbeitung der Aufgabe an den für die URI zuständigen Controller übergeben wird. Dass bei wirklich allen Anfragen an die Anwendung unabhängig von der konkret geforderten URI zunächst das Index-Skript ausgeführt wird, liegt im Verantwortungsbereich des Web-Servers und muss je nach genutzter Software individuell konfiguriert werden. Für die Entwicklung wurde wie bereits beschrieben der Apache HTTP Server verwendet, welcher mit dem `mod_rewrite`-Modul dafür sorgt, dass alle URIs zu `/index.php` umgeschrieben werden. Die ursprüngliche URI wird dem Skript dabei als Parameter übergeben, um im Anschluss vom Router des Slim-Frameworks dekodiert werden zu können. Die Anweisungen in der Datei `.htaccess` enthalten hierfür die Regeln zur URI-Umformung bei Einsatz eines Apache-Servers.

Das Index-Skript inkludiert alle Routinen aus der Datei `Bootstrap.php`, wovon jede zur Initialisierung einer Kernkomponente verantwortlich ist. Hierzu zählen unter anderem der Verbindungsaufbau zur Datenbank und zum Job-Server, das Auslesen und Verarbeiten der Serverkonfiguration und die Registrierung aller vorhandenen Ressourcen und aufrufbaren Methoden. Die Serverkonfiguration befindet sich ebenfalls im `app`-Verzeichnis und folgt den syntaktischen Regeln des `ConfigParser`-Moduls von Python [5]. In Abhängigkeit von der angefragten Ressource wird schließlich eine der Controller-Klassen, die im Verzeichnis `app/controllers/` liegen, instanziiert. Jede Klasse implementiert dabei alle Me-

[40] `https://github.com/earl/beanstalkc/` (abgerufen im Oktober 2014)
[41] `https://github.com/PyMySQL/PyMySQL` (abgerufen im Oktober 2014)
[42] `http://slimframework.com` (abgerufen im Oktober 2014)

```
- HoneySens/
|--- app/
| |--- controllers/                % API-Ressourcen und Methoden
| | |--- Certs.php
| | |--- ...
| |--- models/
| | |--- entities/                 % Domaenenmodell
| | | |--- Event.php
| | | |--- Sensor.php
| | | |--- ...
| | |--- BeanstalkService.php      % Service-Klassen
| | |--- ...
| |--- scripts/                    % Python-Skripte
| | |--- gen-sensorconfig.py
| | |--- ...
| |--- templates/
| | |--- layout.php
| |--- Bootstrap.php               % Initialisierungsroutinen
| |--- config.cfg                  % API-Konfiguration
|--- cache/
|--- data/
| |--- CA/                         % Certificate Authority
| | |--- openssl.cnf
| | |--- cacert.pem
| | |--- cakey.pem
| |--- configs/
| | |--- template/                 % Konfigurationstemplates
| | | |--- honeysens.cfg
| | | |--- ...
| |--- firmware/                   % Firmware-Ablage
| | |--- sd/
| |--- upload/
|--- lib/                          % Externe Frameworks/Bibliotheken
| |--- Slim/
| |--- ...
|--- public/
| |--- .htaccess                   % URI-Rewrite
| |--- index.php                   % Einstiegspunkt der Anwendung
| |--- ...
|--- utils/
| |--- beanstalk/
| |--- docker/
| |--- sensor/
```

Abbildung 8: Serverseitige Verzeichnisstruktur

thoden jeweils genau einer Ressource. Anschließend liegt es im Verantwortungsbereich des Controllers, aus der Datenbank die benötigten Daten abzufragen, bei Bedarf Veränderungen vorzunehmen und das Ergebnis der Operation zurück zum Client zu senden.

Das Generieren von entsprechenden Objekten aus den in der Datenbank gespeicherten Daten ist Aufgabe einer weiteren Abstraktionsschicht, genannt *Object Relational Mapper* (ORM). Sie erleichtert dem Entwickler die Persistenzverwaltung, da dieser das Synchronisieren der Domänenobjekte mit der Datenbank via SQL nicht manuell durchführen muss, was die Komplexität der Anwendung verringert und gleichzeitig die Lesbarkeit des Codes verbessert. Zusätzlich erlaubt es diese Middleware, unabhängig von der zugrundeliegenden Datenbanksoftware zu arbeiten. Für den Prototypen wurde das PHP-Framework *doctrine*[43] eingesetzt, das Unterstützung für alle größeren Datenbankserver bietet, darunter auch MySQL und PostgreSQL. Das Framework stellt nach der Initialisierung einen sogenannten *EntityManager* zur Verfügung, über den die weitere indirekte Kommunikation mit der Datenbank erfolgt. Ein typischer Zyklus der Interaktion mit diesem demonstriert das Codebeispiel 9, in dem zunächst ein Objekt aus der Datenbank gelesen, verändert und wieder zurückgeschrieben wird. Der EntityManager registriert hierfür alle Veränderungen an den ihm bekannten Domänenobjekten, weshalb der parameterlose Aufruf der `flush()`-Methode ausreicht, um anwendungsweit alle bis zu diesem Zeitpunkt vorgenommenen Veränderungen an den Model-Objekten zu synchronisieren. Der Hauptvorteil dieser Abstraktion ist aber, dass der Entwickler ausschließlich mit den PHP-Objekten seines Domänenmodells interagiert. Das Generieren von optimierten SQL-Anfragen wird an das Framework delegiert, für Sonderfälle können diese aber auch vom Entwickler manuell optimiert werden.

```
$sensor = $em->getRepository('HoneySens\app\models\entities\Sensor')
  ->find($id);
$sensor->setLocation($newLocation);
$em->flush(); // synchronisieren aller Aenderungen mit der Datenbank
```

Abbildung 9: Interaktion mit dem EntityManager $em des doctrine-Frameworks

Das Verzeichnis `app/models/entities/` beinhaltet alle Entitäten der Anwendung und stellt im Prinzip eine Replikation des HoneySens-Domänenmodells als PHP-Klassenstruktur dar. Die mit PHP in der Version 5 eingeführten *Namensräume* wurden genutzt, um die verschiedenen Komponenten der Software analog zur Verzeichnisstruktur sauber voneinander zu trennen. Codebeispiel 10 veranschaulicht anhand der (gekürzten) Klasse `User` den Modell-Aufbau. Die Kommentarblöcke über den Klassen- und Attributsdefinitionen dienen als Annotationen für das doctrine-Framework. Sie enthalten alle für die Datenbank-Abstraktionsschicht nötigen Mapping-Informationen, um automatisch ein gültiges Datenbankschema generieren zu können. Zu jedem Attribut existieren weiterhin entsprechende Getter- und Setter-Methoden sowie spezifische Konstanten, falls für eine Eigenschaft ein vorgegebener statischer Wertebereich existiert. Im Beispiel ist es das Attribut `role`, das die Gruppenzugehörigkeit eines Benutzers angibt. Da die Gruppen im Prototypen „hardcoded" sind, werden sie als Konstanten definiert und an allen anderen Stellen im Code ausschließlich in dieser Form referenziert, was die Lesbarkeit verbessert.

In der Konzeption der Anwendung wurde gefordert, dass die Kommunikation mit der REST-API ausschließlich über JSON-Nachrichten erfolgt. Hierfür ist es serverseitig notwendig, die Rückgabewerte der Controller-Methoden in eine entsprechende Repräsentation umzuwandeln. Als Hilfsfunktion besitzt jedes Model zu diesem Zweck eine Funktion `getState()`, die ein assoziatives Array mit allen das

[43] `http://doctrine-project.org` (abgerufen im Oktober 2014)

```php
<?php
namespace HoneySens\app\models\entities;

/**
 * @Entity
 * @Table(name="users")
 */
class User {
  const ROLE_GUEST = 0;
  const ROLE_OBSERVER = 1;

  ...

  /**
   * @Column(type="integer")
   */
  protected $role;

  ...

  public function setRole($role) {
    $this->role = $role;
    return $this;
  }

  public function getRole() {
    return $this->role;
  }

  ...

  public function getState() {
    return array('id' => $this->getId(),
                 'role' => $this->getRole() ...);
  }
}
```

Abbildung 10: Auszug aus der User-Klasse

Objekt repräsentierenden Attributwerten zurückgibt. Dieses kann mit Hilfe der nativen PHP-Methode `json_encode()` in eine String-Repräsentation umgewandelt werden.

Im `models/`-Verzeichnis liegen sogenannte *Service*-Klassen, die Funktionalität bereitstellen, die nicht in Form von Ressourcen ausgedrückt werden kann. Hierzu zählen beispielsweise der `BeanstalkService` zur Abstraktion des Job-Servers und der `ContactService` zum Versenden von E-Mail-Nachrichten bei kritischen Ereignissen oder für wöchentliche Zusammenfassungen. Weiterhin ist das Verzeichnis `scripts/` zu erwähnen, das von der API ausführbare Python-Skripte beinhaltet, die systemnahe Operationen wie Modifikationen am Dateisystem oder Interaktionen mit Kommandozeilenprogrammen ausführen. Das in der Graphik genannte Skript `gen-sensorconfig.py` ist beispielsweise für das Erzeugen eines Sensor-Konfigurationsarchivs zuständig. Das `data/`-Verzeichnis dient sowohl zum Abspeichern von hochgeladenen Dateien (z.B. Sensor-Firmware) als auch zur Bereitstellung aller weiteren statischen Informationen, die selbst keinen Code beinhalten. So nimmt das Verzeichnis `CA/` alle zur *Certificate Authority* gehörigen Dateien auf, mit der die Zertifikate für neue Sensoren generiert werden können. In `configs/` liegen vom zuvor genannten Python-Skript erzeugte Konfigurationsarchive zur Sensorinitialisierung, die von authorisierten Benutzern heruntergeladen werden können. Das `template/`-Unterverzeichnis beinhaltet für die Erzeugung der Konfiguration benötigte Musterdateien, die vom Skript mit den individuellen Sensoreinstellungen gefüllt werden. Die übrigen Verzeichnisse `upload/`, `firmware/` und `sd/` werden zum Speichern der Firmware und bootbaren SD-Images genutzt. In `lib/` sind zudem alle extern eingebundenen Bibliotheken und Frameworks enthalten, darunter die bereits beschriebenen Projekte *Slim* und *doctrine*. Das `utils/`-Verzeichnis nimmt abschließend alle zusätzlichen Daten auf, die für den Betrieb der Anwendung selbst nicht zwingend erforderlich sind, darunter Skripte für das Server-Deployment, die auf den Sensoren laufenden Dienste und die Beanstalkd-Jobs.

Zusätzliche Bibliotheken Zusätzlich zu den bereits beschriebenen Frameworks *Slim* und *doctrine*, die für die Basisfunktionalität der API verantwortlich sind, kommt noch eine Reihe weiterer externer Bibliotheken zum Projekt hinzu, um bei zusätzlichen, teils optionalen Operationen wie dem Versenden von E-Mails oder kryptographischen Methoden zu assistieren. Das Delegieren dieser Aufgaben an etablierte, getestete Softwarekomponenten beschleunigt schließlich die Entwicklung des Prototypen. Die Liste der genutzten externen Bibliotheken umfasst:

FileUpload : Eine Bibliothek, die den Upload großer Dateien in kleineren, *Chunks* genannten Teilen ermöglicht[44]. Somit ist es möglich, die API auch auf restriktiven Web-Servern zu betreiben, deren Konfiguration nicht direkt angepasst werden kann oder soll. Firmware ist in der Regel mehrere hundert Megabyte groß, deren Upload wird daher mit dieser Bibliothek deutlich vereinfacht. Zusätzlich unterstützt die Software das Wiederaufnehmen eines abgebrochenen Uploads, was jedoch auch auf Client-Seite entsprechende Unterstützung erfordert.

ConfigParser : Die Serverkonfiguration nutzt, wie bereits beschrieben, die Syntax des Python *ConfigParser*-Moduls. PHP kann dieses Format nativ nicht interpretieren, was umständliches manuelles Parsen oder den Einsatz einer kompatiblen Bibliothek erfordert. Das von den *Noiselabs Studios* entwickelte Skript `ConfigParser.php`[45] ist genau zu diesem Zweck gedacht und ermöglicht

[44] `https://github.com/Gargron/fileupload` (abgerufen im Oktober 2014)
[45] `https://github.com/noiselabs/configparser.php` (abgerufen im Oktober 2014)

es, den Inhalt der Datei config.cfg zu nativen PHP-Arrays zu konvertieren sowie daran vorgenommene Änderungen wieder im korrekten Format abzuspeichern. Viele Konfigurationseinstellungen müssen somit nicht per Hand im Dateisystem verändert werden, sondern stehen über die settings-Ressource auch über die API zur Modifikation bereit.

Pheanstalk : Zur Kommunikation mit dem Job-Server Beanstalkd ist ebenfalls eine externe Bibliothek erforderlich: *Pheanstalk*, eine reine PHP-Implementierung des Beanstalk-Protokolls ohne zusätzliche Abhängigkeiten[46]. Die API macht von dieser Gebrauch, um neue Aufgaben zur Job-Warteschlange hinzuzufügen, nämlich primär solche zur Konvertierung von Firmware-Archiven zu bootbaren Images für Speicherkarten. Seitens der Python-Skripte, die die Aufgaben schließlich in Empfang nehmen und abarbeiten, ist ebenfalls eine Verbindung zum Beanstalkd-Server nötig. Hierfür wurde das bereits genannte Python-Modul *beanstalkc* eingesetzt.

PHPMailer : In der Konzeption sind E-Mail-Nachrichten als Mittel gewählt worden, um Verantwortliche über plötzliche kritische Vorfälle zeitnah informieren zu können. Über den Kommunikationskanal sollen außerdem zusätzlich regelmäßige Zusammenfassungen über den aktuellen Systemzustand und vergangene Ereignisse versendet werden. Diese Funktionen sind im Prototypen in der ContactService-Klasse zusammengefasst. Sie nutzt nicht die native mail()-Funktion in PHP, da diese ein lokal installiertes Mail-Relay voraussetzt. Die Integration eines zusätzlichen Mail-Servers in das Netzwerk des SVNs ist problematisch und würde Konflikte mit den vielfältigen Firewall-Policies hervorrufen. Vielmehr im Sinne der Integrationsanforderung ist daher die Nutzung eines bereits existierenden SMTP-Servers und die Bereitstellung entsprechender Konfigurationsoptionen im HoneySens-Server, einen solchen zu nutzen. Für diese Funktionalität wurde die Bibliothek *PHPMailer*[47] hinzugefügt, die keine zusätzlichen Abhängigkeiten mit sich bringt und leicht anpassbar ist. Somit konnte auch sichergestellt werden, dass E-Mails unter Nutzung von *STARTTLS* über gesicherte Verbindungen übertragen werden, wenn die Serverseite dies unterstützt.

phpseclib : Diese kleine Bibliothek stellt eine Reihe von kryptographischen Hilfsfunktionen bereit, die typischerweise nicht im Funktionsumfang einer PHP-Distribution enthalten sind[48]. Im HoneySens-Prototypen wird sie lediglich als Adapter benötigt, um zwischen verschiedenen Formaten kryptographischer Zertifikate konvertieren zu können.

[46] https://github.com/pda/pheanstalk/ (abgerufen im Oktober 2014)
[47] https://github.com/PHPMailer/PHPMailer (abgerufen im Oktober 2014)
[48] http://phpseclib.com/about (abgerufen im Oktober 2014)

7.2 Web-Frontend

Die REST-API ist so generisch gestaltet, dass Client-Anwendungen plattform- und technologieunabhängig vom Server entwickelt werden können: Denkbar sind Web- und Thick-Clients, aber auch Apps für Smartphones oder Tablets. Gemeinsame Schnittstelle für alle Lösungen sind immer die über HTTP nutzbare API und JSON als Kommunikationsprotokoll. Im Rahmen dieser Arbeit wurde von diesen Varianten die Bereitstellung einer prototypischen Webanwendung beschlossen, weil sie auf allen denkbaren Endgeräten nutzbar ist. Lediglich ein Web-Browser ist für den Zugriff erforderlich. Zudem kann mit aktuellen Web-Technologien wie *HTML5* und *CSS3* dafür gesorgt werden, dass Benutzerfreundlichkeit auch auf Mobilgeräten mit geringer Rechenleistung und kleinem Display gewährleistet ist. Weiterhin stellt eine Webanwendung, die im Grunde lediglich eine interaktive Website ist, keine besonderen Anforderungen an die Serverseite. Lediglich ein Web-Server wird zum Hosting benötigt, weshalb eine Integration in die bestehende API denkbar ist. Tatsächlich stellt das *Slim*-Framework alle nötigen Ressourcen bereit, um eine REST-API und eine Website zugleich anbieten zu können.

Bei der Realisierung eines Web-Clients beziehungsweise einer Webanwendung im Allgemeinen bieten sich einem Entwickler grundsätzlich zwei Möglichkeiten zur Umsetzung: Einerseits das „traditionelle“ Modell, bei dem überwiegend statische HTML-Dokumente zum Browser gesendet und serverseitig verarbeitet werden, andererseits die „dynamische“ Variante, bei der es Aufgabe des Clients (Browsers) ist, die Darstellung des Dokuments auf Basis der vom Server empfangenen Informationen beständig zu aktualisieren. Letztere Variante wurde im Zuge des „*Web 2.0*“ populär und nutzt die Technologie *AJAX*[49] als Grundpfeiler, bei der im Hintergrund über JavaScript ausgelöste HTTP-Anfragen an den Server geschickt werden, ohne dass deren Antwort eine automatische Aktualisierung der gerade vom Browser dargestellten Website auslöst. Somit ist es möglich, einzelne Bestandteile einer Website vom Server erst dann nachzuladen, wenn sie benötigt werden (sogenanntes „*Lazy Loading*“), oder aber alle Elemente bei der ersten Anfrage bereits auszuliefern und die weitere Kommunikation mit dem Server auf knappe API-Aufrufe zu beschränken, die von der JavaScript-Anwendung im Browser interpretiert werden. Dieses Verfahren spart Bandbreite und vermittelt dem Benutzer ein Gefühl des „flüssigen“ Interagierens mit der Website, da nach einem Klick nicht alle Elemente neu geladen werden müssen. Derartige Software ähnelt in der Bedienung einem lokal installierten, graphischen Programm, woraus die Bezeichnung *Web-Anwendung* resultiert. Der Ansatz wurde aus den genannten Gründen als Referenz für den Prototypen ausgewählt.

7.2.1 Architektur

Eine funktionierende moderne Webanwendung ist das Ergebnis des Zusammenspiels einer Vielzahl von Komponenten und Technologien, sowohl server-, hauptsächlich aber clientseitig. Die nachfolgende Auflistung gibt einen knappen Überblick über die im Web-Frontend genutzten Techniken und Standards.

HTML5 : Die *Hypertext Markup Language* ist eine Auszeichnungssprache für elektronische Dokumente und Grundbaustein des *World Wide Webs*. Mit ihr werden Bestandteile von Dokumenten (Überschriften, Absätze, Tabellen usw.) mit sogenannten *Tags* gekennzeichnet, die beim Rendern einer graphischen Darstellung vom Web-Browser interpretiert werden. Die Spezifikation für die

[49] Asynchronous JavaScript and XML

an die gewachsenen Anforderungen der Netzgemeinde angepasste fünfte Revision der Sprache wurde am 28. Oktober 2014, noch während der Entwicklung dieses Projekts, offiziell verabschiedet [43]. Einzelne Bestandteile des Standards wurden allerdings bereits im Vorfeld von diversen Browserherstellern implementiert und stehen browserübergreifend zur Verfügung. Zum Nachrüsten der Unterstützung neuer Features in älteren Browsern existieren zudem optionale Bibliotheken, die versuchen, plattformübergreifend eine einheitliche „User Experience" zu gewährleisten. Die Entwicklung des HoneySens-Prototypen erfolgte weitestgehend HTML5-konform, mit stellenweisen Einschränkungen, wenn Elemente noch nicht in allen gängigen Browserversionen verfügbar waren.

CSS3 : Die Aufgabe der *Cascading Style Sheets* ist es, die optische Darstellung und Usability eines mit HTML ausgezeichneten Dokuments zu beschreiben. Hierfür stellt CSS eine große Zahl von Eigenschaften wie Vorder- und Hintergrundfarbe, Größen- oder Abstandsparameter zur Verfügung, die in einem sogenannten *Stylesheet* auf HTML-Elemente angewendet werden können. Während es zu Zeiten von HTML4 noch die Regel war, auch innerhalb eines HTML-Dokuments optische Anpassungen vorzunehmen, schreiben die aktuelleren Standards HTML5 und CSS3 eine *strikte Trennung von Markup (HTML) und Design (CSS)* vor. In der dritten Revision ermöglicht CSS zudem die dynamische Anpassung von Websites an die Gegebenheiten des darstellenden Clients, beispielsweise durch optische Veränderungen für die Anzeige einer Website auf einem Mobilgerät mit kleinem Display, was auch unter dem Begriff *Responsive Web Design* bekannt ist [28]. Beim Prototyping der Webanwendung wurde Wert darauf gelegt, zum CSS3-Standard konform zu bleiben und auch auf Smartphones und Tablets eine akzeptable Darstellung zu ermöglichen.

