HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
NETWORKED SOLUTIONS

Google OSS-Fuzz

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2017

Autor(en): Sven Defatsch
Betreuer: Tobias Brunner / Prof. Dr. Andreas Steffen

Abstract

Im Jahr 2016 hat Google das sogenannte OSS-Fuzz Programm angekündigt, welches sich zum Ziel gemacht hat, ausgewählte Open Source Projekte sicherer zu machen. Dazu wird die Möglichkeit angeboten, die enorme Rechenleistung der Google Infrastruktur für kontinuierliches Fuzzing, inklusive übersichtlichem Reporting, zu nutzen. Die Open Source IPSec Umsetzung strongSwan wird an der HSR entwickelt und weltweit eingesetzt. Für eine Software im Sicherheitsbereich ist es besonders wichtig, die Anzahl an Fehlern im Code so gering wie möglich zu halten. Fuzzing ist eine gute Möglichkeit, latente Fehler zu entdecken und somit die Codequalität zu steigern.

Für das Fuzzing wurde libFuzzer eingesetzt, da es zu diesem Zeitpunkt die einzige von Google unterstütze Fuzzing Engine ist. Als ein vielversprechendes Fuzz Target stellte sich der Zertifikatsparser heraus, für welchen sich ein Fuzz Target relativ leicht umsetzen liess. Das Fuzzing wurde zuerst lokal getestet, um Erfahrungen im Umgang mit der Nutzung von libFuzzer zu sammeln und dabei Änderungen, welche im strongSwan Build Prozess nötig sind, aufzudecken. Anschliessend wurde das Fuzzing auf die Google Infrastruktur verlagert, um den Code auf Herz und Nieren zu testen.

Ein Resultat dieser Arbeit ist die Erkenntnis über die Integration von strongSwan in das Google OSS-Fuzz Projekt und auf welche Stolpersteine dabei besonders geachtet werden muss. Des Weiteren sind beim Fuzzing selbst einige Fehler unterschiedlicher Tragweite im Code von strongSwan aufgedeckt worden, welche bereits behoben und in Form von Patches wieder eingeflossen sind. Somit ist das Hauptziel des Fuzzings, die Verbesserung der Codequalität, zu einem Teil bereits erreicht worden.

Inhaltsverzeichnis

I. Einleitung

Aufgabenstellung

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

Studienarbeit 2017

Google OSS-Fuzz - Security Testing des strongSwan Codes

Student: Sven Defatsch

Betreuer: Tobias Brunner / Prof. Dr. Andreas Steffen

Ausgabe: Montag, 22. Februar 2017

Abgabe: Freitag, 2. Juni 2017

Einführung

Die Google OSS-Fuzz Initiative [1] gibt Open Source Projekten die Möglichkeit ihren Source Code mit Hilfe moderner und mächtiger Fuzzing [2] Methoden auf Sicherheitsprobleme zu testen, mit dem Ziel eine höhere Software-Qualität zu erreichen. In dieser Arbeit sollen im strongSwan Source Code [3] geeignete Fuzz Targets definiert und in den strongSwan Build Prozess integriert werden.

Aufgabenstellung

- Einarbeiten in die Google OSS-Fuzz Testumgebung und in die Funktionsweise moderner Fuzzing-Tools.
- Lokales Fuzzing des strongSwan X.509 Zertifikatsparsers.
- Upload des strongSwan X.509 Fuzzing Target auf die Google OSS-Fuzz Plattform.
- Dokumentation des Fuzzing Test Setups (lokal und unter Google OSS-Fuzz)
- Dokumentation der erzielten Resultate, insbesondere von allfällig gefundenen Bugs und Schwachstellen.
- **Optional:** Definition und Testen weiterer strongSwan Fuzz Targets.

Links

[1] Google OSS-Fuzz Info Page
https://github.com/google/oss-fuzz

[2] Wikipedia “Fuzzing” Definition
https://en.wikipedia.org/wiki/Fuzz_testing

[3] strongSwan Project - Github Repository
https://github.com/strongswan/strongswan

Rapperswil, 22. Februar 2017

a. steffen

Prof. Dr. Andreas Steffen

INS -Institute for Networked Solutions 22.02.2017 Steffen

Management Summary

Ausgangslage

Diese Arbeit befasst sich mit der Frage, wie die freiverfügbare Netzwerk Software strongSwan, welche an der HSR entwickelt wird, mittels Fuzzing sicherer gemacht werden kann. Fuzzing beschreibt einen Prozess, bei welchem ein Programm mit zufälligen Daten gefüttert wird, um damit ein eventuelles Fehlerverhalten der Software auszulösen. Es wäre natürlich wünschenswert, wenn keinerlei Fehler auftauchen würden. Aber Software wird von Menschen gemacht und ist daher nicht perfekt. Es gibt in vielen Programmen Kombinationen von Eingabewerten, die zu einem Fehler führen würden. Solche Werte können von einer Person fast nicht ausgedacht werden, weil diese zum Teil äusserst absurd sind. Fuzzing findet genau diese Fehler. Aus dem Fehlverhalten der Software können Entwickler lernen, um ähnliche Fehler nicht wieder zu begehen und somit die eigene Software robuster gestalten. Besonders im Rahmen einer Software im Sicherheitsbereich, wie strongSwan, spielt die Robustheit eine entscheidende Rolle.

Fuzzing an sich ist zu vergleichen mit der Suche von der Nadel im Heuhaufen. Dies wiederum bedeutet, dass viel Rechenleistung dafür gebraucht wird, welche sich nicht alle Softwareentwickler leisten können. Der Internet Riese Google hat im Jahr 2016 ein Projekt gestartet, um einer ausgewählten Anzahl an Software Projekten die enorme Rechenleistung von Google für diesen Prozess kostenfrei zur Verfügung zu stellen. Auch sonst wird der Prozess von Anfang bis Schluss von Google unterstützt.

