

---

## 2 Bughunting

Was versteht man eigentlich unter *Bughunting*? Obwohl es relativ unwahrscheinlich ist, dass du dieses Buch in Händen hältst ohne zumindest eine vage Vorstellung davon zu haben, was sich hinter dem Begriff verbirgt, gehe ich in diesem Kapitel u.a. kurz auf die Beantwortung dieser Frage ein.

Unter dem Begriff Bughunting versteht man in der Regel den Vorgang, Fehler (sogenannte Bugs) innerhalb von Software oder Hardware ausfindig zu machen. Im Rahmen dieses Buches wird der Begriff jedoch ausschließlich dazu verwendet, um den Vorgang zum Finden von sicherheitsrelevanten Softwarefehlern zu beschreiben. Sicherheitsrelevante Softwarefehler, häufig auch Softwareschwachstellen genannt, erlauben es einem Angreifer, bspw. in ein entferntes System einzubrechen, oder vorhandene lokale Rechte zu erweitern (Privilege Escalation). Der Begriff Bughunter, wie er innerhalb dieses Buches verwendet wird, hat also nicht viel mit der deutschen Übersetzung, Insektensammler, zu tun, obwohl in gewisser Weise durchaus die eine oder andere Gemeinsamkeit besteht (gerade was das Wort »sammeln« betrifft). Was vor einigen Jahren noch weitgehend als Hobby, für den Eigengebrauch oder als werbewirksame Leistungen (siehe [XFORCE] und [EEYE]) durchgeführt wurde, hat mittlerweile seinen Weg in den Mainstream gefunden, nachdem bekannt wurde, dass sich mit Softwareschwachstellen und entsprechenden Programmen, um diese auszunutzen (so genannte Exploits), sogar Geld verdienen lässt (siehe [MILLER 2007], [AMINI 2009], [TIPP] und [IDEF]).

Softwareschwachstellen und Exploits erfreuen sich heutzutage eines relativ hohen Medieninteresses. So leben wir in einer Zeit, in der es ein Conficker-Wurm, der u.a. eine Softwareschwachstelle (siehe [MS08-067]) für seine Verbreitung ausnutzte, bisweilen bis in die Tagesschau schafft (siehe [OSTERHAGE 2009]). Darüber hinaus gibt es mittlerweile eine fast schon unüberschaubare Anzahl an Büchern und Informationen aus dem Netz, die sich mit der Ausnutzung solcher Schwachstellen beschäftigen. Zudem werden innerhalb der IT Security Community zuweilen heiße Debatten darüber geführt, wie und wem man eine Schwachstelle gefälligst zu melden hat, falls man eine findet. Aber trotz all dieser Aufmerksamkeit, die Softwareschwachstellen zuteil wird, gibt es erstaunlicherweise nur sehr wenige Informationen darüber, wie man sol-

che Schwachstellen überhaupt findet. Obwohl Begriffe wie »Softwareschwachstelle« oder »Exploit« mittlerweile wie selbstverständlich verwendet werden, ist es selbst gestandenen IT-Sicherheitsspezialisten oft schleierhaft, wie man die für Exploits zugrunde liegenden Schwachstellen überhaupt findet.

Würde man zehn verschiedene Bughunter nach ihrem Vorgehen fragen, um sicherheitsrelevante Softwarefehler ausfindig zu machen, so würde man mit großer Wahrscheinlichkeit zehn unterschiedliche Antworten erhalten. Dies ist wohl einer der Gründe dafür, warum es kein »Kochbuch zum Finden von Schwachstellen« gibt und wahrscheinlich nie geben wird. Genau deshalb habe ich es erst gar nicht versucht, ein solches »Kochbuch« zu erstellen. Ich habe in diesem Buch vielmehr meine eigenen Vorgehensweisen und Erkenntnisse festgehalten, die mir geholfen haben, ein paar dieser begehrten Schwachstellen innerhalb verschiedener Softwarelösungen ausfindig zu machen. Ich hoffe, dass dir die eine oder andere Beschreibung neue Erkenntnisse bringt und das Buch es zumindest ein wenig vermag, die vorhandene Informationslücke in puncto Bughunting etwas auszufüllen.

