Peter Yaworski

Hacking und Bug Hunting

Wie man Softwarefehler aufspürt und damit Geld verdient – ein Blick über die Schulter eines erfolgreichen Bug Hunters

dpunkt.verlag

Über den Autor

Peter Yaworski hat sich das Hacken selbst beigebracht, was er dem umfassenden Wissen der vielen Hacker verdankt, die vor ihm schon aktiv waren, einschließlich derjenigen, die in diesem Buch erwähnt werden. Er ist auch ein erfolgreicher Bug-Hunter, u. a. für Salesforce, Twitter, Airbnb, Verizon Media und das amerikanische Verteidigungsministerium. Zurzeit arbeitet er bei Shopify als Application Security Engineer und hilft dabei, den E-Commerce etwas sicherer zu machen.

Über den Fachkorrektor

Tsang Chi Hong, auch als FileDescriptor bekannt, ist Pentester und Bug-Hunter. Er lebt in Hongkong. Auf *https://blog.innerht.ml* schreibt er über Websicherheit. Er hört gerne Original-Soundtracks und besitzt einige Kryptowährungen.

Peter Yaworski

Hacking und Bug Hunting

Wie man Softwarefehler aufspürt und damit Geld verdient – ein Blick über die Schulter eines erfolgreichen Bug Hunters

dpunkt.verlag

Peter Yaworski

Lektorat: René Schönfeldt
Übersetzung: Peter Klicman
Projektkoordinierung/Lektoratsassistenz: Anja Weimer
Copy-Editing: Claudia Lötschert, *www.richtiger-text.de*
Satz: G&U Language & Publishing Services GmbH, *www.gundu.com*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-86490-734-0
PDF 978-3-96088-969-4
ePub 978-3-96088-970-0
mobi 978-3-96088-971-7

1. Auflage 2020

Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

5 4 3 2 1 0

Inhalt

Vorwort

Der beste Weg zu lernen, ist, einfach machen. So haben wir das Hacken gelernt. Wir waren jung. Wie alle Hacker vor uns und alle, die noch kommen werden, wurden wir von einer unkontrollierbaren, brennenden Neugier getrieben, zu verstehen, wie die Dinge funktionieren. Wir haben meist Computerspiele gespielt und uns im Alter von 12 Jahren entschieden, zu lernen, wie man selbst Software entwickelt. Wir haben uns das Programmieren in Visual Basic und PHP mit Leihbüchern aus der Bibliothek und durch die praktische Anwendung beigebracht.

Unser Wissen um die Softwareentwicklung ließ uns schnell erkennen, dass wir damit auch die Fehler anderer Entwickler aufspüren können. Wir verlegten uns vom Entwickeln auf das Knacken, und das Hacken ist seitdem unsere Leidenschaft. Zur Feier unseres Highschool-Abschlusses übernahmen wir den Übertragungskanal eines Fernsehsenders, um unserer Klasse zu gratulieren. Das war seinerzeit natürlich amüsant, doch wir lernten schnell, dass es Konsequenzen gab und dass das nicht die Aktionen der Art Hacker sind, die die Welt braucht. Der Fernsehsender und die Schule waren nicht gerade erfreut, und wir mussten zur Strafe den ganzen Sommer Fenster putzen. Am College brachten

wir unser Wissen in ein lukratives Consulting-Unternehmen ein, das zu seinen Hochzeiten Kunden aus dem öffentlichen und privaten Sektor auf der ganzen Welt beriet. Unsere Hacking-Erfahrungen führten zu HackerOne, einem von uns im Jahr 2012 gegründeten Unternehmen. Wir wollten es jedem Unternehmen im Universum ermöglichen, erfolgreich mit Hackern zusammenzuarbeiten, und das ist auch heute noch die Mission von HackerOne.

Wenn Sie diese Zeilen lesen, besitzen Sie ebenfalls diese Neugier, die nötig ist, um ein Hacker und Bug-Jäger zu werden. Wir glauben, dass dieses Buch ein ausgezeichneter Begleiter auf Ihrem Weg sein wird. Es ist voller echter Beispiele für Bug-Reports, die zu realen Bug-Bounties führten. Dazu liefert der Autor (und Hacker-Kollege) Pete Yaworski hilfreiche Analysen und Reviews. Er ist Ihr Partner, während Sie lernen, und das ist unbezahlbar.

Ein weiterer Grund, warum dieses Buch so wichtig ist, ist die Konzentration darauf, ein ethischer Hacker zu werden. Die Kunst des Hackings zu beherrschen, ist eine sehr mächtige Fähigkeit, die Sie hoffentlich zum Guten einsetzen. Die meisten erfolgreichen Hacker wissen, wie man sich beim Hacking auf dem schmalen Grat zwischen Gut und Böse bewegt. Viele Leute können Dinge knacken und versuchen auch, damit das schnelle Geld zu machen. Doch stellen Sie sich vor, dass Sie das Internet sicherer machen können, mit tollen Unternehmen auf der ganzen Welt zusammenarbeiten und dafür ganz nebenbei auch noch bezahlt werden. Ihr Talent hat das Potenzial, Milliarden von Menschen und deren Daten zu schützen. Wir hoffen, dass Sie genau das anstreben.