Vorgehen und Technologien

Beim Fuzzing wird die Zielsoftware zuerst genau angeschaut und alle Bestandteile, welche die Software ausmachen analysiert. Einer dieser Teile war der Zertifikatsparser, welcher für den Anfang sehr vielversprechend erschien. Um diesen Softwareteil mit möglichst vielen Zufallswerten zu testen, braucht es ein ausgeklügeltes Programm, welches dies übernimmt, da es nicht sinnvoll, ist x-beliebige Werte einzuspielen. Google gibt dafür den sogenannten libFuzzer vor. Mit dieser Vorgabe wurde also das Vorgehen zuerst auf dem eigenen Computer ausprobiert, um allfällige Stolpersteine aus dem Weg zu räumen, bevor das ganze Prozedere danach auf die Computer von Google verlegt wurde.

Ergebnisse

Ein Ergebnis der Arbeit sind die gewonnenen Erkenntnisse über die Integration von Fuzzing in strongSwan und wo es dabei überall zu Problemen kommen kann. Die Integration ist nicht ohne Probleme verlaufen und daher konnte auch geklärt werden, wie eine entsprechende Lösung aussehen kann. Weitere Ergebnisse sind tatsächlich gefundene Fehler verschiedenen Ausmasses im strongSwan Code, welche mittlerweile von den Entwicklern behoben werden konnten. Somit konnte die Codequalität wie gewünscht bereits verbessert werden.

Ausblick

Für die Zukunft können die gemachten Erfahrungen dieser Arbeit genutzt werden, um weitere Ziele in strongSwan auszuwählen und mittels Fuzzing zu testen. Somit kann die Qualität stetig weiter gesteigert werden.

II. Technischer Bericht

1 Fuzz Testing

Fuzz Testing ist eine Technik um Software auf Fehler zu überprüfen. Dabei werden ausgewählten Funktionen Zufallswerte übergeben bis, falls vorhanden, ein latenter Bug das Programm zum Absturz bringt. Besonders bei sicherheitskritischen Softwareprojekten ist es essentiell, die Fehlerrate möglichst tief und damit die Angriffsfläche klein zu halten. Ein solches Problem kann schwerwiegende Folgen haben, was spätestens seit der „Heartbleed" Lücke in OpenSSL bekannt sein sollte. Im Nachhinein hat sich gezeigt, dass der verantwortliche Buffer Overflow mittels Fuzzing relativ schnell entdeckt worden wäre.[1]

Fuzz Testing funktioniert automatisiert und kann in Kombination mit genügend Rechenleistung eine immense Menge an Inputs pro Sekunde verarbeiten, was der Technik viel Potential einräumt.
Fuzzing ist bereits länger bekannt und wurde bereits Ende der 80er Jahre an der Universität von Wisconsin in den USA beschrieben.[2] Über die Jahre hinweg sind die Techniken aber um einiges ausgereifter geworden.

1.1 Fuzzing Arten[3]

1.1.1 Mutation Based Fuzzing

Bei dieser Technik werden gültige Eingabewerte genutzt, um daraus wieder neue Inputs zu generieren. In der Hoffnung, gültige Inputwerte zu erzeugen, mutiert die Fuzzing Engine kleine Teile des Inputs und nutzt diese erneut für das Fuzzing. Hier ist darauf zu achten, dass völlig zufällige Mutationen zu ungültigem Input führen können. Strukturen, welche Prüfsummen nutzen, wie zum Beispiel TCP/IP, dürfen nur so mutiert werden, dass die Prüfsumme ebenfalls verändert wird. Dasselbe gilt für Programme wie XML Parser, welche mit stark strukturierten Eingabewerten arbeiten.

1.1.2 Generation Based Fuzzing

Als Generation Based Fuzzing werden Techniken bezeichnet, welche die Eingabewerte von Grund auf selbst erzeugen, anstatt gültige Werte zu mutieren. Dabei sind einige Einschränkungen bezüglich gültigem Inputformat zu empfehlen, sodass die Werte nicht zu 100% aus zufälligen Zeichen sind. Dies würde funktionieren, macht aber in den wenigsten Fällen Sinn. Software sollte immer Gültigkeitsprüfungen von Inputs besitzen (über Prüfsummen oder Struktur), welche solche Inputs gar nicht erst zulassen. Als Beispiel dafür dient wieder ein XML Parser. XML besteht aus vordefinierten Schlüsselwörter und aus diesen lassen sich, unter Einhaltung der XML Richtlinien, automatisch gültige XML Strukturen generieren.

1.1.3 Evolutionäres Fuzzing

Evolutionäres Fuzzing kann als eine Mischung von Generation Based und/oder Mutation Based Fuzzing mit zusätzlichen Techniken angesehen werden. Während der Arbeit wurde auf libFuzzer gesetzt, welcher in dieser Kategorie angesiedelt ist. Zur Laufzeit wird die Codeabdeckung überwacht, welche von den Eingabewerten erreicht wird. Dabei lernt die Fuzzing Engine laufend hinzu, mit was für Inputs ein bestimmter Codepfad abgedeckt werden kann und was für Mutationen an den Eingabewerten dafür nötig sind. Dadurch kann schlussendlich die Codeabdeckung des gesamten Fuzzing Vorgangs gesteigert werden.

[1] (https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html)
[2] (http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf)
[3] (https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/)

1.2 Vorteile

Fuzzing dient nicht als Ersatz für normales Testing, sondern als Stresstest. Ein Computer kann enorm viele Inputs pro Sekunde an ein Programm oder Funktion übergeben und somit Randfälle entdecken. Das können Fälle sein, welche für den Menschen auf den ersten Blick nicht zu sehen waren, oder schlicht übersehen wurden. Es besteht die Möglichkeit, verschiedenste Codepfade automatisiert zu testen.

1.3 Nachteile

Es ist nicht immer mit wenig Aufwand verbunden, ein geeignetes Ziel ausfindig zu machen und dieses mit den bestmöglichen Werten zu testen. Des Weiteren können zu strikte Definitionen der erlaubten Inputwerte dazu führen, dass Fehler, welche mit völligen Zufallswerten aufgetreten wären, nie gefunden werden.

