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

TODO: Corpus, Seeds, Dictionaries

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, das heißt 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, insbesondere 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

Mit Checkpoint/Restore, kurz C/R, manchmal auch Checkpoint/Restart, bezeichnet man das Speichern eines Prozesses oder einer Andwendung in einer Art und Weise die eine spätere Wiederherstellung und weitere Ausführung der Anwendung erlaubt. C/R-Systeme finden Anwendung in verschiedenen Bereichen, beispielsweise im Debugging und Fault Recovery durch das Zurücksetzen eines Programmes auf einen vorherigen Zustand, in der Beschleunigung oder Optimierung von Prozessen durch das Fortsetzen von Anwendungen von Checkpoints anstatt rechenzeitaufwändigen Komplettausführungen und dem Aussetzen der Ausführung von Programmen wenn diese grade nicht benötigt werden oder, je nach Fähigkeiten des zugrundeliegenden C/R-Systems, sogar die Migration von laufenden Anwendungen zwischen Maschinen.

TODO: Was alles muss gespeuchert werden?

2.1 C/R Typen und Unterscheidungen

Für das Checkpointing/Restoring von Anwendungen gibt es nicht einen einzelnen Standard sondern eine ganze Reihe von Modellen die wir grob nach zwei Faktoren klassifizieren: Typ bzw. Implemetierungsebene und nach den Fähigkeiten des Checkpoint/Restore-Systems, also welche Aspekte des Programms genau wiederhergestellt werden können und welche nicht.

2.1.1 Nach Typ

Bei der Klassifizierung nach Typ oder Implemetierungsebene unterscheiden wir grob nach drei Kategorien: Das Checkpointen im Userspace, aus dem zu checkpointenen Pro-

gramm selbst, das Checkpointen auf Kernelebene und das Checkpointen von Virtuellen Maschinen beziehungsweise Containern.

Im klassischen Userspacecheckpointing findet das Checkpointing größtenteils transparent gegenüber dem Kernel statt, das heißt das C/R-Tool nutzt aus dem Userspace zugängliche Ressourcen wie /proc/ oder Intercepts von Library- oder Syscalls zum sammeln der notwendigen Checkpoint-Informationen und ebenfalls nur ohne besondere Privilegien zugängige Ressourcen zur Wiederherstellung. Dieses Vorgehen führt in reinen Userspace-C/R-Systemen wie DMTCP (Distributed MultiThreaded CheckPointing) unter Umständen zu einigen Fallstricken oder Problemen, so können zum Beispiel nicht alle Eigenschaften eines Programmes, beispielsweise die PID, nicht ohne weiteres korrekt wiederhergestellt werden, die Intercepts benötigen alternative Libraries die spätestens zur Startzeit mit der Anwendung gelinkt werden müssen, häufig Beschränkungen darauf haben mit welchen Programmen (bspw. Sprachen) sie genutzt werden können und unter Umständen die Funktionsweise des Programmes ändern könnten sowie mögliche Performanceeinbußen durch das duplizieren von Kernelstrukturen im Userspace.

Dem gegenüber steht das Checkpointing auf Kernelebene, beispielsweise BLCR (Berkeley Lab's Linux Checkpoint/Restart) oder Linux-C/R. C/R-Systeme nutzen den Kernel direkt um alle nötigen Informationen wärend des Checkpointings zu sammeln, benötigen also keine Libraries zum Sammeln eben dieser Informationen durch das Abfangen von Syscalls o.ä., und haben alle nötigen Privilegien um, theoretisch, falls implementiert, alle Eigenschaften der Anwendung auch korrekt wiederherzustellen. Allerdings muss der Kernel eben diese Funktionalitäten anbieten damit Kernelebene-C/R-Tools eingesetzt werden können und obwohl es wiederholt versuche der Integration von C/R-Tools in den Mainline Linux Kernel gab ist dies bisher nicht geschehen, das heißt es wird zur Nutzung ein gepatchter Kernel oder ein Kernel mit zusätzlichen Modulen benötigt.
Das Checkpointing mit Kernelebene-C/R-Tools folgt zumeist einem Schema änhlich diesem:

1. Einfrieren und Synchronisieren der Prozesse und Threads
2. Erfassen von globalen Informationen (d.h. Informationen die nicht direkt Teil der zu checkpointenden Anwendung sind wie Namespaces oder Container)
3. Erfassen der Prozess- bzw. Thread-Hierachien und -Informationen wie Parent/Child-Verhältnisse, PIDs, ...
4. Erfassen des Statuses der einzelnen Prozesse und Threads, bspw. Signals, geöffnete Files und FDs, ...
5. Beenden oder Weiterausführung der Anwendung, Schreiben all der erfassten Informationen