Wir sind Pete unendlich dankbar dafür, dass er sich die Zeit genommen hat, all das so eloquent zu dokumentieren. Wir hätten uns eine solche Quelle gewünscht, als wir damals angefangen haben. Es ist ein Vergnügen, Petes Buch zu lesen, und es enthält alle Informationen, die Sie benötigen, um Ihren Weg als Hacker zu gehen.

Viel Spaß beim Lesen und beim Hacken! Achten Sie darauf, verantwortungsbewusst zu hacken.

Michiel Prins und Jobert Abma, Gründer von HackerOne

Danksagung

Dieses Buch wäre ohne die HackerOne-Community nicht möglich gewesen. Ich möchte dem HackerOne-CEO Mårten Mickos danken, der sich an mich wandte, als ich mit der Arbeit an diesem Buch begann. Er lieferte unermüdlich Feedback und Ideen, um das Buch zu verbessern, und zahlte sogar das professionell entworfene Cover der im Eigenverlag erschienenen Ausgabe.

Ich möchte auch den HackerOne-Mitgründern Michiel Prins und Jobert Abma danken, die Vorschläge machten und einige Kapitel beisteuerten, als ich an den frühen Versionen dieses Buchs arbeitete. Jobert machte einen umfassenden Review und editierte jedes Kapitel, um Feedback und technische Erkenntnisse zu liefern. Seine Überarbeitung stärkte mein Vertrauen und lehrte mich sehr viel mehr, als ich je für möglich gehalten hätte.

Adam Bacchus las das Buch, fünf Tage nachdem er zu HackerOne stieß. Er steuerte Überarbeitungen bei und erläuterte, wie es sich anfühlt, auf jener Seite zu stehen, die Sicherheitslücken-Reports entgegennimmt (was mir beim Schreiben von Kapitel 19 half). HackerOne hat nie eine Gegenleistung erwartet. Sie wollten nur die Hacking-Community unterstützen, indem sie dieses Buch so gut machten, wie es nur ging.

Es wäre nachlässig von mir, mich nicht insbesondere auch bei Ben Sadeghipour, Patrik Fehrenbach, Frans Rosen, Philippe Harewood, Jason Haddix, Arne Swinnen, FileDescriptor und den vielen anderen zu bedanken, die am Anfang meines Wegs im Chat über Hacking mit mir diskutierten, ihr Wissen teilten und mich ermutigten.

Darüber hinaus wäre dieses Buch nicht möglich gewesen ohne die Hacker, die ihr Wissen teilen und Bugs veröffentlichen, insbesondere die Bugs, die in diesem Buch behandelt werden. Danke euch allen.

Zuletzt wäre ich nicht da, wo ich heute bin, ohne die Liebe und die Unterstützung meiner Frau und meiner zwei Töchter. Sie sind der Grund, warum ich ein erfolgreicher Hacker wurde und dieses Buch fertigstellen konnte. Natürlich geht mein Dank auch an den Rest meiner Familie, insbesondere an meine Eltern, die mir keine Nintendo-Systeme schenken wollten, sondern mir stattdessen Computer kauften und erklärten, dass das die Zukunft sei.

Einführung

Dieses Buch führt in die große Welt des *ethischen Hackings* ein, also dem Prozess, Schwachstellen verantwortungsvoll aufzudecken und diese dem Eigner der Anwendung zu melden. Als ich mit dem Hacken begann, wollte ich nicht nur wissen, *welche* Schwachstellen Hacker gefunden hatten, sondern auch *wie* sie diese Lücken aufgedeckt hatten.

Ich suchte nach Informationen, doch es blieben immer die gleichen Fragen:

- Welche Schwachstellen finden Hacker in Anwendungen?
- Wie finden sie diese Schwachstellen?
- Wie beginnen sie mit der Infiltration einer Site?
- Wie sieht das Hacking aus: Läuft alles automatisiert, oder ist es Handarbeit?
- Wie kann ich selbst mit dem Hacking beginnen und Schwachstellen aufspüren?

Letztlich landete ich bei HackerOne, einer sogenannten Bug-Bounty-Plattform. Ihr Ziel ist es, ethische Hacker mit Unternehmen zusammenzubringen, die nach Hackern suchen, um ihre Anwendungen zu testen. HackerOne umfasst Funk-

tionen, die es Hackern und Unternehmen erlauben, aufgedeckte und behobene Bugs zu veröffentlichen.

Während ich die veröffentlichten HackerOne-Reports las, kämpfte ich damit zu verstehen, welche Lücken die Hacker gefunden hatten und wie man sie ausnutzen konnte. Häufig musste ich den gleichen Report zwei- oder dreimal lesen, um ihn zu verstehen. Mir wurde schnell klar, dass ich (und andere Einsteiger) von leicht verständlichen Erläuterungen realer Schwachstellen sehr profitieren würde. Und so kam es schließlich zu diesem Buch.

Hacking und Bug Hunting ist eine Referenz, die Ihnen dabei hilft, die unterschiedlichen Arten von Sicherheitslücken im Web zu verstehen. Sie werden lernen, wie man solche Schwachstellen findet, wie man sie meldet, wie man dafür bezahlt wird und (gelegentlich) auch, wie man defensiven Code entwickelt. Doch das Buch enthält nicht nur erfolgreiche Beispiele. Es zeigt Ihnen auch Fehler und wichtige Erkenntnisse aus der praktischen Arbeit; viele davon sind meine eigenen.