2 Google OSS-Fuzzing

Im Dezember 2006 hat Google das OSS-Fuzz Programm angekündigt.[4] Der Begriff OSS steht dabei für „Open Source Software“ und das erklärte Ziel ist, weitverbreitete Software sicherer zu machen. Google selbst setzt für seine Programme wie z.B. den Chrome Webbrowser ebenfalls schon länger auf Fuzzing. Viele Open Source Software Projekte sind heutzutage weltweit im Einsatz und dabei auch oft in kritischen Bereichen im Hintergrund anzutreffen (Bsp. OpenSSL, strongSwan). Google möchte solchen Software Projekten die Ressourcen zur Verfügung stellen, welche ein kontinuierliches Fuzzing benötigt. Zum Zeitpunkt dieses Dokumentes sind ca. 70 Open Source Projekte für OSS-Fuzz zugelassen.[5]

2.1 Prozess

Fuzzing soll dank des Projektes einen einheitlichen Prozess erhalten und die dafür nötige Infrastruktur wird kostenlos zur Verfügung gestellt. Der Prozess ist in 6 folgende Schritte unterteilt.

Der Prozess im Detail (inkl. Uploads und Synchronisationen) wird in diesem Bild übersichtlich illustriert.

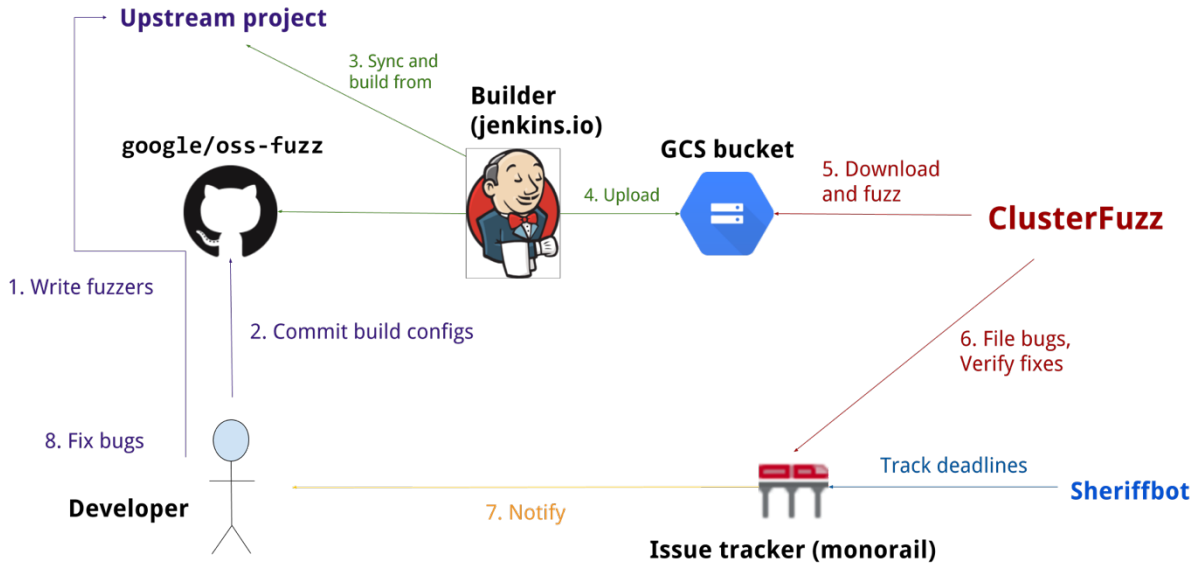


Abbildung 1 OSS-Fuzz Prozess[6]

[4] (https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html)
[5] (https://github.com/google/oss-fuzz/tree/master/projects)
[6] (https://github.com/google/oss-fuzz)

2.1.1 Google akzeptiert Teilnahme

Damit eine Software dem Projekt beitreten kann, muss eine berechtigte Person (benötigt Zugriff auf das eigene Git Repository) OSS-Fuzz „forken“. Dem Projekt anzufügen sind einige Daten wie ein Ordner mit dem Projektnamen und eine „project.yaml“ Datei mit Kontaktdaten zum Projekt. Nach einem Pull Request muss die Antwort von Google abgewartet werden

Dieser Schritt wurde vorgängig von Tobias Brunner erledigt. Weitere Dateien mussten zu einem späteren Zeitpunkt noch erstellt und bereitgestellt werden.

```
5 lines (4 sloc)  120 Bytes
1  homepage: "https://www.strongswan.org"
2  primary_contact: "security@strongswan.org"
3  auto_ccs:
4    - "tobias@strongswan.org"
```

Abbildung 2 Inhalt von „project.yaml“ für strongSwan

2.1.2 Erstellen der Fuzzer

Als Fuzzer wird eine ausführbare Datei bezeichnet, welche ein bestimmtes Fuzz Target testet. Ein Fuzz Target ist eine oder mehrere Funktionen, welche getestet werden. Die Fuzz Targets werden in den eigenen Build Prozess integriert und in das eigene Git Repository geladen.

2.1.3 Gefundene Bugs

Die Fuzzer werden automatisch erstellt und auf dem Cluster von Goolge ausgeführt. Wird dabei ein Fehler entdeckt, löst dies die Erstellung eines Tickets auf einem separaten Issue Tracker aus und der in „project.yaml“ eingetragene Kontakt wird benachrichtigt. Das Problem ist nicht öffentlich und nur von den Projektbetreuern einsehbar.

2.1.4 Behebung von Bugs

Ein Entwickler nimmt sich den Bug an und behebt diesen. In der Commit Message muss der String „Credit to OSS-Fuzz“ stehen.