JavaScript : Die auch unter dem Namen *ECMAScript* bekannte Skriptsprache wird von allen gängigen Web-Browsern unterstützt und erlaubt die Entwicklung dynamischer Websites. Der Code wird vom Browser innerhalb einer abgeschotteten virtuellen Maschine ausgeführt, gewährt dem Entwickler aber trotzdem Möglichkeiten, die mit reinem HTML und CSS undenkbar wären. Diese reichen von optischen Veränderungen einer bereits gerenderten Website über für den Nutzer transparente, parallele Anfragen an den Web-Server bis hin zur Realisierung umfangreicher, interaktiver Spiele direkt im Browser. Die Programmlogik und Interaktivität von Webanwendungen wie dem HoneySens-Frontend sind vollständig in JavaScript realisiert und nutzen – analog zu serverseitigen Programmiersprachen – eine Reihe von Frameworks und Bibliotheken, um die Entwicklung zu beschleunigen.

AJAX und JSON : Die Grundidee hinter der *Asynchronous JavaScript and XML* genannten Technologie ist das dynamische Nachladen von Daten, nachdem eine Website vom Browser bereits vollständig gerendert wurde. JSON eignet sich in diesem Kontext als Format zum Datenaustausch mit dem Server, da es breite Unterstützung in Programmiersprachen und Bibliotheken genießt und zusätzlich zu den übertragenen Daten nur einen geringen strukturellen Overhead mit sich bringt [19] – die Nachrichten sind somit sehr kompakt. Aus diesen Gründen wird JSON sowohl von der REST-API als auch auf Seite der Clients genutzt. Bei der Implementierung einer auf AJAX basierenden *Single-Page*-Webanwendung ist weiterhin darauf zu achten, dass die konkrete Implementierung der Technik browser- und versionsspezifisch ist. Es existieren jedoch Frameworks, die diesen Prozess abstrahieren, alle gängigen Browser unterstützen und dem Entwickler ein generisches AJAX-Objekt zur Verfügung stellen. Im HoneySens-Prototypen wird *jQuery* zu diesem Zweck eingesetzt.

Voraussetzung zur Entwicklung einer modernen Webanwendung ist ein passender Softwarestack aus Bibliotheken und strukturgebenden Frameworks. Die für diese Aufgabe ausgewählte Software wird nachfolgend knapp mit ihrer jeweiligen Funktion präsentiert, um dem Leser die grundsätzliche Arbeitsweise der Webanwendung zu vermitteln. Auf dem Zusammenspiel der einzelnen Komponenten liegt dabei das Hauptaugenmerk. Das Programmieren einer Anwendung mit einer graphischen Benutzungsoberfläche (*GUI*) stellt die Entwickler immer vor die Herausforderung, die Darstellung von der Programmlogik zu trennen. Gamma et al. entwickelten zur Lösung dieses Problems das in der Softwaretechnik beliebte Entwurfsmuster *Model-View-Controller* (MVC) [22], das eine Trennung in eine *Model*-Komponente zur Repräsentation des Domänenmodells, den *View* zur Steuerung der Darstellung und den *Controller* vorschlägt, der die Schnittstelle für den Benutzer darstellt und auf dessen Anfragen hin Model und View entsprechend verändert. Zusätzlich ist es den View-Komponenten erlaubt, beim Rendern direkt die Eigenschaften der Model-Elemente auszulesen. Das Entwurfsmuster hat sich für die Umsetzung umfangreicher graphischer Benutzeroberflächen auch im Web-Bereich in Form von Frameworks durchgesetzt, die die Trennung des JavaScript-Codes in entsprechende Verantwortungsbereiche erleichtern und beispielsweise abstrakte Basisklassen für Controller und Views bereitstellen.

Der HoneySens-Prototyp basiert auf dem JavaScript-Framework *Backbone.js*[50], das alle genannten Komponenten um spezifische, an die Gegebenheiten des WWWs angepasste Funktionen wie das Konvertieren und Interpretieren von JSON-Strings und die vollautomatische Synchronisation mit einer *RESTful API* durch die HTTP-Methoden `GET`, `PUT`, `POST` und `DELETE` erweitert. JavaScript unterstützt nativ keine statischen Klassendefinitionen, weshalb das Framework einen solchen Mechanismus nachrüstet und es ermöglicht, durch Ableiten der Klasse `Backbone.Model` Domänenobjekte mit allen gewünschten Eigenschaften und Methoden zu implementieren. Alle Models eines gemeinsamen Typs werden schließlich in einer *Collection*, repräsentiert durch die gleichnamige Klasse `Backbone.Collection`, gesammelt. So kann beispielsweise zur Auflistung aller Ereignisse einfach die gesamte Collection `Events` an eine geeignete View-Klasse übergeben werden, die dann selbstständig die einzelnen Models extrahiert und rendert. Zusätzlich stellt das Framework einen *Router* bereit, der es ermöglicht, die URL im Browser in Abhängigkeit vom dargestellten View zu modifizieren. Somit funktioniert auch innerhalb der Webanwendung der „Zurück“-Button des Browsers und Nutzer haben die Möglichkeit, Lesezeichen für bestimmte Anwendungszustände anzulegen.

Der Einsatz des Backbone-Frameworks erfordert leider noch immer einen umfangreichen, *Scaffolding* genannten Prozess, bei dem domänenunabhängige Hilfsklassen und -funktionen erstellt werden, auf deren Basis anschließend erst die eigentliche Anwendung errichtet werden kann. Um dennoch das Prototyping zu beschleunigen, wurde zusätzlich das Framework *Marionette.js* eingebunden, das die Autoren knapp als „Composite application library for Backbone.js“[51] beschreiben. Es setzt häufig zusammen mit Backbone gebrauchte Entwurfsmuster um, indem es eine Vielzahl zusätzlicher Klassen anbietet, die spezifischer sind als die des sehr allgemein gehaltenen Backbone-Frameworks. Das Hauptaugenmerk der Software liegt daher auf der Bereitstellung vieler von `Backbone.View` abgeleiteten Klassen, die zur Darstellung von spezifischen, oft gebrauchten Elementen genutzt werden können. Darunter sind Views zum Rendern einzelner Models, ganzer Collections oder zum Definieren von „*composite*“ (geschachtelten) Layouts. Zusätzlich erweitert der `Marionette.AppRouter` den `Backbone.Router`, sodass bestimmte Teilbereiche der Anwendung, spezifiziert durch ihre eindeutige URL, von gesonderten *Controller*-Klassen behandelt werden. Dieses Element des MVC-Entwurfmusters fehlte zuvor in der Backbone-Implementierung.

[50] `http://backbonejs.org` (abgerufen im Oktober 2014)
[51] `http://marionettejs.com` (abgerufen im Oktober 2014)

Die bisher beschriebenen Komponenten betrafen ausschließlich die Anwendungslogik in JavaScript. Es erscheint jedoch sinnvoll, auch für die Markupsprache HTML und die CSS-Stylesheets ein erprobtes, vorgefertigtes Grundgerüst zu nutzen. Ein ausgeklügeltes individuelles optisches Design des Prototypen würde den Rahmen dieser Arbeit sprengen und ist auch nicht gefordert, eine grundsätzlich ästhetisch ansprechende Schnittstelle ist einer rudimentären allerdings immer vorzuziehen. *Twitter Bootstrap* ist ein freies „front-end framework for faster and easier web development“[52], das eine einheitliche Designsprache, Gestaltungsregeln und fertig nutzbare Elemente speziell für den Einsatz in Webanwendungen anbietet. Es beinhaltet eine umfangreiche Bibliothek an in JavaScript implementierten Komponenten, darunter modale Dialogboxen, interaktive Navigationsleisten oder animierte Fortschrittsbalken, aber auch ein umfangreiches Standard-Stylesheet, das allen HTML-Tags ein einheitliches, konsistentes Aussehen verleiht. Dank vordefinierter CSS-Klassen erleichtert das Framework außerdem die Umsetzung eines *Responsive Designs*, für welches der Entwickler lediglich die Elemente der Website in flexible Spalten und Zeilen einsortieren muss. Zudem beinhaltet es *Glyphicons*, eine umfangreiche Sammlung von vektorbasierten, frei skalierbaren Icons. Zuletzt liefert das Bootstrap-Framework noch ein standardkonformes HTML5-Grundgerüst mit, das als Basis für eine neue Webanwendung genutzt werden kann.

7.2.2 Implementierung

Die Frameworks Backbone.js Version 1.1.2, Marionette.js Version 1.8.3 und Bootstrap in der Version 3.1.1 bilden das Fundament des HoneySens Web-Frontends. Die Anwendung wird zusammen mit der REST-API von einem einzelnen Web-Server ausgeliefert und ist somit ebenfalls Teil des auf dem *Slim*-Framework basierenden PHP-Projektes, das in Kapitel 7.1 beschrieben wurde. Das Frontend befindet sich gemäß der in Abbildung 8 gezeigten Projektstruktur im `public/`-Verzeichnis und setzt sich aus den in Abbildung 11 dargestellten Dateien und Verzeichnissen zusammen.

Einstiegspunkt der Webanwendung wie auch der REST-API ist das Skript `index.php`, welches das Slim-Framework und alle zur Bearbeitung einer Anfrage nötigen Komponenten initialisiert. Während des Bootstrap-Prozesses entscheidet sich anhand der angeforderten URI, ob die API genutzt oder das Web-Frontend ausgeliefert werden soll. So wird beispielsweise die HTTP-Anfrage

```
GET http://honeysens-server/api/users/22
```

an den Users-Controller der API weitergereicht, wohingegen

```
GET http://honeysens-server/#sensors
```

auf das Web-Frontend verweist und mit dem zusätzlichen Parameter `#sensors` den AppRouter des Marionette-Frameworks anweist, die Sensorliste auszugeben.

Das `css/`-Verzeichnis beinhaltet alle Stylesheets, die die optische Gestaltung der HTML-Elemente bestimmen. Die beiden Bootstrap-Stylesheets definieren allgemeine Stilvorgaben und geben allen Komponenten ein einheitliches Aussehen. In der Datei `honeynet.css` sind hingegen projektspezifische, für den Prototypen benötigte Stile hinterlegt. Weitere, zu externen Bibliotheken gehörige Stylesheets – in der Abbildung beispielsweise `bootstrapValidator.min.css`, dessen Modul die Formularvalidierung ermöglicht – liegen ebenfalls in diesem Verzeichnis. `Fonts/` enthält die vektorbasierten

[52] `http://getbootstrap.com` (abgerufen im Oktober 2014)

```
- public/
|--- css/
| |--- bootstrap.min.css
| |--- bootstrap-theme.min.css
| |--- bootstrapValidator.min.css
| |--- honeynet.css
| |--- ...
|--- fonts/
| |--- glyphicons-halflings-regular.svg
| |--- ...
|--- images/
| |--- back_disabled.png
| |--- logo.png
| |--- ...
|--- js/
| |--- backbone.marionette/
| |--- datatables/
| |--- validator/
| |--- bootstrap.min.js
| |--- honeysens.js
| |--- honeysens.models.js
| |--- honeysens.routing.js
| |--- honeysens.views.js
| |--- honeysens.controller.js
| |--- ...
|--- .htaccess
|--- index.php
```

Abbildung 11: Verzeichnisstruktur der Webanwendung

Glyphicons für verschiedene Browser in diversen Formaten, während das `images/`-Verzeichnis alle Pixelgraphiken aufnimmt, die in der Anwendung referenziert werden. Jeglicher JavaScript-Code liegt schließlich in `js/`, wobei umfangreiche Frameworks wie Marionette.js wieder eigene Unterverzeichnisse besitzen. Die Anwendungslogik nehmen die `honeysens.*.js`-Dateien auf, die im Anschluss detaillierter beschrieben werden.

Zuletzt sei noch auf die Datei `layout.php` im `templates/`-Verzeichnis der Abbildung 8 hingewiesen, die ebenfalls zur Webanwendung gehört, sich aber aus organisatorischen Gründen außerhalb des `public/`-Verzeichnisses befindet. In dieser sind das HTML-Grundgerüst und alle *Templates* gespeichert. Dabei handelt es sich um HTML-Blöcke, aus denen die View-Objekte schließlich die im Browser dargestellte Website rendern. Einen Ausschnitt dieser Datei demonstriert Abbildung 12. Zu sehen ist das HTML5-Grundgerüst mit gültigem `DOCTYPE`- und `html`-Tag, sowie die Liste aller zu inkludierenden Stylesheets aus dem `css/`-Verzeichnis im HTML-`head`. Im `body` wird zunächst das HTML-Grundgerüst beschrieben, bestehend aus einer Navigationsleiste am oberen Seitenrand, einer Sidebar und einem großen Inhaltsbereich. Anschließend folgt eine Liste aller Templates, die in `script`-Tags vom Typ `text/template` abgelegt sind, anhand ihrer eindeutigen `id` mit JavaScript gelesen werden

können und aus denen letztendlich eine fertig gerenderte Seite der Webanwendung zusammengesetzt wird. Das Mitschicken aller Templates beim Aufruf der Anwendung verlangsamt den ersten Request, was jedoch letztendlich zu einer flüssigeren Benutzerführung beiträgt: Da die HTML-Snippets alle bereits vorbereitet sind und nur noch mit Nutzdaten gefüllt und vom Browser dargestellt werden müssen, entstehen für einen Anwender während der Bedienung der Software kaum unangenehme Pausen.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>HoneySens</title>
    ...
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <link href="css/honeynet.css" rel="stylesheet">
    ...
  </head>
  <body>
    <nav class="..."></nav>
    <div class="container-fluid">
      <div class="row">...</div>
    </div>
    <!-- Templates -->
    <script type="text/template" id="honeysens-navigation">
      <div class="navbar-header">...</div>
    </script>
    ...
  </body>
</html>
```

Abbildung 12: HTML-Grundgerüst mit Templates

Anwendungslogik Die domänenspezifische Logik ist in den `honeysens.*.js` genannten JavaScript-Dateien der Abbildung 11 implementiert, deren Struktur strikt dem MVC-Entwurfsmuster folgt. So beinhaltet `honeysens.models.js` eine Abbildung des Domänenmodells, das auch serverseitig vorhanden ist. Die Methoden zur Datensynchronisation entfallen allerdings, da die Model-Klassen des Backbone-Frameworks die dafür nötigen Methoden bereitstellen. Auch Getter- und Setter-Methoden werden vom Framework über die Methoden `get()` und `set()` abstrahiert, was die Klassen im Wesentlichen auf eine Auflistung aller Attribute reduziert. Nur in Ausnahmefällen sind gesonderte Methoden zur Initialisierung oder Auflösung von Objektbeziehungen hinzugekommen. Für jede Model-Klasse existiert zudem noch eine `Backbone.Collection`-Instanz, die stellvertretend für alle Ressourcen eines bestimmten Typs steht und zur Darstellung von Listen herangezogen wird. Das Beispiel in Abbildung 13 demonstriert das Genannte anhand eines Codeausschnittes mit der Klasse `User`, die einen Benutzer der Anwendung beschreibt. Alle Attribute einer Ressource werden im `defaults`-Objekt mit ihren optionalen Initialwerten aufgelistet. Das Objekt `models.User.role` dient zudem als Ersatz für eine Aufzählung (*enum*),

die nicht nativ in JavaScript existiert. Zusätzlich ist anzumerken, dass alle Klassendefinitionen innerhalb eines *Closure* genannten Konstrukts definiert sind, um Namensraumüberschneidungen mit anderen JavaScript-Bibliotheken und -Modulen zu vermeiden. Die anonyme Funktion wird durch das syntaktische Gebilde `function() { ... }()` sofort ausgeführt und liefert ein die Klassendefinitionen enthaltendes Objekt zurück, das schließlich in der Anwendung als `honeysens.models` referenziert werden kann.

```
honeysens.models = function(){
  var models = {};
  ...
  models.User = Backbone.Model.extend({
    defaults: {
      'id': '',
      'name': '',
      'password': '',
      'role': 1
    }
  });