Wenn Sie mit dem Buch durch sind, haben Sie die ersten Schritte unternommen, um das Web zu einem sichereren Ort zu machen, und sollten dabei auch noch etwas Geld verdienen können.

Wer dieses Buch lesen sollte

Dieses Buch richtet sich an Hacker-Neulinge. Es spielt keine Rolle, ob Sie Webentwickler, Webdesigner, in Elternzeit, ein 10-jähriges Kind oder ein 75-jähriger Rentner sind.

Zwar ist es keine Voraussetzung für das Hacking, doch etwas Programmiererfahrung und die Vertrautheit mit Webtechnologien sind hilfreich. Sie müssen beispielsweise kein Webentwickler sein, um ein Hacker zu werden, doch das Verständnis der HTML-Struktur einer Webseite oder Kentnisse darüber, wie CSS (Cascading Style Sheets) ihr Aussehen definiert und wie JavaScript dynamisch mit Webseiten interagiert, hilft Ihnen dabei, Lücken aufzuspüren und die Auswirkung des entdeckten Bugs zu beurteilen.

Programmieren zu können ist hilfreich, wenn man Schwachstellen sucht, die die Logik einer Anwendung betreffen, und wenn man sich Gedanken darüber macht, welche Fehler ein Programmierer gemacht haben könnte. Wenn Sie sich in den Programmierer hineinversetzen und absehen können, wie er etwas implementiert hat, oder (falls verfügbar) seinen Code lesen können, erhöhen sich Ihre Erfolgsaussichten.

Wenn Sie etwas über Programmierung lernen wollen, finden Sie, u. a. beim dpunkt.verlag, eine Vielzahl hilfreicher Bücher. Sie können sich auch die kostenlosen Kurse auf Udacity und Coursera ansehen. Anhang B führt weitere Ressourcen auf.

Wie man dieses Buch liest

Jedes Kapitel, das einen bestimmten Schwachstellen-Typ beschreibt, hat die folgende Struktur:

1. Eine Beschreibung des Schwachstellen-Typs
2. Beispiele für diese Art von Schwachstelle
3. Eine Zusammenfassung mit Schlussfolgerungen

Jedes Beispiel einer Schwachstelle umfasst:

- meine Einschätzung des Schwierigkeitsgrads, die Schwachstelle aufzuspüren und zu belegen
- den URL mit dem Fundort der Schwachstelle
- einen Link auf den Original-Report oder die Rezension
- das Datum, an dem die Sicherheitslücke gemeldet wurde
- den Betrag, der für die Schwachstelle gezahlt wurde
- eine klare Beschreibung der Schwachstelle
- die Kernpunkte, die man für sein eigenes Hacking nutzen kann

Sie müssen dieses Buch nicht von vorne bis hinten durchlesen. Wenn Sie ein bestimmtes Kapitel besonders interessiert, dann lesen Sie es zuerst. In manchen Fällen spreche ich Konzepte an, die in früheren Kapiteln behandelt wurden. Ich gebe dann aber auch an, wo ein Begriff definiert wurde, damit Sie den entsprechenden Abschnitt schnell finden. Sie sollten das Buch neben sich haben, während Sie hacken.

Was Sie in diesem Buch finden

Hier eine Übersicht dessen, was Sie in den einzelnen Kapiteln finden:

Kapitel 1: Bug-Bounty-Grundlagen erklärt, was Schwachstellen und Bug-Bounties sind, sowie den Unterschied zwischen Clients und Servern. Es erläutert auch, wie das Internet funktioniert, was HTTP-Requests, -Responses und -Methoden sind und was »HTTP ist zustandslos« bedeutet.

Kapitel 2: Offene Redirects behandelt Angriffe, die das in eine gegebene Domain gesetzte Vertrauen ausnutzen, um Nutzer auf eine andere Domain umzuleiten.

Kapitel 3: HTTP-Parameter-Pollution zeigt, wie Angreifer HTTP-Requests manipulieren, zusätzliche Parameter einschleusen (denen die verwundbare Website vertraut) und wie dies zu unerwartetem Verhalten führt.

Kapitel 4: Cross-Site-Request-Forgery zeigt, wie ein Angreifer eine bösartige Website nutzen kann, um einen angegriffenen Browser dazu zu bringen, einen

HTTP-Request an eine andere Website zu senden. Die andere Website agiert dann so, als wäre der Request legitim und gewollt gesendet worden.

Kapitel 5: HTML-Injection und Content-Spoofing erläutert, wie böswillige Nutzer eigene HTML-Elemente in die Webseiten einer angegriffenen Site einschleusen.

Kapitel 6: Carriage Return/Line Feed-Injection zeigt, wie Angreifer codierte Zeichen in HTTP-Nachrichten einfügen, um deren Interpretation durch Server, Proxies und Browser zu verändern.

Kapitel 7: Cross-Site-Scripting erläutert, wie Angreifer Sites ausnutzen, die Benutzereingaben nicht ausreichend prüfen, um eigenen JavaScript-Code auf der Site auszuführen.