2.1.5 Verifikation

Der Build Vorgang wird vom Cluster in unregelmässigen Abständen ausgeführt. Dabei wird sowohl das Fuzzing mit der neuen Version gestartet, so wie auch Regressionstests durchgeführt. So werden Werte, welche das Programm vorher zum Abstürzen gebracht haben, mit der neusten Programmversion getestet. Wird dabei festgestellt, dass der Fehler nicht mehr auftritt, wird im Issue Tracker ein Kommentar dazu eingetragen und der Eintrag geschlossen.

2.1.6 Veröffentlichung

Es gibt definierte Richtlinien, wann ein gefundener Fehler der Öffentlichkeit zugänglich gemacht wird, so sollen Zero-Day Lücken vermieden werden. Im Detail werden folgende Punkte beschrieben.

2.1.6.1 Deadline

Wurde ein Problem festgestellt, hat ein Entwickler 90 Tage Zeit für dessen Behebung. Nach Ablauf dieser Frist wird der Fehler veröffentlicht. Wird das Problem früher als in den geforderten 90 Tagen behoben erfolgt die Veröffentlichung, nach gemachter Verifikation des Issue’s, ebenfalls.

2.1.6.2 Wochenende und Feiertage

Läuft eine Frist am Wochenende oder an allgemeinen Feiertagen aus, wird sie von selbst auf den nächsten Arbeitstag verschoben.

2.1.6.3 Gnadenfrist

Reichen 90 Tage für die Behebung nicht aus, können weitere 14 Tage Aufschub beantragt werden.

2.2 Vorteile dank OSS-Fuzz

Grundsätzlich hält einen nichts davon ab, seine Fuzz Targets zu schreiben und lokal zu testen. Das OSS-Fuzz Projekt bietet allerdings ein kontinuierliches Fuzzing mit hohem Automatisierungsgrad, welches von der Rechenleistung, Error Reporting bis hin zu Regressionstests alles zur Verfügung stellt. Das Fuzzing selbst läuft auf „ClusterFuzz“, dies ist Google’s verteilte Infrastruktur und liefert die Rechenleistung. Die enorme Rechenkraft ist ein Hauptvorteil und ermöglicht den Entwicklern ein x-faches mehr an Instruktionen pro Sekunde, als es lokal möglich wäre.

2.3 Unterstützte Fuzzer und Programmiersprachen

Das Projekt steht noch in der Beta Phase und unterstützt zum jetzigen Zeitpunkt als Fuzzing Engine nur libFuzzer von LLVM. Für die Zukunft ist der Support von weiteren Engines geplant, allen voraus für „American Fuzzy Lop“ (AFL).

Als unterstützte Programmiersprachen werden zur Zeit C und C++ angeboten, wobei diese Vorgabe von libFuzzer herkommt. Fuzzing macht bei diesen Programmiersprachen besonders Sinn, da das Memory Management dem Entwickler überlassen wird.

3 LibFuzzer

LibFuzzer ist ein “in-process, coverage-guided, evolutionary Fuzzing“ Programm[7] für C und C++ Code. Gemäss der Definition aus Kapitel 1 arbeitet libFuzzer mittels Mutationen von gültigen Inputs, welche auf Fuzz Targets angewandt werden. LibFuzzer merkt sich während der Laufzeit, welche Codeblöcke erreicht wurden und generiert daraus neue Mutationen, um diese wieder als neue Inputwerte zu erhalten und so die Pfadabdeckung in die Höhe zu treiben. Entwickelt wird libFuzzer von LLVM[8], dies sind dieselben Entwickler wie des Compilers Clang und sind beide unter Linux ausführbar.

3.1 Erstellen eines Fuzzers

Ein Fuzzer ist ein eigenständiges Programm, welches ein anderes Programm oder eine Funktion testet. Eine solche Datei muss zuerst für jedes Fuzz Target erstellt werden.
Dafür müssen die Fuzz Targets mit Clang speziell kompiliert und libFuzzer als statische Bibliothek eingebunden werden.

3.1.1 Erstellen der libFuzzer Bibliothek

Um die Library „libFuzzer.a“ zu erstellen wird das LLVM Repository mit Git geklont:

```
git clone https://chromium.googlesource.com/chromium/llvm-project/llvm/lib/Fuzzer
```

Im entsprechenden Ordner findet sich ein „build.sh“ Script welches die Library erstellt:

```
./Fuzzer/build.sh
```

3.1.2 Kompilieren eines Beispiels

Die eigenen Fuzz Target müssen nun jeweils mit der libFuzzer Library beim kompilieren zusammen gelinkt werden. Als weitere Argumente können Sanitizers angegeben werden, welche weitere Bugs zur Laufzeit finden können. Da libFuzzer eine „coverage-guided“ Fuzzing Engine ist, muss als Option für Clang ebenfalls die Code Coverage hinzugeschaltet werden.

```
clang -fsanitize-coverage=trace-pc-guard -fsanitize=address
library1.c library2.c fuzz_target.c libFuzzer.a -o fuzzer
```

[7] (http://llvm.org/docs/LibFuzzer.html#introduction)
[8] (http://www.llvm.org/)

