Software Testmethoden

Fuzzing

Foto: pixelio.de

Fuzzing ist eine Testmethode zur Aufdeckung von Problemen und Schwachstellen in Softwareapplikationen. Der Einsatz dieser Testmethode bietet vor allem Hersteller von closed-source Produkten einen signifikanten Vorteil, da sie exklusiv auf ihren Quellcode zugreifen können.

Während das Fuzzing in der traditionellen Qualitätssicherung bis heute nur selten angewendet wird, ist es im Umfeld der Forschung schon lange etabliert. Fuzzing wurde bereits 1989 an der Universität von Wisconsin konzipiert und angewandt.

Es gibt keine Software ohne Fehler

Traditionelles Software Testing geht in der Regel von einer perfekten Welt aus. In dieser Welt gibt es keine Schwachstellen in der Software Implementierung und auch keine böswilligen Absichten der Benutzer. Der Fokus liegt auf funktionalen Tests, also Tests, bei denen die spezifizierten Funktionalitäten in allen Ausführungen getestet werden: Funktion A führt zu Zustand B. Beim Sicherheitstest hingegen sind durch einen Paradigmenwechsel genau solche Testfälle zu betrachten, welche im traditionellen Softwaretest vernachlässigt werden; die nicht-funktionalen Tests. Bei nicht-funktionalen Tests wird bewusst über die Spezifikation hinaus getestet. Äquivalenzklassentests für die Grenzbereiche von Eingaben spielen dabei eine ebenso wichtige Rolle wie Negativbereiche, welche mehr aus dem Rahmen fallen [1]. In der Literatur wird auch von Seiteneffekten gesprochen [2], auf die ein Sicherheitstester zu achten hat. Funktion A führt zu Zustand B und beeinflusst C.

Definition und Geschichte

Fuzzing bietet die geeignete Technik für das Auffinden von sicherheitsrelevanten Schwachstellen in Softwareprodukten. Der Begriff „Fuzzing“ ist nicht eindeutig definiert. Der Ursprung des Begriffs ist in der Übertragung von Daten über instabile (fuzzy) Telefonleitungen zu suchen, wie es zu „Modem-Zeiten“ üblich war. Durch Übertragungsfehler wurden die Befehle mit willkürlichen Zeichen verändert, die darauf hin zum Absturz von Komponenten auf dem Zielsystem führten. Heute versteht man unter Fuzzing das Testen der Robustheit von Softwarekomponenten mit Hilfe von willkürlichen oder zielgerichteten (intelligenten) Daten.

Der erste Fuzzer („Fuzz“) entstand 1989 an der Universität von Wisconsin und benutzte zufällige Patterns um lokale Fehler in UNIX Dienstprogrammen zu finden. Die Funktionen waren bis dahin eingeschränkt und Fuzzing noch unbe-

kannt. Erst 1999, als an der Universität von Oulu die PROTOS Fuzzing Testsuite entwickelt wurde, entfaltete es sich in vielfältigen Bereichen. Aus PROTOS entstand 2003 mit Codenomicon eines der ersten Unternehmen, die Fuzzer kommerziell vertreiben. Ungefähr zu gleichen Zeit (2002) veröffentlichte Dave Aitel das SPIKE Fuzzing Framework. SPIKE hat die Entwicklung von vielen weiteren Applikationen und Frameworks beeinflusst. SPIKE bietet verschiedene Funktionen auf höherem Abstraktionsniveau an, mit denen eigene Fuzzer und Testdaten erstellt werden können. Mit Hilfe von bestehenden Templates für beispielsweise das IMAP, PPTP oder MS-RPC Protokoll wird der Einstieg in das Framework vereinfacht.

