Checkpoint/Restore in Fuzzing

Malte Klaassen

tbd

NEEDS REWORK Fuzzing von komplexen Programmen als ganzes ist meist langsam, Fuzzing komplexer Programme in vielen kleinen Einzelblöcken ist aufwändig für den Tester da für sie jeweils die Fuzz-Targets geschrieben werden müssen. In diesem Stück Papier untersuchen wir die Nutzbarkeit von Checkpoint/Restore-Mechanismen zur Lösung solcher Probleme im Fuzzing, insb. mit Blick auf Netzwerkprotokolle mit einer wohldefinierten State Machine, präsentieren eine mögliche Architektur zum Fuzzen von Implemetierungen ebensolcher Protokolle und untersuchen verschiedene Fuzzer und Checkpoint/Restore Tools auf ihre Anwendbarkeit.

Inhaltsverzeichnis

1 Einleitung

Mit zunehmender Prävalenz und Prominenz von IT-Systemen ist das Thema der Sicherheit eben dieser Systeme zunehmend in den Fokus gerückt, sowohl in Fällen häufig fehlerbehafteter Nutzung von Sofware in denen Nutzer selbst, direkt betroffen sind wie bei Hacks oder Leaks bei Online-Plattformen wie Facebook oder Malware-Kampagnen wie WannaCry, häufig ausgelöst durch technische Sicherheitslücken und verzögerte Patches, als auch vereinzelt in Fällen von rein technischen Sicherheitslücken wie beispielsweise Heartbleed.
Proaktiv vs Reaktiv
Über die Jahrzehnte haben sich dabei verschiedene Ansätze und Methoden zur proaktiven Erkennung, Behebung und Verhinderung von ebensolchen Sicherheitsproblemen entwickelt, beispielsweise:

- Formulierung von Sicherheitsanforderungen bereits während und vor der Designphase
- Formale Beweise beispielsweise bei kryptografischen oder Netzwerk-Protokollen oder sogar von Programmen, insbesondere in der Funktionalen Programmierung
- Compilerchecks und -safeguards sowie statische Codeanaylse
- (Externe) Reviews der Software und Penetrationtesting
- Klassisches Sicherheitstesting

Leider sind nicht immer alle diese Methoden anwendbar und haben jeweils ihre eigenen Stärken und Schwächen. So sind Formale Beweise der Sicherheit von Programmen außerhalb von einigen modernen Hochsprachen häufig nicht möglich, klassisches Sicherheitstesting ist zumeist beschränkt auf die Vorstellungskraft des Testers beziehungsweise auf sein Verständnis der zu testenden Software, Grenzfällen oder komplett unerwartete aber mögliche Eingaben sind mit klassischem Testing schwer umfassend abzudecken.

1.1 Fuzzing

Fuzzing ist eine Test-Methode die teil-automatisiert versucht genau diese Art von mit klassischen Test-Methoden schwer abzudeckenden Fällen umfassend zu erkennen indem,

statt wie im klassischen Testing nur in von Menschen mehr oder weniger präzise beschriebenen Testfällen getestet wird, ein Codestück, das so genannte Fuzz-Target, wiederholt mit computergenerierten zufälligen Inputs getestet wird, dabei wird die Ausführung genau von Fuzzer überwacht um beispielsweise Programmabstürze, das Lesen aus unitialisiertem Speicher oder auch undefiniertes Verhalten zu Erkennen und zu der problematischen Codestelle zurück zu verfolgen.

1.2 Klassifizierung von Fuzzern

Fuzzer werden anhand mehrerer Eigenschaften unterschieden, insbesondere ob sie sich beim Testen der Struktur des Programmes und der Struktur der erwartetenden Eingaben bewusst sind. Hierbei

- Wenn der Fuzzer sich der Struktur des Programmes bewusst ist und dieses Wissen im Testing nutzt um bessere Code-Coverage zu erreichen spricht man, je nach Technologie, von einem White-Box- oder Gray-Box-Fuzzer. Besitzt der Fuzzer keine Kenntnisse über die Programmstruktur und generiert programmunabhängige Zufallsinputs spricht man von einem Black-Box-Fuzzer.

- Wenn der Fuzzer sich der erwarteten Struktur des Inputs, beispielsweise bei einem bestimmten Dateiformat oder Netzwerkprotokoll, bewusst ist spricht man von einem Smart Fuzzer, ansonsten von einem Dumb Fuzzer.

