Fuzzing von Embedded Software
Grundlagen und Erfahrungen aus der Praxis

Axel Wintsche, Philotech

Die Sicherheit von Software ist ein Kriterium welches immens an Bedeutung gewonnen hat, sich aber nur unzureichend als Anforderung formulieren und testen lässt. Teststrategien wie das Fuzzing bietet allerdings eine Möglichkeit automatisiert die Robustheit von Software zu testen und damit die Sicherheit zu erhöhen. Hier beschreiben wir was Fuzzing ausmacht, welche Hürden es beim Testen von Embedded Software gibt und mögliche Lösungsansätze.

Sichere Software Entwicklung: Ziel und Zeitpunkt von Software Test

Das V-Model ist ein klassisches Beispiel dafür wie sich Anforderungen aus den verschiedenen Phasen der Softwareentwicklung durch geeignete Testverfahren prüfen lassen. Und auch bei heutzutage oft verwendeten inkrementellen, agilen Methoden gibt es meist eine enge Verzahnung zwischen Entwicklung und Test, mit dem Ziel die Qualität zu steigern. Um die Sicherheit von Software zu erhöhen können entsprechende Methoden bei Entwicklung und Tests angewandt werden, mit dem Unterschied das sich das Kriterium Sicherheit nur unzureichend als Anforderung formulieren und testen lässt. Selbst wenn durch eine umfassende Bedrohungsmodellierung entsprechende Anforderungen abgeleitet werden und von der Software erfüllt werden können sich durch neue, bisher unbekannte Bedrohungen nicht erfüllte Anforderungen ergeben. Diese Fälle lassen sich durch entsprechendes Incidence Response Management behandeln. Neue Bedrohungen, also Situationen die nicht durch Anforderungen spezifiziert wurden lassen sich aber auch proaktiv durch entsprechende Testverfahren wie z.B. Fuzzing identifizieren.

Sichere Software ist daher als ein Prozess zu verstehen, welcher durch Einsatz verschiedener Methoden und Verfahren, vom Entwurf über die Implementierung bis zur Auslieferung und Wartung gelebt werden muss. Als Beispiele sind hier der SDL (Secure Development Lifecycle) von Microsoft und die Ressourcen von der Open Web Application Security Project (kurz OWASP) zu nennen - beide empfehlen den Einsatz von Fuzzing. Welche Konsequenzen Schwachstellen in Software eingebetteter Systeme haben können bewiesen verschiedene Untersuchungen von Sicherheitsforschern. Ob erfolgreiche Hacks zum Fernsteuern von Autos [1,2], das Manipulieren von Herzschrittmachern und anderer Medizintechnik [3] oder aufgedeckte Sicherheitslücken und Angriffe auf Infrastruktur [4,5]. Die Notwendigkeit einen Security Testprozess auch für diese Systeme zu etablieren ist daher groß.

Fuzzing Tests

Fuzzing ist eine Methode mit der Software durch ungültige oder zufällige Eingaben getestet wird um ein Fehlverhalten zu provozieren. In ihrer simpelsten Form werden durch Fuzzing eine zufällige Abfolge von Bits erzeugt oder von einer bestehenden Bitfolge eine beliebige Anzahl Bits zufällig verändert und diese als neue Eingabe für die zu testende Software verwendet. Da die allermeisten auf diese Weise erzeugten Eingaben keiner formalen Anforderung entsprechen lässt sich vor allem Testen, wie stabil die Software mit fehlerhaften Eingaben umgeht. Fuzzing Tests zielen also auf die Robustheit von Software gegenüber unerwarteten Eingaben ab. Ein Kriterium für Robustheit ist z.B. das die Software zu jeder Zeit in einem definierten Zustand ist und nicht etwa abstürzt. Prinzipiell lässt sich so jede datenverarbeitende Software oder Softwarekomponente mittels Fuzzing testen.

Fuzzing besteht im Wesentlichen aus drei aufeinander folgenden Schritten: 1) Daten generieren, 2) Daten bereitstellen und 3) Monitoring der Software. Diese Schritte sind unabhängig voneinander, werden aber oft in einer Schleife ausgeführt. Häufig werden dieselben Fehler mehrfach gefunden so dass am Ende eines Fuzzing Tests sinnvoller Weise gefundene Fehler zusammen gefasst werden (Bug Triage).