Bis heute wurden zahlreiche weitere Fuzzer entwickelt, darunter universelle Frameworks wie „Sulley“, „Peach“ oder „SMUDGE“ und spezialisierte wie zum Beispiel „iFUZZ“ oder „COMbust“. Eine ausführliche Liste können Sie [3] entnehmen. Als besonders bekannte Vertreter der Fuzzer sind außerdem „mangleme“ und darauf aufbauend auch „htmler“ für das Testen von Web Browsern zu nennen. Durch diese wurde eine Vielzahl von Sicherheitsmängeln in allen bekannten Browsern entdeckt. Der wohl berühmteste Fund von htmler ist die Schwachstelle, die beim verarbeiten von iframes auftaucht. Diese Schwachstelle führte unter anderem zu dem berühmten W32/mydoom Wurm.

Arten, Methoden und Typen

Unterscheiden lassen sich Fuzzing Applikationen (Fuzzer) grundsätzlich in zwei grobe Bereiche [4]: lokale und remote (entfernt). Lokale Fuzzer werden direkt auf dem Testobjekt, beziehungsweise dem Zielsystem ausgeführt. In diese Kategorie fallen alle Kommandozeilen-Parameter Fuzzer, wie beispielsweise „Fuzz“ und Datei-Fuzzer, wie „FileFuzz“ oder „SPIKEfile“. Die Remote-Variante eignet sich für den Test von netzwerkbasierten Applikationen oder Komponenten. In diese Kategorie gehören beispielsweise Netzwerkprotokoll-Fuzzer (oder Frameworks) wie „SPIKE“ und „PEACH“ oder Web Applikations- und Browser-Fuzzer, wie „htmler“ und „WebScarab“.

Fuzzer lassen sich auch bezüglich ihrer Methodik in zwei grobe Bereiche unterteilen: „dumb“ und „smart“. So genannte dumme Fuzzer kennen die Logik, Zusammenhänge, oder den Aufbau des Testobjekts nicht. Das Testobjekt wird in diesem Fall mit willkürlichen Werten getestet, bis es zu einem Absturz oder einer Speicherzugriffsverletzung kommt. Ein einfaches Beispiel für dummes Fuzzing, dass selbst heutzutage immer noch zu verwunderlichen Ergebnissen führen kann, ist

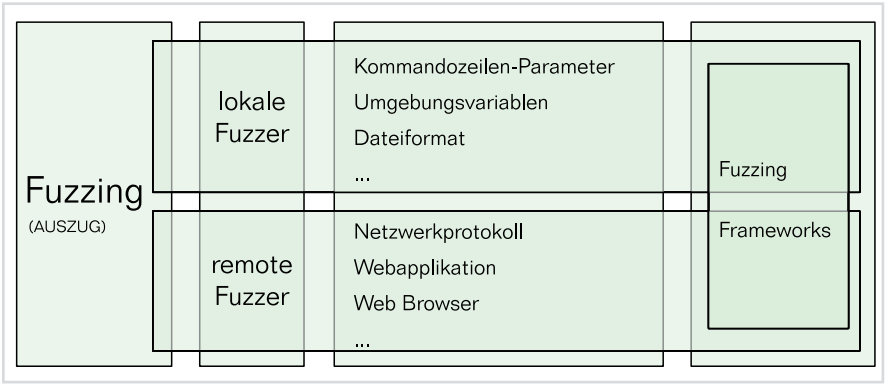

Bild 1: Eine Übersicht) der Fuzzer-Typen (Auszug.