## 2.1 Nur zum Spaß?

Die Motivation für Bughunting sowie die dadurch angestrebten Ziele können durchaus vielfältig sein. Manch unabhängiger Bughunter möchte vielleicht die Sicherheit einer Software verbessern, wobei andere nach Ruhm und öffentlicher Aufmerksamkeit streben und wieder andere einfach nur auf einen guten Job aus sind. Ein Unternehmen hegt vielleicht Interesse daran, Schwachstellen in einem Konkurrenzprodukt zu finden, um sich im Markt besser zu positionieren oder Inhalte für Marketingkampagnen zu liefern. Nicht zu vergessen die vielen »bösen Menschen«, die stets auf der Suche nach neuen Möglichkeiten sind, um in Computersysteme oder Netzwerke einzudringen, oder diejenigen, die schlicht und ergreifend einfach Spaß am Bughunting haben.

## 2.2 Techniken und Vorgehensweisen

Obwohl keine formale Beschreibung von Bughunting verfügbar ist, gibt es dennoch einige Techniken und Vorgehensweisen für das Suchen nach Schwachstellen, von denen ich im Anschluss einige näher beschreiben werde. Diese Techniken lassen sich dabei in die Kategorien statische und dynamische Analysetechniken unterscheiden. Bei statischen Analysetechniken, oftmals auch statische Codeanalysen genannt, wird der Quellcode einer Software oder das Disassembly eines ausführbaren Binärprogramms hinsichtlich Fehler untersucht, ohne die Software bzw. das Programm dazu auszuführen. Bei der dynamischen Analysetechnik wird die Software hingegen mittels Debuggern und Fuzzern während der Ausführung nach Schwachstellen untersucht. Beide Techniken haben ihre jeweiligen Vor- und Nachteile, sodass sie im wahren Leben in der Regel miteinander kombiniert werden.

## Statische und dynamische Analyse

Ich für meinen Teil tendiere meist eher zu statischen Analysen. Das heißt, ich lese mir in der Regel den Quellcode oder das Disassembly der betreffenden Software Zeile für Zeile durch. Dabei versuche ich die Abläufe innerhalb der Software zu verstehen, um dadurch mögliche Fehler aufzudecken.

Würden wir uns beide gerade über dieses Thema unterhalten, bin ich mir ziemlich sicher, dass du mir jetzt wohl gerne folgende Frage gestellt hättest: »Wo genau fängst du an, den Quellcode oder das Disassembly durchzulesen?«. Wenn ich eine Software nach Schwachstellen untersuche, beginne ich üblicherweise damit, möglichst alle Eintrittspunkte von Eingabedaten der Software ausfindig zu machen. Dabei kann es sich um Netzwerkdaten, Daten aus Dateien oder der weiteren Ausführungsumgebung handeln, um nur einige Beispiele zu nennen. Falls ich solche Eintrittspunkte finde, verfolge ich die Eingabedaten bei ihrem Weg durch die Software, wobei ich stets nach möglicherweise fehlerhaften Codebereichen Ausschau halte, die bei der Verarbeitung der Eingabedaten zu Schwachstellen führen könnten. In manchen Fällen kann ich diese Eintrittspunkte von Eingabedaten bereits anhand des Quellcodes (siehe Kapitel 3) oder des Disassembly (siehe Kapitel 7) ausfindig machen. In anderen Fällen muss ich die statischen Analysen mit den Ergebnissen eines Debuggers kombinieren, um die relevanten Stellen innerhalb der Software ausfindig zu machen (siehe Kapitel 6). Neben dem eigentlichen Bughunting kombiniere ich statische und dynamische Analysetechniken ebenfalls dann, wenn es um die Ausnutzung einer gefundenen Schwachstelle geht. Falls ich eine Schwachstelle gefunden habe, möchte ich in der Regel auch wissen, ob und wie einfach sich die Schwachstelle ausnutzen lässt. Der einzige Weg, um dies herauszufinden, besteht darin, einen Exploit für die Schwachstelle zu schreiben. Um einen solchen Exploit zu erstellen, verbringe ich die meiste Zeit innerhalb eines Debuggers.