Kapitel 8: Template-Injection erklärt, wie Angreifer Template-Engines ausnutzen, wenn Sites die Benutzereingaben nicht ausreichend prüfen, die in den Templates genutzt werden. Das Kapitel enthält client- und serverseitige Beispiele.

Kapitel 9: SQL-Injection beschreibt, wie es Schwachstellen in einer datenbankgestützten Anwendung einem Angreifer ermöglichen, die Datenbank der Site abzufragen oder anzugreifen.

Kapitel 10: Server-Side-Request-Forgery erläutert, wie ein Angreifer einen Server dazu bringt, unbeabsichtigte Netzwerk-Requests durchzuführen.

Kapitel 11: Externe Entitäten bei XML zeigt, wie Angreifer die Art und Weise ausnutzen, in der eine Anwendung XML-Eingaben verarbeitet und externer Entitäten in die Eingabe einbindet.

Kapitel 12: Remote-Code-Execution diskutiert, wie Angreifer einen Server oder eine Anwendung missbrauchen, um eigenen Code auszuführen.

Kapitel 13: Speicher-Schwachstellen erklärt, wie Angreifer das Speichermanagement einer Anwendung ausnutzen, um unerwartetes Verhalten herbeizuführen, einschließlich der möglichen Ausführung eingeschleuster Befehle.

Kapitel 14: Übernahme von Subdomains zeigt, wie es zur Übernahme von Subdomains kommt, das heißt, wie ein Angreifer eine Subdomain einer gültigen Domain kontrollieren kann.

Kapitel 15: Race Conditions erklärt, wie Angreifer Situationen ausnutzen, in denen die Prozesse einer Site ihre Arbeit basierend auf Ausgangsbedingungen abschließen wollen, die während der Ausführung der Prozesse nicht mehr gelten.

Kapitel 16: Insecure Direct Object References beschreibt Sicherheitslücken, die auftreten, wenn ein Angreifer die Referenz auf ein Objekt (eine Datei, einen Datensatz aus einer Datenbank, einen Account) nutzen oder modifizieren kann, auf die er eigentlich keinen Zugriff haben sollte.

Kapitel 17: OAuth-Schwachstellen behandelt Bugs in der Implementierung des Protokolls, das die sichere Autorisierung für Web-, Desktop- und mobile Anwendungen vereinfachen und standardisieren soll.

Kapitel 18: Schwachstellen in Anwendungslogik und -konfiguration erläutert, wie ein Angreifer Fehler in der Programmlogik oder der Konfiguration einer Anwendung ausnutzen kann, um die Site einige unbeabsichtigte Aktionen ausführen zu lassen, die zu einer Schwachstelle führen.

Kapitel 19: Eigene Bug-Bounties gibt Tipps, wo und wie man (basierend auf meiner Erfahrung und Methode) nach Sicherheitslücken suchen kann. Dieses Kapitel ist keine Schritt-für-Schritt-Anleitung zum Hacken einer Site.

Kapitel 20: Bug-Reports diskutiert, wie man glaubwürdige und informative Reports zu Schwachstellen verfasst, damit die entsprechenden Bug-Bounty-Programme ihre Meldungen nicht ablehnen.

Anhang A: Tools stellt beliebte Werkzeuge für Hacker vor, darunter Web-Traffic-Proxies, Subdomain-Auflistung, Screenshots und vieles mehr.

Anhang B: Ressourcen führt zusätzliche Ressourcen auf, die Ihr Hacking-Wissen erweitern. Dazu gehören Online-Trainings, beliebte Bounty-Plattformen, empfohlene Blogs und so weiter.

Ein Disclaimer zum Hacking

Wenn Sie in den Medien über Schwachstellen lesen und sehen, wie viel Geld einige Hacker verdienen, ist es nur natürlich zu glauben, dass das Hacking eine einfache und schnelle Möglichkeit ist, reich zu werden. Doch das ist es nicht. Hacking kann lohnenswert sein, doch Geschichten über das Scheitern auf diesem Weg werden Sie kaum finden (außer in diesem Buch, wo ich einige sehr peinliche Geschichten mit Ihnen teilen werde). Da Sie hauptsächlich von den Erfolgen einiger Hacker hören werden, könnten Sie unrealistische Erwartungen in Bezug auf ihre eigene Hacker-Karriere entwickeln.

Sie können schnell Erfolg haben, doch oft wird es so sein, dass Sie Bugs nicht auf Anhieb finden und sich die Suche dann zeitaufwendig gestaltet. Geben Sie aber nicht auf. Entwickler werden immer neuen Code schreiben, und Bugs finden immer ihren Weg in den Produktiv-Code. Je öfter Sie Bugs suchen, desto routinierter werden Sie sein.

In diesem Sinne ermuntere ich Sie, mir eine Nachricht auf Twitter @yaworsk zu senden und mir zu schreiben, wie es Ihnen mit dem Bug Hunting geht. Selbst wenn Sie keinen Erfolg haben, würde ich gerne von Ihnen hören. Die Jagd nach Bugs kann eine einsame Arbeit sein. Doch es ist auch großartig, miteinander zu feiern, und vielleicht finden Sie ja etwas, das ich in die nächste Auflage dieses Buchs aufnehmen kann.

Viel Glück und viel Spaß beim Hacken!