  models.Users = Backbone.Collection.extend({
    model: models.User,
    url: 'api/users/'
  });
  models.User.role = { GUEST: 0, OBSERVER: 1, MANAGER: 2, ADMIN: 3 };
  ...
  return models;
}();
```

Abbildung 13: Model-Implementierung im Frontend

Das Skript `honeysens.views.js` definiert alle *View*-Klassen der Anwendung, die für das Rendern von HTML-Elementen verantwortlich sind. Die dafür nötigen Basisklassen werden vom Marionette.js-Framework zur Verfügung gestellt und erlauben die Konzentration auf die domänenspezifische Implementierung. Von `Backbone.Marionette.Layout` abgeleitete Klassen verkörpern beispielsweise Teile einer Website, die von Model-Daten unabhängig sind und daher in der Regel statisch gerendert werden, aber selbst weitere geschachtelte *Regionen* beinhalten können. In diesen können erneut beliebige Views gerendert werden. So sind beispielsweise die Login-Seite, die bei einem Aufruf der Anwendung als nicht authentifizierter Benutzer angezeigt wird, und das *Dashboard* (eine Übersichtsseite, die direkt nach dem Login erscheint) als solche Layouts realisiert. Der größte Teil der View-Objekte instanziert die Klassen `Backbone.Marionette.ItemView` und `Backbone.Marionette.CompositeView`, die zum Rendern eines einzelnen Models respektive einer ganzen Collection benötigt werden. Abbildung 14 versucht den Zusammenhang zwischen Model- und View-Komponenten anhand der Ereignisstruktur graphisch wiederzugeben. Es fällt auf, dass `CompositeView`- und `ItemView`-Objekte immer das Domänenmodell referenzieren, das gerendert werden soll. Dies erlaubt den direkten Zugriff auf die Attribute des Models innerhalb der HTML-Templates. Wenn ein `CompositeView` zu rendern ist, wird ihm bei der Initialisierung die darzustellende `Collection` übergeben, die sich ihrerseits wieder aus einer Liste von `Model`-Objekten zusammensetzt. Die Models werden beim Laden der Collec-

tion anhand des Rückgabewerts vom Server instanziiert, das Backbone-Framework stellt dabei für Synchronisierung und JSON-Interpretation entsprechende Methoden bereit. Während der Initialisierung des `CompositeViews` wird über die `Collection` iteriert und für jedes vorhandene Model ein zugehöriger neuer `ItemView` initialisiert, in einer privaten Liste gespeichert und letztendlich dargestellt. Mit diesem Pattern lässt sich die Darstellung langer Listen von Objekten des gleichen Typs, im HoneySens-Prototypen sind das beispielsweise Sensoren, Ereignisse oder Firmware-Images, auf kurze, überschaubare Renderprozesse herunterbrechen.

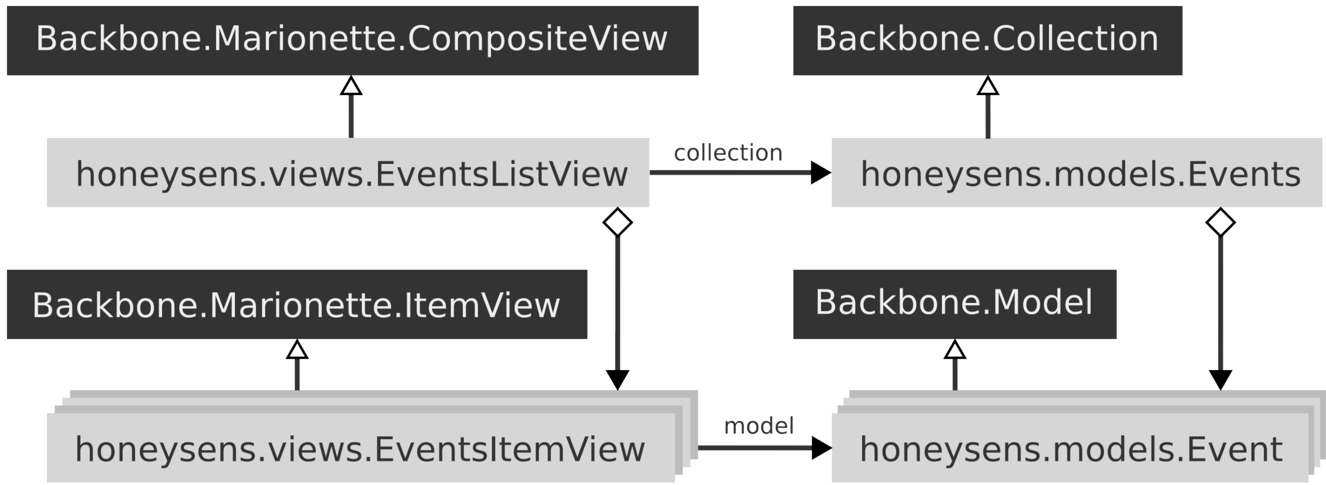


Abbildung 14: Model-View-Integration im Web-Frontend

Die Aufgaben des Controllers werden an die Skripte `honeysens.routing.js` und `honeysens.controller.js` delegiert. Ersteres initialisiert zunächst den `AppRouter` des Marionette.js-Frameworks mit allen denkbaren Routen, die im Prinzip den verfügbaren „Unterseiten" der Webanwendung entsprechen und jeweils über ein eigenes URL-Handle verfügen. Mit jeder Route ist weiterhin eine Funktion verknüpft, die von einem separaten Controller-Objekt bereitgestellt wird. Der HoneySens-Prototyp verwendet aufgrund der geringen Anzahl nötiger Seiten allerdings nur einen einzelnen generischen Controller, der im Skript `honeysens.controller.js` definiert ist. Jede der Callback-Methoden prüft dann zunächst, ob der gerade angemeldete Benutzer zur angefragten Operation überhaupt berechtigt ist, initialisiert anschließend einen View und zeigt ihn im Content-Bereich an. Zuletzt wird ein Ereignis ausgelöst, das den Abschluss des Render-Vorgangs signalisiert. Auf dieses können andere UI-Elemente nach Belieben reagieren und ihre eigene Darstellung anpassen, was im Prototypen beim An- und Abmelden genutzt wird, um die Navigations- und die Sidebar ein- bzw. auszublenden. Das Skript `honeysens.js` fügt abschließend alle MVC-Komponenten zusammen, indem es ein `Backbone.Marionette.Application`-Objekt erzeugt und eine Reihe von Initialisierungsfunktionen hinzufügt, in denen alle weiteren Elemente instanziiert und miteinander verknüpft werden.

Die REST-API bietet wie zuvor beschrieben die `state`-Ressource an, über die von Clients Änderungen in der Datenbank inkrementell abgefragt werden können. Das Web-Frontend besitzt einen Timer, der nach Ablauf von zehn Sekunden eine HTTP-`GET`-Abfrage an den Server veranlasst und bei eventuellen Änderungen das clientseitige Backbone-Model aktualisiert. Durch die direkte Bindung der Models an die zugehörigen Views aktualisieren diese sich bei einer Änderung selbstständig, nicht betroffene Anwendungsbereiche bleiben hingegen bestehen.

Zuletzt initialisiert das `honeysens.js`-Skript noch ein `honeysens.data` genanntes Objekt, das Daten aufnimmt, die clientseitig vorgehalten werden müssen und für die keine Repräsentation im Domänenmodell existiert. Hierzu gehören Metadaten über das letzte Status-Update über die `state`-Ressource, Sessiondaten (angemeldeter Benutzer, dessen Gruppenzugehörigkeit usw.) sowie ein Cache für die E-Mail-Einstellungen der Anwendung (SMTP-Server, Zugangsdaten). Einem noch nicht angemeldeten Benutzer wird immer ein `User`-Objekt zugewiesen, das die Rolle *Guest* besitzt und lediglich das Login-Formular einsehen darf.

Zusätzliche Bibliotheken Auch das Web-Frontend hängt abgesehen von den strukturgebenden Frameworks, die im vorherigen Kapitel beschrieben wurden, von einer Reihe externer Skripte und Bibliotheken ab. Diese dienen primär der optischen Gestaltung und fügen graphische Elemente wie interaktive Diagramme oder sortierbare Tabellen hinzu. Die nachfolgende Liste gibt eine Übersicht über alle bisher noch nicht beschriebenen JavaScript-Komponenten.

jQuery ist eine beliebte JavaScript-Bibliothek zur Manipulation des *Document Object Models* (DOM) einer Website[53]. Die Software wird von Bootstrap, Backbone und Marionette für alle dynamischen Modifikationen einer bereits gerenderten Website benötigt, allerdings im Prototypen auch häufig direkt genutzt. Sie stellt mittels der `ajax()`-Methode zudem das „Backend“ für alle Anfragen an die REST-API bereit. jQuery wird innerhalb der Webanwendung in Version 1.11.0 eingesetzt.

DataTables ist ein Plugin für jQuery, das HTML-Tabellen um dynamische Interaktionsmöglichkeiten bereichert[54]. So erlaubt es beispielsweise dem Benutzer, die Sortierung nach Spalten und die Sortierreihenfolge (aufwärts oder abwärts) zu ändern, ohne dass die Website mit der Tabelle neu geladen werden muss. Zudem können Tabellen auf eine Maximalzahl von Zeilen reduziert und auf mehrere Seiten aufgeteilt werden. Weiterer Bestandteil der Standardfunktionen ist zudem eine dynamische Volltextsuche in allen Zeilen. Eine Herausforderung bei der Integration des Plugins war jedoch die Synchronisation des von der Tabelle genutzten Models mit dem Backbone-Domänenmodell im Falle eines vom Server empfangenen oder dem Benutzer angestoßenen Updates. DataTables wird in der zum Zeitpunkt dieser Arbeit aktuellen Version 1.10.0 genutzt.

bootstrapValidator heißt ein weiteres jQuery-Plugin, über das die dynamische Formularvalidierung im Prototypen realisiert ist. Es unterstützt verschiedene Validierungsmodi (erst beim Abschicken des Formulars oder schon während der Eingabe) und erlaubt es, jedem Formularfeld mehrere *Validatoren* zuzuordnen, die beispielsweise die Gültigkeit eines E-Mail-Strings oder einer IP-Adresse überprüfen. Je nach Erfolgs- oder Fehlerfall werden optische Symbole und – falls gewünscht – Fehlermeldungen in das Formular eingefügt, die optisch auf die Bootstrap-Stylesheets abgestimmt sind. Das Plugin kommt im Prototypen in Version 0.5.1 in allen HTML-Formularen zum Einsatz.

Chart.js ist eine kleine Bibliothek zur Erzeugung dynamischer Diagramme[55], die im HoneySens-Frontend zur graphischen Aufbereitung der gesammelten Daten eingesetzt wird.

[53] `http://jquery.com` (abgerufen im Oktober 2014)
[54] `http://datatables.net` (abgerufen im Oktober 2014)
[55] `http://chartjs.org` (abgerufen im Oktober 2014)

jQuery File Upload ist ein weiteres jQuery-Plugin[56], das ein Widget samt nötigem HTML-Template, Stylesheets und JavaScript-Code zum Upload von Dateien bereitstellt. Es unterstützt die graphische Anzeige des Upload-Fortschritts und das Hochladen von in kleinere *Chunks* zerstückelten großen Dateien. Dies verbessert die Kompatibilität mit einigen Browsern – während der Entwicklung erwies sich speziell der *Mozilla Firefox* als Herausforderung, der ausschließlich mit dieser Methode den Upload von mehreren hundert Megabyte zugleich erlaubte – und restriktiv konfigurierten Web-Servern. In der Webanwendung ermöglicht das Widget in Version 6.41.0 den reibungslosen Upload von neuen Firmware-Archiven.

Spin.js ist eine kleine Bibliothek zur Erzeugung von Graphiken, die einen Ladevorgang anzeigen. Sie werden dynamisch als Vektorgraphik erzeugt und sind somit frei skalierbar und im Aussehen konfigurierbar. Sie kommen im Frontend zum Einsatz, um den Wartevorgang auf eine via AJAX vom Server angeforderte Antwort anzuzeigen.

7.2.3 Funktionsumfang

Dieses Kapitel gibt einen abschließenden Überblick über die verschiedenen Aktionsbereiche und Interaktionsmöglichkeiten der Webanwendung aus Sicht eines Benutzers. Zunächst ist zwingend eine Authentifizierung erforderlich, um auf irgendeine andere als die `session`-Ressource zugreifen zu können. Aus diesem Grund wird ein unbekannter Anwender bei jedem Zugriffsversuch zunächst auf die Login-Seite umgeleitet, die zur Eingabe eines Benutzernamens mit zugehörigem Passwort auffordert. Nach deren Eingabe wird eine AJAX-Anfrage mit den Daten an die API gesendet, die bei Erfolg mit einer umfangreichen HTTP-Nachricht und dem vollständigen Datenstand des Servers zur Initialisierung des Domänenmodells im Client antwortet. Zudem wird ein HTTP-Cookie gesendet, über das der Benutzer auch nach einem Verbindungsverlust ohne erneute Authentifizierung validiert werden kann. Falls die Webanwendung – und nicht die API – dann gezielt mit einem gültigen Cookie angefordert wird, werden alle Datenstrukturen zur Initialisierung des Models als `base64`-kodierte JavaScript-Variablen an die HTTP-Nachricht angehangen. Aus Sicherheitsgründen wurde bei der Implementierung berücksichtigt, dass die via JavaScript angelegten Objekte über Browser-Plugins ausgelesen werden können, weshalb beim Abmelden vom Webinterface alle Datenstrukturen aus dem Speicher entfernt werden.

Layout Nach einem erfolgreichen Login wird dem Benutzer zunächst das sogenannte *Dashboard* präsentiert, zu sehen in Abbildung 15. Die Anwendung ist insgesamt in drei Bereiche aufgeteilt: Am oberen Rand stellt das HoneySens-Logo mit Schriftzug zugleich einen Link dar, um aus jeder Unterseite heraus zurück zur Übersicht gelangen zu können. Außerdem werden der Name des angemeldeten Benutzers und die Option zum Abmelden in der rechten oberen Ecke dargestellt. Die *Sidebar* dient der Navigation und erlaubt den Wechsel zwischen den diversen Unterseiten der Webanwendung, die im Inhaltsfeld aktuell dargestellte Seite ist dabei dominant blau markiert. Auf weißem Hintergrund werden schließlich die Views mit dem datengetriebenen Inhalt gerendert, seien dies Listen von Ereignissen, Sensoren oder die Benutzerverwaltung. Die folgenden Absätze fassen die innerhalb der verschiedenen Bereiche erreichbaren Funktionen zusammen.

[56] `https://blueimp.github.io/jQuery-File-Upload/` (abgerufen im Oktober 2014)

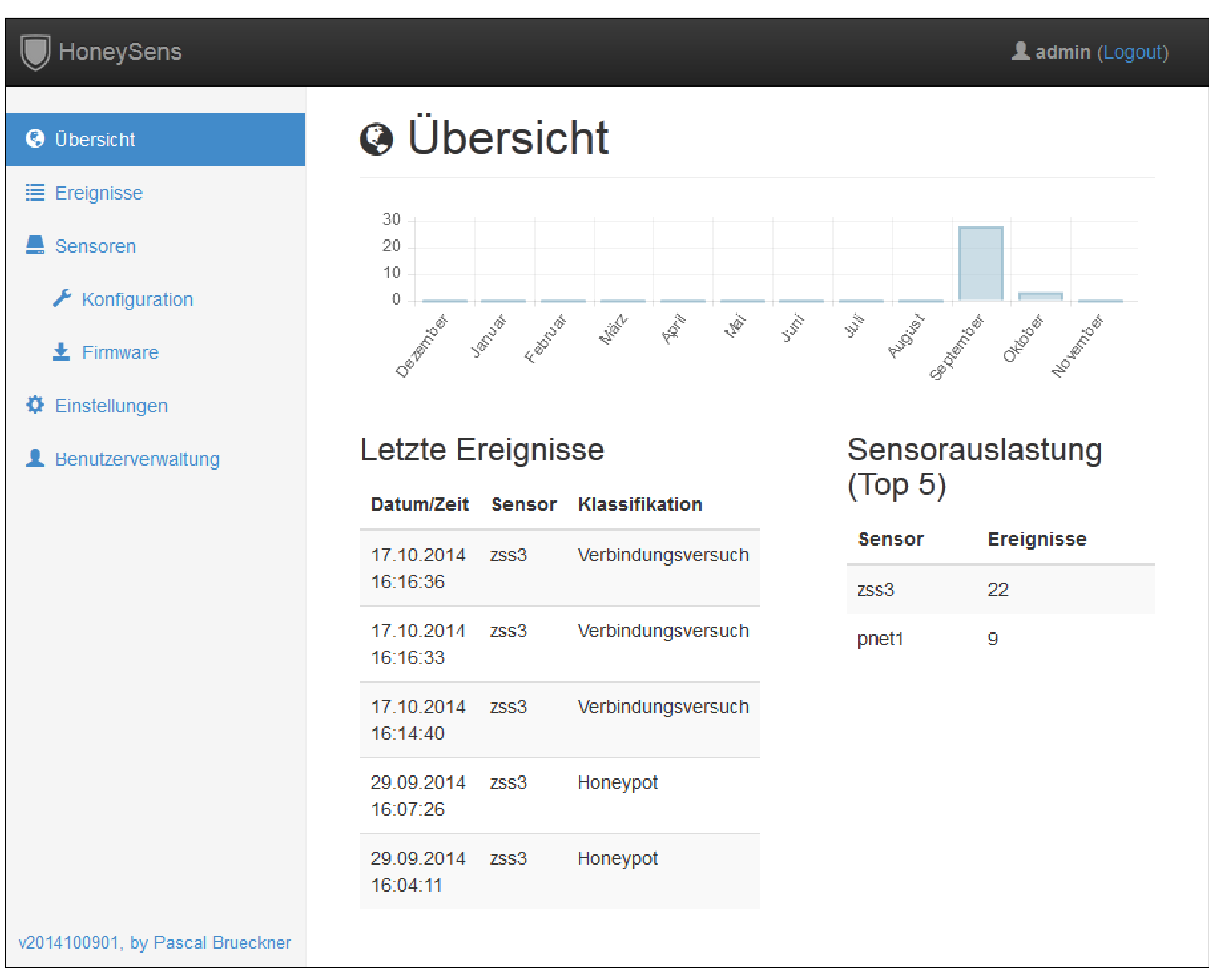


Abbildung 15: Dashboard der Webanwendung

Dashboard Die Übersichtsseite bietet im oberen Bereich ein animiertes, interaktives Balkendiagramm an, das die über das vergangene Jahr aufgezeichneten Ereignisse gruppiert nach Monaten aufsummiert. Den exakten Wert eines Monats erfährt der Benutzer, indem er die einzelnen Balken mit der Maus berührt. Zusätzlich präsentiert die Seite noch in gekürzter Form die letzten fünf aufgezeichneten Ereignisse sowie die fünf „aktivsten" Sensoren im Sinne der registrierten Vorfälle.

Ereignisliste Nach einem Klick auf „Ereignisse" werden alle Vorfälle, die von den Sensoren aufgezeichnet wurden, in einer großen Tabelle aufgelistet. Um Unübersichtlichkeit zu vermeiden, ist für die mit dem DataTables-Plugin realisierte Liste die Aufteilung des Inhalts auf beliebig viele Seiten aktiviert. Zudem kann vom Benutzer durch einen Klick auf einen Spaltenkopf die Sortierrichtung für diese spezifische Spalte geändert werden. Ein Suchfeld erlaubt weiterhin die Volltextsuche in der Tabelle und kann beispielsweise auch als Filter genutzt werden, um nur alle Einträge eines bestimmten Sensors anzeigen zu lassen. Abbildung 16 zeigt einen Ausschnitt der Ereignistabelle. Die Bedeutung der Spalten gestaltet sich folgendermaßen:

ID ist ein von der Datenbank vergebenes Handle, mit dem ein Datensatz eindeutig identifiziert werden kann. Es wird primär als Referenz im Schriftverkehr oder anderen Personen gegenüber gebraucht, wenn bestimmte Ereignisse thematisiert werden sollen.

Zeitpunkt Datum und auf die Sekunde genaue Uhrzeit des Vorfalls. Bei Ereignissen, die mehrere Netzwerkpakete beschreiben (beispielsweise Portscans), wird der Zeitpunkt des Eintreffens des ersten Pakets als Zeitstempel gespeichert.

Sensor Der Name des Sensors, durch den ein Vorfall aufgezeichnet wurde. Da Sensornamen per Definition im Domänenmodell einmalig sind, ist die Zuordnung immer eindeutig möglich. Es ist Aufgabe des Web-Frontends und der API, die Integrität dieses Feldes sicherzustellen.

Klassifikation Diese wird von den Sensoren selbstständig bei der Registrierung eines Ereignisses beim Server vorgenommen und beschreibt die Art des Vorfalls. Im Prototypen verwendete Werte sind *Verbindungsversuch* für TCP- oder UDP-Anfragen an geschlossene Ports, *Honeypot* für Verbindungen, die von einem auf dem Sensor laufenden Honeypot-Service beantwortet wurden und *Portscan* für den vermeintlichen Versuch eines Angreifers, die auf dem Gerät aktiven Netzwerkdienste in Erfahrung zu bringen. Der Wert wird zusätzlich für die Farbkodierung der gesamten Ereigniszeile genutzt (Portscan gelb, Honeypot rot).

Quelle nennt die IP-Adresse, die einen aufgezeichneten Vorfall verursacht hat.

Details Wenige Zeichen umfassende Zusammenfassung des Ereignisses, wird von den Sensoren beim Anlegen auf dem Server hinzugefügt und ist somit je nach Art des Vorfalls unterschiedlich. Benennt bei TCP/UDP-Verbindungsversuchen den betroffenen Port, bei Honeypot-Ereignissen das meldende Honeypot-Modul und im Falle eines Portscans die Anzahl der beteiligten Pakete.

Status Dient zur Markierung von Ereignissen für die das System nutzenden Administratoren. Der Status kann zwischen den Optionen „Neu", „In Bearbeitung", „Erledigt" und „Ignoriert" wechseln. Dessen Änderung ist mit der nebenstehenden Schaltfläche möglich, die ein entsprechendes Dialogfeld erscheinen lässt. Zusätzlich kann für jedes Ereignis noch ein Text hinterlegt werden, beispielsweise um durch manuelle Nachforschung gewonnene Erkenntnisse über den Vorfall festzuhalten.

Ereignisse

Update in 6s

Suchen:

ID	Zeitpunkt	Sensor	Klassifikation	Quelle	Details	Status	Aktionen
259	17.10.2014 16:16:36	zss3	Verbindungsversuch	192.168.1.195	UDP Port 1714	Neu	
254	29.09.2014 16:04:11	zss3	Honeypot	192.168.1.10	SMB/CIFS (dionaea)	Neu	
252	29.09.2014 15:33:17	zss3	Verbindungsversuch	192.168.1.10	TCP Port 123	Neu	
250	26.09.2014 18:06:31	zss3	Verbindungsversuch	192.168.1.10	TCP Port 999	Erledigt	
247	11.09.2014 18:28:47	zss3	Verbindungsversuch	192.168.1.10	TCP Port 123	Neu	
245	11.09.2014 18:24:00	zss3	Honeypot	192.168.1.10	SSH Port 22 (kippo)	Neu	
244	11.09.2014 17:32:23	zss3	Honeypot	192.168.1.10	SSH Port 22 (kippo)	Neu	
243	11.09.2014 17:11:24	zss3	Portscan	192.168.1.10	586 Pakete	Neu	
242	11.09.2014 17:11:12	zss3	Portscan	192.168.1.10	10 Pakete	Neu	
241	11.09.2014 16:03:12	zss3	Honeypot	192.168.1.10	SSH Port 22 (kippo)	Neu	
240	11.09.2014 16:01:52	zss3	Honeypot	192.168.1.10	SSH Port 22 (kippo)	Neu	
220	11.09.2014 12:07:58	zss3	Verbindungsversuch	192.168.1.10	TCP Port 123	Neu	

1 bis 12 von 12 Einträgen

Zurück 1 Nächste

Abbildung 16: Ereignisliste

Aktionen Erlaubt das Entfernen von Ereignissen und ermöglicht die Auflistung zusätzlicher, ereignisspezifischer Details. Diese präsentieren sich in Form eines modalen Dialogs, der im Falle von Portscans alle betroffenen Ports auflistet und bei Honeypot-Vorfällen möglicherweise zusätzlich gewonnene Informationen über den Eindringling benennt. Hierzu gehören die *Sensorinteraktion*, eine Liste von Ereignissen, die die Vorgehensweise des potentiellen Angreifers vom Verbindungsaufbau bis zum -abbruch darlegen, sowie beliebige weitere, vom Honeypot hinzugefügte Daten (beispielsweise der Name eines erkannten Exploits).

Sensorliste Übersicht über alle im System registrierten Sensoren gibt erneut eine Tabelle, die in Kürze alle nötigen Informationen zusammenfasst – zu sehen in Abbildung 17. Jeder Sensor besitzt eine im System eindeutige *ID* und einen einzigartigen *Namen*. Der *Standort* ist rein informeller Natur, um ihn für Nachforschungen leicht auffindbar zu halten. Die *SW-Version* nennt die aktuell laufende Firmware-Revision, die Spalte *IP-Adresse* ist selbsterklärend. Das *Statusfeld* zeigt in Klammern, wann die letzte Statusnachricht mit dem Server ausgetauscht wurde. Mögliche Zustände sind „Online“, falls die letzte Nachricht innerhalb des frei konfigurierbaren Update-Intervalls liegt, „Timeout“, falls außerhalb, sowie mehrere blau hinterlegte Status, die über den Prozess einer Neuinstallation oder eines Firmware-Updates Auskunft geben. Mit einem Klick auf den Button in der *Zertifikat*-Spalte wird das öffentliche Sensorzertifikat und dessen Fingerprint in einer Dialogbox angezeigt. Die möglichen Aktionen umfassen das Auflisten der letzten Statusmeldungen mit dem jeweiligen Zeitpunkt, der gesendeten Firmware-Revision und zusätzlichen Systemparametern (im Prototypen exemplarisch der noch freie Arbeitsspeicher), das Editieren der Sensordaten (Name, Standort und das Zertifikat können aktualisiert werden) und das Entfernen eines Sensors zusammen mit allen ihn referenzierenden Einträgen im System.

Das Hinzufügen eines neuen Sensors ist über den gleichnamigen Button möglich. Er zeigt das in Abbildung 18 dargestellte Dialogfeld an. Die Felder *Name* und *Standort* sind selbsterklärend. Dass der Name im System nur einmalig vergeben sein darf, wird mit Hilfe der Formularvalidierung sichergestellt. Im Abschnitt „Netzwerkkonfiguration“ kann das primäre Netzwerkinterface des Sensors konfiguriert werden, wahlweise mit *DHCP* für die dynamische Allokation einer IP-Adresse oder durch die Spezifikation eigener, statischer Werte. Die *Server-URL* beschreibt für den Sensor, wie die REST-API des HoneySens-Servers zu erreichen ist. Sie kann für jeden Sensor individuell definiert werden, da diese typischerweise in verschiedenen Subnetzen platziert werden und die Serverinstanz deshalb möglicherweise unter verschiedenen Namen oder IP-Adressen zugleich erreichbar ist, speziell falls der Server *multi-homed* ist, also mehrere IP-Adressen besitzt. Nachdem mit einem Klick auf *Zertifikat erzeugen* vom Server ein neues Paar bestehend aus privatem Schlüssel und gültigem, signiertem Zertifikat erzeugt wurde, kann der Dialog geschlossen werden. Der Server generiert in diesem Moment eine gültige Sensorkonfiguration und bietet sie zusammen mit der aktuellen Firmware (im bootbaren, umgewandelten Format) zum Download an. Die Details des Installationsprozesses werden in Kapitel 9.2 beschrieben.

Sensorkonfiguration Dieser Teilbereich der Anwendung erlaubt zunächst die Vorgabe einer globalen Konfiguration für alle Sensoren, die sich aus der Definition des Update-Intervalls in Minuten sowie einer Auswahl der zu betreibenden Dienste zusammensetzt. Aktive Dienste umfassen im Prototypen den *Passiven Modus*, der ankommende Pakete ohne auf sie zu antworten aufzeichnet, sowie die Honeypots *kippo* und *dionaea*, die im Kapitel zur Sensorimplementierung genauer beschrieben werden. Sensoren fragen ihre Konfiguration nach Ablauf des Update-Intervalls ab und aktualisieren somit ihre lokal vorgehaltenen Einstellungen. Es ist auch möglich, für einzelne Sensoren eine individuelle Konfiguration zu hinterlegen.

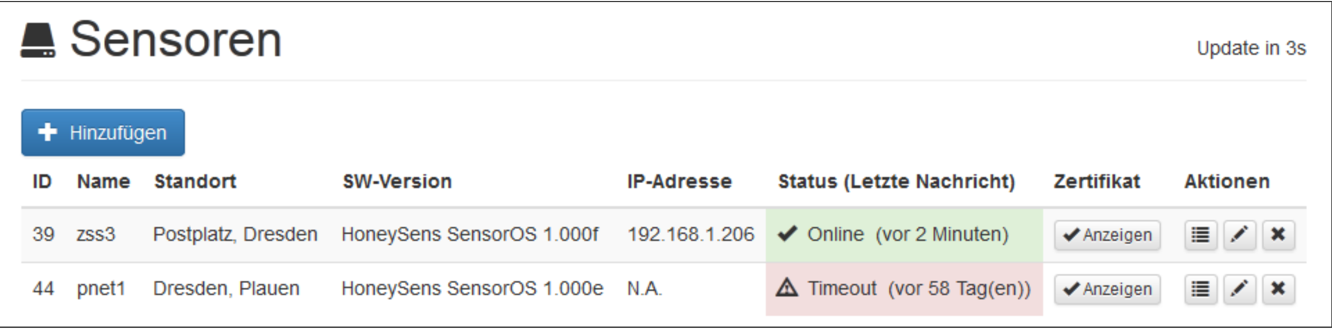


Abbildung 17: Sensorliste

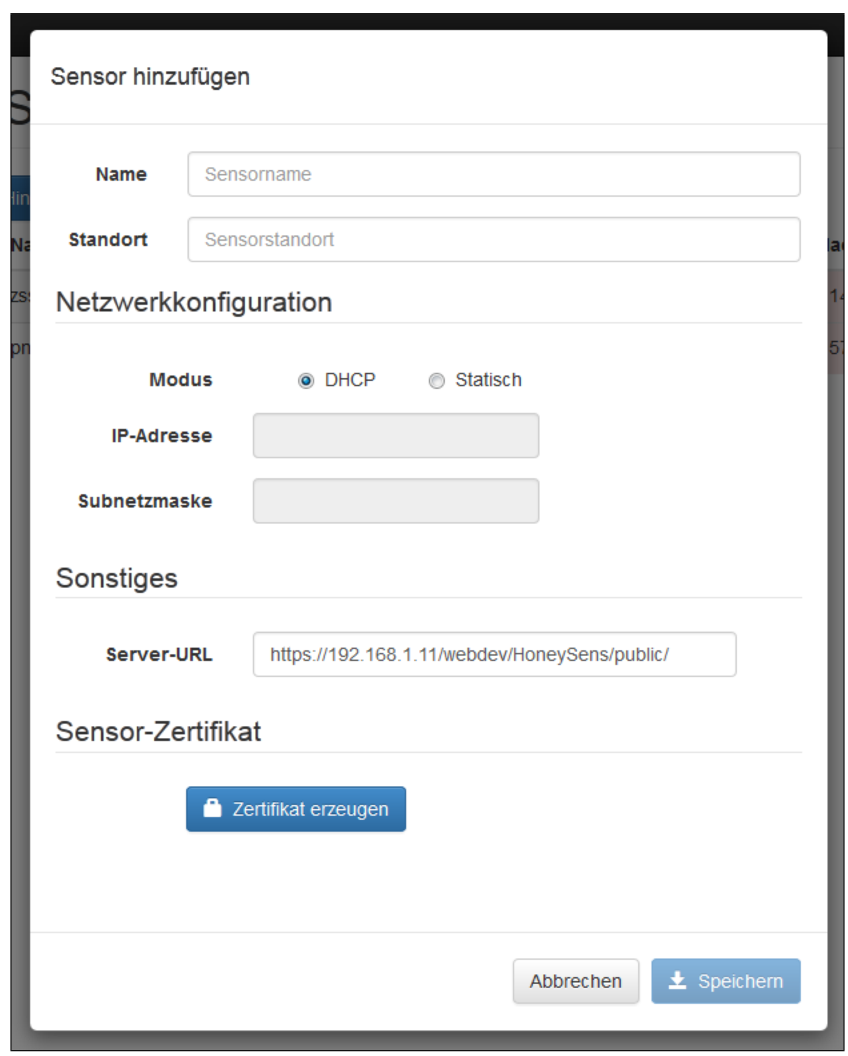


Abbildung 18: Dialog zum Hinzufügen eines Sensors

Dabei stehen die bereits genannten Parameter und zusätzlich die Auswahl einer zu betreibenden Firmware zur Verfügung. Diese Option erlaubt es, Sensoren an die spezifischen Anforderungen des Teilnetzes, in dem sie aufgestellt wurden, anzupassen (und beispielsweise rein passiv zu betreiben). Die gesonderte Firmware-Auswahl kann weiterhin genutzt werden, um eine neue experimentelle Revision zunächst nur mit einzelnen Sensoren zu testen.

Firmware Dem Anwender wird eine Liste aller auf dem Server registrierten Firmware-Images präsentiert. Jeder Eintrag beinhaltet neben dem Namen und der Versionsbezeichnung auch eine einzeilige Zusammenfassung, die bei der Erstellung der Firmware in den Metadaten spezifiziert werden muss. Die Spalte *SD-Image* zeigt an, ob die betreffende Firmware bereits in ein bootbares Abbild einer Speicherkarte umgewandelt wurde (oder dies gerade im Gange ist). Trifft das zu, kann sie über eine gesonderte *Download*-Aktion heruntergeladen werden. Andernfalls wird ein Button dargestellt, der einen zusätzlichen Task zum Job-Server schickt und somit die Umwandlung in Auftrag gibt. Zusätzlich kann ein Firmware-Image noch entfernt oder zum systemweiten Standard erhoben werden. Dies hat zur Folge, dass alle bestehenden Sensoren – falls nötig – ein Update auf diese Version durchführen und dass jeder zukünftig neue Sensor mit dieser Revision ausgestattet sein wird. Über die gleichnamige Schaltfläche kann zudem neue Firmware hochgeladen werden. Es erscheint eine Dialogbox, über die eine Datei aus dem lokalen Dateisystem des Clients ausgewählt und zum Server gesendet werden kann. Nach erfolgtem Upload wird sie vom Server auf Gültigkeit überprüft, um im Erfolgsfall die Metadaten zu extrahieren und das Archiv in die Firmwareliste aufzunehmen.

Einstellungen Dieser Teil der Anwendung erlaubt das Setzen der serverseitigen SMTP-Einstellungen (Server, Absenderadresse, Benutzer und Passwort) und des globalen Server-Endpunkts, der im Dialog zum Hinzufügen eines neuen Sensors die Vorgabeeinstellung ist. Zusätzlich können beliebig viele Kontakte bestimmt werden, an die E-Mails mit wöchentlichen Zusammenfassungen und/oder spontanen kritischen Vorfällen gesendet werden sollen.

Benutzerverwaltung Der letzte Menüpunkt zeigt eine Liste von im System registrierten Benutzern mit ihrer jeweiligen ID, dem Anmeldenamen und ihrer Gruppenzugehörigkeit. Benutzer können hinzugefügt, bearbeitet und auch wieder entfernt werden. Das Interface stellt dabei sicher, dass der Benutzer mit der ID 1 nicht verändert werden kann, um ein Aussperren des Administrators aus dem Frontend zu vermeiden.

7.3 Deployment

Der letzte hinsichtlich der Implementierung zu betrachtende serverseitige Aspekt ist das sogenannte *Deployment*, also das „Nutzbarmachen“ der Software für den produktiven Einsatz. Im Einklang mit den in Teil III der Arbeit genannten Anforderungen soll eine transparente, leicht wartbare Integration in bestehende Netzwerkstrukturen möglich sein. Bei der Serverkomponente handelt es sich um ein Webprojekt, das einen Web-Server, eine Datenbank, Python in Version 2 oder 3 und den Beanstalkd-Job-Server voraussetzt. Die Python-Skripte nutzen im Prototypen Werkzeuge, die ausschließlich unter Linux zur Verfügung stehen. Eine Migration dieser Skripte ist allerdings denkbar und würde den Betrieb der Software auch auf anderen Betriebssystemen ermöglichen, da der überwiegende Teil der Anwendung plattformunabhängig gestaltet ist. Je nach für den Server angedachter Soft- und Hardwareplattform ergeben sich daraus einige Varianten zur Inbetriebnahme:

- Der Einsatz eines **Bare-Metal**-Serversystems, das entweder als reiner HoneySens-Server aufgesetzt wird oder noch zusätzliche weitere Dienste anbietet. Dies erfordert die Installation und Konfiguration aller genannten Serverprozesse. Speziell der Apache HTTP Server und dessen PHP-Modul benötigen je nach Distribution eine Reihe von Konfigurationsänderungen für ein reibungsloses Routing der API-Anfragen und den Firmware-Upload. Die HoneySens-Software muss jedoch nicht gesondert vorbereitet werden, es genügt diese auf den zukünftigen Server zu kopieren, in der Datei `app/config.cfg` die Zugangsdaten zur Datenbank einzutragen und – falls nötig – weitere Einstellungen wie den Pfad zum Serverzertifikat oder den Port des Beanstalkd-Dienstes anzupassen. Für die Initialisierung der Datenbank sorgt ein PHP-CLI-Skript, das im `utils/`-Verzeichnis der HoneySens-Distribution enthalten ist. Zudem muss sichergestellt sein, dass die Verzeichnisse `cache` und einige Unterverzeichnisse von `data` für den Server beschreibbar sind. Nachteil dieser Lösung ist, dass die Softwareumgebung jedes weiteren HoneySens-Servers erneut an die spezifischen Anforderungen der Software angepasst werden muss, was speziell auf einem Server, der mehrere Webangebote gleichzeitig bereitstellt, zu Konflikten führen kann.
- Nutzung von **Virtualisierung** (QEMU/KVM, VirtualBox, VMWare...): Die zuvor genannten Schritte zur Installation der Software und aller nötigen Abhängigkeiten sind nur ein einziges Mal zum Erstellen einer virtuellen „Referenzinstanz“ nötig. Diese kann anschließend beliebig oft geklont werden, um mehrere HoneySens-Server in Betrieb nehmen zu können. Das virtuelle System läuft anschließend als abgeschotteter Container, was die Folgen für die IT-Sicherheit als Konsequenz einer eventuellen Übernahme durch einen Angreifer reduziert. Das Betriebssystem des physischen Hosts spielt für den Betrieb keine Rolle und muss lediglich sicherstellen, dass die Schnittstellen HTTP, HTTPS und bei Bedarf SSH von außerhalb erreichbar sind. Ein weiterer Vorteil ist die Möglichkeit der schnellen Migration von Instanzen zwischen verschiedenen physischen Hosts, was sich als nützlich erweisen kann, wenn sich die Netzwerkstruktur ändert oder zusätzliche physische Ressourcen benötigt werden [33]. Ein wichtiger Nachteil der Lösung ist eine im Vergleich zum nativen Betrieb auf einem Host reduzierte maximal erzielbare Performance der virtuellen Maschinen, speziell da innerhalb einer virtuellen Instanz ein weiteres vollständiges Betriebssystem simuliert wird. Zusätzlich erfordert das System, nachdem es in Betrieb genommen wurde, genauso umfangreiche Wartungsprozeduren wie ein physischer Host. Es muss gegen Angriffe abgeschirmt und regelmäßig mit Sicherheitsupdates versorgt werden.
- Eine leichtgewichtige Alternative zur Virtualisierung stellt das Konzept der **Container** (auch: OS-Virtualisierung) dar, die ebenfalls alle benötigte Software innerhalb einer virtuellen Instanz betreiben, allerdings keine Hardware simulieren. Stattdessen nutzen sie direkt den Betriebssystem-

> Kernel und die Ressourcen des Hostsystems, was sich in einer nahezu nativen Performance widerspiegelt. Größte Einschränkung dieser Technik ist, dass es architekturbedingt nicht möglich ist, Instanzen verschiedener Betriebssysteme zu „mischen“, also beispielsweise einen Windows-Container auf einem Linux-Host zu betreiben [33]. Container beinhalten zumeist nur eine minimale, für den Betrieb der angebotenen Dienste nötige Softwareausstattung und besitzen umfangreiche Werkzeuge zur Manipulation, was die Migration und das Deployment zusätzlich vereinfacht. Verbreitete Containerlösungen sind die auf FreeBSD verfügbaren *Jails*[57], das für Linux entwickelte *OpenVZ*[58] und das zum Zeitpunkt dieser Arbeit bisher nur auf Linux verfügbare, aber mit austauschbarem Backend entwickelte *Docker*[59].

Da der HoneySens-Server keine speziellen Hardwareanforderungen stellt und eine Beschränkung auf Linux-Systeme im Einklang mit den Anforderungen des SVNs steht, wurde das noch sehr junge **Docker**-Projekt für das HoneySens-Deployment ausgewählt. Es steht zum Zeitpunkt dieser Arbeit in Version 1.2.0 bereit und überzeugt mit einem umfangreichen Toolkit, das den Anwender während des *Lifecycles* eines Containers vom Erstellen eines Abbildes über dessen Produktiveinsatz bis hin zu Migration und Update unterstützt [21]. Eine weitere Stärke im Vergleich zu ähnlichen Lösungen ist die Integration eines Online-Repositorys mit freien, direkt nutzbaren Docker-Images[60], die ihrerseits wieder Basis eines neuen, selbst erstellten Abbildes sein können. Somit existieren bereits unzählige, virtuelle Instanzen mit Serversoftware und Entwicklungsumgebungen aller Art. Das `docker`-Kommandozeilenwerkzeug greift auf das Repository direkt zu und lädt alle benötigten Abhängigkeiten bei Bedarf nach, sodass allein der Aufruf von

