Schwachstellenerkennung mittels Reverse Code Engineering

Seminarausarbeitung – finale Version vom 11. Juni 2013

Cornelius Diekmann

Fakultät für Informatik,
Technische Universität München
diekmann@in.tum.de

Zusammenfassung Diese Arbeit liefert einen Überblick über das Thema Schwachstellenerkennung mittels Reverse Code Engineering. Schwachstellenerkennung kann sowohl bei Open als auch bei Closed Source Software eingesetzt werden, um sicherheitskritische Fehler in der Anwendung oder deren Design aufzudecken. Es werden etablierte, sowie aktuelle Techniken aus der Forschung vorgestellt. Diese beinhalten Offline und Live Code Analyse, Traces, Slicing, Taint Analyse, Trace Alignment, Symbolic Execution, Abstract Interpretation, Binary Diff und Differential Slicing. Die Anwendung der Techniken in diversen Einsatzgebieten wird anhand vielfältiger Beispiele demonstriert. Es zeigt sich, dass die präsentierten Techniken in einigen generischen Anwendungsgebieten wie der Crashanalyse sehr hilfreich sein können, jedoch immer noch großes Wissen vom Reverse Engineer notwendig ist. Des Weiteren gibt es einige Vorstöße in spezielle Anwendungsgebiete wie der Erkennung sicherheitskritischer Felder in Konfigurationsdateien. Allerdings besteht im Allgemeinen noch Forschungs- und Entwicklungsbedarf bei der Tool- und Technikunterstützung der Schwachstellenerkennung mittels Reverse Code Engineerings.

1 Einleitung

Als Reverse Engineering wird der Prozess der Analyse eines Zielsystems bezeichnet, um die Komponenten und ihre Zusammenhänge zu identifizieren und eine Darstellung des Systems in einer anderen Form oder auf einem höheren Abstraktionslevel zu erzeugen[1]. Diese Arbeit beschäftigt damit, wie Reverse Engineering eingesetzt werden kann, um sicherheitskritische Schwachstellen zu erkennen. Dabei fokussiert sich diese Arbeit auf die folgenden Schwachstellen: Software Designfehler, unsichere Programmierpraktiken, Implementierungsfehler, Konfigurationsfehler und Durchsickern sicherheitskritischer Metainformationen.

Im folgenden Abschnitt 2 wird die Schwachstellenerkennung in Open Source Projekten vorgestellt. Dabei handelt es sich um einen einführenden Abschnitt, der dem Leser einen Einblick verschaffen soll, ohne Ihm die in Abschnitt 3 beschriebenen Probleme bei Closed Source Binaries als Steine in den Weg zu legen. Die folgenden Abschnitte beschäftigen sich danach primär mit Closed Source Binaries. Abschnitt 4 stellt grundlegende Werkzeuge und Techniken vor, die für den eigentlichen Prozess des Reverse Engineerings gebraucht werden. Anschließend präsentiert Abschnitt 5, wie diese Techniken eingesetzt werden können, um Schwachstellen zu identifizieren. Abschnitt 6 bewertet anhand der in dieser Arbeit vorgestellten Techniken den aktuellen Stand der Forschung. Anschließend fasst Abschnitt 7 die Ergebnisse kurz zusammen.

[1] Übersetzung des Autors, Original: "Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."[CC90]

2 Schwachstellenerkennung in Open Source Projekten

Wie aus der Definition bereits ersichtlich, beschäftigt sich Reverse Engineering keineswegs ausschließlich mit Closed Source Programmen in ihrer binären Form. Allerdings ist dies vermutlich das bekannteste Einsatzgebiet des Reverse Engineerings. In diesem einführendem Abschnitt wird aufgezeigt, wie sich Reverse Engineering auch in Open Source Projekten zur Schwachstellenerkennung einsetzen lässt.

2.1 Statische Programmanalyse

Dass sich durch statische Analyse des Quellcodes von Programmen kritische Fehler finden lassen, ist wohlbekannt [BBC+10]. Dass es sich dabei auch um direkte Schwachstellen handeln kann, zeigt sich beispielsweise an einer realen Studie am Linux Kernel [Cov04]. Spätestens bei Betrachtung der strengen Richtlinien mit denen Ergebnisse der statischen Code Analyse von Open Source Programmen an ausgewählte Maintainer weitergegeben werden[2], wird ersichtlich, dass diese Informationen extrem sicherheitsrelevant sind. Da statische Programmanalyse nur entfernt mit Reverse Engineering zusammenhängt, wird darauf an dieser Stelle nicht weiter eingegangen.

2.2 Bug Tracker und Code Repository Meta-Data

Für einen Reverse Engineer kann es interessant sein, die Entwicklungs-Repositories von Projekten zu beobachten. Mit Zugang zu Bug Trackern und Code Repositories, wie beispielsweise in Open Source Projekten, lassen sich versteckte sicherheitsrelevante Informationen wiederherstellen. Eine Informationsquelle sind sicherheitsrelevante Bug-Nummern, welche erst erkannt werden müssen. Bug Tracker-Einträge zu diesen Bug-Nummern sind beispielsweise nicht öffentlich zugänglich, wodurch sie leicht identifiziert werden können. Auch Patches für Schwachstellen in einem öffentlichen Repository sind eine wertvolle Information. Diese müssen allerdings auch als solche erkannt werden. Die Commit-Message kann Hinweise darauf geben. Bei Projekten, die ein dediziertes Sicherheitsteam besitzen, kann auch der Autor eines Patches ein Hinweis darauf sein, ob der Patch sicherheitsrelevant ist. Die Autoren in [BLRS11] benutzten folgende zwei Herangehensweisen für ihre Studie:

1. Patch Beschreibung enthält Bug-Nummer, Bug-Nummer nicht öffentlich zugänglich $\Rightarrow$ sicherheitsrelevanter Patch.
2. Ein Algorithmus zum maschinellen Lernen wird trainiert um sicherheitsrelevante Patches zu erkennen. Der Trainingsdatensatz kann automatisch erstellt werden, da eine Sicherheitslücke nachdem sie geschlossen wurde oft einen CVE Eintrag erhält und somit die sicherheitsrelevanten Patches automatisch identifiziert werden können.

In ihrer Studie beschäftigen sich die Autoren mit dem weitverbreiteten Webbrowser Firefox. Sie zeigen unter Anderem, dass in Firefox 3 alle 0.86 Tage ein sicherheitsrelevanter Patch durch die Beziehung zwischen Commit Messages und Bug-Nummer entdeckt werden konnte. Zum Zeitpunkt der Entdeckung wurden noch keine Informationen über die Sicherheitslücke veröffentlicht und ein Patch wurde noch nicht an die Endnutzer ausgerollt. Folglich waren alle so gefundenen Schwachstellen in der offiziellen Endnutzerversion ausnutzbar.

[2] Coverity Scan Initiative für Open Source Projekte: `http://scan.coverity.com/`

3 Probleme bei Closed Source Binaries

Im Gegensatz zu Open Source Programmen, ist das Reverse Engineering von Closed Source Binaries im Allgemeinen weitaus komplexer. Dies liegt daran, dass der Reverse Engineer sich meistens mit Maschinencode auseinandersetzen muss. Im Gegensatz zu verbreiteten Programmiersprachen ist Maschinencode nicht für Menschen geschrieben. Wenn ein optimierender Compiler zur Erzeugung der Binary verwendet wurde, kann der entstehende Maschinencode sehr unintuitiv werden und sich weit vom originalen Programmcode distanzieren. Zusätzlich kann das zu untersuchende Programm noch passiv und aktiv versuchen das Reverse Engineering zu erschweren. Als passive Erschwernis kann beispielsweise die Verwendung von Packern, dynamische Code Erzeugung zur Laufzeit, Nachladen von Funktionalität zur Laufzeit oder allgemeine Code Verschleierung gezählt werden. Als aktive Erschwernis zählt unter Anderem das Erkennen von Debuggern. Weitere Herausforderungen sind die Unmengen von Daten, die beim Reverse Engineering anfallen können. Potenziell gefährliche Programme (Malware) dürfen einerseits bei der Analyse keinen Schaden anrichten, andererseits sollen sie auch korrekt ausgeführt werden.