1

Bug-Bounty-Grundlagen

Ist Hacking etwas Neues für Sie? Dann legen Sie sich jetzt ein grundlegendes Verständnis der Funktionsweise des Internets zu und lernen, was hinter den Kulissen passiert, wenn Sie einen URL in der Adressleiste des Browsers eingeben. Auch wenn der Besuch einer Website einfach aussieht, so umfasst er viele verborgene Prozesse, etwa den Aufbau eines HTTP-Requests, die Identifikation der Domain, an die der Request gesendet werden soll, die Übersetzung der Domain in eine IP-Adresse, die Rückgabe einer Response und so weiter.

In diesem Kapitel lernen Sie grundlegende Konzepte und Begriffe kennen wie etwa Schwachstellen, Bug-Bounties, Clients, Server, IP-Adressen und HTTP. Sie bekommen eine grundsätzliche Vorstellung davon, wie unerwartete Aktionen und unerwartete Eingaben sowie der Zugriff auf private Informationen zu Schwachstellen führen. Dann sehen wir uns an, was passiert, wenn Sie einen URL in der Adressleiste Ihres Browsers eingeben, wie HTTP-Requests und -Responses aussehen sowie die verschiedenen HTTP-Aktionsverben. Wir beenden das Kapitel mit der Erklärung, was »HTTP ist zustandslos« bedeutet.

1.1 Schwachstellen und Bug-Bounties

Eine Schwachstelle (engl. *vulnerability*) in einer Anwendung, ermöglicht es einer böswilligen Person, unerwünschte Aktionen auszuführen oder Zugriff auf Informationen zu erhalten, auf die sie normalerweise nicht zugreifen darf.

Während Sie Anwendungen testen, sollten Sie daran denken, dass Angreifer solche Schwachstellen durch beabsichtigte und unbeabsichtigte Aktionen öffnen können. Wenn Sie etwa die ID eines Datensatzes ändern, um auf Informationen zuzugreifen, die Sie eigentlich nicht sehen sollten, dann ist das ein Beispiel für eine (vom Entwickler) nicht beabsichtigte Aktion.

Nehmen wir an, Sie können auf einer Website ein Profil mit Name, E-Mail, Geburtsdatum und Adresse anlegen. Diese Informationen sollen vertraulich behandelt werden und nur für jene Benutzer sichtbar sein, die als Ihre Freunde bekannt sind. Wenn es die Website aber jedem erlaubt, Sie ohne Ihre Zustimmung als Freund aufzunehmen, dann ist das eine Schwachstelle. Denn selbst wenn die Site Ihre Daten vor Nicht-Freunden schützt, kann Sie jeder als Freund hinzufügen und so auf diese Informationen zugreifen. Während Sie eine Website testen, sollten Sie immer darüber nachdenken, wie jemand die vorhandene Funktionalität missbrauchen könnte.

Ein *Bug-Bounty* ist eine Belohnung, die eine Website oder ein Unternehmen an jemanden bezahlt, der (ethisch sauber) eine Schwachstelle entdeckt und meldet. Die Belohnung ist oft Geld und reicht von ein paar Zehn bis zu Tausenden von Dollar. Andere Beispiele für Bounties sind Kryptowährungen, Flugmeilen, Belohnungspunkte, Gutschriften und so weiter.

Bietet ein Unternehmen Bug-Bounties an, legt es ein *Programm* auf. Wir verwenden diesen Begriff in diesem Buch für die Regeln und das Rahmenwerk, die Unternehmen für Leute aufstellen, die das Unternehmen auf Schwachstellen testen wollen. Beachten Sie, dass sich das von den sogenannten *Vulnerability Disclosure Programs (VDPs)* anderer Unternehmen unterscheidet. Bug-Bounties bieten eine monetäre Belohnung, während ein VDP keine Bezahlung bietet (auch wenn das Unternehmen eine Prämie gewähren kann). Ein VDP ist nur eine Möglichkeit für ethische Hacker, Schwachstellen an ein Unternehmen zu melden, die es dann beheben kann. Zwar wurden nicht alle Reports in diesem Buch finanziell belohnt, doch alle Beispiele stammen von Hackern, die an Bug-Bounty-Programmen teilnehmen.

1.2 Client und Server

Ihr Browser ist auf das Internet angewiesen, ein Netzwerk aus Computern, die einander Nachrichten senden. Wir nennen diese Nachrichten *Pakete*. Pakete

umfassen die von Ihnen gesendeten Daten sowie Informationen darüber, wo diese Daten herkommen und wohin sie gehen. Jeder Computer im Internet hat eine Adresse, an die Pakete gesendet werden können. Doch einige Computer akzeptieren nur bestimmte Arten von Paketen, während wieder andere nur Pakete von einer beschränkten Liste anderer Computer empfangen. Der empfangende Computer muss dann entscheiden, was mit den Paketen geschehen und wie reagiert werden soll. In diesem Buch konzentrieren wir uns nur auf die in den Paketen enthaltenen Daten (die HTTP-Nachrichten), nicht auf die Pakete selbst.