```
docker run -t -i ubuntu:14.04 /bin/bash
```

genügt, um in einem *Ubuntu Linux* in Version 14.04 eine Shell zu starten. Das Abbild des Betriebssystems wird beim Absetzen des Befehls automatisch heruntergeladen, falls es lokal noch nicht vorhanden ist. Anschließend erstellt und startet der lokale `docker`-Daemon eine neue virtuelle Instanz auf Grundlage des Ubuntu-Images, in der schließlich die Anwendung `/bin/bash` als alleinige PID 1 ausgeführt und die weitere Kontrolle an den Benutzer übergeben wird. Container können selbstverständlich auch als Daemon im Hintergrund gestartet werden, was der bevorzugte Modus für Serverinstanzen ist. Das Docker-Design ist zwar betriebssystemunabhängig, zum Zeitpunkt dieses Projektes existierte allerdings nur ein exklusiv unter Linux-Systemen nutzbares Backend, das auf die nur unter diesem Kernel verfügbaren *Control Groups* und *Namespaces* zur Isolation der Prozesse und Ressourcen der Container setzt. Für andere Betriebssysteme existieren jedoch offizielle Lösungen, die mit Hilfe von Virtualisierung den Docker-Betrieb innerhalb einer zusätzlichen virtuellen Maschine ermöglichen. Für Microsoft Windows wurde zudem bereits eine native Implementierung angekündigt [45].

Erstellung des HoneySens-Images Um einen Docker-Container zu erstellen, empfiehlt sich der Einsatz eines *Dockerfiles*. Die gleichnamige Datei ähnelt in ihrer Funktion einem *Makefile* und enthält Anweisungen, wie das Image zu erstellen ist. Über das `docker`-Tool wird schließlich ein *Build*-Prozess angestoßen, der die Anweisungen im Dockerfile zeilenweise einliest und im fertigen Image vermerkt. Das Codebeispiel in Abbildung 19 zeigt einen Ausschnitt aus dem HoneySens-Dockerfile. Die erste Zeile be-

[57] `https://freebsd.org/doc/handbook/jails.html` (abgerufen im Oktober 2014)
[58] `http://openvz.org` (abgerufen im Oktober 2014)
[59] `http://docker.io` (abgerufen im Oktober 2014)
[60] `https://registry.hub.docker.com` (abgerufen im Oktober 2014)

stimmt mit der Anweisung `FROM`, welches bereits existierende Docker-Image als Basis eingesetzt werden soll. Das hier genutzte *baseimage-docker*[61] beinhaltet ein minimales Ubuntu, das eine Reihe von Modifikationen zum Einsatz mit Docker erfahren hat. Es stellt unter anderem den Daemon `/sbin/my_init` zur Verfügung, der als rudimentäres Init-System agiert und alle Prozesse im Container verwaltet. Zusätzlich sind Basisdienste wie *Syslog*, *SSH* und *cron* bereits vorinstalliert. Die `MAINTAINER`-Anweisung ist rein informeller Natur und wird zu den Metadaten hinzugefügt, während via `ENV` Umgebungsvariablen bestimmt und später innerhalb des Containers gesetzt werden. `DEBIAN_FRONTEND` deaktiviert beispielsweise die interaktiven Nachfragen während der Installation neuer Pakete, die bei `deb`-basierten Distributionen wie Debian und Ubuntu üblich sind. Dies garantiert, dass die Installation zusätzlicher Pakete unbeaufsichtigt verlaufen kann.