3.2 Interface und Erklärung

Im folgenden Bild ist ein Beispiel zu sehen, was für Output auf der Konsole generiert wird, wenn ein Fuzzer läuft. Anhand des Bildes wird die Bedeutung der Einzelnen Zeilen und Output erklärt.

```
INFO: Seed: 3377016662
INFO: Loaded 2 modules (8042 guards): [0x7f5a95291360, 0x7f5a952990f4), [0x74cf30, 0x74cf44),
Loading corpus dir: ../../CORPUS/CT_generated/
Loading corpus dir: ../../CORPUS/CT_clean/
INFO: -max_len is not provided, using 2475
#0      READ units: 1135
#1024   pulse  cov: 1124 ft: 1757 corp: 122/132Kb exec/s: 512 rss: 234Mb
#1135   INITED cov: 1144 ft: 1857 corp: 153/196Kb exec/s: 567 rss: 255Mb
#1462   NEW    cov: 1147 ft: 1860 corp: 154/197Kb exec/s: 487 rss: 309Mb L: 1429 MS: 2 ChangeASCIIInt-ChangeByte-
#2048   pulse  cov: 1147 ft: 1860 corp: 154/197Kb exec/s: 512 rss: 406Mb
#2177   NEW    cov: 1148 ft: 1861 corp: 155/199Kb exec/s: 435 rss: 427Mb L: 1403 MS: 2 ChangeByte-ShuffleBytes-
#2422   NEW    cov: 1148 ft: 1862 corp: 156/201Kb exec/s: 484 rss: 455Mb L: 2475 MS: 2 ChangeBinInt-ShuffleBytes-
#2447   NEW    cov: 1149 ft: 1863 corp: 157/202Kb exec/s: 489 rss: 455Mb L: 1362 MS: 2 ChangeBinInt-ChangeBinInt-
#2799   NEW    cov: 1149 ft: 1864 corp: 158/204Kb exec/s: 466 rss: 475Mb L: 1429 MS: 4 ChangeASCIIInt-ShuffleBytes
#3134   NEW    cov: 1150 ft: 1867 corp: 159/205Kb exec/s: 447 rss: 480Mb L: 1429 MS: 4 CMP-CMP-ChangeByte-ChangeBy
#3313   NEW    cov: 1151 ft: 1868 corp: 160/206Kb exec/s: 473 rss: 481Mb L: 1122 MS: 3 ChangeBit-ShuffleBytes-Shuf
#3406   NEW    cov: 1151 ft: 1875 corp: 161/208Kb exec/s: 486 rss: 484Mb L: 1536 MS: 1 CopyPart-
#4096   pulse  cov: 1151 ft: 1875 corp: 161/208Kb exec/s: 455 rss: 492Mb
#4208   NEW    cov: 1151 ft: 1883 corp: 162/209Kb exec/s: 467 rss: 492Mb L: 1217 MS: 3 ChangeBinInt-InsertRepeated
```

Abbildung 3 libFuzzer Interface und Output

Auf dem Bild sind nicht alle möglichen Informationen zu sehen, weil diese zum Teil seltener auftauchen. Der Vollständigkeit wegen werden diese ebenfalls erklärt. Die Informationen stammen zum Teil aus der Hilfe und wurden durch eigene Kommentare ergänzt.[9]

Begriff	Bedeutung
INFO Seed:	Der Fuzzer wurde mit dem Random Seed „3377016662“ gestartet.
INFO: (XXX Guards)	LibFuzzer nutzt für die Coverage „SanitizerCoverage“ von LLVM. Der Compiler fügt bei jeder Funktion, Basic-Block und Edge eine Instruktion ein, um die Coverage bei einem ausgeführten Programm zu verfolgen.[10] Diese Instruktionen werden Guards genannt und deren Anzahl wird bei jedem Start ausgegeben. Somit hat jede Software eine unterschiedliche Anzahl Guards.
Loading corpus dir:	Jeder Corpus Ordner, welcher beim Aufruf angegeben wird, ist hier in der entsprechenden Aufrufs Reihenfolge aufgelistet.
INFO: -max_len is not provided	Dem Fuzzer wurden keine Grössenbegrenzungen für Inputwerte angegeben. Als Standardwert nimmt libFuzzer immer die Grösse von 64 Bytes oder kleiner an. Sind grössere Dateien im Corpus und keine Länge angegeben, wird die Grösse des grössten Files genommen, hier 2475 Bytes.
READ units:	Der Fuzzer hat insgesamt 1135 Files in den Corpus Ordnern gefunden.
#*Beliebige Zahl*	Alle Zahlen nach einem # geben an, der wievielte Input es ist. Bei der letzten Zeile waren es bereits 4208.
INITED	Der Fuzzer ist bereit für die Arbeit.
NEW	Der Fuzzer hat einen neuen Eingabewert generiert, welcher einen neuen Bereich im Code aufgedeckt hat. Somit gilt ein solcher Input als „interessant“ und wird in das erste angegebene Corpus gespeichert.
pulse	Der Fuzzer überprüft seine eigene Funktionalität alle 2^n Inputs mit einem solchen leeren Input.
DONE	Der Fuzzer hat alle Operationen abgeschlossen oder die Zeit ist abgelaufen. Diese Meldung erscheint nur, falls ein Zeitlimit oder die maximale Anzahl Inputs vorgegeben sind.
RELOAD	Wenn mehrere Fuzzer parallel laufen (mehrere Jobs), generiert jeder davon eigene neue Eingabewerte. Von Zeit zu Zeit lädt ein Prozess das Corpus neu nach, um auch die Inputs zu bekommen, welche andere Jobs erzeugt haben.

[9] (http://llvm.org/docs/LibFuzzer.html#output)
[10] (https://clang.llvm.org/docs/SanitizerCoverage.html#tracing-pcs-with-guards)

cov:	Anzahl an Code Blocks, welche unter Verwendung der aktuellen Corpora abgedeckt wurden.
ft:	libFuzzer nutzt verschiedene Techniken, um die Code Coverage herauszufinden und alle diese Techniken liefern Feedback. Die Summe der Feedbacks ergibt diese Zahl und dient der Statistik.
corp:	Anzahl der Corpus Einträge, welche gerade im Memory gehalten werden inkl. deren Grösse. Jeder „NEW Eintrag“ zählt diesen Zähler hoch.
exec/s:	Anzahl an Fuzzing Operationen in der Sekunde.
rss:	Aktueller Memory Verbrauch. Sollte diese Zahl grösser als das Argument „-rss_limit_mb“ werden bricht der Vorgang ab.
L:	Tritt nur bei einem „NEW Event“ auf und gibt die Grösse des neuen Inputs in Bytes an.
MS:	Tritt nur bei einem „NEW Event“ auf. Die Zahl davor gibt an, wie viele Mutationsoperationen durchgeführt wurden und dahinter stehen die einzelnen Operationen.

Tabelle 1 Erklärung des Fuzzer Outputs

3.3 Verfügbare Optionen

Die Liste aller verfügbaren Command Line Optionen ist in der Hilfe dokumentiert.[11] Hier wird auf die für die Arbeit relevanten eingegangen.