Ich bezeichne diese Computer entweder als Clients oder als Server. Der die Requests (Anforderungen) initiierende Computer ist üblicherweise der *Client*, unabhängig davon, ob der Request durch einen Browser, die Kommandozeile und so weiter initiiert wird. *Server* stehen für die Websites und Webanwendungen, die diese Requests empfangen. Ist ein Konzept sowohl auf Clients als auch auf Server anwendbar, spreche ich ganz allgemein von Computern.

Da im Internet eine beliebige Anzahl von Computern miteinander reden kann, benötigen wir Richtlinien, auf welche Art sie über das Internet miteinander kommunizieren sollen. Diese liegen in Form der sogenannten *Request for Comments (RFCs)* vor, die Standards definieren, wie Computer sich zu verhalten haben. Zum Beispiel definiert das *Hypertext Transfer Protocol (HTTP)*, wie ein Internet-Browser mit einem entfernten Server über das *Internet Protocol (IP)* kommuniziert. Bei diesem Szenario müssen sowohl Client als auch Server die gleichen Standards implementieren, um die Pakete verstehen zu können, die sie senden und empfangen.

1.3 Was beim Besuch einer Website passiert

Weil wir uns in diesem Buch auf HTTP-Nachrichten konzentrieren, geben wir in diesem Abschnitt eine (auf hohem Niveau angesiedelte) Übersicht des Prozesses, der durchlaufen wird, wenn Sie einen URL in der Adressleiste Ihres Browsers eingeben.

1.3.1 Schritt 1: Extrahieren des Domainnamens

Sobald Sie *http://www.google.com/* eingegeben haben, bestimmt Ihr Browser den Domainnamen aus dem URL. Ein *Domainname* identifiziert die Website, die Sie besuchen wollen, und muss bestimmten Regeln folgen, die durch die RFCs definiert sind. Beispielsweise darf ein Domainname nur alphanumerische Zeichen und Unterstriche enthalten. Eine Ausnahme sind internationalisierte Domainnamen, die aber den Rahmen dieses Buchs sprengen würden. Wer mehr über

sie erfahren will, sei auf RFC 3490 verwiesen, das deren Nutzung definiert. Die Domain ist eine Möglichkeit, die Adresse eines Servers zu ermitteln.

1.3.2 Schritt 2: Auflösen der IP-Adresse

Nachdem der Domainname ermittelt wurde, nutzt Ihr Browser das IP, um die mit der Domain verknüpfte *IP-Adresse* zu bestimmen. Dieser Prozess wird als Auflösung (engl. Resolving) der IP-Adresse bezeichnet, und jede Domain im Internet muss zu einer IP-Adresse aufgelöst werden, um funktionieren zu können.

Es gibt zwei Arten von IP-Adressen: das Internet Protocol Version 4 (IPv4) und das Internet Protocol Version 6 (IPv6). IPv4-Adressen sind als vier durch Punkte voneinander getrennte Zahlen organisiert, und jede dieser Zahlen liegt im Bereich von 0 bis 255. IPv6 ist die neueste Version des Internetprotokolls. Sie wurde entwickelt, um das Problem ausgehender IPv4-Adressen zu lösen. IPv6-Adressen bestehen aus acht Gruppen von vier Hexadezimalzahlen, die durch Doppelpunkte voneinander getrennt sind, doch es gibt auch Methoden, um IPv6-Adressen zu verkürzen. So ist 8.8.8.8 beispielsweise eine IPv4-Adresse und 2001:4860:4860::8888 eine verkürzte IPv6-Adresse.

Um eine IP-Adresse über den Domainnamen nachzuschlagen, sendet Ihr Computer einen Request an einen Server des *Domain Name Systems (DNS)*. Das DNS besteht aus spezialisierten Servern im Internet, die ein Register aller Domains und der zugehörigen IP-Adressen vorhalten. Die obigen IPv4- und IPv6-Adressen sind Google-DNS-Server.

In unserem Beispiel würde der DNS-Server, mit dem Sie die Verbindung herstellen, die Domain *www.google.com* zur IPv4-Adresse 216.58.201.228 auflösen und an Ihren Computer zurückschicken. Um mehr über die IP-Adresse einer Site zu erfahren, können Sie z.B. den Befehl `dig A site.com` auf einer Linux-Kommandozeile eingeben, wobei Sie *site.com* durch die gewünschte Site ersetzen.

1.3.3 Schritt 3: Herstellen einer TCP-Verbindung

Als Nächstes versucht der Computer, eine *TCP*-Verbindung (*Transmission Control Protocol*) mit der IP-Adresse an Port 80 herzustellen, weil Sie die Site über *http://* besuchen. Details über TCP sind an dieser Stelle nicht wichtig. Es handelt sich nur um ein weiteres Protokoll, das die Kommunikation zwischen Computern definiert. TCP stellt eine Zwei-Wege-Kommunikation bereit, sodass die Empfänger der Nachrichten die empfangenen Informationen verifizieren können und nichts bei der Übertragung verloren geht.