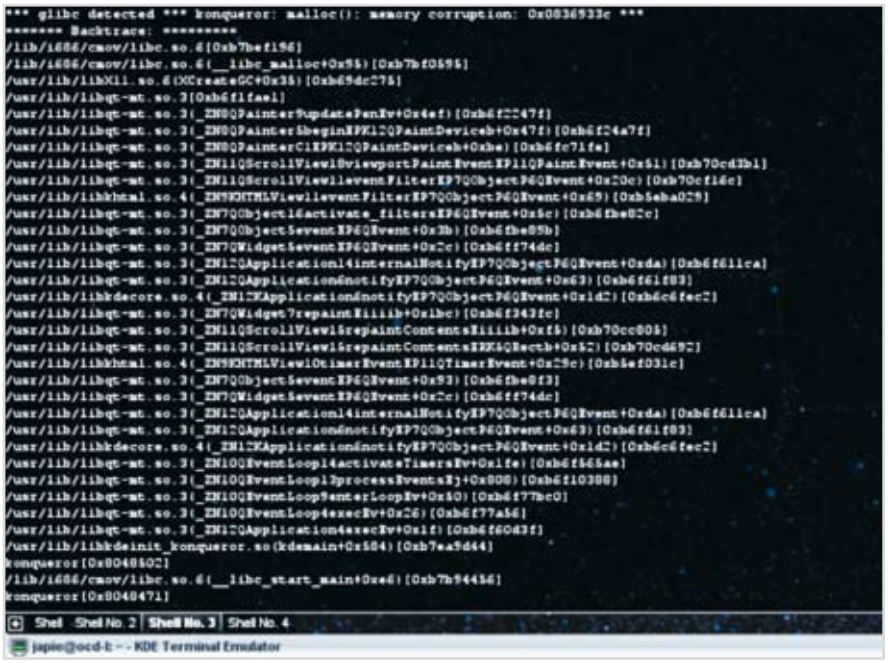

Bild 2: Fuzzing Ablaufverfolgung – Konqueror Browser 3.5.9

die Umleitung der /dev/urandom/ Ausgabe auf einen Netzwerkport des Testobjektes.

Eine höhere Effizienz und Testabdeckung kann mit Hilfe von intelligenten Fuzzern erreicht werden. Viele Applikationen, Protokolle und Dateien verlangen einen bestimmten, festgelegten Aufbau der verarbeiteten beziehungsweise übertragenen Daten. Dieser Aufbau ist bei dummen Fuzzern nicht gegeben, wodurch Daten noch vor ihrer Verarbeitung und damit vor den interessanten Programmstellen verworfen werden. Bei der intelligenten Variante werden dem Fuzzer die Strukturen des Testobjektes beigebracht, damit die Applikation alle Fuzzing-Anfragen verarbeiten kann. Es wird deshalb von semi-validem Input gesprochen. Ein Nachteil dieser Methode ist der initiale Zeitaufwand für die Analyse und Programmierung der Applikations-, Protokoll- oder Dateistrukturen. Dieser Mehraufwand wird jedoch durch eine höhere Fehlerausbeute belohnt. Die besseren Ergebnisse amortisieren den initialen Zeitaufwand nach wenigen Iterationen. In der Praxis wird daher die intelligente Methode bevorzugt, wobei der Fokus zu Beginn meist auf Kernfunktionen des Testobjekts begrenzt wird.

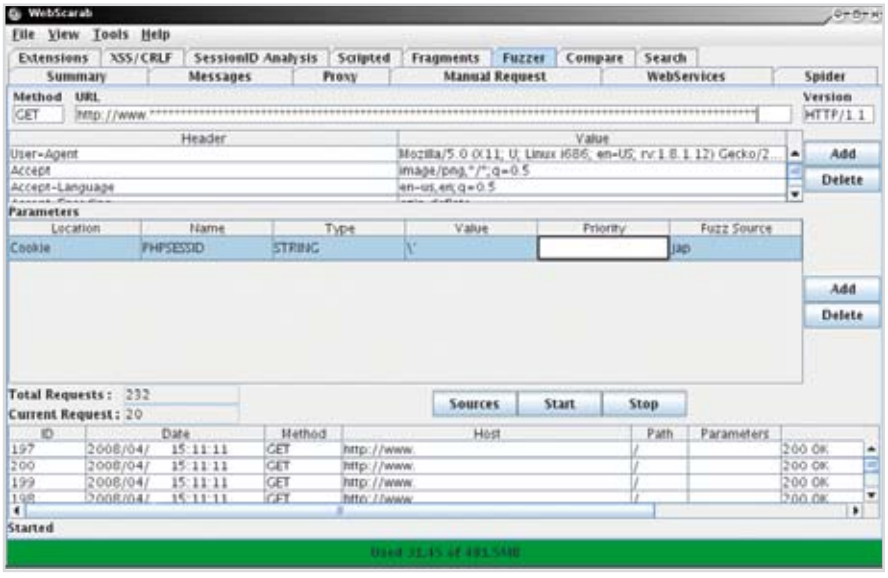

Bild 3: Webscarab Fuzzer.

