

Checkpoint/Restore in Fuzzing

Begleitseminar Bachelorarbeit

Malte Klaassen

2018-12-14

Inhaltsverzeichnis

- 1 Fuzzing
- 2 Checkpoint/Restore
- 3 C/R-Fuzzing: Ansätze
- 4 CRIU
- 5 C/R-Fuzzing: Implementierung
- 6 Ergebnisse und Ausblick

Fuzzing

- Testen mit (zufälligen) computergenerierten Inputs
- insb. zum Abdecken von Grenzfällen etc.
- Überwachung der Ausführung auf unerwünschtes Verhalten
- Verschiedene Strategien, bspw. zur Erzeugung der Inputs

Fuzzing: Probleme/Beschränkungen

- Effektivität von Fuzzing beruht auf hohem Durchsatz
- Zerlegung komplexer Anwendungen in simple Subsysteme
 - Zusätzlicher Aufwand für mehr Fuzz-Targets
 - Benötigt Kenntnisse über Struktur der Anwendung

Fuzzing: Fuzzing-Engines

- libFuzzer
 - Ziel: Leicht zu nutzende, mächtige Fuzzing-Engine
 - Integration mit LLVM/clang
 - Nutzt clang-Compilerfeatures (Code-Coverage, Sanitizer, Fuzzing-Engine)
- afl
 - Ziel: Mächtige, effiziente Fuzzing-Engine
 - Nutzt einige Compilerfeatures (insb. Code-Coverage, einige Sanitizer)
 - Benötigt eigene Fuzzing-Binary, spezielle Compiler-Versionen, ...

Checkpoint/Restore

- C/R : Speichern des Zustandes einer Anwendung zur späteren Wiederherstellung
- Debugging, Laufzeitorientierung, Lastoptimierung, Migration
- Userspace vs. Kernel vs. Container

C/R: Userspace

- Checkpointen einer Anwendung durch eine Userspace-Anwendungen
- Sammeln/Wiederherstellen der nötigen Informationen durch:
 - Lesen in bspw. `/proc/`
 - Intercept von System- und Library-Calls (bspw. DMTCP)
 - Nutzung von OS-Features bspw. für Speicherinhalte (BLCR, CRIU)

C/R: Kernel

- Checkpointen einer Anwendungen durch den Kernel
- Kernel hat direkten Zugriff auf alle benötigten Informationen
- Benötigt Custom Kernel (Legacy OpenVZ C/R) oder Kernelpatches (Linux-CR)

C/R: Container

- Container oder Virtuelle Maschinen als zu checkpointene Anwendung
- Integration existierender C/R-Tools in Containerverwaltungssystem (Docker: CRIU)

C/R-Tools: Capabilities

- Verschiedene C/R-Tools können verschiedene Features checkpointen und wiederherstellen
 - Threads?
 - Mehrere Prozesse?
 - Netzwerksockets? In welchen Zuständen?
 - PIDs? Namespaces?
- C/R-Tools arbeiten primär auf Prozessen, nicht Threads

C/R-Fuzzing: Ansätze

- Ziel: Lösung von Fuzzing-Problemen mittels C/R-Mechanismen
 - Laufzeitoverhead durch wiederholte Setups
 - Notwendigkeit der Zerlegung in Subsysteme
- Frage: Wo wird C/R-Funktionalität implementiert?
 - Im Fuzzer, transparent gegenüber dem Fuzz-Target
 - Im Fuzz-Target, ohne C/R-Support des Fuzzers
- Frage: Was für C/R-Funktionalität?

C/R-Fuzzing: Bisherige Ansätze

- afl Forkserver
 - Einfrieren des Fuzz-Target-Prozesses nach `execve`, Linking, Initialisierung (oder noch später mit `__AFL_INIT()` ;)
 - Fuzzing findet auf Copy-on-Write-fork dieses Prozesses statt
 - Forkserver muss vor Nutzung von Childprocesses, Threads, Filedeskriptoren, ... aufgesetzt werden → Reduktion des Setupoverheads aber nur bei simplem Overhead
- Mit mächtigen C/R-Tools: Keine (bekannt)

C/R-Fuzzing: "Naiver" C/R-Ansatz

- Ähnlich zu afl Forkserver
- Zerlege Fuzz-Target in Setup und eigentlichen Test
- Führe das Setup einmal aus, checkpointe den Prozess, steige für weitere Ausführungen an dem Checkpoint wieder ein
- Implementiert durch den Fuzzer mit entsprechendem Breakpoint in Fuzz-Target oder implementiert im Fuzz-Target selbst
- Laufzeitgewinn pro Iteration:

$$\begin{aligned}\frac{T_{Non-C/R}(n) - T_{C/R}(n)}{n} &= \frac{(n-1)T_{Setup} - T_{Checkpoint} - nT_{Restore}}{n} \\ &=_{n \rightarrow \infty} T_{Setup} - T_{Restore}\end{aligned}\tag{1}$$

Naiver C/R-Ansatz: C/R-Fuzzer

- Mächtigere Variante des afl Forkservers
- Inputübergabe muss nach dem Checkpoint passieren
- Welche Codesegmente in den Setup-Teil verlagert werden können wird durch das C/R-Tool bedingt
- $T_{Restore}$ muss klein sein damit ein Laufzeitgewinn vorliegt
- Parallel Fuzzing? Timeouts?

Naiver C/R-Ansatz: C/R-Fuzz-Target

- Gleiche Überlegungen wie beim C/R-Fuzzer, zusätzlich:
- Kompatibilität von C/R-Tool und Fuzzer muss gewährleistet sein
 - Genutzte Sanitizer, Fuzzing von Multi-Process-Anwendungen, Restore-In-Place, ...
- Übergabe des Inputs, Speicherung des Zustands, ...