Auf dem Server, an den Sie einen Request senden, können mehrere Dienste laufen (stellen Sie sich einen Dienst als Computerprogramm vor), weshalb er

Ports nutzt, um die Prozesse festzulegen, die die Requests empfangen sollen. Ohne Ports müssten die Dienste um die Informationen konkurrieren, die an den gleichen Ort gesendet werden. Das bedeutet, dass wir einen weiteren Standard benötigen, der definiert, wie Dienste miteinander kooperieren, und sicherstellt, dass Daten für einen Dienst nicht von einem anderen gestohlen werden. Port 80 ist zum Beispiel der Standard-Port für das Senden und Empfangen unverschlüsselter HTTP-Requests. Ein weiterer gängiger Port ist 443, der für verschlüsselte HTTP-Requests genutzt wird. Zwar ist Port 80 Standard für HTTP und 443 für HTTPS, doch die TCP-Kommunikation kann an jedem Port erfolgen, je nachdem, wie der Administrator die Anwendung konfiguriert.

Sie können eine eigene TCP-Verbindung zu einer Website an Port 80 herstellen, indem Sie ein Linux-Terminal öffnen und `nc <IP-ADRESSE> 80` ausführen. Sie verwenden dabei den Netcat-Befehl `nc`, um eine Netzwerkverbindung zum Lesen und Schreiben von Nachrichten zu erzeugen.

1.3.4 Schritt 4: Senden eines HTTP-Requests

Zurück zu unserem *http://www.google.com/*-Beispiel. Wird die Verbindung in Schritt 3 erfolgreich hergestellt, erzeugt und sendet Ihr Browser einen HTTP-Request wie in Listing 1–1:

```
❶ GET / HTTP/1.1
❷ Host: www.google.com
❸ Connection: keep-alive
❹ Accept: application/html, */*
❺ User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
```

Listing 1–1 *Senden eines HTTP-Requests*

Der Browser erzeugt einen `GET`-Request auf den Pfad / ❶, der den Stamm (engl. root) der Website darstellt. Der Inhalt einer Website ist in Pfaden organisiert, ähnlich den Ordnern und Dateien auf Ihrem Computer. Wenn Sie tiefer in einen Ordner eintauchen, wird der von Ihnen genommene Pfad durch den Namen des Ordners, gefolgt von einem / angezeigt. Wenn Sie die Startseite einer Website besuchen, greifen Sie auf den Stamm-Pfad zu, also auf /. Der Browser zeigt auch an, dass er das HTTP-Protokoll in der Version 1.1 verwendet. Ein `GET`-Request ruft schlicht Informationen ab. Wir werden später mehr darüber erfahren.

Der *Host-Header* ❷ enthält eine weitere Information, die als Teil des Requests gesendet wird. HTTP 1.1 benötigt sie, um bestimmen zu können, wohin der Server an der gegebenen IP-Adresse den Request senden soll, da IP-Adressen mehrere Domains hosten können. Ein Verbindungs-Header (*connection header* ❸) gibt

an, dass die Verbindung mit dem Server offen bleiben soll, um den Overhead durch das fortlaufende Öffnen und Schließen von Verbindungen zu verhindern.

Die zu erwartende Antwort sehen Sie an ❹. In diesem Fall erwarten wir `application/html`, akzeptieren aber jedes Format, was durch das Wildcard-Symbol (`*/*`) angedeutet wird. Es gibt Hunderte möglicher Typen, doch in unserem Fall werden Sie `application/html`, `application/json`, `application/octet-stream` und `text/ plain` am häufigsten sehen. Abschließend gibt der User-Agent ❺ an, welche Software den Request gesendet hat.

1.3.5 Schritt 5: Die Response des Servers

Als Response auf unseren Request antwortet der Server mit Code:

```
❶ HTTP/1.1 200 OK
❷ Content-Type: text/html
   <html>
     <head>
       <title>Google.com</title>
     </head>
     <body>
❸     --schnipp--
     </body>
   </html>
```

Listing 1–2 *Server-Response*

Hier haben wir eine HTTP-Response mit dem HTTP/1.1-Statuscode 200 ❶ empfangen. Der Statuscode ist wichtig, weil er angibt, wie der Server antwortet. Die Codes sind ebenfalls durch einen RFC definiert und verwenden üblicherweise dreistellige Zahlen, die mit 2, 3, 4, oder 5 beginnen. Zwar müssen Server keine bestimmten Codes nutzen, doch üblicherweise geben die 2*xx*er-Codes an, dass ein Request erfolgreich war.

Da es keine strikten Regeln gibt, wie der Server die Verwendung von HTTP-Codes implementiert, kann es sein, dass einige Anwendungen mit 200 antworten, obwohl der HTTP-Nachrichten-Rumpf besagt, dass es einen Fehler in der Anwendung gab. Ein *HTTP-Nachrichten-Rumpf* (engl. *message body*) ist der zu einem Request oder einer Response gehörende Text ❸. In unserem Beispiel haben wir den Inhalt entfernt und durch `--schnipp--` ersetzt, da der Response-Body von Google sehr groß ist. Der Text in einer Response ist üblicherweise der HTML-Code einer Webseite, kann bei einer Programmierschnittstelle aber auch JSON sein oder der Inhalt einer Datei bei einem Download und so weiter.

Der Content-Type-Header ❷ informiert die Browser über den Mediatyp des Body. Der Mediatyp bestimmt, wie der Browser den Inhalt rendert. Doch Brow-

ser nutzen nicht immer den von der Anwendung zurückgegebenen Wert, sondern führen ein sogenanntes *MIME-Sniffing* durch, das heißt, sie lesen den Anfang des Inhalts ein und bestimmen den Mediatyp selbst. Anwendungen können dieses Verhalten des Browsers deaktivieren, indem sie den Header *X-Content-Type-Options: nosniff* einfügen (der im obigen Beispiel nicht genutzt wird).