Ein weiteres Problem bei der Schwachstellenerkennung ist, zu unterschieden, wann es sich um einen normalen Bug und wann um eine echte sicherheitskritische Schwachstelle handelt. Wie in Abbildung 1 bereits zu sehen ist, ist nicht jeder Crash automatisch exploitable. Bei genauerer Betrachtung der Abbildung ist zu erkennen, dass die Bezeichnung "Other cases require data flow analysis" sehr generisch ist. Dies verrät schon, dass es keine allgemeine Möglichkeit gibt zu entschieden, ob es sich nur um einen Bug oder um eine echte Schwachstelle handelt. Hier wird die Aufgabe des Reverse Engineers deutlich: Er muss sich einen Crash sehr genau anschauen und individuell für jeden Crash entscheiden, ob es sich um eine potenzielle Sicherheitslücke handelt. Der reverse Engineer muss herausfinden, zu welchem Ausmaß ein Angreifer den Crash kontrolliert und ob der Angreifer genug Kontrollmöglichkeiten hat, einen Crash in eine ausnutzbare Schwachstelle zu verwandeln. Diese Entscheidung beschränkt sich nicht nur auf die Stelle des Crashes, sondern kann über das gesamte Programm verteilt sein.

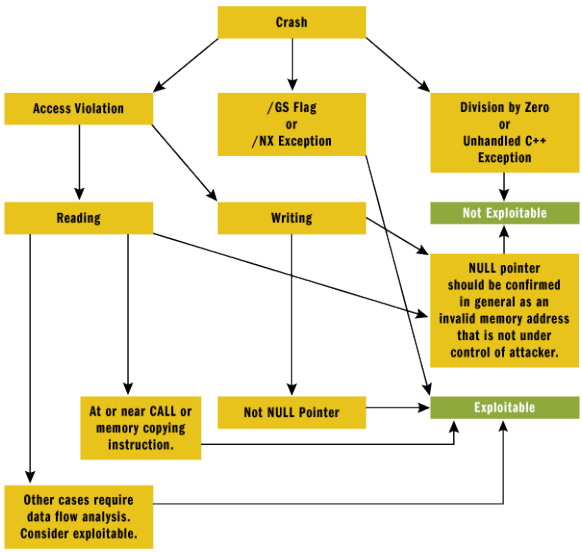

Abbildung 1. Access Violation Analysis Path [AHLW, Figure 1]

4 Techniken und Werkzeuge für Closed Source Binaries

Dieser Abschnitt stellt hilfreiche Werkzeuge für die Schwachstellenerkennung durch Reverse Engineering vor. Bei allen aufgelisteten Werkzeugen und Konzepten wurde

darauf geachtet, dass sie auf Binärprogrammen ohne zusätzliche Debuggingsymbole funktionieren und dass Implementierungen existieren. Die Abschnitte 4.1 und 4.2 präsentieren die grundlegenden Werkzeuge eines Reverse Engineers. In den Abschnitten 4.3, 4.4 und 4.5 werden fortschrittliche Techniken der Analyse vorgestellt. In den darauf folgenden Abschnitten 4.6, 4.7, 4.8 und 4.9 werden aktuelle Techniken aus der Forschung vorgestellt. Abschließend präsentiert Abschnitt 4.10 kurz, wie sich auch Visualisierung als nützliches Tool verwenden lassen kann und Abschnitt 4.11 gibt einen kurzen Einblick in die verwandte Disziplin des Fuzzings.

4.1 Offline Code Analyse

Offline Code Analyse – auch bekannt als Dead-Listing [Eil05, Chapter 4, S. 110] oder static analysis – bezeichnet den Prozess bei dem Versucht wird, die Semantik eines Programms zurückzugewinnen, ohne dieses auszuführen. Üblicherweise wird versucht, ein Programm zu disassemblieren, um eine menschenlesbare Repräsentation des Maschinencodes zu erhalten. Einfache Disassembler wie `objdump` oder die umfangreiche IDA [Gui] Suite erleichtern diesen Prozess erheblich. Die Entwicklerdokumentation für die entsprechende Plattform ist jedoch teilweise auch nötig [Int].

Vor jeder Analyse ist es hilfreich, sich den generellen Aufbau des zu untersuchenden Programms anzuschauen. Die vorhandenen Sektionen und importierten Symbole können bereits einige Hinweise über die Funktion der Binary enthalten. Für das Executable and Linkable Format (ELF) eignen sich Programme wie `objdump`, `ldd` und `readelf`, sowie weitere Programme aus der GNU Binutils Sammlung [GNU]. Für das Portable Executable Format (PE) – welches für Windows Systeme bevorzugt wird – eignet sich hierzu das Programm `DUMPBIN` [dum] welches in Microsoft Visual C++ enthalten ist.

Spezielle Reverse Engineering Tools wie BinNavi [zynb] oder generische Frameworks wie Vine aus dem BitBlaze Projekt [SBY+08,Bit,MJ10] bieten zudem die Möglichkeit, nach dem Disassemblieren weitere Schritte durchzuführen. Dies beinhaltet beispielsweise die Erstellung von Kontrollflussgraphen, Slicing (siehe Abschnitt 4.4), Integer Intervall Analysen und Weiteres. Auch Optimierungsmöglichkeiten sind vorhanden, welche in erster Linie aus Performance Gründen verwendet werden. Allerdings zeigte Spasojevic in [Spa10], dass durch Optimierung auch automatische Code Entschleierung möglich ist.

4.2 Live Code Analyse

Live Code Analyse – auch bekannt als dynamische Analyse – bezeichnet den Prozess, bei dem das zu untersuchende Programm ausgeführt und beobachtet wird, um die Semantik des Programms zurückzugewinnen. Ein klassisches Werkzeuge hierfür ist ein Debugger mit eingebauten Disassembler wie gdb [GP], OllyDbg [Yus], IDA [Gui] oder WinDbg [Mica]. Letztere zwei sind besonders mächtig wenn es darum geht, die Interaktion eines Programms mit dem Windows Betriebssystem zu untersuchen; Es ist ihnen möglich, automatisch entsprechende Debug Symbole (PDB Dateien) vom Microsoft-Symbolserver zu beziehen.

Falls auch auf Kernel-Ebene debugged werden soll, um ein genaueres Bild des ganzen Systems zu erhalten, eignen sich Numega's SoftICE [Eil05, Chap. 4] oder WinDbg. Ersteres unterstützt nur Windows Systeme bis XP und wird seit 2006 nicht mehr offiziell unterstützt [sof12, 2.1 Termination]. WinDbg wird offiziell von Microsoft unterstützt. Allerdings sind zwei Computer notwendig, um die vollen Kernel-Mode Fähigkeiten von WinDbg zu nutzen. Dank den Fortschritten in der Virtualisierung ist dies jedoch heutzutage ein kleines Hindernis.

Des Weiteren lassen sich auch Monitoring Programme wie Process Explorer [Rus] oder Analyse-Sandboxen wie cwsandbox [WHF07] zu den Werkzeugen der dynamischen Analyse zählen. Da sie jedoch für die Schwachstellenerkennung von geringerer Bedeutung sind, wird auf sie an dieser Stelle nicht weiter eingegangen.

Generische Debugging Frameworks wie Valgrind [NS07] ließen sich auch für die Live Code Analyse verwenden. So könnten beispielsweise von Valgrind aufgezeigte Speicherfehler ein Hinweis für eine mögliche Schwachstelle sein.

Ein dediziertes generisches Framework für die Schwachstellenerkennung ist TEMU aus dem BitBlaze Projekt [SBY+08,Bit,MJ10]. TEMU kann ein komplettes System in einer virtuellen Umgebung für die Analyse laufen lassen. Dafür greift es auf QEMU [Bel05] zurück. Dies ermöglicht TEMU sehr feingranular auf Maschinencode Ebene zu arbeiten und gleichzeitig eine uneingeschränkte Sicht auf das gesamte System bereitzustellen. Um TEMU eine Sicht auf das virtualisierte System zu ermöglichen, muss TEMU in der Lage sein die Semantik des verwendeten Betriebssystems zu verstehen. Bei Linux Gastsystemen extrahiert TEMU direkt von Außen die notwendigen Informationen. Diese Daten erhält TEMU indem es die Adressen der globalen statischen Symbole des Kernels aus der `System.map` extrahiert. Ein automatisiertes, flexibleres und dynamischeres Vorgehen wie in [SPE11] beschrieben wäre denkbar. Bei Windows Gastsystemen greift TEMU auf ein speziell entwickeltes Kernel Modul zurück, welches die nötigen Informationen extrahiert.