```
FROM phusion/baseimage:0.9.15
MAINTAINER Pascal Brueckner <pascal.brueckner@mailbox.tu-dresden.de>
ENV HOME /root
ENV DEBIAN_FRONTEND noninteractive
...
ADD add /opt/HoneySens/app/
ADD lib /opt/HoneySens/lib/
...
RUN /bin/bash /opt/HoneySens/utils/docker/setup.sh
EXPOSE 22 80 443
CMD ["/sbin/my_init"]
```

Abbildung 19: HoneySens-Dockerfile

Anschließend fügt das Skript all jene Dateien aus dem HoneySens-Projektverzeichnis zum Image hinzu (nach `/opt/HoneySens`), die für den Betrieb der Serveranwendung benötigt werden. Nachdem dies erfolgt ist, wird das Skript `setup.sh` gestartet, das das zukünftige Serversystem konfiguriert. Es unternimmt im Wesentlichen folgende Schritte:

1. Update des Basissystems und Installation aller benötigten Pakete und deren Abhängigkeiten, darunter der Apache HTTP Server, Beanstalkd, MySQL, Python und OpenSSL.
2. Anpassung der Benutzerberechtigungen des HoneySens-Projekts, damit Verzeichnisse wie `cache/` oder `data/upload/` beschreibbar sind.
3. Start des MySQL-Daemons, Hinzufügen eines Benutzers `honeysens` und Anlegen der zugehörigen Datenbank.
4. Modifikation der Konfigurationsdateien der installierten Serverdienste zur Anpassung an die Docker-Gegebenheiten.
5. Aktivieren der Module `rewrite` und `ssl` des Apache-Servers sowie Konfiguration der anzubietenden Websites.
6. Kopieren der HoneySens-Skripte `01_regen_selfsigned_cert.sh` und `02_regen_honeysens_ca.sh` in ein Initialisierungsverzeichnis, sodass beim Erstellen eines neuen Containers automatisch alle Zertifikate neu generiert werden.
7. Kopieren der Serviceskripte von `utils/docker/services/` in das dafür zuständige Verzeichnis `/etc/service`.

[61] `https://github.com/phusion/baseimage-docker` (abgerufen im Oktober 2014)

Das Schlüsselwort `EXPOSE` definiert die TCP-Ports, die ein Container anbieten soll, und `CMD` legt fest, welcher Prozess später in der virtuellen Instanz zu starten ist. Nach dem Ende des Build-Prozesses steht auf dem Entwicklungssystem ein Docker-Image des HoneySens-Servers zur Verfügung, das mit `docker save` exportiert und auf andere Docker unterstützende Rechner übertragen werden kann.

Betrieb des Containers Wenn ein gültiges Docker-Image vorliegt, muss es mit `docker run` nur noch als neuer Container gestartet werden. Eine vollständige Befehlszeile inklusive aller nötigen Parameter könnte lauten

```
docker run -d -h honeysens-server -p 2222:22 \
           -p 80:80 -p 443:443 --privileged pbrueckner/honeysens
```

Mit `-d` ist sichergestellt, dass der Prozess im Hintergrund startet. Die `-p`-Parameter bestimmen die Mappings der lokalen Ports auf die des Containers und `-h` legt den Hostnamen der Serverinstanz fest, was für die Erzeugung der Zertifikate von Bedeutung ist. Über `--privileged` erhält der Container erweiterte Rechte, die benötigt werden, um `loop`-Devices im Linux-Kernel des Hostsystems zu registrieren. Diese Funktion ist essentieller Bestandteil des Skriptes, das Firmware-Archive in bootbare Speicherkartenabbilder umwandelt. Der letzte Parameter ist der Name des Images, aus dem der neue Container erstellt werden soll. Später können die Befehle `docker start <id>` und `docker stop <id>` genutzt werden, um das HoneySens-System herunterzufahren oder neu zu starten. Die Administration des Containers kann zudem über SSH erfolgen, das zum Einloggen nötige Schlüsselpaar wird ebenfalls bei der Konfiguration der Dienste innerhalb der virtuellen Instanz registriert.

Abschließend kann festgehalten werden, dass Docker nach einmaligem Einrichten der Build-Umgebung das Deployment der Serversoftware stark vereinfacht hat. Das Image kann jederzeit aus den Quellen des Projektes neu erstellt werden, beispielsweise wenn Änderungen am Server-Code vorgenommen wurden. Es ist nicht nötig, das zugrundeliegende Basissystem gesondert zu warten, da dieses beim Bauen einer neuen Image-Revision automatisch neu heruntergeladen und später vom Setup-Skript mit aktuellen Sicherheitsupdates versorgt wird.

Während der Prototyp-Testphase hat sich herausgestellt, dass ein Update des laufenden Servers beschwerlich ist. Manueller Zugriff via SSH ist für Updates zwar möglich, aber nicht sehr praktikabel. Zur Lösung des Problems wäre die Nutzung des `VOLUME`-Befehls im Dockerfile denkbar, mit dem Verzeichnisse des Hostsystems in den Container eingebunden werden können. Auf diese Art können alle veränderlichen Anwendungsdaten, wie beispielsweise das Verzeichnis der Datenbank, an separater Stelle gesichert und auch in Backup-Zyklen mit eingeschlossen werden. Nach einem anschließenden Austauschen des HoneySens-Images durch eine neuere Version genügt es, die Verzeichnisse mit den bestehenden Daten wieder einzubinden.

8 Sensoren

In diesem Kapitel soll ein Blick auf die Implementierung der Sensorsoftware geworfen werden. Deren Hauptaufgabe ist das Detektieren und Melden von verdächtigen, über das Netzwerk empfangenen Paketen oder Datenströmen. Die nachfolgenden Abschnitte beschreiben zunächst die Hardwareplattform, gefolgt von einem Abriss der entwickelten Sensorsoftware.

8.1 Hardwareplattform und Betriebssystem

Dass das reine Monitoring im Sächsischen Verwaltungsnetz keine hohen Systemanforderungen mit sich bringt, war eine Schlussfolgerung aus der in Kapitel 5 durchgeführten Trafficanalyse. Die in diesem Zusammenhang eingesetzten Kleinstrechner vom Typ *BeagleBone Black* waren der Aufgabe gewachsen und überzeugten zudem durch einen geringen Stromverbrauch und kleinen Formfaktor. Aus diesen Gründen und der Tatsache, dass eine größere Menge solcher Boards aufgrund der Analyse bereits vorhanden war, wurde die Hardware als Plattform für den Prototypen ausgewählt. Dies erwies sich im Laufe der Entwicklung als gute Entscheidung, da sie im Gegensatz zu vergleichbaren Plattformen wie dem *Raspberry Pi*[62] einen integrierten nichtflüchtigen Speicher besitzen, der im Zusammenspiel mit einer zusätzlichen Speicherkarte als temporärer Zwischenspeicher für vollständige Systemupdates genutzt werden konnte. Die Implementierung der Updateprozedur wird ausführlich als eine der Problemstellungen des neunten Kapitels beschrieben. Weiterhin erwuchs aus der Anforderungsanalyse der Wunsch, die Sensoren auch über *Power over Ethernet* (PoE) mit Strom versorgen zu können, was im Falle des BeagleBone Black über speziell für diese Plattform entwickelte Adapterkabel möglich ist.

Als Betriebssystem wurde erneut die speziell an das BeagleBone angepasste ARM-Variante von *Debian GNU/Linux*[63] installiert, mit der bereits während der Trafficanalyse gute Erfahrungen gesammelt wurden. Die Standardinstallation ist sehr schlank gehalten und besteht nur aus für den Betrieb zwingend notwendigen Paketen, weshalb die Distribution sehr platzsparend ist. Da das HoneySens-System sowohl vom 2 GB großen internen eMMC-Speicher als auch von externen Karten gestartet werden soll, ist dies zugleich auch die Maximalgröße für die Sensorsoftware.

Die Firmware der Sensoren übernimmt strikt das durch die Debian-Installation entstandene Partitionsschema: Eine 100 MB große Boot-Partition wird in `/boot/uboot/` eingehängt und ist mit dem Dateisystem *FAT32* formatiert, während das restliche System auf einer einzelnen mit *ext4* formatierten Root-Partition liegt. Grund für die Wahl der FAT-Boot-Partition ist, dass das Dateisystem von allen gängigen Betriebssystemen nativ unterstützt wird und somit überall sofort gemountet werden kann. Die Partition muss zudem zwingend dieses Format aufweisen, damit das Board überhaupt bootet.

[62] `http://raspberrypi.org` (abgerufen im November 2014)
[63] `http://debian.org` (abgerufen im Oktober 2014)

8.2 Implementierung

Die Sensorfirmware besteht aus dem Basisbetriebssystem Debian GNU/Linux, einer Reihe von im Rahmen des Prototypen entwickelten Skripten und für die Verwendung im HoneySens-Projekt modifizierter Honeypot-Software. Zunächst soll ein Überblick über diese Komponenten und ihr Layout im Dateisystem gegeben werden, woraufhin eine detaillierte Beschreibung der Management-Skripte und der laufenden Dienste folgt, die für die Sensorfunktionalität verantwortlich sind. Die Software wurde überwiegend in *Python* implementiert, für einige Management-Funktionen erwies sich jedoch aufgrund der starken Abhängigkeit von Systemwerkzeugen das Schreiben eines *Bash*-Shellskriptes als effektiver.

8.2.1 Komponenten

Jegliche zusätzliche, nicht über das Debian-Repository beziehbare Software wurde im dafür vorgesehenen Verzeichnis `/opt/` abgelegt, was Abbildung 20 zu visualisieren versucht. Die Verzeichnisse `dionaea/` und `kippo/` beinhalten die gleichnamigen Honeypots, die im Anschluss näher beschrieben werden. In `honeysens/` liegen hingegen alle weiteren für den Sensorbetrieb erforderlichen Skripte.

Die Shell-Skripte `am335x_evm.sh` und `beaglebone-black-flasher.sh` wurden mit der Debian-Distribution für die BeagleBone-Boards ausgeliefert und für das HoneySens-Projekt geringfügig modifiziert. Ersteres wird von einem der während des Bootens laufenden Skripte ausgeführt und hat u.a. die Aufgabe, die Ethernet-over-USB-Schnittstelle zu initialisieren und einen Flashvorgang des laufenden Systems vom internen Speicher auf eine SD-Karte oder umgekehrt zu starten, falls die „Trigger"-Datei `/boot/uboot/flash-eMMC.txt` existiert. Es wurde dahingehend modifiziert, dass es stattdessen das `*flasher.sh`-Skript im HoneySens-Verzeichnis ausführt. Dieses teilt wiederum im Gegensatz zum Originalskript den aktuellen Status des Flashvorgangs zusätzlich noch über die vier steuerbaren LEDs des Boards mit.

Das Perl-Skript `fat-set-uuid.pl` stammt ebenfalls aus einer externen Quelle[64] und dient dazu, das Label bzw. die *UUID* (Universally Unique Identifier) einer FAT-Partition zu ändern. Das ist mit den im Repository verfügbaren Werkzeugen nicht möglich, weshalb auf eine gesonderte Lösung zurückgegriffen werden musste. Die Funktionalität wird während der Installations- und Update-Prozesse benötigt. Das Shell-Skript `firewall.sh` initialisiert die *Netfilter*-Firewall des Linux-Kernels und gewährleistet die Erreichbarkeit der Honeypot-Dienste und die Funktionalität des nachfolgend beschriebenen *Passive Scan Modes*. Es wird aus dem Skript `/etc/rc.local` heraus aufgerufen und somit bei jedem Bootvorgang ausgeführt. In diesem wird zudem das *USB Autosuspend* abgeschaltet, welches den Energieverbrauch reduzieren soll. Aus unbekannten Gründen bereitete dieses Feature Probleme im Zusammenhang mit den Honeypot-Diensten, nach deren Start sich die Log-Dateien des Sensorsystems sekündlich mit USB-Fehlern füllten. Die Datei `version.txt` enthält den Namen und die Revision der aktuell betriebenen Firmware als Referenz für potentielle Systemupdates.

[64] `https://github.com/nickandrew/fat-set-uuid` (abgerufen im Oktober 2014)

```
- opt/
|--- honeysens/
| |--- Crypto/
| |--- requests/
| |--- am335x_evm.sh
| |--- beaglebone-black-flasher.sh
| |--- config_sensor.sh
| |--- fat-set-uuid.pl
| |--- firewall.sh
| |--- honeysens.cfg
| |--- passive_scan.py
| |--- poll.py
| |--- set_mode.py
| |--- update.sh
| |--- version.txt
|--- dionaea/
|--- kippo/
```

Abbildung 20: Sensor-Verzeichnisstruktur

Die Verzeichnisse `Crypto/` und `requests/` enthalten das „Python Cryptography Toolkit“ *PyCrypto*[65] respektive die HTTP-Bibliothek *Requests*[66], die die Interaktion mit einer REST-API innerhalb von Python-Skripten für den Entwickler stark vereinfacht. Beide werden von allen Anwendungen eingesetzt, die direkt mit dem HoneySens-Server kommunizieren. Die Datei `honeysens.cfg` hält die letzte bekannte Sensorkonfiguration vor und wird von den anderen Skripten eingelesen, um beispielsweise die URI des Servers zu erfahren. Alle verbleibenden Dateien dienen entweder dem Management oder der Honeypot-Funktionalität des Sensors.

8.2.2 Management

Die Sensoren wurden für den autonomen Betrieb konzipiert und hängen deshalb in ihrer Funktion stark vom HoneySens-Server ab, der sie mit aktuellen Konfigurationsdaten versorgt und aufgezeichnete Ereignisse entgegennimmt und speichert. Die wichtigste Managementaufgabe, die der Aufrechterhaltung des Sensorbetriebs dient, ist das periodische *Polling*. Jeder Sensor kennt hierfür ein Zeitintervall, nach dessen Ablauf er einen Abriss des aktuellen Systemzustandes über einen gesicherten Kanal zum Server schickt und von diesem die aktuelle Sensorkonfiguration zurückerhält. Eventuelle Änderungen an den laufenden Diensten oder dem Update-Intervall werden sofort lokal übernommen. Falls der Server einmal nicht erreichbar ist, versucht der Sensor weiterhin in regelmäßigen Abständen, diesen zu erreichen. Der Stand der letzten empfangenen Konfiguration bleibt dabei erhalten und übersteht auch Neustarts des Systems ohne jeden Serverkontakt.

Der Polling-Vorgang ist im umfangreichen Skript `poll.py` implementiert, das innerhalb eines frei wählbaren Intervalls über einen *Cron-Job* aufgerufen wird. Während der Testläufe stellte es sich jedoch als

[65] `https://dlitz.net/software/pycrypto/` (abgerufen im Oktober 2014)
[66] `http://python-requests.org` (abgerufen im Oktober 2014)

Problem heraus, dass beispielsweise ein Intervall von 20 Minuten, in die `crontab`-Syntax übersetzt `*/20`, dazu führt, dass alle Sensoren zugleich zum Zeitpunkt der Minuten 0, 20 und 40 einer jeden Stunde den Polling-Vorgang starten. Dieses Verhalten ist nicht optimal, da die Serverlast mit wachsender Sensoranzahl in diesen kurzen Momenten deutlich ansteigt, während das System die restliche Zeit überwiegend unbeschäftigt ist. Aus diesem Grund wurde eine für jeden Sensor individuelle Verteilung der exakten Zeitpunkte bei Beibehaltung des gleichen Intervalls angestrebt. Als Lösung des Problems wurde schließlich ein Verfahren implementiert, das aus dem systemweit eindeutigen Hostnamen eines Sensors (der dem „Namen“ im Webinterface entspricht) einen Hashwert berechnet, der über eine Modulo-Operation mit dem geforderten Intervall eine *Startverzögerung* ergibt, die jeder Sensor zu Beginn einer vollen Stunde in den Zyklus einfügt. Der zugehörige Codeausschnitt kann zum besseren Verständnis in Abbildung 21 betrachtet werden. Die von der Funktion in das Cron-Skript eingefügte Zeile beinhaltet eine Minutendefinition der Form `5-59/20`, womit der Job beispielsweise immer zu den vollen Minuten 25, 45 und 05 ausgeführt wird.

```
def setInterval(interval):
  sensor = open('/etc/hostname', 'r').read().strip()
  delayedStart = int(hashlib.sha1(sensor.encode('ascii')) \
          .hexdigest(), 16) % interval
  with open('/etc/cron.d/honeysens', 'r') as f:
    fc = f.readlines()
  if fc[0].split([0] != '{}-59/{}'.format(delayedStart, interval):
    with open('/etc/cron.d/honeysens', 'w') as f:
      f.write('{}-59/{}␣*␣*␣*␣*␣root␣/opt/honeysens/poll.py' \
        + '>/dev/null␣2>&1\n'.format(delayedStart, interval))
```

Abbildung 21: Funktion zur Ermittlung des sensorspezifischen Polling-Delays

Das Skript `poll.py` orientiert sich bei der Vorgehensweise zur Durchführung des Pollings an den in Kapitel 6.4 beschriebenen konzeptionellen Abläufen. Die im Prototypen gesammelten Statusdaten sind der Zeitpunkt der Skriptausführung, ein allgemeiner Statuscode (Online/Installation/Update), die IP-Adresse des primären Netzwerkinterfaces, der freie Arbeitsspeicher und die betriebene Firmware-Revision. Jeder Sensor besitzt einen individuellen privaten Schlüssel, der bei dessen Installation vom Server generiert und im Zuge der Initialkonfiguration auf den Sensor verschoben wurde. Er wird benutzt, um über alle genannten Daten eine Signatur mit dem sensoreigenen Zertifikat zu bilden, die zusätzlich mit zum Server an die `sensorstatus`-Ressource der API via `HTTP-POST` gesendet wird. Der Server validiert die Signatur und antwortet im Erfolgsfall mit der serialisierten Sensorkonfiguration. Im Prototypen ist die Rückmeldung des Servers nicht signiert, da die Kommunikation mit diesem bereits über einen verschlüsselten Kanal erfolgt, bei dessen Initialisierung eine Überprüfung des Serverzertifikats stattfindet. Zur vollständigen Absicherung wäre eine Erweiterung dieses Prozesses jedoch denkbar. Die vom Server empfangenen Konfigurationsdaten werden dann vom Skript geparst und in die Datei `honeysens.cfg` geschrieben. Falls der Server nicht mehr erreichbar ist, kann daraus jederzeit die letzte bekannte Konfiguration gelesen werden. Anschließend wird das Skript `set_mode.py` ausgeführt, das die aktuelle Konfiguration erneut einliest und Honeypot-Dienste den Anweisungen entsprechend startet oder beendet. Letzter Schritt ist ein Vergleich der aktuellen mit der vom Server gewünschten Firmware-Revision. Falls eine Abweichung besteht, wird durch Aufruf des Skriptes `update.sh` die Prozedur zum Firmware-Update gestartet, die der zur Installation ähnelt und auf die im Anschluss in Kapitel 9 genauer eingegangen wird. Während ein Sensor die Installations- oder Updateprozedur durchläuft, wird der Pollingvorgang automatisch jede

Minute wiederholt, um dem Benutzer im Frontend zeitnah den aktuellen Status und das Ende des Vorgangs zügig mitteilen zu können. In diesem Fall unterbricht das Skript `poll.py` bereits seine Arbeit, noch bevor die neue Sensorkonfiguration geschrieben wurde.

Bei den Prototyp-Testläufen entstand die Situation, dass gerade gestartete Sensoren erreichbar waren, jedoch nicht beim Server registriert wurden. Bei der Ursachenforschung stellte sich die Zeitsynchronisation der Geräte als Flaschenhals heraus. Die BeagleBone-Plattform besitzt keine batteriegestützte Hardware-Uhr und verliert somit bei einem „kalten“ Neustart die Systemzeit. In der Folge schlug die Überprüfung des Serverzertifikats fehl, da dieses aus Sicht des Sensors in der Zukunft ausgestellt wurde. Da im Zuge der Anforderungsanalyse eine möglichst große Unabhängigkeit von externer Serverinfrastruktur gefordert ist, synchronisieren sich die Sensoren nun nicht über das zu diesem Zweck üblicherweise genutzte *Network Time Protocol*, sondern direkt beim Web-Server über HTTP. Der sendet die aktuelle Serverzeit im Header einer jeden HTTP-Nachricht, was ebenfalls zur Synchronisation mit geringfügigen Abweichungen von ca. 0,5 Sekunden verwendet werden kann. Das Tool *htpdate*[67] stellt einen Daemon bereit, der die Systemzeit auf diesem Weg aktuell hält und deshalb auf den Sensoren zum Einsatz kommt. Falls während des Polling-Vorgangs vom Server die Fehlermeldung `Invalid Timestamp` zurückgegeben wird, erzwingt das Skript zunächst eine Aktualisierung der Systemzeit und wiederholt anschließend den Vorgang.

Falls ein manueller Eingriff in den Sensorbetrieb gewünscht ist, bietet jeder Sensor über seine USB-Schnittstelle eine virtuelle Ethernetverbindung und einen darüber erreichbaren SSH-Server an. Über die physische Ethernet-Schnittstelle ist dieser jedoch nicht zu erreichen. Die Zugangsdaten für den Dienst sind im Prototypen fester Bestandteil der Firmware. Die für das Sensor-Management essentiellen Prozesse für Installation und Update sind komplexer und werden aus Sicht des Gesamtsystems erst in Kapitel 9 genauer beschrieben.

8.2.3 Passive Scan Mode

Dieser Betriebsmodus ist einer der von den Sensoren angebotenen Dienste, die individuell aktiviert werden können. Seine Hauptaufgabe ist das Detektieren von TCP-Verbindungsversuchen oder UDP-Paketen an Ports, auf denen kein passender Service betrieben wird. Alle auf diese Art aufgezeichneten Ereignisse erscheinen im Web-Frontend als *Verbindungsversuche* samt Quell-IP-Adresse, betroffenem Protokoll und Port. Die Anwendung arbeitet rein passiv und kann somit auf derartige Anfragen nicht reagieren, weshalb es bei solchen Vorfällen im Verantwortungsbereich der zuständigen Administratoren liegt, die Ursache zu ergründen. Das Skript stellt weiterhin eine rudimentäre *Portscan*-Erkennungsroutine bereit, die eine Vielzahl eintreffender Pakete von einer einzelnen Quelle entsprechend klassifiziert und nur ein einzelnes Ereignis beim HoneySens-Server registriert.
Die genannte Funktionalität ist im Skript `passive_scan.py` implementiert und basiert auf dem Netfilter-Framework des Linux-Kernels. Während des Bootvorgangs werden Firewall-Regeln definiert, woraufhin alle Anfragen an Ports, die nicht von einem der separaten Honeypot-Dienste abgedeckt werden, in einer gesonderten Protokolldatei vermerkt werden. Das Skript registriert für diese anschließend mit Hilfe des `select`-Moduls der Python-Bibliothek ein *Polling*-Objekt, um auf Änderungen sofort reagieren zu können. Das Netfilter-Log wird anschließend zeilenweise gemäß dem in Abbildung 22 gezeigten Pseudocode verarbeitet.

[67] `http://vervest.org/htp/` (abgerufen im Oktober 2014)