Daten generieren:
Dies ist der eigentliche Fuzzing Prozess in welchem die Testdaten erzeugt werden. Man unterscheidet hier zwischen zwei Methoden, „Generieren“ und „Mutieren“. Beim Generieren werden anhand einer Spezifikation oder eines Modells Testdaten erzeugt und bei Inhalt und Struktur zufällige Änderungen vorgenommen. Werden hingegen bestehende Daten mit zufälligen Änderungen versehen wird dies Mutieren genannt. Beide Varianten kommen in unterschiedlichen Szenarien zum Einsatz.

Daten bereitstellen:
Die gefuzzten Daten werden anschließend an die zu testende Software übergeben, je nach Art der Eingabeschnittstelle z.B. als Parameter beim Programmaufruf, als Datei oder in Form einer Netzwerknachricht.

Monitoring:
Fehler bei der Ausführung der Software werden durch das Monitoring ermittelt. Dies kann von der Analyse der Rückgabewerte bis hin zur Nutzung von Debuggern reichen welche den internen Zustand detektieren können. Die Art des Monitoring entscheidet welche Fehler überhaupt detektiert werden können.

Fuzzing kann als Blackbox und als White-Box Test eingesetzt werden. Programme für Blackbox Fuzzing sind oft einfacher zu konfigurieren und zu automatisieren und meist universell einsetzbar. Sie finden tendenziell aber eher simple Fehler (Dumb Fuzzing). Sind Spezifikation oder gar Quellcode verfügbar lohnt es sich mehr Aufwand in das Erzeugen der Daten und das Monitoring zu investieren da hierdurch die Effizienz erheblich gesteigert werden kann (Smart Fuzzing). Mittlerweile gibt es eine große Palette an verfügbaren Fuzzing Werkzeugen (https://github.com/secfigo/Awesome-Fuzzing#tools). Neben vielen spezialisierten Tools wie z.B. Fuzzer für bestimmte Dateiformate oder Netzwerkprotokolle existieren auch Frameworks welche durch entsprechende Konfiguration vielseitig einsetzbar sind.

Durch Fuzzing gefundene Fehler sind in ihrer Natur oft schwerwiegend, wie z.B. Buffer Overflows, Integer Overflows, Denial of Service (DoS) oder Code Injection Schwachstellen (XSS, SQLi) [6]. Darüber hinaus ist Fuzzing ein gut automatisierbares, kosteneffektives Testverfahren mit dem Potenzial die bei Code Review, Unit Test und Co nicht entdeckten Fehler zu detektieren (extreme oder nicht sinnvolle Eingaben). Beachtet werden muss das die Ausführen von Millionen von Testfällen mitunter sehr zeitaufwendig sein kann und daher eine gute Planung und Integration in den Entwicklungsprozess erfordert. Wie bei allen Testmethoden kann auch der Fuzzing Test kein komplettes Bild vorhandener Schwachstellen wiedergeben und eine Software ohne Fehler beim Fuzzing ist damit nicht automatisch sicher.

Fuzzing von eingebetteten Systemen
Für Software auf Steuergeräten gelten oft andere Rahmenbedingungen woraus sich neue Herausforderungen an das Fuzzing ergeben, hauptsächlich dadurch das Fuzzing Tool und getestete Software auf getrennten Systemen laufen. Als Testdatenformat sind meist spezielle Nachrichtenprotokolle wie CAN, LIN oder ARINC einzuhalten und die Übertragung an das

Steuergerät erfordert oft zusätzliche Hardware und Software. Mit sogenannte Breakout Boxen lässt sich zudem eine Systemumgebung simulieren (Sensoren und Nachrichten Bus). Beim Monitoring besteht die Schwierigkeit Fehler bei der Ausführung auf dem Steuergerät zu detektieren. JTAG Debugger sind hier oft die einzige Möglichkeit den internen Zustand zu beobachten. In einem Blackbox Testszenario können Time-out Kriterien eine praktikable Lösung darstellen wobei ein vorhandener Watchdog deaktiviert werden sollte.

Beispiel CAN Nachrichten Fuzzing
Getestet werden soll eine Softwarekomponente welche auf der Anwendungsschicht CAN Nachrichten auf einem Steuergerät verarbeitet. Eine CAN Nachricht besteht vereinfacht aus einer ID, der Angabe zur Nachrichtenlänge und dem Nachrichteninhalt. Nicht protokollkonforme Nachrichten würden bereits bei der Übertragung über die CAN Hardware blockiert werden und bei Nachrichten mit ungültiger ID würden diese nicht an die entsprechende Softwarekomponente übergeben werden. Spezielle Login und Logoff Nachrichten müssen zu Beginn und Ende einer Verbindung gesendet werden und auf jede gesendete Nachricht folgt eine Antwortnachricht vom Steuergerät.