Dabei werden all diese Schritte natürlich nicht von der zu checkpointenden Anwendung selbst vorgenommen sondern von einem dafür gespawnten externen Prozess. Das Wiederherstellen funktioniert generell in änhlicher Weise:

1. Wiederherstellen der globalen Information, beispielsweise Container bei Multiprocess-Anwendungen oder dem Prozess bei Multithreaded aber Singleprocess Anwedungen, wenn nötig

2. Erstellen einer Prozess- bzw. Threadhierachie entsprechend der erfassten Hierachie (in eingefrorenem Zustand)

3. Wiederherstellung des Statuses der einzelnen Prozesse und Threads

4. "Fortsetzen"beziehungsweise Auftauen der Prozesse/Anwendung

CRIU (Checkpoint/Restore in Userspace) ist ein Hybrid aus Userspace- und Kernelebene-Systemen, das heißt es agiert größtenteils in Userspace, nutzt jedoch Calls auf privilegierte Operationen wie das Forken mit bestimmten PIDs. All diese von CRIU benötigten Kernel-Capabilities sind seit einiger Zeit im Linux Mainline-Kernel vorhanden. Das Checkpointing und Restoring in CRIU ähnelt stark dem der Kernelebene-C/R-Systeme, benötigt insbesodere keine Intercept-Libraries wie normale Userspace-C/R-Tools, jedoch können nicht all diese Informationen von externen Programmen erfasst oder hergestellt werden. Dies umgeht CRIU indem es mittels ptrace sogennanten parasite code in die zu checkpointenen Prozesse einfügt (beziehungsweise den restorer blob bei der Wiederherstellung) die eben diese Informationen erfassen.

Das Checkpointen von Virtuellen Maschinen oder Containern nutzt zumeist einen der obigen Ansätze um die laufenden Virtuelle Maschine (als Anwendung auf dem Host-System) als ganzes zu Checkpointen. So nutzt Docker beispielsweise CRIU, verwaltet durch entsprechenden Aufrufe des docker Kontrollprogramms `docker checkpoint create [...]` und `docker start --checkpoint [...]`.

2.1.2 Nach Capabilities

Zusätzlich unterscheiden sich verschiedene C/R-Tools auch noch danach was genau sie Checkpointen und Restoren können. Einige Tools wie frühere BLCR-Versionen unterstützen nur einzelne multi- oder singlethread Prozesse, Linux-CR unterstützt das Checkpointen beliebiger Container oder Subtrees von Prozessen. Tools die innerhalb von multithreaded Prozessen arbeiten, beispielsweise einzelne Threads checkpointen und wiederherstellen können existieren anscheinend nicht, zumindest fanden wir keine. Dies ist auch plausibel, da solche Threads sich eben Ressourcen teilen und daher ein teilweises Wiederherstellen nur einiger Threads zu undefiniertem Verhalten führen würde.
Weitere Unterschiede gibt es in der Frage welche Eigenschaften von laufenden Programmen wiederhergestellt werden können, beispielsweise Netzwerkverbindungen wie TCP- oder UDP-Sockets, Dateien oder Namespaces, sowie bezüglich der Frage ob beliebige Programme gecheckpointed werden können oder ob bereits beim Start der Anwendung bekannt sein muss, dass diese gecheckpointed werden soll und entsprechende Vorbereitungen wie das Preloading von Libraries oder das Informieren des Kernels getroffen werden müssen.

3 Fuzzing mit C/R

Aim of this thesis: Answer the question wether we could solve some or all of aboves issues in using fuzzing for security testing by using C/R. In this chapter we will look at a few approaches and their requirements, general feasibility and what issues they would solve. There are two generally possible approaches: 1. Using C/R in a Fuzzer that is not just aware that C/R is happening but does the C/R parts for you and makes active use of C/R for performance 2. Using a "normal"fuzzer and do all the C/R stuff in the fuzztarget

3.1 C/R-Fuzzer

How would this generally work? What issues might exist? What would this solve? How would this impact performance?

3.2 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?

3.2.1 Naiv

3.2.2 Exploration des Zustandsgraphen in Client/Server Systemen

4 CRIU

CRIU is probably the only C/R tools that could make sense right now, two reasons why: 1. It's actually still getting updates unlike most of the other things 2. It's capabilities.

4.1 Funktionsweise

Detailed description of internal functionality of CRIU.

4.2 Capabilities

What can we restore with CRIU?

4.3 Probleme

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

4.4 Performance

Here are estimates for performance.

5 CRIU + Fuzzing Engines

We have a quick look at modern fuzzing engines.
Now, this is where it gets annoying: CRIU doesn't work with pretty much any modern fuzzing engines, here is why. But this wouldn't just be CRIU but any C/R in existence. To fix this we would either need 1. In-Process Thread-Restoring (why wouldn't this be a good idea?) 2. Multi-Process Fuzzing Engines.

6 Fazit und Ausblick