```
while True:
  testForScan() # move suspicious events to scan queue
  if length(scanQueue) > 0:
    completedScans = []
    for event in scanQueue:
      if timeSinceAdded(event) >= 10s:
        completedScans.append(event)
    if length(completedScans) > 0:
      sendEvents(completedScans)
      refresh(scanQueue)
  if length(eventQueue) >= 50 or (length(eventQueue) >= 1 \
        and timeSinceAdded(eventQueue.last()) >= 5):
    sendEvents(eventQueue)
    refresh(eventQueue)
  if newDataFromLog():
    event = composeFromRawData()
    eventQueue.append(event)
  else:
    sleep(1s)
```

Abbildung 22: Arbeitsweise des Passive Scan Modes (Pseudocode)

Dieser zeigt die Existenz von zwei Warteschlangen auf: der „*Event Queue*“, in der zunächst alle eintreffenden Ereignisse abgelegt werden, und der „*Scan Queue*“, die (nach IP-Adresse der Quelle sortiert) alle Ereignisse speichert, die mutmaßlich zu einem Scanversuch gehören. Die Funktion `testForScan()` verschiebt dabei immer dann Ereignisse in die Scan-Warteschlange, wenn für die entsprechende Quell-IP-Adresse bereits ein Eintrag existiert oder aber Ereignisse für mehr als drei unterschiedliche Ports gleichzeitig (von einer einzelnen Quell-IP-Adresse) in der Ereigniswarteschlange vorgefunden werden. In diesem Fall wird ein neuer Eintrag in der Liste der gefundenen Scans eingefügt. Wenn nun einem solchen Scan für mehr als zehn Sekunden keine weiteren Pakete mehr zugeordnet werden können, wird über `sendEvents()` der HoneySens-Server mittels einer `HTTP-POST`-Nachricht an die `events`-Ressource vom Vorfall informiert. Ähnlich wird mit der Ereigniswarteschlange verfahren, wenn seit mindestens fünf Sekunden keine neuen Pakete hinzugekommen sind oder eine Menge von 50 Ereignissen erreicht wurde, woraufhin der bisherige Zwischenstand dem Server mitgeteilt wird. Dabei handelt es sich um eine Schutzmaßnahme, um einen Speicherüberlauf im Sensor zu verhindern. Sie nutzt die Fähigkeit der REST-API, mehrere Ereignisse in einer einzelnen Anfrage verarbeiten zu können. Die genannten Parameter des Skriptes wurden in mehreren Testläufen mit dem Scanwerkzeug *nmap*[68] ermittelt. Eine potentielle Erweiterung des Prototypen könnte die Integration dieser Variablen in die Sensorkonfiguration sein, sodass eine exakte Anpassung an die Gegebenheiten der jeweiligen IT-Landschaft möglich ist.

[68] `http://nmap.org/` (abgerufen im November 2014)

8.2.4 Honeypots

Die prototypische HoneySens-Implementierung liefert mit der Sensor-Firmware zwei Honeypot-Dienste aus, die auf Basis der Erfahrungen der Trafficanalyse in Kapitel 5 und der von der ENISA durchgeführten Analyse von Honeypot-Software [14] ausgewählt wurden. Beide können über die Sensorkonfiguration gezielt aktiviert oder abgeschaltet werden.

kippo Diese von den Autoren selbst als *Medium-Interaction-Honeypot* eingestufte Software[69] simuliert einen vollwertigen SSH-Dienst auf dem TCP-Port 22. Das Programm ist vollständig in Python implementiert und muss deshalb nicht erst gesondert kompiliert werden, die Integration in die Sensor-Firmware verlief deshalb nach Installation aller Abhängigkeiten aus dem Debian-Software-Repository reibungslos. Seine größte Stärke spielt der Honeypot aus, wenn es einem Angreifer gelingt, sich mit einem „gültigen" Benutzeraccount am System anzumelden. Die Software simuliert in einem solchen Fall eine interaktive Shell, mit der der Angreifer versuchen kann, Schadsoftware nachzuladen. Deren Ausführung innerhalb der virtuellen Umgebung wird jedoch verhindert [16]. Falls einem Angreifer ein solcher Einbruch gelingt, legt kippo umfangreiche Protokolle an, um die Shell-Session zur Auswertung vollständig rekonstruieren zu können. Andernfalls werden lediglich Verbindungsversuche und die versuchten Kombinationen aus Benutzername und Passwort registriert. Eine umfassende Evaluation des Honeypots kann der diesem Projekt vorausgehenden Belegarbeit entnommen werden [16]. Da SSH innerhalb des SVNs häufig zur Administration von unixoiden Serversystemen eingesetzt wird, war der Betrieb eines solchen Honeypot-Dienstes unausweichlich.

Für das Zusammenspiel mit dem HoneySens-Prototypen war es nötig, die Software um ein Modul zu erweitern, das aufgezeichnete Ereignisse an die REST-API schickt und deren Protokoll implementiert. kippo ist erfreulicherweise bereits modular konzeptioniert und liefert eine Reihe von ebenfalls in Python verfassten `dblog`-Modulen mit, die das Protokollieren in verschiedene Datenbanken, Textdateien oder via XMPP erlauben. Im Rahmen des Projektes wurde deshalb ein weiteres Modul geschrieben, das die abstrakte Klasse `dblog.DBLogger` implementiert, alle während einer Verbindung mit dem Honeypot gewonnenen Daten sammelt und nach dem unvermeidlichen Verbindungsabbruch zum HoneySens-Server sendet. Das Modul liegt im Sensor-Dateisystem unter
`/opt/kippo/kippo/dlog/honeysens.py` und nutzt wie auch die zuvor bereits beschriebenen Skripte die externen Python-Bibliotheken *Requests* und *PyCrypto*. Die Implementierung der Logging-Klasse definiert zunächst eine Methode `start(cfg)`, der die aktuelle kippo-Konfiguration übergeben wird. Diese wurde um die URI des HoneySens-Servers, das zu validierende Serverzertifikat und den privaten, kryptographischen Schlüssel des Sensors erweitert. Die Methode liest all diese Parameter und initialisiert ein *Session*-Array, in das alle potentiellen Vorfälle aufgenommen werden. Bei jedem Verbindungsversuch mit dem Honeypot wird dann die Operation
`createSession(peerIP, peerPort, hostIP, hostPort)` aufgerufen, in der ein neuer Vorfall initialisiert und all seine Parameter gespeichert werden. Eine Reihe von Callback-Methoden dienen anschließend der Registrierung von im Rahmen des Angriffs gewonnenen Informationen, beispielsweise `handleLoginSucceeded()` für ein erfolgreiches Login oder `handleCommand()` für einen vom Angreifer an der simulierten Shell abgesetzten Befehl. Wenn die Verbindung durch die Gegenstelle wieder geschlossen wird, sorgt die Funktion `handleConnectionLost()` für das Senden des Vorfalls zum HoneySens-Server und das Aufräumen der angelegten Datenstrukturen.

[69] `https://github.com/desaster/kippo` (abgerufen im November 2014)

dionaea Bei dionaea[70] handelt es sich wie auch bei kippo um einen serverseitigen Honeypot, der allerdings eindeutig als *Low-Interaction* klassifiziert werden kann [16]. Erklärtes Ziel der Software ist es, durch das Simulieren von häufig unter Angriffen leidenden Netzwerkdiensten Malware einzusammeln, die Angreifer auf dem System einzuschleusen versuchen. Hierfür emuliert der Dienst gezielt eine Reihe von bereits bekannten Sicherheitslücken in Protokollen wie dem für die Windows-Dateifreigabe genutzten *Server Message Block* (SMB) oder dem *Session Initiation Protocol* (SIP), das für die VoIP-Kommunikation genutzt wird. Falls ein bekannter Exploitversuch detektiert wird, versucht das stark modularisierte dionaea den vom Eindringling bei derartigen Angriffen typischerweise übertragenen *Shellcode* zu finden und zu interpretieren. Wenn dieser versucht, weitere Malware aus dem Internet nachzuladen, kann diese mit weiteren Modulen heruntergeladen und zur späteren Inspektion gesichert werden. Eine detaillierte Beschreibung und Evaluation der Funktionsweise ist ebenfalls in der bereits erwähnten Belegarbeit *Honeypots und Honey-Netze* [16] enthalten.

Der dionaea-Daemon kann eine Vielzahl von Diensten anbieten, von denen jedoch nur wenige im Rahmen des SVNs sinnvoll eingesetzt werden können. Hinzu kommt, dass der erzielbare Informationsgewinn von Modul zu Modul stark schwankt, weshalb im Rahmen des HoneySens-Prototypen ausschließlich die sehr umfangreiche [14] SMB/CIFS-Implementierung (TCP-Port 445) und der auf Windows-Systemen typischerweise geöffnete *Portmapper RPC Service* (bzw. dessen Emulation) auf TCP-Port 135 aktiviert wurden. Entsprechende Ausnahmeregeln, die verhindern, dass solcher Datenverkehr zusätzlich noch vom *Passive Scan Mode* registriert wird, sind Teil der Firewallregeln.

Die Integration des Honeypots in das Sensornetzwerk erforderte trotz der zusätzlichen Abhängigkeiten und der Tatsache, dass die Software kompiliert werden muss, einen mit dem kippo-Honeypot vergleichbar geringen Aufwand, da all diese Schritte ausführlich von den Entwicklern dokumentiert wurden. Ebenfalls analog zum SSH-Honeypot erlaubt dionaea die Erweiterung um in Python implementierte *ihandler*, die für beliebige Ereignisse im System Callback-Funktionen bereitstellen. Im Umfang des dionaea-Downloads sind bereits eine Reihe solcher Module enthalten, die wiederum das Logging zu verschiedenen Datenbanksystemen oder über XMPP-Nachrichten unterstützen. Zusätzlich können solche Erweiterungen aber auch zur Emulation neuer Protokolle geschrieben werden, die Honeypot-Distribution wird beispielsweise bereits mit Modulen für die Protokolle FTP und TFTP ausgeliefert. Es wurde für das HoneySens-Projekt wiederum ein gesondertes Modul für die Integration der REST-API geschrieben. Dieses implementiert die abstrakte `ihandler`-Klasse und nutzt erneut die Bibliotheken *Requests* und *PyCrypto* zur gesicherten Kommunikation mit der Serverseite. Die `init()`-Methode liest die um HoneySens-spezifische Erweiterungen versehene Konfiguration ein und legt ein `attacks` genanntes Array an, in dem alle Vorfälle gesammelt werden. Callback-Methoden wie `handle_incident_dionaea_connection_tcp_accept()` füllen dieses letztendlich mit den gewonnenen Informationen. Nachdem eine Verbindung geschlossen wurde, werden alle vorhandenen Daten des spezifischen Vorfalls, darunter beispielsweise der Name des vom Angreifer ausgenutzten Exploits, im JSON-Format und mit gültiger, base64-kodierter Signatur an die API gesendet.
Für den reibungslosen Betrieb wurde die dionaea-Konfiguration zuletzt so angepasst, dass ausschließlich kritische Fehlermeldungen in die Protokolldateien geschrieben wurden, was für das manuelle Debugging bei Problemen hilfreich war. Die Standardkonfiguration ist in dieser Hinsicht weniger strikt und erzeugt zu jedem Vorfall mehrere Kilobyte an Logdaten, was beim Betrieb über einen längeren Zeitraum zu Speicherplatz-Engpässen geführt hätte. Die Honeypot-Prozesse besaßen Rahmen der Containment-Maßnahmen reduzierte Rechte und liefen unter separaten Benutzern, um die Risiken im Falle eines Softwarefehlers zu reduzieren.

[70] `http://dionaea.carnivore.it` (abgerufen im Oktober 2014)

9 Problemstellungen

Im letzten Abschnitt zur Implementierung sollen ausgewählte Aspekte genauer beleuchtet werden, die sich aufgrund ihrer Komplexität weder der Server- noch der Sensorseite eindeutig zuordnen lassen. Konkret wurden hierfür die Vorgänge des Firmwareupdates und das an den Prozess angelehnte Einrichten zusätzlicher Sensoren ausgewählt, die während der Implementierung die größten Hürden darstellten.

9.1 Firmwareupdate

Das Betriebssystem der Sensoren und die darauf laufenden Anwendungen sind so konfiguriert, dass nur wenige Protokollnachrichten generiert werden, um ein Überlaufen des Dateisystems zu verhindern. Alle registrierten Ereignisse werden an den Server weitergeleitet und nicht lokal gespeichert, womit ein langanhaltender Betrieb der Sensorsysteme beabsichtigt wird. Trotzdem kann es erforderlich sein, sei es aufgrund von Softwareupdates für die Honeypots oder das zugrundeliegende Debian-System, oder um neue Features oder Fehlerbehebungen in den laufenden Betrieb zu übernehmen, die Firmware eines laufenden Sensors durch eine andere Version zu ersetzen. Um die Prozedur zur Erstellung von Firmware-Abbildern für den Endanwender möglichst zu vereinfachen, wurde im Prototypen auf inkrementelle Updates verzichtet, die zudem im laufenden Betrieb nur bestimmte Komponenten aktualisieren und beispielsweise nicht das Partitionsschema verändern können. Eine neue Firmware-Revision besteht deshalb immer aus einem vollständigen Abbild des Betriebssystems. Dieses Kapitel beschreibt die einzelnen Schritte, welche zur Aktualisierung der Sensorfirmware nötig sind. Es handelt sich um eine Zusammenfassung der Implementierung der in Kapitel 6.4 konzeptionell beschriebenen Einzelschritte.

HoneySens-Firmware wird in Form eines unverschlüsselten `.tar.gz`-Archives verteilt, das immer drei Dateien enthält: `metadata.xml` speichert Informationen wie den Namen und die Revision der Firmware, eine Kurzbeschreibung sowie eine Liste der Änderungen der vergangenen Versionen (Changelog). Weiterhin sind die Dateien `boot.img` und `root.img` enthalten, die die Dateisysteme für die Boot- und die Root-Partition des Sensors beherbergen. Nach Erhalt einer neuen Firmware ist es zunächst Aufgabe des Administrators, diese über den *Firmware*-Bereich des Web-Frontends auf den Server zu laden. Dieser testet das Archiv nach einem erfolgreichen Upload, indem er es entpackt, auf Vollständigkeit überprüft und das XML-Dokument zu validieren versucht. Eine Prüfsumme zur Gewährleistung eines korrekten Uploads ist im Prototypen allerdings nicht implementiert und könnte Teil einer zukünftigen Erweiterung sein. Im Erfolgsfall werden die Metadaten ausgelesen, ein neues `Image`-Objekt zur Datenbank hinzugefügt und das gültige Archiv in das `data/firmware/`-Verzeichnis des Servers verschoben.

Es liegt nun an den Wünschen des Administrators, ob die neue Firmware direkt zum Systemstandard für alle Sensoren erhoben werden soll oder ob zunächst einzelne Geräte über eine individuelle Konfiguration damit ausgestattet werden sollen. In jedem Fall wird ein zu aktualisierender Sensor bei seinem nächsten Polling-Vorgang – nach Ablauf des dafür definierten Intervalls – eine Abweichung zwischen der lokal installierten und der in der Konfiguration geforderten Firmware-Revision feststellen und aus dem `poll.py`-Skript heraus den Updateprozess über das Shellskript `update.sh` in Gang setzen. Der Aktualisierungsvorgang selbst besteht nun aus zwei getrennten Phasen, deren Übergang ein automatischer Neustart des Geräts markiert.

Phase 1: Firmware -> eMMC Das Skript `update.sh` ist für die ersten Phase verantwortlich und lässt zunächst jeweils zwei der vier LEDs des BeagleBone-Boards periodisch aufblinken, um den Aktualisierungsprozess anzuzeigen. Anschließend werden alle Honeypot-Dienste und der *Passive Scan Mode* beendet sowie das Pollingintervall über eine Modifikation des zugehörigen Cron-Jobs auf eine Minute reduziert. Dies stellt sicher, dass der Benutzer sofort über eventuelle Fortschritte im Update informiert wird.

Anschließend wird mit Hilfe des Tools `mktemp` ein temporäres Verzeichnis erstellt und das für den Sensor aktuell gültige Firmwareimage vom HoneySens-Server heruntergeladen. Hierfür wird ein HTTP-GET-Request an die URI