## Verdächtige Codebereiche untersuchen

Das beschriebene Vorgehen stellt nur eine Herangehensweise dar, um nach sicherheitsrelevanten Fehlern in Software zu suchen. Eine andere Vorgehensweise besteht beispielsweise darin, den Quellcode oder das Disassembly nach potenziell »unsicheren« Codebereichen und -konstrukten oder Bibliotheksfunktionen zu durchsuchen, die oftmals zu Sicherheitsproblemen führen. Dies können beispielsweise berühmt berüchtigte C/C++-Bibliotheksfunktionen wie `strcpy()` und `strcat()` sein. Oder man durchsucht die entsprechenden Binärprogramme nach Assembler-Instruktionen wie `movsx`, um dadurch mögliche Schwachstellen bei der Vorzeichenerweiterung (sign extension) ausfindig zu machen. Findet man solche möglicherweise problematische Stellen innerhalb der Software, so verfolgt man die an diesen Stellen verarbeiteten Daten zurück bis zu ihrem Ursprung. Findet man dabei heraus, dass es sich um benutzerdefinierbare Daten handelt, so stehen die Chancen gut, dass man auf diesem Weg eine Schwachstelle entdecken kann. Ich persönlich kann dieser Vorgehensweise nicht allzu vieles abgewinnen und setze sie daher nur sehr sporadisch ein. Ich bin jedoch sicher, dass es Bughunter

gibt, die auf diese Herangehensweise schwören und dagegen meiner präferierten Methode nicht viel abgewinnen können.

## Fuzzing

Ein komplett anderer Ansatz, um sicherheitsrelevante Fehler innerhalb von Software ausfindig zu machen, ist das sogenannte Fuzzing. Bei diesem Ansatz handelt es sich um eine dynamische Analysetechnik, deren Ziel darin besteht, die zu untersuchende Software mit bewusst fehlerhaften Eingaben zu konfrontieren. Obwohl ich mich nicht als Fuzzing-Experte bezeichnen würde – ich kenne Bughunter, die ihre eigenen Fuzzing-Frameworks entwickelt haben und damit relativ erfolgreich Fehler ausfindig machen –, nutze ich diesen Ansatz ebenfalls von Zeit zu Zeit, um dadurch Eintrittspunkte von Eingabedaten oder manchmal auch einen Fehler ausfindig zu machen (siehe Kapitel 9).

»Wie soll man denn bitteschön mit Fuzzing Eintrittspunkte für Eingabedaten in Software ausfindig machen können?« Naja, stell dir eine komplexe Applikation vor, die du nach Schwachstellen untersuchen möchtest und die du nur in binärer Form vorliegen hast. In solchen Fällen ist es nicht immer ganz einfach, diese Eintrittspunkte zu identifizieren. Komplexe Software hat aber auch eine weitere Eigenschaft, die wir uns zunutze machen können: Mit der Komplexität steigt in der Regel ebenfalls die Fehleranfälligkeit. Dies ist gerade bei Software der Fall, die viele verschiedene Dateiformate verstehen und verarbeiten muss. Dazu zählen beispielsweise Office-Programme, Media-Player und Antivirus-Produkte. Auch wenn die durch Fuzzing provozierten Fehler meist nicht sicherheitsrelevant sind, wie beispielsweise ein Programmabbruch aufgrund einer Division durch null innerhalb eines Office-Programms, so liefern mir diese Abstürze häufig wertvolle Hinweise darauf, wo benutzerdefinierte Daten innerhalb des Programms verarbeitet werden. Diese Stellen lassen sich dann meist relativ einfach zu ihren Eintrittspunkten zurückverfolgen.