Option	Bedeutung
-seed	Random Seed, von welchem gestartet wird um Zufallsmutationen zu generieren. Durch diese Option kann vom selben Seed gestartet werden um dieselben Resultate zu erzielen.
-max_len	Maximale Grösse eines Inputs. Ohne Angabe versucht libFuzzer anhand der Dateien im Corpus eine geeignete Grösse zu wählen. (Sucht das grösste File)
-rss_limit_mb	Memory Verbrauchslimit in MB. Sobald ein Input mehr Memory verbraucht wird dieser abgebrochen. Standard ist 2048
-merge	Minimiert Corpus. Siehe Kapitel 3.5. Standard ist 0
-jobs	Anzahl Jobs, welche durchgeführt werden. Sollte ein Job abbrechen, wird ein neuer erstellt, bis diese Zahl erreicht wurde. 1 Job wird jeweils in einem Worker Prozess ausgeführt. Standard ist 1. Wird die Option „-workers“ nicht gleichzeitig gesetzt sind maximal „Anzahl CPU Cores/2“ Jobs möglich.
-workers	Anzahl an Prozessen, welche Fuzzing Jobs durchführen, pro Worker 1 Job. Wenn die Option nicht gesetzt wird, was Standard ist, ergibt sich die Anzahl gleichzeitiger Prozesse wie folgt: min(jobs, (Anzahl CPU Cores/2))
-dict	Pfad zu einem Wörterbuch für das spezifische Fuzz Target.
-minimize_crash	Minimiert Crash Dump. Siehe Kapitel 3.6.2.

Tabelle 2 libFuzzer Command Line Options

[11] (http://llvm.org/docs/LibFuzzer.html#options)

3.4 Seed Corpus

Effizientes Fuzzing setzt voraus, dass die Inputwerte nicht zu 100% aus Zufallswerten bestehen. Als moderner Fuzzer mutiert libFuzzer daher gütige Eingabewerte und testet diese danach erneut. Gültige Eingabewerte können verschiedensten Ursprungs sein, zusammengefasst werden diese Startwerte Seed Corpus genannt. Vorzugsweise beinhaltet der Seed Corpus vielfältige Werte, welche möglichst viele verschiedene Codepfade auslösen und somit die Codeabdeckung erhöhen. Dafür wiederrum ist es notwendig, den zu testenden Code genau zu kennen, andernfalls können gute und abwechslungsreiche Inputwerte kaum erzeugt werden. Je nachdem wie die Funktion aussieht, ist das ein leichteres oder schwereres Unterfangen.

3.4.1 Verwendung

Um mit einem Seed Corpus zu arbeiten, wird dem Fuzzer der Pfad zum Ordner als Argument übergeben. Der Fuzzer liest Inputs rekursiv von dem Ordner ein und speichert bei der Ausführung neue Eingabewerte im selben Ordner ab.

```
./fuzzer PATH_TO_SEED_CORPUS
```

Der Vorgang verhält sich anders, sobald mehrere Corpora angegeben werden.
Es werden alle Dateien in allen Ordnern rekursiv abgesucht und dem Fuzzer übergeben. Neue, interessante Inputs werden bei dieser Methode aber nur in einem, dem ersten Ordner gespeichert. (Hier: PATH_TO_CORPUS1)

```
./fuzzer PATH_TO_CORPUS1 PATH_TO_SEED_CORPUS
```

3.4.2 Auswirkungen

Wieso ein guter Seed Corpus für effizientes Fuzzing unerlässlich ist, wird mit folgenden Bildern deutlich:

Es ist zu sehen, dass die Coverage nicht mehr steigt, weil die Inputs keine neuen Pfade im Code mehr aufdecken. Dies ist die Auswirkung der Anwendung von Zufallswerten auf Funktionen, welche strukturierte Werte erwarten. Ein solches Verhalten muss vermieden werden.

```
#0      READ units: 206
#206    INITED cov: 896 ft: 1170 corp: 56/454Kb exec/s: 0 rss: 73Mb
#1024   pulse  cov: 896 ft: 1170 corp: 56/454Kb exec/s: 512 rss: 217Mb
#1366   NEW    cov: 896 ft: 1171 corp: 57/454Kb exec/s: 455 rss: 277Mb
#2048   pulse  cov: 896 ft: 1171 corp: 57/454Kb exec/s: 409 rss: 400Mb
#4096   pulse  cov: 896 ft: 1171 corp: 57/454Kb exec/s: 455 rss: 461Mb
#6612   NEW    cov: 896 ft: 1172 corp: 58/454Kb exec/s: 440 rss: 464Mb
#8192   pulse  cov: 896 ft: 1172 corp: 58/454Kb exec/s: 431 rss: 466Mb
#8673   NEW    cov: 896 ft: 1173 corp: 59/483Kb exec/s: 433 rss: 466Mb
#10237  NEW    cov: 896 ft: 1174 corp: 60/483Kb exec/s: 426 rss: 467Mb
```