C/R-Fuzzing: Exploration eines Zustandsgraphen

- Ziel: Effizientes Fuzzing einer komplexen Anwendung mit mehrstufigen Eingaben und wohldefiniertem Zustandsgraphen
 - Bspw. Implementierung eines Netzwerkprotokolls
- Ansatz: Exploration des Zustandsgraphen durch Anwendung von neuen Inputs auf bereits bekannte Zustände
- Checkpointing der Anwendung bei neu gefundenen Zuständen → Ausführung mit neuen Inputs kann direkt dort fortgeführt werden

C/R-Fuzzing: Exploration eines Zustandsgraphen

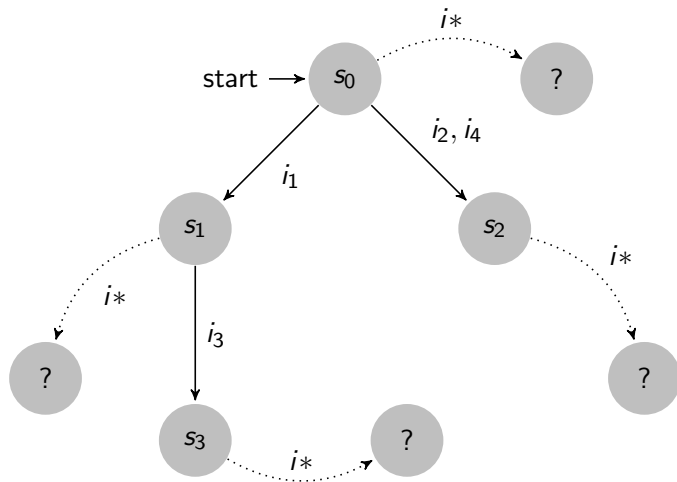


Abbildung: Zustandsübergangsexploration mit Input i^*

C/R-Fuzzing: Exploration eines Zustandsgraphen

- Woran erkennt man einen Zustand?
- Übergabe des Inputs?
- Welche Mittel stehen zur geführten Generierungen von Inputs zur Verfügung?
- Bei einer Implementierung des C/R im Fuzz-Target:
 - Persistente Speicherung der Zustände
 - Anwendung der Inputs auf verschiedene Serverzustände?
 - Wie bei dem naiven Ansatz: Kompatibilitätsprobleme

CRIU

- "Checkpoint/Restore in Userspace"
- Als Userspace-Ersatz zu OpenVZ-Kernel-CR entwickelt (2011)
- Viele unterstützte Features
- Wird noch aktiv unterstützt
- Nutzt unveränderte Binaries, benötigt keine speziellen Initialisierungen bei Programmstart
- Nutzung entweder über RPC/C-API oder Commandline-Tool

CRIU: Funktionsweise

- CRIU arbeitet (größtenteils) im Userspace
- Nutzt einige Kernelfeatures und privilegierte Operationen (seit 3.11 im Mainline Linux-Kernel)
 - ptrace, CONFIG_CHECKPOINT_RESTORE u.A. für prctl
 - CONFIG_NAMESPACES sowie weitere Namespace-Features
 - Socketmonitoring

CRIU: Funktionsweise - Checkpointing

- Einfrieren des Prozessbaumes (freezer cgroup oder ptrace)
- Extraktion der Prozessinformationen
 - Extern durch Lesen von /proc/ und ptrace
 - Intern durch Injektion eines Parasite-Blobs mittels ptrace
- Schreiben des Images, Auftauen oder Beenden der Anwendung

CRIU: Funktionsweise - Restore

- Zerlegung des Images in einzelne Prozesse und Zuweisung von Shared Ressources
- Erstellung eines entsprechenden Prozessbaumes durch Forken der CRIU-Anwendung
- Wiederherstellen der Prozessinformationen
 - Extern durch die CRIU-Anwendung (bspw. Speicherinhalte, Sockets, Namespaces)
 - Intern durch die Nutzung eines Restorer-Blobs in den geforkten Prozessen
- Unmapping des Restorer-Contextes, Fortsetzen der Anwendung

CRIU: CRIU+Fuzzing?

- CRIU wurde nicht gezielt für Fuzzing entwickelt
- Es ergeben sich einige (lösbare) Probleme:
 - Wiederholtes Wiederherstellen von Anwendungen mit etablierter TCP-Verbindung → Firewall-Regel
 - Genutzte PID bereits wieder neu vergeben → Isolation, bspw. mit PID-Namespace
 - TCP-Timeouts → Vergrößerung des Fensters, Manipulation des Images
- Performance?

CRIU: Performance

- Performance, insb. $T_{Restore}$ von hoher Relevanz für C/R-Fuzzing

Iterationen	1byte	10^2 byte	10^4 byte	10^6 byte	10^8 byte	TCP
10	0.292s	0.290s	0.292s	0.301s	0.627s	0.982s
50	1.269s	1.269s	1.256s	1.298s	2.332s	4.615s
100	2.459s	2.494s	2.477s	2.558s	4.482s	9.170s
500	12.205s	12.228s	12.250s	12.648s	21.564s	45.541s
$T_{Restore}$	0.024s	0.024s	0.024s	0.025s	0.043s	0.091s

- insb. $T_{Restore} \geq 24ms$ (auf diesem Laptop)
- Linux-CR erreicht Restore-Zeiten von $\leq 1ms$

C/R-Fuzzing: Implementierungsversuche

libfuzzer + CRIU

afl + CRIU

Ergebnisse

Ausblick



Lesser-known features of afl-fuzz; Michał Zalewski
[https://lcamtuf.blogspot.com/2015/05/
lesser-known-features-of-afl-fuzz.html](https://lcamtuf.blogspot.com/2015/05/lesser-known-features-of-afl-fuzz.html)