4.3 Traces

Die aufgezeichnete Ausführung eines Programmes wird Trace genannt. Unterschiedliche Tracing Varianten existieren, welche sich unter Anderem in ihrem Tracing Umfang unterscheiden [NOA09]. Es existieren grobgranulare Tracer welche beispielsweise auf System Call Ebene arbeiten. Am anderen Ende des Spektrums existieren sehr feingranulare Tracer welche die gesamte Ausführung eines Programms auf Maschinenebene aufzeichnen können.

Ein Vorteil der Betrachtung von feingranularen Traces anstelle von Programm Binaries ist, dass der komplette tatsächlich ausgeführte Code sichtbar ist. Dies kann beispielsweise sehr nützlich bei Programmen sein, bei denen Code nur verschlüsselt vorliegt und zur Laufzeit dynamisch entpackt wird. Des Weiteren kann so auch dynamisch nachgeladener Code aus dem Netz betrachtet werden. Wenn ein ausreichend genauer Trace vorliegt, lassen sich auch die Methoden der offline Code Analyse auf diesen Trace anwenden. Eine Möglichkeit um solche Traces zu erstellen ist das Tracecap Plugin aus dem BitBlaze Framework [MJ10].

4.4 Slicing

Slicing wurde ursprünglich als Abstraktionswerkzeug für die Programmwartung entwickelt [Wei81]. Der Hintergrundgedanke ist, sich nur auf die Codebereiche zu beschränken, die eine bestimmte Menge an Variablen beeinflussen. Als Slice wird ein ausführbares Programm bezeichnet, dessen Verhalten identisch zu der spezifizierten Teilmenge des Verhaltens des originalen Programms ist[3]. Aus einem großen Originalprogramm wird somit ein kleineres Programm erzeugt, welches die gleiche Semantik im Bezug auf gewisse Variablen aufweist. Slicing im Allgemeinen und insbesondere den kleinsten Slice zu finden sind unentscheidbare Probleme. Jedoch ist Slicing ein mächtiges Werkzeug um Komplexität zu verringern. Ein gültiger (i.A.

[3] Freie Übersetzung des Autors, Original: "A slice is itself an executable program, whose behavior must be identical to the specified subset of the original program's behavior." [Wei81]

nicht minimaler) Slice kann gefunden werden, indem sukzessiv Anweisungen aus dem Originalprogramm gelöscht werden, welche die zu betrachtenden Variablen nicht beeinflussen.

Ursprünglich wurde Slicing für die statische Analyse von Quellcode entwickelt. Slicing angewendet auf der Ausführung eines Programms (Trace) wird dynamisches Slicing genannt [KL88]. Somit enthält ein dynamischer Slice alle Instruktionen des Traces, welche bis zu einem gegebenen Zeitpunkt den Wert einer Menge von Zielvariablen beeinflusst haben. Da sich die Analyse nur auf tatsächlich ausgeführten Code beschränkt, kann der resultierende Slice so signifikant kleiner werden [KL88]. Ein dynamischer Slice kann iterativ berechnet werden: Es wird die Menge an Variablen bestimmt, welche die festgelegten Zielvariablen direkt beeinflussen. Diese sind einerseits an direkten Zuweisungen oder Berechnungen beteiligte Variablen und andererseits Variablen, die die festgelegten Zielvariablen indirekt beeinflussen. Beispielsweise kontrollflussändernde Tests. In der nächsten Iteration werden nun die Variablen bestimmt, welche die in der letzten Iteration gefundenen Variablen beeinflussen. Dieser Prozess wird wiederholt, bis keine neuen Variablen gefunden werden. Ein gültiger Slice sind mindestens alle Instruktionen, welche Einfluss auf mindestens eine der gefundenen Variablen nehmen. Zusätzliche Instruktionen, wie beispielsweise Schleifenbedingungen, können optional hinzugefügt werden, um die Struktur des Slices zu erhalten und damit die Lesbarkeit zu erhöhen.

Für das BitBlaze Framework existiert ein dynamischer Slicer für aufgezeichnete Traces, genannt x86_slice. Die Ausgabe ist ein reduzierter Trace, welcher sich auf die Anweisungen beschränkt, die einen vorgegeben Wert beeinflussen [MJ10].

4.5 Taint Analyse

Unter einer dynamischen Taint Analyse versteht man das Markieren bestimmter Daten und deren Verfolgung zur Laufzeit [CLO07]. Die markierten Daten werden als Taint-Quelle bezeichnet. Diese könnte beispielsweise eine globale Speicheradresse, Tastatureingabe oder die Rückgabe einer Betriebssystemfunktion sein. Im Gegensatz zum Slicing werden also die Auswirkungen einer Taint-Quelle auf das System betrachtet, während beim Slicing die Auswirkungen von Programmteilen auf eine Zielvariable untersucht wird.

Bei der Verbreitung der Taint-Information eines Wertes unterscheidet man zwischen explizitem und impliziten Informationsfluss [CLO07]. Bei Ersterem ist das markierte Datum direkt an der Berechnung eines neuen Wertes beteiligt und die Markierung wird weitergeben. Bei Letzteren beeinflusst der Wert eines markierten Datums den Wert eines weiteren Datums indirekt. Die Weitergabe von Markierungen bei explizitem Informationsfluss wird datenflussbasiert auf Maschinencodeebene berechnet. Dafür werden für jede ausgeführte Maschinenanweisung die Quell- und Zieloperanden ermittelt. Die Markierungen des Quelloperanden werden an die Zieloperanden propagiert. Eine wichtige Optimierung hier ist die Erkennung konstanter Funktionen wie beispielsweise `XOR EAX, EAX`, welche das `EAX` Register auf 0 setzt. Hier sollten die bestehenden Markierungen von `EAX` nicht weitergegeben werden, da `EAX` immer den konstanten Wert 0 haben wird, unabhängig von der vorherigen Belegung. Der Bezug zur ursprünglich definierten Taint-Quelle ist aufgehoben. Eine solche Optimierung kann das resultierende Ergebnis der Taint Analyse erheblich verkleinern und somit die Genauigkeit erhöhen. Indirekter Informationsfluss kann mit Hilfe des Kontrollflussgraphen berechnet werden. Auf Maschinencodeebene werden dazu die Markierungen aller ausgeführten kontrollflussändernden Instruktion an alle Instruktionen des neuen Kontrollflusszweiges weitergegeben. Das Ende eines Kontrollflusszweiges kann anhand der Post-Dominatoren bestimmt werden, welche aus dem Kontrollflussgraphen berechnet werden können.

Ein generisches Framework für die dynamische Tanit Analyse ist Dytan [CLO07]. Auch TEMU aus dem BitBlaze Framework [SBY+08,Bit] unterstützt dynamische ganzheitliche Taint Analyse. Dies beinhaltet den Taint-Status jeder Speicherzelle des physischen Speichers und der Festplatte, aller Register, sowie dem Buffer der Netzwerkkarte [SBY+08]. Es lässt sich somit beobachten, wie sich Daten aus einer Taint-Quelle im gesamten System verteilen.

4.6 Normalisierung

Unter Normalisierung im Sinne des Reverse Code Engineerings versteht man den Prozess, ein gegebenes (meist binäres) Objekt in eine einheitliche Darstellung zu bringen. Ziel der Normalisierung ist es, eine einfachere Darstellung für die weitere Verarbeitung zu erhalten.

Vine unterstützt beispielsweise die x86 und ARMv4 Befehlssatzarchitektur. Dies wird erreicht, indem Vine mit einem normalisierten abstrakten Assembler – genannt Vine IL [SBY+08,Bit,MJ10] – arbeitet. x86 beziehungsweise ARMv4 Assembler wird nach Vine IL übersetzt bevor der Maschinencode weiterverarbeitet wird. Der Vorteil von Vine IL besteht darin, dass die Sprache sehr kompakt und genau definiert ist. Vine IL ist funktional gehalten. So wird beispielsweise dem `store` Befehl ein kompletter Speicher übergeben und ein kompletter Speicher zurückgegeben. So lassen sich Änderungen am Speicher gut nachvollziehen. Um dies effizient zu realisieren, lässt sich der Speicher beispielsweise als Liste oder Lambda Ausdruck [NK12, state] realisieren. Zusätzlich ist Speicher in VINE IL typisiert. Das heißt, Typfehler wie beispielsweise durch Endianness werden vermieden. Da Zugriffe auf Speicherzellen der Größe von 8 Bit atomar sind, lassen sich dennoch Konvertierungen durchführen.