Die meisten modernen Fuzzer können sowohl als Black- als auch als Gray- beziehungsweise White-Box-Fuzzer und sowohl als Smart als auch als Dumb Fuzzer agieren, diese Terme werden zumeist auch zur Beschreibung von Operationsmodi ebensolcher moderner Fuzzer verwendet.

1.3 Historischer Überblick

Testing mit randomisierten Inputs ist kein sonderlicher neuer Ansatz, jedoch ist die Effizienz von naivem Random Input Test (also Black-Box-Fuzzing) offensichtlich beschränkt und der größte Teil der Fehler die dauf diese Art und Weise gefunden werden können relativ simpel und selten in den Tiefen des Programmes versteckt, d.h. es handelt sich zumeist um Fehler die vergleichsweise simpel auch durch Code-Reviews oder klassische Tests gefunden werden könnten, entsprechenden hatte Random Input Testing für recht lange Zeit nur eine relativ geringe Relevanz, sowohl in der Qualitätssicherung als auch in der Sicherheitsforschung.
Mit dem Aufkommen von effizienteren Gray- und White-Box-Fuzzern und zunehmender Relevanz von IT-Sicherheit in den späten 2000ern entwickelte sich ein breiteres Interesse an Fuzzing, inklusive einiger Open Source Fuzzern und Fuzzing Frameworks, hin zu Fuzzing als Kernbestandteil des Sicherheits-Testings vieler relevanter Open Source Software Projekte, vorrangetrieben insbesondere durch Projekte wie Googles OSS-Fuzz, ein Service zum kontinuierlichen Testen verschiedenster Open Source Software durch mehrere verschiedene Fuzzer.

1.4 Beschränkungen im Fuzzing

Leider ist natürlich auch Fuzzing nicht ohne Probleme und Beschränkungen. Da Fuzzing zu großen Teilen darauf beruht eine große Menge an Eingaben zu Testen sinkt die Effizienz des Fuzzings deutlich wenn die Ausführug des Fuzz-Targets zu lange dauert. Dies führt dazu, dass das Fuzzen komplizierter, insb. interaktiver Systeme nicht ohne weiteres effizient möglich ist. Dieses Problem wird zumeist gelöst indem man komplexe Systeme in kleinere, simple Subsysteme aufteilt und diese unabhängig voneinander fuzzt und so beispielsweise unnötigen Overhead im Setup des Fuzz-Targets vermeidet. Aber auch dies hat seine Nachteile: Der Tester muss hier nun mit der Struktur des zu testenden Systems bekannt sein, muss das komplexe System in kleinere Systeme selbst zerlegen und dann für jedes der Subsysteme einzelne Fuzz-Targets schreiben - dies ist zeit- und arbeitsaufwändig und, ähnlich wie beim klassischen Testing, anfällig für Fehler des Testers.

2 Checkpoint/Restore

C/R is the idea of saving the state of a some running code to be continued at some later point.

2.1 C/R in anderen Kontexten

This mostly sees use around debugging though there are libraries that make migration of prcoesses between possible.

2.2 C/R Typen und Unterscheidungen

2.2.1 Nach Typ

There are multiple types of C/R tools by what they restore: Running enviroments (Docker), Process sub-trees, single processes, threads.

2.2.2 Nach Capabilities

There are different capabilities they bring to the table, e.g. if they can restore Network connections, and they of course have different efficiencies.

3 CRIU

CRIU is such a tool.

3.1 Funktionsweise

It works in some way.

3.2 Warum CRIU

It has some capabilities that make it interesting.

3.3 Probleme

It does not work out of the box in a fuzzing context, here is how we can fix those issues.

3.4 Performance

Here are estimates for performance.

4 Fuzzing mit C/R

4.1 C/R-Fuzzer

Would be nice, would look something like this. It would have the following behaviour and problems.

4.2 Non-C/R-Fuzzer mit C/R Ansätze

Can we make something work with normal Fuzzers? What could it look like? What could we use it for?

4.2.1 Naiv

4.2.2 Exploration des Zustandsgraphen in Client/Server Systemen

5 CRIU + Fuzzing Engines

Doesn't work, this is why.

6 Fazit und Ausblick

Was können wir grade machen? Nichts! Was bräuchten wir um was machen zu können?