Abbildung 4 Coverage steigt nicht

Wird nun ein Seed Corpus angegeben, steigt dank den gültigen Inputs die Codeabdeckung stetig an. Nach einiger Zeit verlangsamt sich dieser Prozess ebenfalls, die Verbesserung ist aber zu sehen.

```
#0      READ units: 1
#1      INITED cov: 761 ft: 756 corp: 1/1b exec/s: 0 rss: 34Mb
#2      NEW    cov: 763 ft: 758 corp: 2/2b exec/s: 0 rss: 35Mb L: 1 M
#16     NEW    cov: 766 ft: 761 corp: 3/7b exec/s: 0 rss: 37Mb L: 5 M
#22     NEW    cov: 766 ft: 771 corp: 4/9b exec/s: 0 rss: 38Mb L: 2 M
#23     NEW    cov: 766 ft: 779 corp: 5/12b exec/s: 0 rss: 38Mb L: 3
#24     NEW    cov: 767 ft: 780 corp: 6/75b exec/s: 0 rss: 38Mb L: 63
#34     NEW    cov: 767 ft: 787 corp: 7/139b exec/s: 0 rss: 40Mb L: 6
#59     NEW    cov: 767 ft: 790 corp: 8/157b exec/s: 0 rss: 43Mb L: 1
#60     NEW    cov: 767 ft: 804 corp: 9/221b exec/s: 0 rss: 44Mb L: 6
#62     NEW    cov: 767 ft: 808 corp: 10/285b exec/s: 0 rss: 44Mb L:
#64     NEW    cov: 767 ft: 810 corp: 11/349b exec/s: 0 rss: 44Mb L:
#80     NEW    cov: 767 ft: 811 corp: 12/413b exec/s: 0 rss: 47Mb L:
#167    NEW    cov: 767 ft: 812 corp: 13/477b exec/s: 0 rss: 60Mb L:
#591    NEW    cov: 771 ft: 816 corp: 14/478b exec/s: 591 rss: 126Mb
#734    NEW    cov: 771 ft: 817 corp: 15/542b exec/s: 734 rss: 149Mb
#782    NEW    cov: 772 ft: 818 corp: 16/544b exec/s: 782 rss: 156Mb
```

Abbildung 5 Coverage steigt an

Bricht das Fuzzing nach einiger Ausführungszeit ab oder wird gestoppt sind alle interessanten Inputs bereits zurückgespeichert. Interessant daran ist die Tatsache, dass der Fuzzer beim erneuten Aufruf mit demselben Corpus wieder dort weitermacht wo er aufgehört hat und somit die Coverage nicht wieder von vorne abdecken muss.

3.5 Minimierung des Corpus

Mit der Zeit wird ein Corpus durch Mutationen immer grösser. LibFuzzer verfügt über eine eingebaute Option, welche es dem Nutzer erlaubt, einen Corpus zu durchsuchen und Eingabewerte, welche eventuell einen anderen Inhalt haben, aber den gleichen Codepfad auslösen würden, zusammenzuführen. Der Platzverbrauch wird so reduziert und die Codeabdeckung bleibt dieselbe.

Dieser Befehl reduziert alle Eingabewerte im Ordner „NEW_CORPUS“:

```
./fuzzer NEW_CORPUS OLD_CORPUS -merge=1
```

Ebenfalls können Werte aus einem Corpus in einen existierenden kopiert werden, wobei nur die Dateien kopiert werden, welche die Codeabdeckung steigern.

```
./fuzzer EXISTIERENDER_CORPUS ANDERER_CORPUS -merge=1
```

3.6 Crashes und Leakes

Wird durch Fuzzing ein Fehler in der Software ausgelöst, bricht der Prozess ab und gibt auf der Konsole einen „Stack Trace“ aus. Wurde der Fuzzer mit mehreren Jobs gestartet, beginnt im Hintergrund automatisch ein neuer. Bei einem gefundenen Fehler wird am Ausführungsort des Fuzzers eine Datei mit dem Präfix „Crash“ oder „Leak“, gefolgt vom SHA1 Hash der Datei erzeugt. Darin ist für die weitere Analyse der Input welcher den Fehler ausgelöst hat enthalten.

```
INFO: Seed: 18003156
INFO: Loaded 2 modules (8042 guards): [0x7efd91d63360, 0x7efd91d6b0f4), [0x74cf30, 0x74cf44),
/home/sven/git-work/strongswan/fuzz/.libs/lt-fuzz_certs: Running 1 inputs 1 time(s) each.
Running: ../../CORPUS/SA/crashes/crash1/crash-7f5584ebbc2c41dc7daa3655cd240bbc6a44ed50
ASAN:DEADLYSIGNAL
================================================================
==30709==ERROR: AddressSanitizer: FPE on unknown address 0x7efd8c2145ba (pc 0x7efd8c2145ba bp 0x7fffad69a2f0 sp 0x7fffad69a260 T0)
    #0 0x7efd8c2145b9 in __gmp_exception (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x95b9)
    #1 0x7efd8c2145ed in __gmp_divide_by_zero (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x95ed)
    #2 0x7efd8c229a64 in __gmpz_powm_sec (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x1ea64)
    #3 0x7efd8c4a0b63 in rsaep /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:86:2
    #4 0x7efd8c4a076b in rsavp1 /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:106:9
    #5 0x7efd8c49eb08 in verify_emsa_pkcs1_signature /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:144:16
    #6 0x7efd8c49e1a8 in verify /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:311:11
    #7 0x7efd8caf9107 in issued_by /home/sven/git-work/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:1627:10
    #8 0x7efd8cae2584 in parse_certificate /home/sven/git-work/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:1491:7
    #9 0x7efd8cade8c6 in x509_cert_load /home/sven/git-work/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:2519:7
    #10 0x7efd9191cdf4 in create /home/sven/git-work/strongswan/src/libstrongswan/credentials/credential_factory.c:129:16
    #11 0x7efd8c6adafd in load_from_blob /home/sven/git-work/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:428:10
    #12 0x7efd8c6acd0c in pem_load /home/sven/git-work/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:494:10
    #13 0x7efd8c6ace8c in pem_certificate_load /home/sven/git-work/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:524:9
```

Abbildung 6 Beispiel eines Crashes

3.6.1 Reproduzierung

Ein Fehler kann beliebig oft reproduziert werden, indem der Fuzzer direkt mit einem Crashfile gestartet wird. Dieser Vorgang setzt eine Fuzzer Version mit der gleichen Schwachstelle voraus:

```
./fuzzer crash-7f5584ebbc2c41dc7daa3655cd240bbc6a44ed50
```