Um auch Traces mehrerer Programmdurchläufe vergleichen zu können, kann eine Normalisierung des Speichers hilfreich sein. Dies ist insbesondere wichtig, wenn Maßnahmen wie Address Space Layout Randomization (ASLR) [BDS03] dafür sorgen, dass die gleichen Objekte an unterschiedlichen Speicherstellen in verschiedenen Programmdurchläufen liegen. Daher ist es wichtig, Pointer zu normalisieren um eine Gleichheitsaussage zu treffen. Dafür unterscheidet man drei Arten von Pointern, in Abhängigkeit von dem Bereich auf den sie verweisen: *Heap Pointer*, *Stack Pointer* und *Data Section Pointer*. Zwei Heap Pointer in zwei Traces sind gleich, wenn sie in jedem Trace an der gleichen Trace-Stelle alloziert wurden. Mit Bezug auf Abschnitt 4.8 sagt man, die Speicherallokationen sind aligned. Zwei Stack Pointer sind gleich, wenn sie den gleichen Offset vom Beginn des Stacks haben. Zwei Data Section Pointer sind gleich, wenn sie den gleichen Offset vom Beginn des Moduls haben. Im Adressbereich eines Programms können sich mehrere Module (beispielsweise DLLs) befinden. Daher muss zuerst festgestellt werden, auf welches Modul ein Pointer sich bezieht. Die hier vorgestellten Pointer-Normalisierungen werden in [JCC+11] verwendet.

4.7 Symbolic Execution und Abstract Interpretation

Eine symbolische Ausführung eines Programms ist ein theoretisches Konzept, in dem ein Programm mit symbolischen Werten (z.B. "x") anstelle von konkreten Werten (z.B. "5") ausgeführt wird [Kin76]. Als abstrakte Interpretation bezeichnet man eine Berechnung, welche auf abstrakten Objekten (z.B. Intervalle) anstelle von konkreten Werten (z.B. Zahlen) durchgeführt wird [CC77].

Rudder aus dem BitBlaze Projekt [SBY+08,Bit,MJ10] nutzt einige dieser Konzepte. Der Nutzer kann gewisse Eingaben eines Programms symbolisch spezifizieren. Rudder exploriert dann alle Ausführungspfade des Programms in Abhängigkeit der symbolischen Eingabe. Um mehrere alternative Pfade zu explorieren bedient sich

Rudder an TEMU's Snapshot Funktion, welche aus QEMU geerbt wurde. Als weiteres Werkzeug stellt BitBlaze das Tool `valset_ir` bereit. Mit Hilfe von Taint Informationen ist es damit möglich abzuschätzen, welcher Bereich eines Wertes von einem anderen abhängt [MJ10]. Somit lässt sich beispielsweise die Frage beantworten, welcher Wertebereich eines Datums unter der Kontrolle eines möglichen Angreifers stehen.

4.8 Trace Alignment

Trace Alignment ist eine Technik, mit der es möglich ist, korrespondierende Instruktionen in zwei Traces zu identifizieren. Dabei müssen die Instruktionen zwar auf den gleichen Variablen arbeiten, jedoch können die Variablen unterschiedliche Werte haben. Da jedoch Instruktionen in einem Trace mehrfach auftreten können, muss ein semantischer Zusammenhang zwischen den zwei Traces gefunden werden. Diesen Prozess nennt man Trace Alignment [JCC+11]. Man sagt, zwei gleiche Instruktionen aus zwei Traces, die einander semantisch entsprechen, sind *aligned*. Ein Beispiel wäre eine While-Schleife im Quellcode des zu untersuchenden Programms. Es werden zwei Traces A und B erzeugt. Die Aufgabe des Trace Alignment ist es nun, die Regionen der beiden Traces zu identifizieren, die die Ausführung der While-Schleife bilden und sich daher entsprechen. Diese korrespondierenden Instruktionen sind demnach aligned. Die While-Schleife kann in jedem Trace auf unterschiedlichen Daten arbeiten, jedoch darf sich der Kontrollfluss nicht unterscheiden. Angenommen in Trace A wird die Schleife 8 mal und in Trace B nur 5 mal durchlaufen. Die drei zusätzlichen Durchläufe in A finden keine Entsprechung in B und sind daher nicht aligned.

Trace Alignment Informationen werden in zwei Schritten berechnet. Dies lässt sich wie folgt skizzieren: Zuerst wird der Kontrollflussgraph des Programms erstellt. Anhand dieses Graphs werden die Knoten ermittelt, an denen Verzweigungen wieder zusammenlaufen. Diese Punkte werden in der Graphentheorie auch Post-Dominatoren genannt. Im zweiten Schritt werden die Instruktionen mit Hilfe eines Stacks aligned. Dazu werden die Traces parallel traversiert, wobei beide Traces mit einer aligned Instruktion anfangen müssen. Bei einer Kontrollflussverzweigung wird ein für die aktuelle Instruktion eindeutiger Index auf den Stack gepushed. Wenn der Post-Dominator dieses Indexes erreicht wird, wird er vom Stack gepopped. Intuitiv heißt das, wenn die zwei Traces einen unterschiedlichen Ausführungszweig nehmen, können sie frühestens im Post-Dominator wieder zusammenlaufen [JCC+11]. Wenn die beiden Traces parallel die gleichen Instruktionen im selben Zweig ausführen, sind sie aligned.

4.9 Differential Slicing

Beim Differential Slicing [JCC+11] ist das Ziel, zwei Programmläufe miteinander zu vergleichen. Dabei liegt der Fokus auf Unterschieden in den Traces, die durch Unterschiede in der Programmumgebung hervorgerufen wurden. Als Programmumgebung werden unter Anderem die Parameter eines Programms oder Dateien, die das Programm verarbeitet gezählt. Der resultierende *causal difference graph* enthält nur die Teile des Traces, die sich bedingt durch die unterschiedliche Umgebung unterscheiden. Somit lässt sich die Frage beantworten, *warum* sich zwei Programmabläufe unterschieden. Meistens ist nur ein Unterschied wie ein Crash relevant, jedoch unterscheiden sich oftmals viele unabhängige Abschnitte zweier Traces. Daher ist es wünschenswert, den resultierenden causal difference graph weiter zu verkleinern. Deshalb beschränkt sich das Differential Slicing auf eine sogenannte Zieldifferenz (beispielsweise ein Crash). Das Differential Slicing kann bei Angabe zweier Traces und einer Zieldifferenz mit Hilfe von Trace Alignment, Slicing und Normalisierung automatisiert werden [JCC+11].

4.10 Binary Diff und Visualisierung

Um den Unmengen von Daten welche beim Reverse Engineering anfallen können entgegenzuwirken, kann Visualisierung ein hilfreiches und intuitives Vorgehen sein. Ein klassisches Beispiel um den Unterschied zwischen zwei BLOBs[4] zu visualisieren ist der Binary Diff. Auch für dediziertere Aufgaben gibt es spezielle Binary Diff Toolunterstützung. So kann DarunGrim [OLG] beispielsweise direkt Unterschiede zwischen zwei Programmen oder einem Programm und einem Patch aufzeigen. Die Unterschiede werden graphisch im Kontrollflussgraphen des Programms aufgezeigt. Spezielle Metriken erlauben es, automatisch sicherheitsrelevante Änderungen gesondert hervorzuheben. Auch zynamics BinDiff [zyna] ist darauf spezialisiert Binärdateien über graphentheoretische Ansätze zu vergleichen, um so die Schwachstellenanalyse zu unterstützen. Auch spezielle Visualisierungen, welche Änderungen am Speicher eines aktiv laufenden Programms erzeugen, wurden entwickelt [BLB10]. Eine weitere generische Möglichkeit den Speicher eines Prozesses graphisch aufzuarbeiten, stellt das Programm VMMap [RC11] dar. Es visualisiert den virtuellen Speicher eines Prozesses aus Betriebssystem-Sicht und stellt unter Anderem Memory Mappings und threadlokale Bereiche übersichtlich dar. Um die Verbreitung von Taint-Informationen besser zu verfolgen, gibt es auch einen Taint-Graph als IDA Pro Erweiterung [MJ10].