Ferner gilt es sich anhand der Projektanforderungen zwischen bestehenden Fuzzern, Fuzzer Frameworks oder einer möglichen Eigenentwicklung zu entscheiden. Der Vorteil eigener Fuzzer ist in der Regel ihr perfekter Zuschnitt auf das Testobjekt, jedoch sollten dabei folgende Punkte in die Entscheidungsfindung einbezogen werden:

- Wiederverwendbarkeit
- Nachvollziehbarkeit
- Testabdeckung
- Monitoringmöglichkeiten zur Fehlererkennung

Im Gegensatz zur Eigenentwicklung hat ein Fuzzing Framework

Referenzen

[1] Oehlert, P. „Violating assumptions with fuzzing“. Security & Privacy Magazine, IEEE, 2005.

[2] Thompson, H.H. „Why Security Testing is hard“. Security & Privacy Magazine, IEEE, 2003.

[3] ThreatMind Security Wiki. http://www.threatmind.net/secwiki/FuzzingTools

[4] Sutton, Green and Amini. „Fuzzing, Brute Force Vulnerability Discovery“. Addison-Wesley, 2007.

[5] Neystadt, John. „Automated Penetration Testing with White-Box Fuzzing“. MSDN Library. 2008.

[6] Frei, Tellenback und Plattner. „0-day Patch – Exposing Venders (In)security Performance“ TIK, ETH Zurich, 2008.

[7] Howard, Michael, and Steve Lipner. „The Security Development Lifecycle“. Redmond, WA: Microsoft Press, 2006.

den Vorteil, dass die genannten Punkte berücksichtigt werden. So bieten beispielsweise das „Sulley“ und „Peach“ Fuzzing Framework fertige Methoden für die Ablaufverfolgung und das Erkennen von Fehlern an. Zu den derzeitigen Nachteilen beider Frameworks gehört die Komplexität der Handhabung. Daher streben beide Frameworks, in neueren Releases, eine höhere Benutzerfreundlichkeit an. „Peach“ bietet beispielsweise in der aktuellen Betaversion 2.1 bereits eine graphische Anwendung zur Erstellung der XML Testbeschreibung an. Das „Sulley“ Framework plant den Einsatz einer Web Anwendung zur Modellierung der Tests.

Testverfahren und Vorteile

Die grundsätzlichen Testarten sind Black-Box, Gray-Box und White-Box Tests. Bei Black-Box Tests ist die interne Programmstruktur nicht bekannt. Daher ist nur eine eingeschränkte Testabdeckung zu erreichen. White-Box Tests basieren auf dem Quellcode des Testobjekts. Testdaten werden daher oft aus der internen Logik abgeleitet. Mit Hilfe von White-Box Tests kann eine vollständige Testabdeckung erreicht werden. Gray-Box Tests stellen eine Kombination von Black-Box und White-Box Testverfahren dar. Beispielsweise wird das Testobjekt mit Wissen über Interna und mit Hilfe eines Debuggers getestet, dabei wird überprüft ob die sensiblen Stellen im Quellcode erreicht werden können.