Für das Fuzzing eines solchen Systems müssen die Voraussetzungen zum Senden und Empfangen auf den CAN Bus geschaffen werden. Die notwendige Hardware und Treiber müssen installiert werden und eine Anbindung des Fuzzing Programms an die Treiber API hergestellt werden. Die Erzeugung der Testdaten sollte Login und Logoff Berücksichtigen und vorwiegend den Datenbereich und die Längenangabe fuzzen. Dies steigert die Effizienz des Fuzzing und somit auch die Chance Fehler zu detektieren. Trotz der benutzerdefinierten Gegebenheiten ist es oft nicht notwendig ein spezialisiertes Fuzzing Programm zu entwickeln da es Fuzzing Frameworks gibt welche sich entsprechend konfigurieren lassen. Für das Monitoring ergeben sich mehrere Möglichkeiten. Z.B. kann bereits das Ausbleiben einer Antwortnachricht vom Steuergerät zur Fehlerdetektion genutzt werden (Timeout Kriterium). Durch Verwendung eines JTAG Debuggers lässt sich z.B. der Stacktrace bei auftretenden Fehlern ermitteln.

Zusammenfassung
Fuzzing ist eine ernstzunehmende Testmethode und wird bereits erfolgreich bei Desktop und Web Applikationen eingesetzt. Im Kontext eingebetteter Systeme werden Fuzzing Tests allerdings kaum eingesetzt obwohl es gewisse Ähnlichkeit zu herkömmlichen Computernetzwerken gibt. Einzelne Steuergräte sind über verschiedene Netzwerktypen wie CAN, LIN oder ARINC miteinander verbunden und somit Fuzzing als Teil des Security Testprozesses gut denkbar. Da immer mehr eingebettete Systeme über Komponenten mit „online“ Funktionalität verfügen lassen sich zudem die Erfahrungen aus Fuzzing Tests klassischer Protokolle (TCP/IP) und moderner Anwendungen (z.B. Web- und Smartphone Apps) nutzen. Security Testmethoden wie Fuzzing könnten sich zukünftig auch in Safety Standards wie der ISO 26262 wiederfinden um den aktuellen Stand der Technik an die Sicherheitsanforderungen zu erfüllen [7].

Literatur
[1] C. Miller and C. Valasek, "Adventures in Automotive Networks and Control Units," DEFCON 21 Hacking Conference, 2013.

[2] “Car Hacking Research: Remote Attack Tesla Motors”, Keen Security Lab, http://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/

[3] "Los, Hacker, brecht mir das Herz!: Sicherheit von vernetzter Medizintechnik auf dem Prüfstand" https://www.heise.de/newsticker/meldung/Los-Hacker-brecht-mir-das-Herz-Sicherheit-von-vernetzter-Medizintechnik-auf-dem-Pruefstand-3145186.html

[4] “Schadsoftware im Atomkraftwerk Gundremmingen”, heise.de News, https://www.heise.de/newsticker/meldung/Schadsoftware-im-Atomkraftwerk-Gundremmingen-3186045.html

[5], „92 Percent of Internet-Available ICS Hosts Have Vulnerabilities”, http://news.softpedia.com/news/92-percent-of-internet-available-ics-hosts-have-vulnerabilities-506204.shtml

[6] Prasanna Padmarajulu, "Discovering vulnerabilities with Fuzzing", PenTest Magazin 04/2017, https://pentestmag.com/download/pentest-intelligent-fuzzing-techniques/

[7] Bayer S., Enderle T., Oka DK., Wolf M. (2016) Automotive Security Testing - The Digital Crash Test. In: Langheim J. (eds) Energy Consumption and Autonomous Driving. Lecture Notes in Mobility. Springer

Autor

Axel Wintsche ist Diplom Informatiker und ist seit 2009 als Softwareentwickler und Tester tätig. Seine Kompetenz im Bereich Software Sicherheit konnte er in verschiedenen Projekten und Weiterbildungen erwerben. Herr Wintsche besitzt darüber hinaus verschiedene Zertifikate aus diesem Bereich und ist Referent zum Thema Sichere Software Entwicklung in der Philotech Academy.

Kontakt:
Email: axel.wintsche@philotech.de

Center of Competence Embedded Security

Philotech GmbH, Bahnhofstraße 30, 03046 Cottbus

BTU Cottbus – Senftenberg, Lehrstuhl Systeme
Erich-Weinert-Str. 1, 03046 Cottbus