4.11 Fuzzing

Unter Fuzzing versteht man eine Software Testmethodik. Dabei wird ein Programm auf Robustheit getestet, indem ihm viele Eingabedaten zum Verarbeiten gegeben werden. Ziel ist es, Fehler im Programm zu finden. Der am leichtesten zu erkennende Fehler in einem Programm ist der klassische Absturz. Die Eingabedaten können entweder zufällig erzeugt werden, oder gezielt mit Wissen über das zu untersuchende Programm. So ist es beispielsweise vorstellbar, gezielt Längenangaben zu verfälschen oder zyklische Baumstrukturen in Dokumenten zu verstecken. Für das Fuzzing an sich gibt es viele fertige Fuzzer für spezielle Domänen und unter Anderem auch ein generisches Framework um Fuzzer zu bauen [Nag].

Fuzzing selbst ist keine Reverse Engineering Technik, jedoch ist ein mit Fuzzing gefundener Absturz eines Programms oftmals der Ausgangspunkt einer Reverse Engineering Session.

5 Anwendung in der Schwachstellenerkennung

Dieser Abschnitt beschäftigt sich nun mit der Anwendung der vorgestellten Techniken. Das Einsatzgebiet der Schwachstellenerkennung wird dabei in den Vordergrund gestellt. Da dieses Gebiet sehr weitläufig ist, gliedert sich der folgende Abschnitt in spezialisierte Unterabschnitte. Die Abschnitte enthalten reale und aktuelle Anwendungsbeispiele. Zur besseren Übersicht erhält jedes Beispiel seinen eigenen Absatz. Im folgenden Abschnitt 5.1 wird gezeigt, wie sich die vorgestellten Techniken zur Evaluierung der Sicherheit eines Programms nutzen lassen. Abschnitt 5.2 geht einen Schritt weiter und zeigt, wie man sicherheitsrelevante High-Level-Designartefakte zurückgewinnen kann. Abschnitt 5.3 zeigt anschließend, wie sich Fehler im Design ohne vorliegende Designdokumente aufdecken lassen. Abschnitt 5.4 demonstriert, wie sich sicherheitsrelevante Konfigurationsdateifelder erkennen lassen. Abschnitt 5.5 beschäftigt sich mit der Analyse von bekannten Schwachstellen und Abschnitt 5.6 beschäftigt sich schließlich mit der Analyse unbekannter Schwachstellen.

[4] Binary Large Object

5.1 Evaluierung und Auditing

Oftmals ist der Quellcode von kommerzieller Drittanbieter Software nicht verfügbar. Dennoch muss evaluiert werden, ob diese Software gewisse Sicherheitsstandards einhält und somit verwendet werden kann. Dies kann sich sowohl auf zugekaufte Bibliotheken beziehen, als auch auf komplette Softwarepakete.

Eilam beschreibt in [Eil05, Chapter 6] beispielhaft, wie durch manuelle offline Code Analyse die interne Arbeitsweise einer Verschlüsselungssoftware rekonstruiert werden kann. Mit diesem Wissen kann eingeschätzt werden, ob die Software den Sicherheitsansprüchen für den geplanten Verwendungszweck genügt. Des Weiteren kann die allgemeine Qualität der Software abgeschätzt werden.

Durch die offline Code Analyse lässt sich zusätzlich ein allgemeines Bild über das vorliegende Produkt gewinnen. Die Suche nach Funktionen wie `strcpy, wcscpy, _mbscpy` welche von Microsoft als gefährlich eingestuft werden [stra], oder deren sichere Alternative `StringCchCopy` [strb] geben grundsätzliche Hinweise über die Qualität der vorliegenden Software. Auch die Verwendung von Stack Cookies [sta,Bra02] kann weitere Hinweise enthalten. Im Hinblick auf Verschlüsselungssoftware kann überprüft werden, ob Salts und zufällige Initialisierungsvektoren verwendet werden.

Ein konkretes Beispiel wird in [Oec09] beschrieben. Die Software EISST E-capsule PrivateSafe bewarb, dass sie unter Anderem zwei Passwörter unterstützt: Ein privates Passwort welches die zu schützenden Daten entschlüsselt und ein Panik Passwort, welches lediglich einen leeren Ordner anzeigt. Oechslin zeigte einen schweren Implementierungsfehler in diesem Produkt auf; Durch Vertauschen der Einträge im Steuerungsblock der verschlüsselten Datei ist es möglich, die privaten Daten mit dem Panik Passwort zu entschlüsseln.

Letztendlich ist es allerdings nicht der Quellcode, der auf der CPU ausgeführt wird, sondern Maschinencode. Trotz korrektem und umfangreich getestetem Quellcode, kann ein Compilerfehler schwerwiegende Sicherheitslücken in das kompilierte Produkt einbringen. Solche Fehler können nur durch Analyse des Maschinencodes entdeckt werden. So wurde beispielsweise der GCC C Compiler beschuldigt, Tests auf Pointer Overflows wegzuoptimieren [Cor08]. Tests bewiesen, dass so tatsächlich in vermeintlich sicherem Code, Buffer Overflows entstanden. Das Problem lag auch bei anderen Compilern vor. Hierbei handelt es sich zwar um unerwartetes Verhalten, allerdings ist dieses Compilerverhalten nach C-Standard valide, da der C-Standard definiert, dass Pointer keinen Overflow haben [Cor08]. Das Beispiel zeigt die Diskrepanz zwischen Quellcode und dem kompiliertem Programm und demonstriert daher, die Notwendigkeit des Reverse Engineerings auf Maschinencodeebene.

5.2 Design Wiederherstellung

Bei der Design Wiederherstellung werden Designabstraktionen aus einer Kombination von Code, bestehenden Designdokumenten (falls verfügbar), persönlicher Erfahrung und generellem domänenspezifischen Wissen nachgebildet[5]. Insbesondere bei sicherheitskritischen Anwendungen muss überprüft werden, ob das real ausgelieferte Produkt mit dem beschriebenen Design übereinstimmt. Häufig werden Änderungen

[5] Übersetzung des Autors, Original: "Designrecovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains" [Big89]

nicht in die Dokumentation übernommen, insbesondere nicht auf einem höhere Level als im Quellcode selbst[6].

So gelang es Oechslin [Oec09] beispielsweise zu zeigen, dass ein nach FIPS 142-3 Level 2 [fip02] zertifizierter USB Crypto-Key nicht dem beschriebenen Design entsprach und Lücken aufwies. Dies geschah primär durch Live Code Analyse. Der Fehler wurde schnell behoben.

Ein umfangreiches Beispiel, welches sich weiterer Techniken bedient, ist das Folgende. Die TCP Zustandstransitionen werden in RFC 793 [Pos81] definiert und sind in Abbildung 2 dargestellt. Mit Hilfe von Slicing auf Quellcodeebene stellten Guha und Mukherjee in [GM97] die realen TCP Zustandsübergangsdiagramme wieder her, wie sie in gängigen Betriebssystemen implementiert wurden. Unter Anderem konnten die Autoren eine Transition von SYN_RCVD zu CLOSE_WAIT identifizieren. Diese direkte Transition – welche nicht im RFC existiert – lässt sich für eine Denial of Service Attacke missbrauchen. Die genannte Transition lässt sich durch ein TCP Paket, mit sowohl gesetztem SYN, als auch gesetztem FIN Flag, provozieren. Das angegriffene System wartet nun auf einen Timeout, bevor es die Verbindung aus dem Speicher entfernen kann. Ähnlich wie beim SYN-Flooding lassen sich die Ressourcen des Systems so erschöpfen. Da die Timeout Werte im CLOSE_WAIT Zustand sehr groß sind, ist dieser Angriff sehr effizient.