Dies war nur ein kurzer Abriss einiger verfügbarer Techniken und Vorgehensweisen, die man zum Finden von Schwachstellen einsetzen kann. Falls du mehr über die Theorie des Bughuntings erfahren möchtest, kann ich dir zum Thema Quellcodeanalyse das Buch [DOWD et al. 2007] und zum Thema Fuzzing das Buch [SUTTON et al. 2007] wärmstens empfehlen.

## 2.3 Speicherfehler

Die innerhalb dieses Buches beschriebenen Programmierfehler bzw. Schwachstellen haben eines gemeinsam: Sie führen jeweils zu ausnutzbaren Speicherfehlern. Solche Speicherfehler treten dann auf, wenn ein Prozess, ein Thread oder der Kernel

- Speicher verwendet, der ihm nicht zugeordnet ist (bspw. NULL Pointer Dereferences, siehe Abschnitt 10.2),
- mehr Speicher verwendet als allokiert wurde (bspw. Buffer Overflows, siehe Abschnitt 10.1),

- uninitialisierte Speicherbereiche verwendet (bspw. uninitialisierte Variablen, siehe [HODSON 2008]) oder
- fehlerhaft mit der Heap-Speicherverwaltung umgeht (bspw. Double Frees, siehe [DOUBLEFREE]).

Die Ursache von Speicherfehlern liegt meist in der inkorrekten Verwendung von maschinennahen C/C++-Funktionalitäten wie expliziter Speicherverwaltung oder Zeigerarithmetik. Eine Unterkategorie von Speicherfehlern sind sogenannte Memory-Corruption-Schwachstellen. Eine solche Memory Corruption tritt dann auf, wenn ein Prozess, ein Thread oder der Kernel

- eine Speicherstelle modifiziert, die ihm nicht gehört, oder
- eine ihm eigene Speicherstelle mit invaliden Daten überschreibt.

Falls du dich bisher noch nicht allzu häufig mit derart maschinennahen Programmierfehlern bzw. Schwachstellen auseinandergesetzt hast, würde ich dir empfehlen, zunächst einen Blick in die Abschnitte 10.1, 10.2 und 10.3 zu werfen. In diesen Kapiteln werden die Grundlagen und Hintergrundinformationen zu den innerhalb dieses Buches beschriebenen Programmierfehlern und Schwachstellen erläutert.

Neben den beschriebenen Speicherfehlern gibt es noch zahlreiche weitere Schwachstellengattungen. Dazu zählen bspw. Logikfehler oder webspezifische Schwachstellen wie Cross-Site Scripting, Cross-Site Request Forgery und SQL Injection. Diese Schwachstellenklassen werden innerhalb dieses Buches jedoch nicht behandelt.

## 2.4 Handwerkszeug

Zum Finden und Ausnutzen von Softwareschwachstellen benötigt man allerhand verschiedene Werkzeuge. Die beiden wichtigsten sind ein Debugger und ein ordentlicher Disassembler.

### 2.4.1 Debugger

Ein Debugger sollte verschiedene Möglichkeiten bereitstellen, um User-Space-Prozesse oder den Kernel des jeweiligen Betriebssystems untersuchen zu können. Dazu zählt beispielsweise die Möglichkeit, beliebige Werte von beliebigen Speicheradressen oder Prozessorregistern einzusehen, Breakpoints zu setzen oder die Software Instruktion für Instruktion auszuführen. Jedes wichtige Mainstream-Betriebssystem stellt dabei seinen eigenen Debugger bereit, der mit den verschiedenen betriebssystemabhängigen Besonderheiten umzugehen weiß. Daneben gibt es gerade im Windows-Umfeld noch einige empfehlenswerte Debugger von Drittanbietern, die in manchen Bereichen durchaus ihre Vorzüge haben. Innerhalb der folgenden Tabelle werden die im Rahmen des Buches beschriebenen Betriebssystemplattformen sowie die jeweils eingesetzten Debugger kurz zusammengefasst:

Betriebssystem	Debugger	Kernel-Debugging
Microsoft Windows	WinDBG (der offizielle Windows-Debugger von Microsoft). OllyDBG und dessen Variante ImmunityDebugger.	WinDBG: Ja  OllyDBG/ImmunityDebugger: Nein
Linux	The GNU Debugger (gdb)	Ja
Sun Solaris	The Modular Debugger (mdb)	Ja
Mac OS X	The GNU Debugger (gdb)	Ja
iPhone OS	The GNU Debugger (gdb)	Ja

**Tab. 2-1** Debugger

Wie erwähnt werden diese Debugger innerhalb des Buches eingesetzt, um Schwachstellen ausfindig zu machen und diese auszunutzen. In den Abschnitten 10.4, 10.5 und 10.6 findest du darüber hinaus jeweils eine kurze Referenz der wichtigsten Kommandos für die verschiedenen Debugger.

## 2.4.2 Disassembler

Wenn du eine Software nach Fehlern untersuchen möchtest, aber keinen Zugriff auf deren Quellcode besitzt, musst du wohl oder übel die Analyse anhand des Assembler-Codes der entsprechenden Binärprogramme durchführen. Obwohl die beschriebenen Debugger in der Lage sind, die Codebereiche von Prozessen bzw. des Kernels in Assembler-Code darzustellen (Disassembly), so ist die Navigation innerhalb des Debuggers jedoch meist relativ umständlich und die Ausgabe häufig unübersichtlich. Aus diesem Grund setze ich für statische Analysen eines Binärprogramms den Interactive Disassembler Professional, besser bekannt als IDA Pro (siehe [IDA]), ein. IDA Pro unterstützt über 50 Prozessorfamilien und bietet neben verschiedenen Möglichkeiten, um sich interaktiv innerhalb des Disassembly bewegen zu können, eine äußerst hilfreiche Code-Graph-Funktion. Solltest du jemals mit dem Gedanken spielen, eine statische Analyse eines Binärprogramms durchzuführen, wirst du wohl an IDA Pro nicht vorbeikommen. Falls du mehr über IDA Pro erfahren möchtest, dann schau dir doch mal das äußerst empfehlenswerte Buch [EAGLE 2008] näher an.

## 2.5 EIP = 41414141

Wie bereits erwähnt, wird die Tragweite bzw. das Risikopotenzial der Schwachstellen innerhalb dieses Buches »lediglich« durch die Möglichkeit zur Kontrolle des Instruction Pointers der CPU demonstriert. Bei dem Instruction Pointer (IP), auch Program Counter (PC) genannt, handelt es sich

### Notiz

Instruction Pointer/Program Counter:  
 EIP - 32-Bit Instruction Pointer (IA-32)  
 RIP - 64-Bit Instruction Pointer (Intel 64)  
 R15 bzw. PC - ARM-Architektur  
 (bspw. iPhone)

um ein spezielles Prozessorregister der CPU, das auf die nächste auszuführende Instruktion verweist (siehe [INTEL 2008]). Kontrolliert man dieses Register, so besitzt man volle Kontrolle über den ausgenutzten Prozess. Um innerhalb des Buches eine solche Kontrolle des Instruction Pointers zu verdeutlichen, werde ich das entsprechende Register mit Werten wie 0x41414141 (Hexadezimalrepräsentation von ASCII »AAAA«) oder 0x41424344 (Hexadezimalrepräsentation von ASCII »ABCD«) füllen. Taucht in den folgenden Kapiteln daher beispielsweise ein EIP = 41414141 auf, so weißt du, was es damit auf sich hat (siehe [GOOG1]).

Um aus einer Kontrolle des Instruction Pointers einen kompletten Exploit zu basteln, bedarf es einiger weiterer Techniken, die an anderer Stelle detailliert beschrieben werden (siehe bspw. [KLEIN 2003], [GOOG2] und [ERICKSON 2008]).