```
api/sensorimages/download/<id>
```

gestellt, wobei die ID durch Interpretation der vom Server zuvor empfangenen Konfigurationsdaten gewonnen wurde. Beim Download des Archivs über die verschlüsselte TLS-Verbindung wird das Serverzertifikat überprüft, welches auf dem Sensor bei dessen Installation hinterlegt wurde. Nach erfolgreichem Download wird das Archiv entpackt. Eine gesonderte Validierung der Datei ist im Prototypen nicht vorgesehen, allerdings als potentielle spätere Erweiterung der Software denkbar.

Nach dem Entpacken des Archivs stehen im temporären Verzeichnis die Boot- und Root-Partitionen im RAW-Format zur Verfügung. An dieser Stelle wird eine Eigenheit des BeagleBones ausgenutzt und macht die prototypische Implementierung von dieser Hardwareplattform abhängig. Das Sensorsystem bootet normalerweise direkt von einer eingesteckten Speicherkarte, auf dem Board ist jedoch auch ein integrierter, *eMMC* genannter, 2 GB großer Flash-Speicher vorhanden, auf dem ebenfalls ein Betriebssystem installiert werden kann. Welches letztendlich beim Booten gestartet wird, ist über den Bootmanager vor einem Neustart einstellbar. Diesen Umstand nutzt der Updateprozess, indem zunächst die beiden Images für die Boot- und Root-Partitionen mit `dd` auf den internen Speicher geschrieben werden. Daraus resultiert auch die maximale Größe einer Firmware von 2 GB. Das laufende Sensorsystem bleibt von diesem Vorgang völlig unbeeinflusst und kann im Fehlerfall zudem noch einmal gestartet werden, um ein erneutes Update anzustoßen.

Nachdem das Flashen des internen Speichers abgeschlossen ist, müssen die *Universally Unique Identifiers* (UUIDs) der Partitionen modifiziert werden, um zu verhindern, dass unter ungünstigen Umständen mehrere Partitionen eine solche identische Zeichenfolge besitzen und nicht mehr eindeutig unterschieden werden können. Eine völlig neue, zufällige ID kann mit Hilfe des Tools `uuidgen` generiert werden, das als Paket im Debian-Repository enthalten und auf den Sensoren vorinstalliert ist. Die Root-Partition der BeagleBones nutzt das Dateisystem *ext4*, welches eine solche Änderung problemlos mit Hilfe der Anwendung `tune2fs` erlaubt. Im Gegensatz dazu wurde trotz Recherche keine mitgelieferte Software gefunden, die diesen Vorgang auch für die FAT32-Boot-Partition durchführen kann. Es war deshalb nötig, auf das Perl-Skript `fat-set-uuid.pl`[71] zurückzugreifen. Weiterhin war der mit `uuidgen` generierte String zu lang und musste mit `awk` auf acht Stellen gekürzt werden.

Der nächste Schritt im Updateprozess ist das Übernehmen der aktuellen Sensorkonfiguration in das temporäre Update-System, das gerade auf den internen Speicher geschrieben wurde. Hierfür sammelt das Update-Skript alle relevanten Konfigurationsdateien im temporären Verzeichnis und generiert ein Konfigurationsarchiv, das mit dem bei der Installation eines neuen Sensors vom HoneySens-Server generierten

[71] `https://github.com/nickandrew/fat-set-uuid` (abgerufen im Oktober 2014)

kompatibel ist. Anschließend wird der interne Speicher ins Dateisystemen eingehängt und die Konfiguration auf die Boot-Partition kopiert. Durch das Anlegen der Datei `flash-eMMC.txt` auf der Boot-Partition des Update-Systems wird sichergestellt, dass nach einem Neustart ein erneuter Flash-Vorgang gestartet wird, der wiederum die SD-Karte überschreibt. Vor einem die erste Phase abschließenden Neustart wird noch der Bootloader aktualisiert, damit das System später vom neuen System auf dem internen Speicher startet.

Phase 2: eMMC -> Speicherkarte Nachdem das System neu gestartet wurde, bootet es vom internen Speicher, der nun ein ausschließlich für das Update vorbereitetes System enthält. Während des Bootens veranlasst die Existenz des Konfigurationsarchivs auf der Boot-Partition den Start des Shell-Skriptes `config_sensor.sh`, das die Konfigurationsdateien in ein temporäres Verzeichnis entpackt und anschließend im laufenden System verteilt. Dies betrifft die Datei `/etc/network/interfaces` mit den Netzwerkeinstellungen, den lokalen Hostnamen, den privaten Schlüssel sowie das öffentliche Serverzertifikat (zur Validierung). Weiterhin ist die Datei `honeysens.cfg` enthalten, aus der die URI des Servers ausgelesen und in den Konfigurationsdateien der Honeypots kippo und dionaea eingetragen wird. Zuletzt erfolgt eine Anpassung der `htpdate`-Konfiguration, die für die Zeitsynchronisation unerlässlich ist.

Ebenfalls während des Bootens löst die Existenz der in der ersten Phase angelegten Datei `/boot/uboot/flash-eMMC.txt` einen erneuten vollständigen Flash-Vorgang der noch immer eingesteckten Speicherkarte aus, auf der noch das alte Sensorsystem liegt. Da dessen Daten bereits auf das temporäre Zwischenbetriebssystem migriert wurden, kann das Überschreiben der Altlasten mit Hilfe des Skriptes `beaglebone-black-flasher.sh` gestartet werden. Es wurde für den Prototypen modifiziert, um wieder den laufenden Flash-Vorgang mit Hilfe der LEDs auf dem Board zu visualisieren. Die Anwendung selbst stammt vom BeagleBone-Projekt und klont das gerade laufende System mit der aktualisierten Firmware-Revision auf das externe Speichermedium. Es stellt dabei ebenfalls sicher, dass die UUIDs angepasst und der externe Speicher zuvor frisch formatiert wird, um den vollen Platz des Speichermediums im „neuen“ Sensorsystem nutzen zu können. Einige weitere Modifikationen an dem Skript bewirken, dass zuletzt erneut die Konfiguration des Bootloaders so geändert wird, dass der nächste – abschließende – Neustart wieder von der frisch beschriebenen Speicherkarte erfolgt. Nach dem Abschluss all dieser Tätigkeiten startet das System ein letztes Mal neu und ist letztendlich – falls keine Fehler auftraten – mit der neuen Firmware-Revision online.

Falls in einer der Phasen Probleme auftreten, was insbesondere im Zusammenhang mit defekter Hardware geschehen kann, empfiehlt sich in der Regel die Neuinstallation des betreffenden Sensors. Insbesondere defekte Speicherkarten können unvorhersehbare Systemzustände auslösen, die nur durch einen Austausch der Komponente zu beheben sind. Ein Verlust der Verbindung zum Server während des Aktualisierungsvorganges stellt ausschließlich während des Firmware-Downloads ein Problem dar, die übrigen Schritte verlaufen autonom. In diesem Fall würde der Server lediglich keine Rückmeldung über den Updateprozess erhalten und im Web-Frontend eine Zeitüberschreitung anzeigen. Das Update-Skript berücksichtigt diesen Fall und bricht den Vorgang ab. Sobald die Verbindung wiederhergestellt ist, beginnt dann ein erneuter Updateversuch.

9.2 Sensorinstallation

Das Hinzufügen eines neuen Gerätes zum Sensornetzwerk verläuft in einigen Schritten identisch zum zuvor beschriebenen Aktualisierungsvorgang, unterscheidet sich jedoch erheblich in der Vorbereitung. Bevor ein neuer Sensor überhaupt hinzugefügt werden kann, muss eine systemweite Standardfirmware mit einem zugehörigen bootbaren Speicherkartenabbild auf dem Server vorliegen. Falls das noch nicht der Fall ist, kann über das Web-Frontend für eine bereits hochgeladene Firmware der Konvertierungsvorgang gestartet werden, der einen entsprechenden Job an den Beanstalkd-Server schickt. Nachdem die Konvertierung abgeschlossen ist, liegt im Verzeichnis `data/firmware/sd/` ein mit externen Speicherkarten kompatibles Abbild der Firmware.

Nachdem die Voraussetzungen erfüllt sind, kann in der Sensorliste der prototypischen Webanwendung mit dem *Hinzufügen*-Button ein Dialog geöffnet werden, in dem alle den neuen Sensor betreffenden Einstellungen zu tätigen sind. Hierzu zählen der systemweit einzigartige Sensorname, der zugleich als Hostname dient sowie eine informelle Beschreibung der Sensorstandorts. Diese ist ausschließlich zur leichteren Lokalisierung für die Administratoren gedacht und hat keine Auswirkungen auf die Funktionalität. Weiterhin ist die Netzwerkkonfiguration vorzunehmen: das primäre Ethernet-Interface kann entweder mit einer statischen IP-Adresse versehen oder später dynamisch über das DHCP-Protokoll konfiguriert werden. Weiterhin kann ein *Endpunkt* für die API des HoneySens-Servers angegeben werden. Diese Einstellung kann in der Regel auf dem Standardwert belassen werden und bestimmt, wie der Sensor später den Server erreichen kann. In Falle von Servern mit mehreren Netzwerkschnittstellen kann es erforderlich sein, den Wert für einzelne Sensoren anzupassen. Zuletzt werden noch ein privater Schlüssel und ein gültiges, von der serverseitigen *Certificate Authority* signiertes Zertifikat angefordert. Nach dem Speichern all dieser Einstellungen generiert der Server aus den im Verzeichnis `data/configs/template/` liegenden Schablonen eine gültige Sensorkonfiguration, die vom im vorherigen Kapitel beschriebenen Skript `config_sensor.sh` verarbeitet werden kann. Anschließend werden dem Benutzer das bootbare Image für eine Speicherkarte und das Konfigurationsarchiv zum Download angeboten.

Es obliegt an dieser Stelle dem Anwender, das vom Server bezogene Abbild auf ein Speichermedium zu flashen, wofür beispielsweise unter unixoiden Betriebssystemen das Tool `dd` verwendet werden kann. Dieses legt auf dem Medium die üblichen zwei Partitionen an: eine kleine, mit dem Dateisystem *FAT32* formatierte Boot-Partition und eine Root-Partition, die *ext4* nutzt. Zumindest die Bootpartition kann von allen gängigen Betriebssystemen gelesen und auch beschrieben werden, was in der Folge nötig ist, um die Sensorkonfiguration in das Image einzubinden. Es genügt, das heruntergeladene Archiv auf diese Partition zu kopieren. Anschließend kann die Speicherkarte entfernt und in das *BeagleBone Black* eingesetzt werden.

Das Board lädt immer zuerst den Bootloader der auf der Speicherkarte befindlichen Boot-Partition, wenn eine solche eingesteckt ist. Somit wird nach der Aktivierung der Stromzufuhr umgehend das darauf befindliche System gestartet. Da das vom Server bezogene Image auf die minimale Größe reduziert wurde (um den Download klein zu halten), sind nach dem Bootvorgang nur noch wenige Megabyte an Speicher im Dateisystem frei. Damit die volle Kapazität der Speicherkarte genutzt werden kann, wird wieder ein aus zwei Phasen bestehender Installationsprozess gestartet, der stark dem zuvor beschriebenen Aktualisierungsvorgang ähnelt. Unterschiede bestehen in der ersten Phase: Beim Start des Sensors wird zunächst wieder die in der Boot-Partition abgelegte Konfiguration vom Skript `config_sensor.sh` verarbeitet. Anschließend wird direkt ein Flash-Vorgang mit dem an diesem Punkt bereits vertrauten Programm `beaglebone-black_flasher.sh` gestartet, der das laufende System auf den internen Speicher

klont. Nach dem abschließenden Neustart und Bootvorgang vom geklonten System wird erneut die bereits beschriebene Phase 2 wiederholt, bei der die Speicherkarte neu formatiert und das System ein letztes Mal kopiert wird. Nach dem abschließenden Neustart ist der Sensor letztendlich einsatzbereit.

Der gesamte Installationsprozess beruht auf der Idee, das Einrichten einer großen Zahl neuer Sensoren für den Administrator möglichst einfach zu gestalten. Durch die Trennung des generischen Sensor-Abbildes von der individuellen Sensor-Konfiguration genügt es, eine beliebige Anzahl von Speicherkarten mit dem bootbaren Image vorzubereiten und anschließend die nur wenige Kilobyte großen Konfigurationsarchive nachzureichen. Eine alternativ in Betracht gezogene Implementierung sollte bereits auf den neuen Sensor zugeschnittene SD-Abbilder generieren, bei denen die individuellen Daten bereits enthalten sind. Dieser Ansatz wurde aber verworfen, nachdem der deutlich höhere administrative Aufwand dieser Methode bei der Initialisierung von vielen Sensoren zugleich deutlich wurde.

Teil VI

Testbetrieb

Der letzte Schritt nach der Konzeption und Implementierung des HoneySens-Systems war die Verifikation von dessen Funktionalität durch die Ausübung praktischer Versuche. Es sollte damit demonstriert werden, dass es die in der Anforderungsanalyse in Teil III dieser Arbeit geforderten Punkte weitestgehend erfüllt, welche auch bereits während des Kapitels zur Implementierung vereinzelt angesprochen wurden.

Zunächst erfolgte eine permanente Überprüfung aller neu implementierten Funktionen während der Entwicklung der Software selbst. Dabei handelt es sich um einen inkrementellen Vorgang, der parallel zur Implementierung erfolgt und auch im Rahmen dieses Projektes konsequent ausgeführt wurde. Derartige kurze Funktionstests können allerdings keinesfalls alle möglichen Szenarien und Eingabewerte nachstellen und ersetzen somit nicht einen ausführlichen Praxiseinsatz. Aus diesem Grund erfolgte der Testbetrieb nach Fertigstellung des Prototypen in zwei Stufen: Zunächst wurden die Grundfunktionen gesondert im Laborbetrieb innerhalb eines abgeschotteten Netzsegments ohne gefährdete Produktivsysteme getestet und gefundene Fehler oder aufgetretene Probleme behoben, um im Anschluss zwei größere Testsysteme innerhalb von produktiv genutzten Subnetzen platzieren zu können. Die folgenden Abschnitte beschreiben Details und Ergebnisse dieser Experimente.

10 Labortests

Zentrale Komponente der Testumgebung für die kontrollierten Testreihen war ein x86-basierter Server mit *Mageia Linux*[72], auf dem die während der Implementierung bereits beschriebenen Komponenten der Entwicklungsumgebung (Apache HTTP Server, MySQL, Beanstalkd) installiert wurden. Auf das Betreiben eines Docker-Containers (siehe Kapitel 7.3) wurde für die Funktionsanalyse verzichtet, da ein vollständiges „physisches“ System leichter inspiziert werden kann, einen normalen Init-Mechanismus besitzt und somit die Installation von zusätzlichen Analysetools wie beispielsweise *Wireshark*[73] zur Betrachtung des ein- und ausgehenden Datenverkehrs ohne Komplikationen möglich ist. Weiterhin wurde in das System bereits zu Beginn der Tests ein funktionsfähiger Sensor aus der Entwicklungsumgebung mit der Firmware-Revision 1.000j eingehangen, die zum Zeitpunkt der Versuche aktuellste Variante. Die letzte Komponente innerhalb des Netzsegments war ein Desktop-Rechner, von dem aus sowohl die Interaktion mit dem Web-Frontend als auch die simulierten Angriffe auf die Sensoren ausgeführt wurden.

Die Testreihe umfasste folgende Versuche:

1. Upload eines zusätzlichen Firmware-Archives, das identisch zur Version 1.000j ist, aber aus Testgründen eine abweichende Revisionsbezeichnung besitzt.
2. Konvertierung dieser Firmware zu einem bootbaren, zum Schreiben auf eine Speicherkarte geeigneten Abbild.
3. Hinzufügen eines neuen Sensors im Frontend auf Basis dieser Firmware; Netzwerkkonfiguration via DHCP oder statisch.
4. Download der nötigen Daten und Schreiben des Update-Systems und des spezifischen Sensor-Konfigurationsarchivs auf eine microSD-Karte.
5. Start des automatischen Sensor-Installationsvorgangs; Überprüfung von dessen Rückmeldungen am Server.
6. Setzen der SMTP-Daten und Einrichten eines Kontaktes zum E-Mail-Versand.
7. Simulierte Angriffe auf den neuen Sensor:
   - Kontakt mit dem Sensor auf zwei beliebigen, aber nicht von Honeypots belegten Ports für jeweils TCP und UDP,
   - Durchführung eines TCP-Scans von mindestens vier Portnummern, um die Portscan-Erkennungsroutine zu testen,
   - Login-Versuch via SSH (testet den kippo-Honeypot),
   - Exploit-Versuch über SMB (TCP 445), um die Funktionalität von dionaea sicherzustellen.
8. Anlegen einer spezifischen Konfiguration für den Testsensor, bei der die Honeypots deaktiviert werden.
9. Testen der neuen Konfiguration nach dem nächsten Polling-Vorgang durch Wiederholung der simulierten Angriffe.
10. Überprüfung, ob bei den Angriffen auf die Honeypots alle E-Mails versendet wurden.
11. Sensor-Update auf die Version 1.000j durch Modifikation der individuellen Konfiguration.
12. Entfernen des Sensors mit Hilfe des Web-Frontends.

Diese Tests wurden alle mehrmals wiederholt, speziell nach größeren Änderungen an der Anwendung. Für die Verbindungsversuche mit den geschlossenen Ports wurde die Software *GNU netcat* genutzt, die

[72] `http://mageia.org` (abgerufen im November 2014)
[73] `http://wireshark.org` (abgerufen im November 2014)

in den Repositories aller gängigen Linux-Distributionen enthalten ist. Alle damit gestarteten Anfragen wurden bei den Sensoren und anschließend dem Server registriert. Beim Portscan half hingegen das Tool *nmap*[74], das eine Vielzahl von Techniken zum Scannen von Hosts anbietet und ebenfalls in den Repositories großer Distributionen enthalten ist. Jeder Portscan wurde auch als solcher erkannt, weswegen angenommen werden kann, dass die verwendeten Parameter der Erkennungsroutine zur rudimentären Detektion geeignet sind. Es kam jedoch vor, dass ein einzelner Scan in Form von zwei getrennten Portscan-Ereignissen im Web-Frontend registriert wurde. Dieses Verhalten war nicht reproduzierbar und konnte deshalb bis zum Abschluss der Arbeit nicht eindeutig erklärt werden, lässt sich jedoch höchstwahrscheinlich mit einer weiteren Optimierung der Parameter verhindern. Für die Tests der SMB-Honeypot-Funktionalitäten wurde schließlich auf das „Penetration Testing Framework" *metasploit*[75] gesetzt, das eine Vielzahl von Modulen enthält, die bereits bekannte Sicherheitslücken in einer Vielzahl von Softwareprodukten attackieren können. Im Test wurde konkret das Modul `exploit/windows/smb/sm04-11-lsass` eingesetzt, das einen Fehler im *Local Security Subsystem Service* (LSASS) in bestimmten Versionen des Windows-Betriebssystems auszunutzen versucht, über den sich der Wurm *„Sasser"* im Jahr 2004 verbreitete [29]. Der Honeypot *dionaea* registrierte den Exploit-Versuch beim HoneySens-Server, konnte zum Angriff selbst allerdings keine zusätzlichen Informationen liefern. Die Login-Versuche via SSH wurden hingegen vollständig mit allen die Session betreffenden Daten, darunter auch die benutzten Authentifizierungsdaten, aufgezeichnet.

Die Tests deckten weiterhin Ungereimtheiten bei der Problembehandlung der Serverseite auf. Durch verschiedene Umstände konnte es vorkommen, dass Fehler während der Behandlung einer Anfrage an die API nicht korrekt abgefangen wurden und dazu führten, dass das Web-Frontend auf eine AJAX-Anfrage hin lediglich einen HTTP-Fehlercode 500 („*Internal Server Error*") zurückerhielt. Der Benutzer konnte in solchen Fällen die aktuell durchgeführte Operation trotz ausbleibender Fehlermeldung nicht beenden. Hauptursache dieser Probleme waren unvorhersehbare Systemzustände, beispielsweise hervorgerufen durch mangelnden Speicherplatz oder defekte Hardware. Es wurde versucht, dem Problem sowohl auf Seite des Servers als auch des Clients zu begegnen. Dies wird speziell beim Firmware-Upload deutlich, dessen Dialogfeld im Fall einer ungültigen, hochgeladenen Datei aussagekräftige Fehlermeldungen präsentiert. Es wurde versucht, dieses Verhalten innerhalb der Anwendung konsequent umzusetzen, indem jede JSON-Antwort des Servers das optionale Feld `success` enthält, das im Problemfall `false` ist und mit einem weiteren Attribut `error` eine Fehlermeldung in Textform zurückgibt, die vom Client angezeigt werden kann.

Während des gesamten Testprozesses wurde eine Reihe weiterer kleiner Fehler aufgedeckt, die hier nicht im Detail aufgelistet werden sollen. Nachdem die Prozedur letztendlich fehlerfrei durchlaufen wurde, erfolgte das Packen der Serversoftware in Form eines Docker-Images und die Installation zweier Testumgebungen, die im Folgenden beschrieben werden.

[74] `http://nmap.org` (abgerufen im November 2014)
[75] `http://metasploit.com` (abgerufen im November 2014)

11 Produktive Testumgebungen

Ausführliche Tests der Sensor-Infrastruktur innerhalb von Produktivnetzwerken mit „kritischen“ Komponenten wurden innerhalb eines Teilnetzes der *Fakultät Informatik* und in einem zu den *Sächsischen Informatikdiensten* (SID) gehörigen Büronetzwerk, das überwiegend aus Windows-Clients bestand, durchgeführt. Ziel der Versuche war es, abgesehen von der reinen Funktionalität des HoneySens-Systems auch Erfahrungen bezüglich der Leichtigkeit der Integration (siehe Kapitel 4), der Transparenz, und der Benutzerfreundlichkeit zu sammeln. Beide Architekturen bestanden aus jeweils einem Server und einer beliebigen Anzahl von Sensoren – die Administration wurde von Verantwortlichen vor Ort erledigt – und wurde für mehrere Wochen betrieben, um anschließend sowohl die gesammelten Ereignisse, als auch die in kurzen Gesprächen gewonnenen Rückmeldungen bezüglich der Benutzbarkeit auszuwerten.

In beiden Institutionen wurde das in Kapitel 7.3 beschriebene Deployment-Verfahren mit Docker-Containern genutzt. Da an den Standorten allerdings serverseitig das noch junge Docker-Projekt nicht existierte, war die Installation eines zusätzlichen Basissystems, auf dem die Container schließlich betrieben werden konnten, notwendig. In beiden Fällen wurde zu diesem Zweck die Linux-Distribution *CoreOS*[76] ausgewählt, die nur geringe Systemanforderungen stellt und speziell auf den Betrieb von Anwendungen innerhalb von Docker-Containern ausgelegt wurde. Als Sensoren kamen erneut die bereits für die Implementierung genutzten Boards vom Typ *BeagleBone Black* mit zusätzlichen microSD-Karten zum Einsatz, die je nach Verfügbarkeit über externe Netzteile oder USB mit Strom versorgt wurden.

Setup 1: Fakultät Informatik Für die Installation des HoneySens-Servers an diesem Standort wurde eine neue virtuelle Maschine auf einem bestehenden *VMware vSphere Server*[77] auf Basis des CoreOS-Betriebssystems erzeugt und mit einer eindeutigen IP-Adresse im Netzwerk versehen. Anschließend genügte es, das bereits vorbereitete Docker-Image zu importieren und nach der in Kapitel 7.3 beschriebenen Vorgehensweise in Betrieb zu nehmen. Es wurden die TCP-Ports 80 (HTTP) und 443 (HTTPS) an die virtuelle Instanz weitergeleitet, womit die API und das Webinterface aus dem Netzwerk heraus erreichbar waren. Es wurde anschließend die zu diesem Zeitpunkt aktuelle Firmware auf den Server geladen und ein neuer Sensor hinzugefügt, der via DHCP seine IP-Adresse bezog. Der war schließlich zwei Wochen lang mit allen im Prototypen vorhandenen Diensten (Passive Scan Mode, kippo und dionaea) aktiv.