Da einem Softwarehersteller der Quellcode seines eigenen Testobjekts vorliegt, kann dieser mit Hilfe von White- und Gray-Box Verfahren wesentlich effizienter testen. Ein Softwarehersteller hat dadurch signifikante Vorteile gegenüber Hackern. Der Hersteller weiß auf welche Stellen er sich im Programmcode konzentrieren muss, wie das eigene binäre Protokoll aufgebaut ist und welche Strukturen akzeptiert werden. John Neystadt beschreibt in [5] wie mit Hilfe von White-Box Fuzzing eine Testabdeckung von 99% erreicht werden kann. Ein Angreifer, der keinen Zugriff auf den Quellcode der Software hat, kann dies über einen Black-Box Ansatz nicht erreichen.

Integration in den Software Development Lifecycle (SDL)

In den letzten sechs Jahren entwickelte sich Fuzzing von einem rein akademischen Forschungsthema zu einer immer populärer werdenden Standardtestmethode vieler Softwarehersteller. Trotzdem hat es sich noch nicht flächendeckend durchgesetzt. Dies liegt nicht zuletzt an der immer noch hohen Komplexität der Fuzzer. Auch die Generierung von Testdaten für komplizierte Protokolle ist nach wie vor zeitaufwendig. Um diesen Zeitaufwand einzuplanen und die Quali-

tät der Softwareprodukte nachhaltig und langfristig zu steigern, ist eine grundlegende Anpassung der Entwicklungsprozesse nötig.

Im traditionellen SDL werden Sicherheitstests, wenn überhaupt, erst gegen Ende des Entwicklungsprozesses durchgeführt. Dabei gefundene Schwachstellen können nur mit großem Aufwand behoben werden. Hierbei gilt der Grundsatz; je später ein Fehler gefunden wird, desto höher ist der Zeit-/Kostenaufwand ihn zu beseitigen.

Sicherheitstests müssen daher bereits in frühen Phasen der Softwareentwicklung eingebettet werden. Bereits die Analyse- und Design Phase bietet die Möglichkeit für die Erhöhung der Sicherheit der Software.

Microsoft hat im Bezug auf die Entwicklung von sicherem Programmcode über die letzten Jahre viel dazugelernt. Laut [6] hat sich die Anzahl der Advisories von Microsoft im Gegensatz zu anderen Herstellern von Jahr zu Jahr verringert. Das hängt einerseits damit zusammen, dass Microsoft die Zusammenarbeit mit Sicherheitsexperten verstärkt hat (Bluehat Security Conference). Andererseits hat Microsoft mit Hilfe des eigenen „Secure Software Development Lifecycle“ [7] den Fokus in Richtung „Sichere Software“ gelenkt. Software- und Sicherheitstests (wie Fuzzing) spielen im SSDL eine wesentliche Rolle und werden in alle Phasen integriert.

Quality Assurance Team

Für den Einsatz von Sicherheitstestmethoden im SDL muss nicht jeder QA Mitarbeiter ein Sicherheitsexperte sein. Wichtig ist jedoch, dass allen das Interesse an sicherer Software vermittelt wird. Ob nun ein in der QA gefundener Fehler tatsächlich auch zu einem gravierenden Sicherheitsproblem führen könnte, müssen dann die Sicherheitsexperten entscheiden. Grundsätzlich lohnt es sich jedem während der Entwicklung gefundenen Fehler nachzugehen.

Fazit

Trotz seiner offensichtlichen Vorteile wird das Fuzzing in der traditionellen Qualitätssicherung nur selten angewendet. Die bisherigen Ergebnisse und Erfolge dieser Testtechnik sprechen allerdings für sich. Sicherheitsexperten und „Vulnerability Researcher“ bedienen sich des Fuzzings ebenso wie die Marktgrößen Mozilla und Microsoft. Hersteller von Software müssen sich genau überlegen, ob sie das Auffinden von Fehlern in der eigenen Software auch weiterhin den Hackern überlassen wollen, oder ob Sie den unvermeidlichen Sicherheitsrisiken künftig proaktiv begegnen.

Jakob Pietzka M. Sc., Dipl. Inf. (FH)
pietzka@oneconsult.com