## 2.6 Was nun folgt

Ich habe dieses Kapitel mit einer Fülle verschiedener Themen vollgepackt, ohne dabei jeweils allzu tief ins Detail zu gehen. Dies hat womöglich dazu geführt, dass du jetzt eine Reihe von Fragen hast, die bisher nicht beantwortet wurden. Sollte dies der Fall sein, dann kann ich dich beruhigen, denn in den folgenden Tagebuchkapiteln (Kapitel 3 bis einschließlich Kapitel 9) werde ich auf die einzelnen Punkte nochmals im Detail eingehen, sodass im Laufe des Buches all deine Fragen bestimmt beantwortet werden. Zumindest hoffe ich das ;) Weitere Hintergrundinformationen zu diesem Kapitel und den Tagebuchkapiteln findest du in Kapitel 10.

Die Reihenfolge der Tagebuchkapitel ist nicht chronologisch geordnet, sondern fachlich begründet.

### Literatur

Die innerhalb dieses Kapitels referenzierten URLs findest du in klickbarer Form unter <http://www.trapkit.de/books/bhd/>. Sollte einer der Links nicht mehr funktionieren, dann lass es mich bitte wissen. Danke!

[AMINI 2009] Amini, P.: *Mostrame la guita! Adventures in buying vulnerabilities*, 2009, [http://docs.google.com/present/view?id=dcc6wpsd\\_20gbbpjxcr](http://docs.google.com/present/view?id=dcc6wpsd_20gbbpjxcr) (Stand: Januar 2010).

[DOUBLEFREE] [http://de.wikipedia.org/wiki/Doppelte\\_Deallokation](http://de.wikipedia.org/wiki/Doppelte_Deallokation) (Stand: Januar 2010).

[DOWD et al. 2007] Dowd, M.; McDonald, J.; Schuh, J.: *The Art of Software Security Assessment*, Addison-Wesley, 2007.

[EAGLE 2008] Eagle, C.: *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*, No Starch Press, 2008.

[EEYE] eEye Digital Security Research, <http://research.eeye.com/> (Stand: Januar 2010).

- [ERICKSON 2008] Erickson, J.: *Hacking – Die Kunst des Exploits*, dpunkt.verlag, 2008.
- [GOOG1] <http://www.google.de/search?q=eip+41414141>
- [GOOG2] <http://www.google.de/search?q=exploitation+techniques>
- [HODSON 2008] Hodson, D.: *Uninitialized Variables: Finding, Exploiting, Automating*, Ruxcon 2008, <http://www.ruxcon.org.au/files/2008/Uninitialized%20Variables%20-%20Live.ppt> (Stand: Januar 2010).
- [IDA] IDA Pro Disassembler, <http://www.hex-rays.com/idaipro/> (Stand: Januar 2010).
- [IDEF] iDefense Labs Vulnerability Contribution Program, <http://labs.idefense.com/> (Stand: Januar 2010).
- [INTEL 2008] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, November 2008, <http://www.intel.com/products/processor/manuals/> (Stand: Januar 2010).
- [KLEIN 2003] Klein, T.: *Buffer Overflows und Format-String-Schwachstellen – Funktionsweisen, Exploits und Gegenmaßnahmen*, dpunkt.verlag, 2003.
- [MILLER 2007] Miller, C.: *The Legitimate Vulnerability Market – Inside the Secretive World of 0-day Exploit Sales*, 2007, <http://weis2007.econinfosec.org/papers/29.pdf> (Stand: Januar 2010).
- [MS08-067] Microsoft Security Bulletin MS08-067, <http://www.microsoft.com/technet/security/Bulletin/MS08-067.msp> (Stand: Januar 2010).
- [OSTERHAGE 2009] Osterhage, J.: *Rechner von Computerwurm infiziert*, <http://www.tagesschau.de/inland/conficker102.html>, ARD Berlin, Tagesschau 15.02.2009 (Stand: Januar 2010).
- [SUTTON et al. 2007] Sutton, M.; Greene, A.; Amini, P.: *Fuzzing: Brute Force Vulnerability Discovery*, Addison-Wesley, 2007.
- [TIPP] TippingPoint Zero Day Initiative, <http://www.zerodayinitiative.com/> (Stand: Januar 2010).
- [XFORCE] IBM X-Force, <http://xforce.iss.net/> (Stand: Januar 2010).