Setup 2: SID-Teilnetz Innerhalb des zum SID gehörigen Verwaltungsnetzes wurde für die CoreOS-Installation ein separates, auf der Hardwareplattform *Intel Atom D510* basierendes System eingerichtet. Auf die Verwendung von Virtualisierungstechniken wurde in diesem Fall also verzichtet. Der Rechner war hinsichtlich der Leistungsdaten eigentlich als schlankes Desktop-System konzipiert und zum Testzeitpunkt bereits veraltet und daher keinesfalls mit einem professionellen Serversystem vergleichbar. Es war an dieser Stelle interessant herauszufinden, ob auf solch schwacher Hardware Performanceprobleme auftreten würden. Nach der Installation des Betriebssystems wurde analog zum virtualisierten System im ersten Setup verfahren, diesmal jedoch insgesamt eine Summe von vier Sensoren mit dynamisch bezogenen IP-Adressen (DHCP) eingerichtet. Diese waren anschließend für einen Zeitraum von vier Wochen in Betrieb und wurden währenddessen nur sporadisch über das Web-Frontend kontrolliert.

[76] `http://coreos.com` (abgerufen im November 2014)
[77] `http://vmware.com/products/vsphere` (abgerufen im November 2014)

In beiden Fällen erfolgte nach der Installation des Gesamtsystems eine Einweisung der zuständigen Verantwortlichen in die Bedienung und Funktionalitäten des Web-Frontends. Zuletzt wurden zur Kontrolle noch einige Testangriffe, im Wesentlichen HTTP- und SSH-Anfragen, an die Sensoren gesendet und deren ordnungsgemäßes Erscheinen in der Ereignisliste der Webanwendung überprüft.

12 Auswertung

Nach Abschluss der Testzeiträume wurde mit Hilfe des Web-Frontends zunächst überprüft, ob alle Sensoren noch ordnungsgemäß „online“ waren und in ihrem konfigurierten Polling-Intervall Statusnachrichten sendeten, was auch der Fall war. Die grundsätzliche Funktionalität der HoneySens-Architektur war zudem durch die Ergebnisse der Laborversuche aus Kapitel 10 und die Testumgebungen sichergestellt, in denen die elementaren Operationen zum Sensor-Management und zur Angriffserkennung noch einmal gesondert getestet wurden. Es sind jedoch innerhalb der kurzen Testzeiträume nur wenige potentielle Angriffe registriert wurden, die nicht zu den beabsichtigten Funktionstests gehörten. Innerhalb des SID-Teilnetzes wurden hierbei innerhalb der vier Wochen gar keine zusätzlichen Ereignisse registriert, die Ereignisliste bestand folglich lediglich aus den neun nach der Installation gezielt durchgeführten Verbindungsversuchen. Es muss an dieser Stelle jedoch auch darauf hingewiesen werden, dass der Einsatz von Honeypots und Sensoren im Allgemeinen eine langfristige Investition und Maßnahme zur Prävention ist, weswegen dieses Resultat nicht den Betrieb einer derartigen Architektur in Frage stellen sollte.

Die Installation in der Informatik-Fakultät registrierte innerhalb der drei Wochen hingegen 523 Einträge (die Funktionstests sind hier nicht enthalten), von denen 503 vom DHCP-Server des Netzwerks aus als UDP-Pakete an Port 68 des Sensors gesendet wurden. Dass keine Paketinhalte aufgezeichnet wurden, erschwerte die exakte Zuordnung dieser Ereignisse, es handelte sich jedoch sehr wahrscheinlich entweder um mit dem DHCP-Protokoll assoziierbare Nachrichten (beispielsweise *IP Renewal Requests*) oder um Pakete in Zusammenhang mit dem *Bootstrap Protocol* (BOOTP). Beide Dienste werden innerhalb des Fakultäts-Teilnetzes genutzt und operieren auf den UDP-Ports 67 und 68. Somit konnten die betreffenden 503 Vorfälle als harmlos klassifiziert werden, was sich jedoch innerhalb des Web-Frontends als umständlich herausstellte. Zum einen ist die manuelle Änderung des Bearbeitungszustandes von mehreren Ereignissen gleichzeitig im Prototypen nicht vorgesehen, weiterhin ist aber ebenfalls zu erwarten, dass der Sensor auch in Zukunft diese Nachrichten erhalten und somit neue belanglose Ereignisse beim Server registrieren wird. Aus dieser Beobachtung heraus entstand schließlich der Wunsch nach einem *Filter-Verfahren*, das es beispielsweise erlaubt, aus bestehenden Paketen Regeln zu erstellen, die zukünftige Ereignisse mit gewissen Eigenschaften automatisch ignorieren oder aber auch den Sensor anweisen, diese gar nicht erst zum Server zu senden. Der anfallende Datenverkehr würde dann geringer sein. Eine Zwischenlösung im bestehenden System wäre die Aktualisierung der Firmware, sodass direkt an den Sensor gerichtete UDP-Pakete auf Port 68 ignoriert werden, da diese für die DHCP-Clientsoftware gedacht und in der Regel erwünscht sind.

Die verbleibenden 20 Vorfälle wurden ebenfalls durch UDP-Pakete ausgelöst, die aber alle an unterschiedliche Ports im Bereich zwischen 30.000 und 60.000 gerichtet waren. Sie gingen von einem als DNS-Server genutzten Host aus, ihr Gefahrenpotential konnte jedoch letztendlich nicht eindeutig bestimmt werden. Dies führte zu der Feststellung, dass zusätzliche Informationen wie der Paketinhalt auch aufgezeichnet werden könnten, um die Bewertung der Ereignisse zu erleichtern. Eine solche Änderung wäre durch Modifikation des *Passive Scan Mode* und des serverseitigen Domänenmodells möglich, wür-

de aber auch zu einem erhöhten Datenaufkommen im Sensornetzwerk führen. Das Komprimieren oder Beschneiden der Paketinhalte wären Möglichkeiten, diesem Umstand zu begegnen.

Die registrierten Vorfälle zeigen, dass die Funktionalität der Erkennungsroutinen im Prototypen grundsätzlich gegeben ist. Hinsichtlich der Skalierbarkeit kann festgehalten werden, dass der nicht sehr leistungsfähige Server im SID-Teilnetz zumindest mit den eingesetzten vier Sensoren keinerlei Probleme verursachte und das Webinterface trotz der alten Hardware ein angenehmes Nutzungsverhalten für die zuständigen Administratoren bereitstellte. Die Anforderungen an die Benutzungsfreundlichkeit und Wartbarkeit können zumindest im Rahmen der kurzen Testzeiträume als erfüllt betrachtet werden, da abgesehen vom eben erläuterten umständlichen Umgang mit „False Positives" keine Beschwerden auftraten und alle Sensoren am Ende des Testzeitraums noch einsatzbereit waren. Lediglich beim Deployment des Server-Systems selbst entstanden im Setup 1 zunächst Probleme, da der Docker-Container nicht mit dem korrekten Hostnamen `honeysens-server` erzeugt wurde. Dies ist allerdings zwingend erforderlich, da beim ersten Start der Instanz das TLS-Zertifikat den Hostnamen des Systems als *Common Name* des Zertifikats verwendet und von den Sensoren später entsprechend verifiziert wird. Diese Überprüfung schlug aber fehl, wenn die explizite Angabe dieses Hostnamens ausgelassen wurde. Die Verwendung von dynamischen Hostnamen ist durch eine Modifikation des Deployment-Prozesses denkbar, erfordert allerdings auch eine Aktualisierung der Firmware-Skripte zur Integration einer neuen Sensorkonfiguration. Die genannten Verbesserungsvorschläge werden im die Arbeit abschließenden *Ausblick* noch einmal zusammengefasst.

Teil VII

Schlussbetrachtung

Im Rahmen dieser Arbeit wurde ein Honeypot-Netz entworfen und in Form eines Prototypen implementiert, das den spezifischen Anforderungen und Gegebenheiten des komplexen Sächsischen Verwaltungsnetzes Rechnung trägt. Zu diesem Zweck wurden zunächst die für den Betrieb von Honeypots notwendigen allgemeinen Grundlagen erörtert und anschließend die Architektur des Zielnetzes beschrieben. Teil III der Arbeit analysierte dann die technischen und organisatorischen Anforderungen, die an die zu entwerfende Architektur gestellt wurden. Die nachfolgenden Kapitel beschreiben ausführlich die Konzeption und Implementierung eines Prototypen, der schließlich im Testbetrieb sowohl unter kontrollierten Laborbedingungen als auch innerhalb eines Produktivnetzwerkes getestet wurde.

Der Schwerpunkt der Arbeit lag auf dem Versuch, Angreifer *aus dem Inneren* aufzuspüren. Das bedeutet, dass die zu betreibenden Honeypots bewusst nicht aus dem Internet, sondern ausschließlich aus dem eigenen oder aus benachbarten internen Teilnetzen erreichbar waren. Weiterhin sollte sich die zu projektierende Infrastruktur insbesondere durch ihre Langlebigkeit, einfache Wartbarkeit und leichte Installation hervorheben. Da zudem nicht sichergestellt werden konnte, dass die für die Administration zuständigen Verantwortlichen Erfahrung im Bereich IT-Security besitzen würden, war die Benutzungsfreundlichkeit der resultierenden Anwendung von großer Bedeutung. Größte Herausforderung war jedoch die Beschaffenheit des SVNs, das aus einer Vielzahl von durch restriktiven Firewalls und IDS abgeschirmten Teilnetzen bestand und die Integration von Honeypots entsprechend erschwerte.

Grundlage der Konzeption war eine zuvor durchgeführte Trafficanalyse im SVN, in der passive Sensorsysteme innerhalb verschiedener Teilnetze platziert wurden und jeglichen direkt an sie adressierten Datenverkehr aufzeichneten. Es stellte sich innerhalb des mehrwöchigen Analysezeitraums heraus, dass das Datenaufkommen an einem solchen System gering war und jedes auf diese Art empfangene Paket prinzipiell als verdächtig angesehen werden konnte. Diese Beobachtung stand in einem starken Kontrast zu direkt über das Internet erreichbaren Honeypots, die tendentiell mehr Traffic empfangen und von Hosts aus der ganzen Welt angegriffen werden können [16]. Die in Kapitel 6.1 durchgeführte Untersuchung von verwandten Arbeiten, die sich ebenfalls mit dem Entwurf komplexer hybrider Honey-Netze beschäftigten, resultierte schließlich in einem *Sensornetzwerk* mit zentraler, als Manager fungierender Serverkomponente. Der anschließend entwickelte und *HoneySens* getaufte Prototyp bestand im Kern aus einer REST-API, die sowohl für die in den verschiedenen Teilnetzen platzierten Sensoren als auch für die für den Administrator entworfene Webanwendung als Backend diente. Um die reibungslose Informationsübermittlung zwischen allen Komponenten durch die restriktiven Netzübergänge hindurch sicherzustellen, wurden HTTP und HTTPS als einzige Kommunikationsprotokolle eingesetzt. Die Sensoren kombinierten einen rein passiven Modus, der jeglichen ankommenden Datenverkehr aufzeichnete und Portscans erkennen konnte, mit den etablierten Low-Interaction-Honeypots *kippo* und *dionaea*, die um eigens entwickelte Module zur Kommunikation mit dem HoneySens-Server erweitert wurden.

Der Prototyp wurde sowohl mehrfach unter kontrollierten Laborbedingungen als auch mit zwei kurzen Einsätzen in produktiven Netzwerken getestet. Bei deren Auswertung stellte sich heraus, dass das entworfene Honey-Netz den Anforderungen prinzipiell gerecht wurde, die prototypische Implementierung allerdings auch noch einige Defizite bei der Installation und im Umgang mit vielen gleichartigen Vorfällen aufwies. Der prinzipielle Einsatz von Honeypots in solch komplexen IT-Infrastrukturen wie dem SVN erscheint als langfristige Investition zum Auffinden von sich selbstständig ausbreitender Malware und gezielt vorgehenden internen Angreifern sinnvoll. Das entworfene System benutzt lediglich Low-Interaction-Honeypots und passive Systeme, die sich durch ein geringes Risiko für das restliche Netzwerk auszeichnen und somit nur wenige administrative Ressourcen für sich beanspruchen. Die Kriterien der Transparenz sowie der einfachen Installation und Wartbarkeit haben sich in diesem Zusammenhang als wichtigste Faktoren für ein derartiges Honey-Netz herausgestellt.

Ausblick Während der Implementierung und des anschließenden Testbetriebs kamen immer wieder neue Ideen und Erweiterungsmöglichkeiten für den Prototypen auf, die im zeitlichen Rahmen dieser Arbeit nicht realisierbar waren, aber potentielle Möglichkeiten zur Erweiterung des Sensornetzwerks darstellen. Die nachfolgende Aufzählung nennt deshalb abschließend all jene Vorschläge, denen während der Auswertung der Ergebnisse dieser Arbeit die höchste Priorität zugewiesen wurde und deren Implementierung die Funktionalität und Bedienbarkeit des Systems weiter verbessern würde.

- Die **manuelle Klassifikation** von registrierten Ereignissen durch Benutzer würde es ermöglichen, harmlose Vorfälle in eine *Whitelist* aufzunehmen und zukünftig in der Webanwendung nicht mehr erscheinen zu lassen. Dies wäre über frei definierbare Filterregeln auf Basis der Ereignisattribute (Quelle, Protokoll, Port usw.) realisierbar, die entweder direkt auf dem Sensor oder nach Registrierung der Daten auf dem Server angewendet werden könnten.
- In der Webanwendung des Prototypen ist es möglich, die Ereignisliste nach beliebigen Kriterien in Form der Spalten zu sortieren. Weiterhin ist eine Volltextsuche vorhanden, die zur Reduzierung der Darstellung auf nur wenige Einträge genutzt werden kann. Es hat sich während der Arbeit mit der Anwendung jedoch herausgestellt, dass die Implementierung von separaten **Anzeigefil-**

tern, die beispielsweise das Ausblenden von „ignorierten" Vorfällen oder Verbindungsversuchen ermöglichen, gerade im Umgang mit großen Datenmengen die Benutzbarkeit des Frontends und die Übersicht in der Ereignisliste stark verbessern würde.

- Die als Verbindungsversuch klassifizierten Ereignisse geben abgesehen von einer Absender-IP-Adresse, dem verwendeten Protokoll und dem Zielport nur wenige Informationen über den potentiellen Angreifer preis. Eine interessante Erweiterung wäre die Implementierung eines Dienstes, der an eigentlich geschlossenen Ports ankommende TCP-Pakete aufzeichnen und dynamisch zugeschnittene **Antwortpakete generieren** kann, um den TCP-Handshake abzuschließen und die vom Angreifer anschließend übermittelten Daten zu protokollieren. Im Falle des verbindungslosen UDP-Protokolls würde es hingegen bereits genügen, wenn die Daten in jedem ankommenden Paket in komprimierter Form gespeichert werden würden. Die Übermittlung dieser Informationen zum Server stellt jedoch eine zusätzliche Herausforderung dar, falls vom Sensor große Datenmengen empfangen werden. Weiterhin wäre die Problematik zu lösen, wie selektiv dieser Mechanismus eingesetzt werden kann, um nicht im Falle eines Portscans mehrere tausend offene Ports zu präsentieren.
- Eine **automatische Generierung von Firmware-Updates** könnte im Falle von vom Debian-Projekt bereitgestellten Sicherheitsaktualisierungen nützlich sein, um direkt über das Webinterface eine bereits vorhandene Revision ohne Zutun des Benutzers zu aktualisieren. Für die Umsetzung der Funktion könnte die Firmware in eine virtuelle Maschine eingebunden werden, die CPUs anderer Architekturen als die des Hostsystems emulieren kann. Das freie Projekt *QEMU*[78] könnte sich hierfür als hilfreich erweisen.
- Beim Upload einer neuen Firmware zum Prototypen findet derzeit keine Überprüfung statt, ob die hochgeladene Datei identisch zum Original ist. Diese Funktion könnte durch die Einführung einer **Prüfsumme**, die vor der Übertragung vom Client berechnet und danach vom Server kontrolliert wird, nachgerüstet werden. Ebenso verhält es sich beim Download der Firmware vom Server, beispielsweise beim Update eines Sensors. Dies würde jedoch auch eine Erweiterung des Domänenmodells um eine entsprechende Prüfsumme erfordern.
- **Gestaffelte Sensorupdates** könnten verhindern, dass eine möglicherweise fehlerhafte Firmware auf alle Sensoren gleichzeitig übertragen wird und damit die gesamte Infrastruktur lahmlegt. Die serverseitige Firmware-Datenstruktur könnte zu diesem Zweck um ein Flag erweitert werden, das anzeigt, ob der letzte Sensor, der eine Aktualisierung auf eine bestimmte Version gestartet hat, diese auch erfolgreich abgeschlossen hat. Solange dies nicht geschehen ist, würde der Server die Aktualisierung weiterer Sensoren verweigern. Diese Technik hätte außerdem den Vorteil, dass nicht alle Sensoren parallel ihre neue Firmware vom Server beziehen und somit das Netzwerk unnötig belasten.
- Der HoneySens-Prototyp ist speziell auf die Gegebenheiten der BeagleBone-Plattform zugeschnitten. Eine **Modularisierung der Firmware** und der damit verbundenen Prozeduren würde die Integration anderer Hardware erleichtern und womöglich auch heterogene Sensornetze ermöglichen. Zusammen mit diesem Schritt wäre es außerdem hilfreich, Sensoren in Gruppen einteilen zu können, für die wiederum eine gemeinsame Konfiguration erstellt werden kann. Dieser Schritt sollte die Verwaltung einer großen Anzahl von Sensoren erleichtern.

[78] `http://qemu.org` (abgerufen im November 2014)

Literatur

[1] 2013 Internet Crime Report. http://www.ic3.gov/media/annualreport/2013_IC3Report.pdf, abgerufen im Oktober 2014.

[2] 2013 Norton Cybercrime Report. http://www.symantec.com/about/news/resources/press_kits/detail.jsp?pkid=norton-report-2013, Abgerufen im Oktober 2014.

[3] 2014 Data Breach Investigations Report. http://www.verizonenterprise.com/DBIR/2014/, abgerufen im November 2014.

[4] Berkeley Packet Filter (BPF) syntax. http://biot.com/capstats/bpf.html. Abgerufen im Oktober 2014.

[5] ConfigParser - Configuration file parser. https://docs.python.org/2/library/configparser.html. Abgerufen im Oktober 2014.

[6] Deception Toolkit. http://www.all.net/dtk/. Abgerufen am 06.06.2014.

[7] Kommunales Datennetz Sachsen. http://www.egovernment.sachsen.de/48.htm. Abgerufen im Juli 2014.

[8] May 2014 Web Server Survey. http://news.netcraft.com/archives/2014/05/07/may-2014-web-server-survey.html. Abgerufen im Oktober 2014.

[9] Nepenthes Documentation. http://nepenthes.carnivore.it/documentation. Abgerufen im Oktober 2014.

[10] Path Traversal vulnerability in VMware's shared folders implementation. http://www.coresecurity.com/content/advisory-vmware. Abgerufen im Juli 2014.

[11] SID im Porträt. http://www.sid.sachsen.de/portraet.htm. Abgerufen im Juli 2014.

[12] Usage of web servers for websites. http://w3techs.com/technologies/overview/web_server/all. Abgerufen im Oktober 2014.

[13] Zend Framework Manual: Recommended Project Directory Structure. http://framework.zend.com/manual/1.12/en/project-structure.project.html. Abgerufen im Oktober 2014.

[14] Kristina Beer. ENISA wirbt für digitale Hackerfallen. http://www.heise.de/security/meldung/ENISA-wirbt-fuer-digitale-Hackerfallen-1758563.html. Abgerufen am 28.05.2013.

[15] Leyla Bilge and Tudor Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 833–844. ACM, 2012.

[16] Pascal Brückner. Honeypots und Honey-Netze. 2013.

[17] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[18] W.R. Cheswick and S.M. Bellovin. Firewalls and Internet Security: Repelling the Wily Hacker. 1994.

[19] Douglas Crockford. The JSON saga. *YUI Theater video*, 2009.

[20] Julie Tate Ellen Nakashima, Greg Miller. U.S., Israel developed Flame computer virus to slow Iranian nuclear efforts, officials say. `http://articles.washingtonpost.com/2012-06-19/world/35460741_1_stuxnet-computer-virus-malware`. Abgerufen im Oktober 2014.

[21] John Fink. Docker: a Software as a Service, Operating System-Level Virtualization Framework. *Code4Lib Journal*, (25), 2014.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[23] Juszczyk Kijewski Pawlinski Grudziecki, Jacewicz. Proactive Detection of Security Incidents. 2012.

[24] Xuxian Jiang, Dongyan Xu, and Yi-Min Wang. Collapsar: a VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *Journal of Parallel and Distributed Computing*, 66(9):1165–1180, 2006.

[25] Saurabh Kulkarni, Madhumitra Mutalik, Prathamesh Kulkarni, and Tarun Gupta. Honeydoop-a system for on-demand virtual high interaction honeypots. In *Internet Technology And Secured Transactions, 2012 International Conference for*, pages 743–747. IEEE, 2012.

[26] Steffen Lehmann. Studienarbeit. 2011.

[27] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[28] Ethan Marcotte. *Responsive Web Design*. Editions Eyrolles, 2011.

[29] Antonio Nucci and Steve Bannerman. Controlled Chaos [Internet Security]. *Spectrum, IEEE*, 44(12):42–48, 2007.

[30] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments, 2007.

[31] Sebastian Poeplau and Jan Gassen. A honeypot for arbitrary malware on USB storage devices. In *Risk and Security of Internet and Systems (CRiSIS), 2012 7th International Conference on*, pages 1–8. IEEE, 2012.

[32] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, first edition, 2007.

[33] Mathijs Jeroen Scheepers. Virtualization and Containerization of Application Infrastructure: A Comparison. 2014.

[34] Christian Seifert, Ian Welch, and Peter Komisarczuk. Taxonomy of Honeypots. 2006.

[35] Michael Shapiro and Ethan Miller. Managing databases with binary large objects. In *Mass Storage Systems, 1999. 16th IEEE Symposium on*, pages 185–193. IEEE, 1999.

[36] A. Smith. *Bluepot: Bluetooth Honeypot*. PhD thesis. Abgerufen im September 2014.

[37] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[38] Lanz Spitzner. Know Your Enemy: Honeywall CDROM roo 3rd Generation Technology. `http://old.honeynet.org/papers/cdrom/roo/`, 2005. Abgerufen im November 2014.

[39] Rogier Spoor. A distributed intrusion detection system based on passive sensors, 2007.

[40] Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005.

[41] Jörg Thoma. Antivirensoftware ist tot. http://www.golem.de/news/symantec-antivirensoftware-ist-tot-1405-106251.html.

[42] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[43] W3C. Open Web Platform Milestone Achieved with HTML5 Recommendation. `http://www.w3.org/2014/10/html5-rec.html.en`. 28.10.2014.

[44] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated Web Patrol with Strider Honeymonkeys. In *Proceedings of the 2006 Network and Distributed System Security Symposium*, pages 35–49, 2006.

[45] Jason Zander. New Windows Server containers and Azure support for Docker. `http://azure.microsoft.com/blog/2014/10/15/new-windows-server-containers-and-azure-support-for-docker/`. Abgerufen im November 2014.