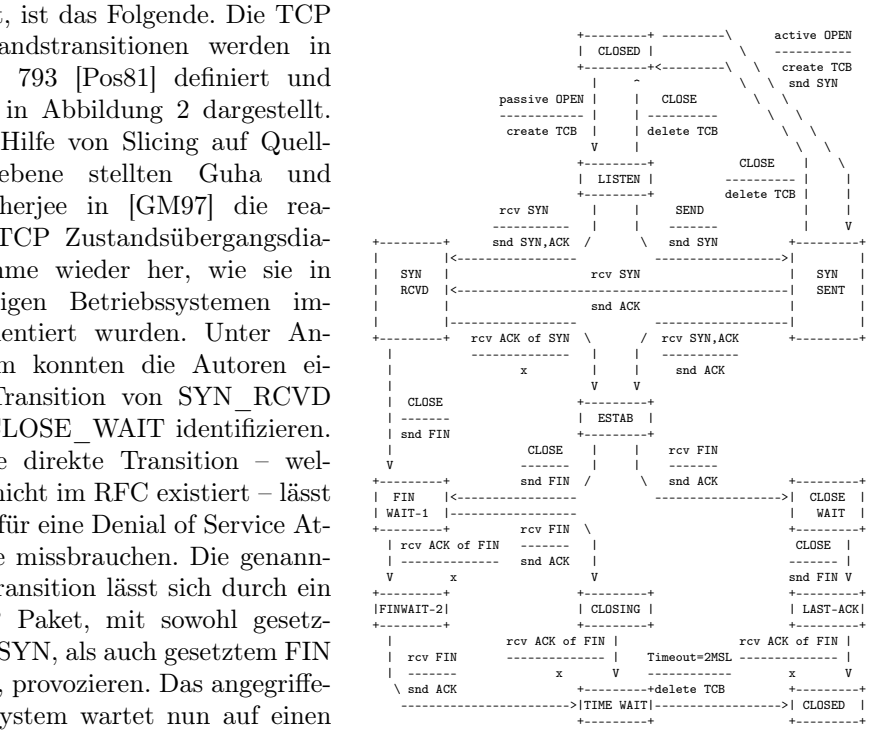

Abbildung 2. TCP Zustandsübergangsdiagramm [Pos81, Abb. 6]

5.3 Designfehlererkennung

Durch das Reverse Engineering können Designfehler in Software aufgezeigt werden. Dies unterscheidet sich zu der Design Wiederherstellung darin, dass keine Designdokumente vorliegen, aber trotzdem ein grundlegender Designfehler gesucht wird.

So zeigen Bursztein et al. in [BLB10] wie sich im Speicher von Videospielen die Spielkarte auffinden lässt. Die Herangehensweise lässt sich dabei wie folgt skizzieren:

1. Erstelle Memorydump
2. Spiele das Videospiel, erkunde dabei nicht die Karte

[6] Übersetzung des Autors, Original: "Modifications are frequently not reflected in documentation, particularly at a higher level than the code itself." [CC90]

3. Erstelle Memorydump
4. Vergleiche die zwei Memorydumps, betrachte nur noch Speicherbereiche die sich nicht geändert haben
5. Spiele das Videospiel, erkunde ausschließlich die Karte
6. Erstelle Memorydump
7. Isoliere die Speicherbereiche die Sich verändert haben
8. Erkenne Spielkarten-Datenstruktur durch Visualisierung

Die aufgezeigte Möglichkeit, die Spielkarte in einigen aktuellen Videospielen zu finden, zeigt einen groben Designfehler in Online Spielen auf: Durch Veränderungen an der Karte ist es möglich, im Spiel zu Betrügen. Es wäre möglich, die Karte komplett aufzudecken, obwohl diese Information dem Spieler nicht zugänglich sein sollte. Der Designfehler hier besteht darin, dass der Onlinespielclientanwendung nur die Informationen über die Karte zugänglich sein sollten, die der Spieler bereits erkundet hat.

5.4 Konfigurationsdateien

Wang et al. präsentieren in [WWZL08] eine automatisierte Möglichkeit, um sicherheitsrelevante Optionen aus Konfigurationsdateien zu identifizieren. Dabei beschränken sie sich auf Optionen in Konfigurationsdateien, welche für die Zugriffskontrolle relevant sind. Als Ergebnis wird die Sprache der Konfigurationsdateien als kontextfreie Sprache in Backus-Naur-Form [BBG$^+$63] präsentiert. Insbesondere werden Werte für Felder, wie beispielsweise *allow* oder *deny*, automatisch identifiziert. Dafür definieren die Autoren zwei Taint-Quellen. Zum Einen werden die Felder der Konfigurationsdatei als Taint-Quelle festgelegt, zum Anderen wird die Benutzeranfrage, wie beispielsweise ein HTTP GET, als zweite Taint-Quelle gesetzt. Wenn sich Daten aus den beiden Taint-Quellen treffen, wird mit hoher Wahrscheinlichkeit gerade eine Benutzeranfrage im Kontext der Konfigurationsdatei bearbeitet. Ein Feld in der Konfigurationsdatei ist relevant für die Zugriffskontrolle, wenn dem Feld ein anderer Wert gegeben wird und bei Wiederholung der gleichen Benutzeranfrage die Entscheidung bezüglich des Zugriffs auf eine Datei sich ändert. Um alle möglichen Werte eines Konfigurationsfeldes zu erkunden, wird das Feld mit einem Zufallswert belegt. Das zu untersuchende Programm ist nun genötigt, den Wert des Feldes mit allen gültigen Werten zu vergleichen. Diese Vergleiche mit Konstanten nutzen die Autoren, um die Sprache der Konfigurationsdatei zu entwickeln. In der Auswertung ihrer Arbeit konnten die Autoren zeigen, dass es mit ihrem Ansatz möglich ist, alle relevanten Felder in komplexen Programmen wie Apache oder Bftpd zu identifizieren.

5.5 Schwachstellen Signaturen und 1-Day Exploits

Sobald ein Hersteller einen Sicherheitspatch für ein Produkt veröffentlicht, ist das Interesse groß, die zugrundeliegende Schwachstelle zu identifizieren. Black Hats können mit dem Wissen danach sogenannte 1-Day Exploits [Oh10] bauen, welche auf alle Nutzer zielen, die den Patch noch nicht eingespielt haben. White Hats können mit einem genauen Verständnis über die Sicherheitslücke Signaturen für Virenscanner und Intrusion Detection Systeme entwickeln.

Mit den vorgestellten Binary Diff Programmen ist es möglich, die Änderungen eines Patches zu identifizieren, um die Suche nach der gepatchten Sicherheitslücke zu beschleunigen. In [Oh10] wird gezeigt, wie sich so Schwachstellen aus Herstellerpatches extrahieren lassen.

Falls bereits ein Exploit für die Sicherheitslücke vorliegt, lässt sich die Suche durch weiteren Technikeinsatz vereinfachen. Dazu werden zwei Traces erstellt: Ein Trace in dem der Exploit erfolgreich ausgeführt wird und ein Trace, in dem der Exploit fehlschlägt. Mit Differential Slicing ist es dann effizient möglich, die Ursache der Unterschiede in beiden Traces aufzudecken. Somit ist auch die zugrundeliegende Sicherheitslücke bekannt. In [JCC+11] zeigte eine Studie den Nutzen von Differential Slicing bei der Analyse einer Schwachstelle im Adobe Reader.

5.6 Malware Ananlyse, 0-Days, Bug Fixing

Eine Sicherheitsanalyse ohne Vorwissen auf einer unbekannten Closed Source Binary durchzuführen, ist die Königsdisziplin der Schwachstellenerkennung durch Reverse Code Engineering. Folglich unterscheidet sich das Vorgehen von Fall zu Fall. Alle vorgestellten Techniken können sich also als hilfreich erweisen. Die Anwendungsgebiete sind vielseitig: Einerseits ist es von Interesse, welche Schwachstelle eine gegebene Malware versucht auszunutzen. Falls es sich dabei um eine bis dato unbekannte Schwachstelle handelt, können die gewonnenen Informationen vom Hersteller genutzt werden, um die Schwachstelle zu beseitigen. Andererseits ist es auch möglich, gezielt 0-Day Exploits zu entwickeln. Durch Fuzzing lassen sich Fehler und Programmabstürze finden. Die Analyse zeigt dann, ob sich der Fehler ausnutzen lässt.