Andere Response-Codes beginnen mit 3 und zeigen einen Redirect (also eine Umleitung) an, der den Browser anweist, einen weiteren Request durchzuführen. Wenn Google Sie also permanent auf einen anderen URL umleiten möchte, könnte es eine 301-Response verwenden. Im Gegensatz dazu ist 302 ein temporärer Redirect.

Wird eine 3*xx*-Response empfangen, führt der Browser einen neuen HTTP-Request zu dem URL durch, der im `Location`-Header wie folgt definiert ist:

```
HTTP/1.1 301 Found
Location: https://www.google.com/
```

Mit 4 beginnende Responses zeigen üblicherweise einen Benutzerfehler an, etwa die Response 403 bei einer fehlerhaften Identifikation beim Zugriff auf autorisierte Inhalte (auch wenn der HTTP-Request selbst gültig ist). Mit 5 beginnende Responses zeigen einen Fehler des Servers an. Beispielsweise gibt 503 an, dass der Server den Sende-Request nicht durchführen kann.

1.3.6 Schritt 6: Rendering der Response

Da der Server eine 200er-Response mit dem Inhaltstyp text/html gesendet hat, beginnt unser Browser mit dem Rendering (also der Ausgabe) des empfangenen Inhalts. Der Body der Response sagt dem Browser, was er dem Nutzer präsentieren soll.

In unserem Beispiel wäre das HTML für die Seitenstruktur, Cascading Style Sheets (CSS) für Styles und das Layout, JavaScript für zusätzliche, dynamische Funktionalität und Medien wie Bilder oder Videos. Der Server kann auch andere Inhalte zurückgeben, etwa XML, doch wir bleiben bei diesem Beispiel bei den Grundlagen. Kapitel 11 geht detaillierter auf XML ein.

Da Webseiten externe Dateien wie CSS, JavaScript und Medien referenzieren können, muss der Browser möglicherweise zusätzliche HTTP-Requests durchführen, um alle für die Webseite benötigten Dateien zu laden. Während der Browser diese zusätzlichen Dateien anfordert, setzt er das Parsing (also die Verarbeitung) der Response fort und stellt den Body als Webseite dar. In unserem Beispiel rendert er die Google-Homepage *www.google.com*.

JavaScript ist eine Skriptsprache, die von jedem wichtigen Browser unterstützt wird. JavaScript ermöglicht Webseiten dynamische Funktionalität, etwa die Aktualisierung einer Webseite ohne das Neuladen der Seite, die Prüfung, ob Ihr Passwort stark genug ist (bei einigen Websites) und so weiter. Wie andere Programmiersprachen auch besitzt JavaScript fest eingebaute Funktionen, kann Werte in Variablen speichern und Code als Reaktion auf Ereignisse auf einer Webseite ausführen. Sie hat auch Zugriff auf verschiedene Application Programming Interfaces (APIs). Diese Programmierschnittstellen ermöglichen es JavaScript, mit anderen Systemen zu interagieren, von denen das Document Object Model (DOM) wohl das wichtigste ist.

Das DOM erlaubt JavaScript den Zugriff und die Manipulation des HTML- und CSS-Codes einer Webseite. Das ist von Bedeutung, weil ein Angreifer, der seinen eigenen JavaScript-Code auf einer Site ausführen kann, Zugriff auf das DOM hat und auf der Site Aktionen gegen das anvisierten Opfer durchführen kann. Kapitel 7 geht weiter auf dieses Konzept ein.

1.4 HTTP-Requests

Die Vereinbarung zwischen Client und Server, wie HTTP-Nachrichten zu verarbeiten sind, schließt auch die Definition von Request-Methoden ein. Eine *Request-Methode* legt den Zweck des Client-Requests fest sowie das vom Client im Erfolgsfall erwartete Ergebnis. In Listing 1–1 haben wir zum Beispiel einen `GET`-Request an *http://www.google.com/* gesendet. Das impliziert, dass nur der Inhalt von *http://www.google.com/* zurückgegeben und keine weiteren Aktionen durchgeführt werden sollen. Da das Internet als Schnittstelle zwischen räumlich weit auseinanderliegenden Computern entworfen wurde, hat man Request-Methoden entwickelt und implementiert, um die ausführenden Aktionen unterscheiden zu können.

Der HTTP-Standard definiert die folgenden Request-Methoden: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `CONNECT` und `OPTIONS` (`PATCH` wurde im HTTP-RFC ebenfalls vorgeschlagen, ist aber nicht oft implementiert). Während dies geschrieben wird, senden Browser nur `GET` und `POST` über HTML. Jeglicher `PUT`-, `PATCH`- oder `DELETE`-Request ist das Ergebnis eines JavaScripts, das den HTTP-Request anstößt. Das wirkt sich aus, wenn wir später Beispiele für Schwachstellen in Anwendungen betrachten, die diese Art von Methoden erwarten.

Der nächste Abschnitt enthält eine kurze Übersicht der Request-Methoden, die Sie in diesem Buch finden.