Durch verteiltes Fuzzing konnte Nagi [Nag09] in Microsoft Word über 200000 Crashes finden. Davon konnten 9 direkt als potenziell exploitable eingestuft werden und nach ausgiebiger Analyse zeigte sich, dass 4 der Crashes tatsächlich exploitable waren. Ein großes Problem beim Fuzzing besteht jedoch darin, dass Fuzzing im Allgemeinen sehr viele Programmabstürze zurückliefern kann. Mit den vorgestellten Techniken ist es möglich, einige Abstürze zu klassifizieren. Trace Alignment könnte beispielsweise dabei helfen, einen zugrundeliegenden Fehler in mehreren Abstürzen wiederzufinden. Spezielle Programme wie Microsoft's !exploitable [Micb] wurden direkt dafür entwickelt, die Ergebnisse einer Fuzzing Session zu klassifizieren. Dabei wird jeder Programmabsturz mit *Exploitable*, *Probably Exploitable*, *Probably Not Exploitable* oder *Unknown* bewertet. Die der Entscheidung zugrundeliegende Fehlerart wird benannt. Des Weiteren wird der Ort des Fehler sehr genau bestimmt.

Für Entwickler ist es hilfreich, bekannte Fehler in Software zu priorisieren. Dabei sollten sicherheitskritische Fehler schneller behoben werden als reine Bugs. Während die Arbeit mit Tools aus dem BitBlaze Projekt eher für Reverse Engineere gedacht ist, richtet sich Microsoft's !exploitable direkt an die Entwickler. Da die Aufnahme der Traces für die weitere Verarbeitung mit BitBlaze sehr lange dauern kann, lohnt sich der Einsatz von BitBlaze nur, wenn weitere Analyseschritte bevorstehen.

Viele Firmen, unter Anderem Microsoft, untersuchten 2010 den berühmten Wurm stuxnet. Nicht nur durch die angebliche politische Motivation wurde stuxnet berühmt, auch durch die hohen Kosten die mit seiner Entwicklung verbunden sein mussten. So konnten Forscher durch Reverse Engineering mehrere bis Dato unbekannte 0-Day Exploits entdecken, welche stuxnet verwendete. Microsoft selbst nutzte statische und dynamische Analyse um die 0-Day Exploits zu entdecken, verstehen und die Probleme zeitnah zu beheben [DF10].

Miller und Johnson zeigen in [MJ10], wie sich ein Crash in Adobe Reader unter Einsatz vielfältiger Techniken schnell und einfach analysieren lässt. Bei dem Crash – welcher beispielsweise durch Fuzzing gefunden werden könnte – handelt es sich um ein Dereferenzierung eines ungültigen Pointers. Als Erstes stellt sich die Frage, ob

der Pointer von einer Benutzereingabe direkt abhängt. Eine Tainting Analyse konnte schnell bestätigen, dass der Pointer nicht tainted ist, daher ist der Crash nicht direkt exploitable. Als nächstes stellt sich die Frage, woher der ungültige Wert kommt. Durch Slicing ließ sich schnell feststellen, dass der ungültige Wert weder aus einem unkontrolliertem Lesevorgang noch aus einem uninitialisierten Bereich[7] stammt. Damit ist sichergestellt, dass dieser Crash nicht exploitable ist. Um den Fehler zu beheben, muss die Fehlerursache gefunden werden. Dafür wurde ein Trace mit dem besagten Crash erstellt und ein Trace ohne Crash. Durch Trace Alignment konnte schnell aufgedeckt werden, dass ein fehlender Sprung zu dem ungültigen Pointer führt. Mit dieser Information ließe sich der Fehler beheben. Um die weitere Möglichkeit des fortschrittlichen Tooleinsatzes zu demonstrieren, sei nun angenommen, dass der Pointer doch von einer Benutzereingabe abhängt. Es stellt sich also die Frage, wie viel Kontrolle hat ein potenzieller Angreifer? Mit `valset_ir` lässt sich der Wertebereich des Pointers abschätzen. Abhängig von dessen Größe und der weiteren Verwendung der Daten, ließe sich so ein Exploit bauen.

Malware kann oft unterschiedliches Verhalten aufzeigen. Durch Symbolic Execution ist es möglich, das komplette Verhalten der Malware zu erkunden. In [JCC+11] zeigen die Autoren, wie sich Differential Slicing einsetzen lässt, um die Ursache von unterschiedlichen Malwareverhalten zu entdecken. Für W32/Conficker.A konnte gezeigt werden, dass die Malware kein schädliches Verhalten an den Tag legt, wenn das Tastaturlayout des Zielrechners ukrainisch ist. Die Erforschung des Verhaltens von Malware liefert wichtige Erkenntnisse, die zu deren Abwehr genutzt werden können.

6 Ausblick

In der Einleitung wurden die in dieser Arbeit behandelten Schwachstellen wie folgt klassifiziert: Software Designfehler, unsichere Programmierpraktiken, Implementierungsfehler, Konfigurationsfehler und Durchsickern sicherheitskritischer Metainformationen. Dieser Abschnitt fasst den aktuellen Stand der Forschung anhand der in dieser Arbeit vorgestellten Techniken und Beispiele zusammen.

Mit den vorgestellten Techniken war es in den genannten Beispielen möglich, das Design der Implementierung wiederherzustellen und mit den Designdokumenten der Spezifikation zu vergleichen. Somit konnten Designfehler der Implementierung, welche nicht in den Designdokumenten vorlagen, entdeckt werden. Selbiges gilt auch für Abweichungen zwischen Quellcode und Maschinencode. Allerdings wurden nur kleine Teile Spezifikation betrachtet. Da diese Aufgabe stark an die entsprechende Spezifikation und Anwendung gebunden ist, existiert keine allgemeine Möglichkeit um das komplette Design der Implementierung mit der Spezifikation zu vergleichen. Gängige Praxis wird es also bleiben, Entwickler ständig darauf hinzuweisen, dass Änderungen im Quellcode sich auch in der Spezifikation wiederfinden müssen. Es besteht weiterhin Forschungsbedarf auf allen Ebenen im Felde des Reverse Code Engineerings um Abweichungen zwischen Design und Implementierung aufzudecken.

Es wurde gezeigt worauf zu achten ist, um sichere und unsichere Programmierpraktiken zu erkennen. Dies kann einerseits allgemein geschehen, wie bei der Suche nach Funktionen die anfällig für Buffer Overflows sind oder anwendungsspezifisch wie zufällige Initialisierungsvektoren bei Kryptoverfahren. Die vorgestellten Kriterien sind allerdings nicht allumfassend und entwickeln sich stetig mit den verwendeten Programmierpraktiken weiter. Während im Bereich der statischen Quellcodeanalyse

[7] Ein uninitialisierter Bereich könnte unter Anderem ein Teil des Stacks sein. Falls ein Angreifer nun die Parameter oder lokalen Variablen einer Funktion bestimmen kann, kann er unter Umständen diesen Bereich des Stacks kontrollieren.

gute Toolunterstützung existiert um unsichere Programmierpraktiken zu vermeiden, fehlt diese Unterstützung für Binärprogramme.

Die Erkennung von Implementierungsfehlern ist stets anwendungsabhängig und daher schwer allgemein zu klassifizieren. Für wohlbekannte Fehler wie Crashes existiert eine Vielzahl von Techniken und Programmen um diese zu analysieren. Da logische Fehler sich jedoch teilweise nur auf logischer Ebene äußern, existiert auch hier weiterer Forschungsbedarf um mehr Gebiete der logischen Fehler abzudecken.

Es wurde gezeigt, dass sich sicherheitsrelevante Felder in Konfigurationsdateien mittels Reverse Engineerings sehr gut aufzeigen lassen. Dadurch ist es möglich, die Konfiguration einer Software besser zu verstehen und Konfigurationsfehler zu vermeiden. Allerdings beschränken sich die hier vorgestellten Techniken nur auf die Zugriffskontrolle von Dateien. Konfigurationsfehler welche rein auf logischer Ebene sicherheitskritisch sind, können nicht erkannt werden und sind im Allgemeinen auch nicht automatisch erkennbar.

Mittels Reverse Engineerings wurde gezeigt, dass sich beispielsweise sicherheitskritische Metainformationen aus öffentlichen Entwicklungsrepositories extrahieren lassen. Dabei wurde festgestellt, dass der Autor eines Commits eine sicherheitskritische Information sein kann. Die Lizenz von einigen Projekten verbietet jedoch, den Autor eines Commits zu entfernen oder zu verfälschen. Hier besteht weiterer Forschungsbedarf um einen Weg zu finden, wie das Veröffentlichen von sicherheitskritischen Metainformationen verhindert werden kann, ohne die Lizenz eines Projektes zu verletzen. Auf Programmebene selbst wurde die Taint Analyse vorgestellt, um den Einfluss sicherheitskritischer Informationen zu verfolgen. Jedoch kann das Ergebnis viel zu groß und komplex sein, um es sinnvoll auszuwerten. Es besteht Forschungsbedarf um die Taint Analyse zu verfeinern. Gegebenenfalls ist domänenspezifisches Wissen nötig um die Komplexität des Ergebnisses zu verringern.

Allgemein lässt sich sagen, dass die Klassiker wie Crash-Analyse und Vendorpatches-Analyse sehr gut verstanden sind und es erfolgreiche Vorstöße für domänenspezifische Schwachstellen gibt. Allerdings muss sich ein Reverse Engineer noch sehr gut mit Maschinencode und der Anwendungsdomäne auskennen. Toolunterstützung ist oftmals entweder nur für Spezialfälle vorhanden oder liefert sehr komplexe Ergebnisse, welche schwer auszuwerten sind.

7 Fazit

In dieser Arbeit wurde ein großflächiger Überblick über die Schwachstellenerkennung mittels Reverse Code Engineerings gegeben. Dabei wurden altbewährte Techniken wie die Offline Code Analyse und Live Code Analyse vorgestellt. Spezialisierte Techniken wie Traces, Slicing, Taint Analyse, Trace Alignment, Symbolic Execution, Abstract Interpretation, Binary Diff und Differential Slicing wurden präsentiert.

Allgemeine Anwendungsgebiete der Techniken in der Schwachstellenanalyse wurden aufgezeigt. Dabei wurde jede Kategorie durch reale Beispiele veranschaulicht.

Literatur

AHLW. Adel Abouchaev, Damian Hasse, Scott Lambert, and Greg Wroblewski. Analyze crashes to find security vulnerabilities in your apps. MSDN Magazine, 2007. `http://msdn.microsoft.com/en-us/magazine/cc163311.aspx`.

BBC+10. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

BBG+63. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *The Computer Journal*, 5(4):349–367, 1963.

BDS03. S. Bhatkar, D.C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium*, volume 120. Washington, DC., 2003.

Bel05. F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.

Big89. T.J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36 –49, jul 1989.

Bit. BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

BLB10. Elie Bursztein, Jocelyn Lagarenne, and Dan Boneh. Kartograph: Finding a needle in a haystack or how to apply reverse engineering techniques to cheat at video games. In *DEFCON 18*, 2010.

BLRS11. Adam Barth, Saung Li, Benjamin I. P. Rubinstein, and Dawn Song. How open should open source be? Technical Report UCB/EECS-2011-98, EECS Department, University of California, Berkeley, Aug 2011.

Bra02. Brandon Bray. Compiler security checks in depth. msdn, February 2002. `http://msdn.microsoft.com/en-us/library/Aa290051`.

CC77. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

CC90. E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13 –17, jan. 1990.

CLO07. James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

Cor08. Jonathan Corbet. GCC and pointer overflows. *LWN*, April 2008. `http://lwn.net/Articles/278137/`.

Cov04. Coverity. Linux kernel software quality and security better than most proprietary enterprise software, 4-year coverity analysis finds. Press Releases, December 2004. `http://www.coverity.com/html/press_story03_12_14_04.html`.

DF10. Bruce Dang and Peter Ferrie. Adventures in analyzing stuxnet. In *27th Chaos Communication Congress*, Dec 2010.

dum. Description of the DUMPBIN utility. Microsoft Support. `http://support.microsoft.com/kb/177429`.

Eil05. E. Eilam. *Reversing: secrets of reverse engineering*. Wiley, first edition, April 2005.

fip02. 140-2: Security requirements for cryptographic modules, March 2002. `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`.

GM97. B. Guha and B. Mukherjee. Network security via reverse engineering of tcp code: vulnerability analysis and proposed solutions. *Network, IEEE*, 11(4):40 –48, jul/aug 1997.

GNU. GNU-Projekt. GNU Binutils. `http://www.gnu.org/software/binutils/`.

GP. GNU-Projekt. GDB: The GNU project debugger. `http://www.gnu.org/software/gdb/`.

Gui. Ilfak Guilfanov. IDA: Interactive disassembler. `http://www.hex-rays.com/products/ida/`.

Int. Intel. Intel® 64 and IA-32 Architectures Software Developer's Manuals.

JCC+11. N.M. Johnson, J. Caballero, K.Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 347 –362, May 2011.

Kin76. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

KL88. Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155 – 163, 1988.

Mica. Microsoft. Debugging tools for windows. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

Micb. Microsoft Security Engineering Center. !exploitable Crash Analyzer - MSEC Debugger Extensions. `http://msecdbg.codeplex.com/`.

MJ10. Charlie Miller and Noah Johnson. Crash analysis using BitBlaze. In *Black Hat USA Las Vegas, NV*, July 2010. `http://securityevaluators.com/files/papers/CrashAnalysis.pdf`.

Nag. Ben Nagy. Metafuzz. `http://rubyforge.org/projects/metafuzz`.

Nag09. Ben Nagy. Finding microsoft vulnerabilities by fuzzing binary files with ruby - a new fuzzing framework. In *SyScan Singapore*, 2009.

NK12. Tobias Nipkow and Gerwin Klein. Concrete semantics, Feb 2012. `http://www4.in.tum.de/lehre/vorlesungen/semantik/WS1112/imp.pdf`.

NOA09. Y. Nakamoto, T. Osaki, and I. Abe. Proposing universal execution trace framework for embedded software using qemu. In *Future Dependable Distributed Systems, 2009 Software Technologies for*, pages 173 –178, march 2009.

NS07. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

Oec09. Philippe Oechslin. Exposing crypto bugs through reverse engineering. In *26th Chaos Communication Congress*, Dec 2009.

Oh10. Jeongwook Oh. ExploitSpotting: Locating vulnerabilities out of vendor patches automatically. In *Defcon 18*, 2010.

OLG. Jeongwook Oh, Byoungyoung Lee, and Jay Gould. DarunGrim: A patch analysis and binary diffing tool. `http://www.darungrim.org/`.

Pos81. J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

RC11. Mark Russinovich and Bryce Cogswell. VMMap v3.1. technet.microsoft.com, May 2011.

Rus. Mark Russinovich. Process Explorer. Microsoft Technet. `http://technet.microsoft.com/en-us/sysinternals/bb896653`.

SBY+08. Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

sof12. SoftICE. wikipedia, March 2012. `http://en.wikipedia.org/wiki/SoftICE` Revision 463555537.

Spa10. Branko Spasojevic. Code deobfuscation by optimization. In *27th Chaos Communication Congress*, 2010. `http://code.google.com/p/optimice`.

SPE11. Christian Schneider, Jonas Pfoh, and Claudia Eckert. A universal semantic bridge for virtual machine introspection. In Sushil Jajodia and Chandan Mazumdar, editors, *Information Systems Security*, volume 7093 of *Lecture Notes in Computer Science*, pages 370–373. Springer, December 2011.

sta. /GS (buffer security check). msdn. `http://msdn.microsoft.com/en-us/library/8dbf701c%28v=vs.80%29.aspx`.

stra. strcpy, wcscpy, _mbscpy. msdn. `http://msdn.microsoft.com/en-us/library/kk6xf663.aspx`.

strb. StringCchCopy function. msdn. `http://msdn.microsoft.com/en-us/library/ms647527%28VS.85%29.aspx`.

Wei81. Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

WHF07. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, pages 32–39, 2007.

WWZL08. Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. Towards automatic reverse engineering of software security configurations. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 245–256, New York, NY, USA, 2008. ACM.

Yus. Oleh Yuschuk. OllyDbg. http://www.ollydbg.de/.
zyna. zynamics.com. BinDiff. http://www.zynamics.com/bindiff.html.
zynb. zynamics.com. BinNavi. http://www.zynamics.com/binnavi.html.