3.6.2 Minimierung des Crashfiles

Je nach Art der Eingabedaten kann ein Crashfile unterschiedlich gross werden. LibFuzzer kann instruiert werden, ein Crashfile mittels mehrfacher Iteration so zu minimieren, dass alle unnötigen Inhalte der Datei gelöscht werden und der Fehler trotzdem zustande kommt. Die Anzahl Iterationen wird mit der Option „-runs“ gesteuert und pro Iteration werden maximal 10000 Mutationen durchprobiert. Ein solches Unterfangen führt daher, besonders bei stark strukturiertem Input, nicht immer zum gewünschten Erfolg.

```
./fuzzer crash-7f5584ebbc2c41dc7daa3655cd240bbc6a44ed50 -minimize_crash -runs=5
```

3.7 Sanitizers

Sanitizers bestehen aus Compiler Anweisungen und/oder zusätzlichen Libraries, welche zur Laufzeit geladen werden, um latente Fehler im Code zu entdecken. Ursprünglich wurde das Projekt von Google ins Leben gerufen, aktuell liegt die Codebasis aber ebenfalls bei LLVM.[12] LibFuzzer und OSS-Fuzz können zurzeit mit folgenden 3 Sanitizern erweitert werden, um die Codequalität zu steigern.

3.7.1 Address Sanitizer (ASAN)

Erkennt Fehler die den Speicher betreffen wie z.B. Memory Leaks oder Speicherzugriffsverletzungen.

3.7.2 Undefined Behaviour Sanitizer(UBSAN)

Erkennt Verhalten, welches in C und C++ dafür bekannt ist, „Undefined Behaviour“ auszulösen. Eine vollständige Liste von Verhaltensweisen, welche zur Laufzeit überwacht werden ist bei LLVM zu finden.[13]

3.7.3 Memory Sanitizer (MSAN)

Erkennt uninitialisierte Lesezugriffe. Ist ein Adressbereich nicht initialisiert und es findet darauf ein Lesezugriff statt, ist dies ebenfalls „Undefined Behaviour“ und wird erkannt.

Dieser Sanitizer kann bei OSS-Fuzz nicht mit den anderen kombiniert werden und verhält sich speziell. Damit alles ohne Probleme funktioniert, muss der gesamte Code inkl. aller genutzten Libraries mit diesem Sanitizer erstellt werden.

3.8 Wörterbuch

Wörterbücher sind eine Möglichkeit das Fuzzing zu verbessern. Das funktioniert am besten für Fuzz Targets, bei denen der Input klar strukturierten Regeln folgt. Als Beispiel dient XML, welches vordefinierte Schlüsselwörter vorgibt. LibFuzzer versucht anschliessend die Inputs aus dem Seed Corpus mit solchen Schlüsselwörtern anzureichern, um möglichst wieder sinnvolle Inputs generieren zu können.

Wörterbücher müssen in einem speziellen, aber einfachen key=“Value“ Format vorliegen:[14]

```
#
# AFL dictionary for XML
# ----------------------
#
# Several basic syntax elements and attributes, modeled on libxml2.
#
# Created by Michal Zalewski <lcamtuf@google.com>
#

attr_encoding=" encoding=\"1\""
attr_generic=" a=\"1\""
attr_href=" href=\"1\""
attr_standalone=" standalone=\"no\""
attr_version=" version=\"1\""
attr_xml_base=" xml:base=\"1\""
attr_xml_id=" xml:id=\"1\""
attr_xml_lang=" xml:lang=\"1\""
attr_xml_space=" xml:space=\"1\""
attr_xmlns=" xmlns=\"1\""
```

Abbildung 7 Beispiel eines XML Wörterbuches[15]

[12] (https://github.com/google/sanitizers)
[13] (https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks)
[14] (https://github.com/rc0r/afl-fuzz/tree/master/dictionaries)
[15] (https://github.com/mirrorer/afl/blob/master/dictionaries/xml.dict)

4 strongSwan Fuzzing

StrongSwan ist eine quelloffene Umsetzung von IPSec mit Fokus auf X.509 Zertifikaten und wird an der Hochschule für Technik in Rapperswil entwickelt. Verfügbar ist die Software für Linux basierte Systeme und unterstützt dabei Verbindungen vom Typ IKEv1 und IKEv2. StrongSwan selbst ist in C geschrieben, was eine der Grundvoraussetzungen für Fuzzing mit libFuzzer ist.

4.1 Fuzz Targets

Als erstes stellt sich die Frage nach einem sinnvollen Fuzz Target. Der Fokus der Arbeit liegt in der Integration von strongSwan in OSS-Fuzz und nicht primär im Schreiben von zahlreichen Fuzz Targets.

4.1.1 Auswahl Sinnvoller Targets

Grundsätzlich stellt sich die Frage: „Welche Funktionen sind für Fuzzing geeignet?“. Als erstes setzt dies wieder voraus, den Code, welcher getestet werden soll, gut zu kennen. Praktisch jedes Programm kennt sogenannte „Trust Boundaries“[16]. Dies sind Grenzen, bei denen Daten über verschiedene Vertrauensebenen hinweg wechseln. Übergibt ein Programm intern Daten an ein anderes ist dies meist vertrauenswürdiger als Input von extern.

Typischerweise sind solche Grenzen[17]:

- Network Sockets
- RPC Interfaces
- Pipes
- User Input
- Daten aus dem Internet
- Konfigurationsdateien

Aus diesen Grenzen lassen sich Fuzz Targets ableiten, gut geeignet sind dafür also z.B. Parser.

Zur Diskussion standen zu Anfang verschiedene Möglichkeiten von X.509 Zertifikaten bis hin zum schwierigen Fuzzing der Kommunikation von strongSwan zwischen Server und Client. Aufgrund des niedrigeren Schwierigkeitsgrades wurde der Zertifikatsparser mit seinen Plugins als erstes Fuzz Targets ausgewählt, was die schnellsten Erfolge versprach.

[16] (https://en.wikipedia.org/wiki/Trust_boundary)
[17] (https://msdn.microsoft.com/en-us/library/cc162782.aspx#Fuzzing_topic5)

4.1.2 Erstes Fuzz Target: fuzz_certs

Das ausgewählte Fuzz Target wurde innert kurzer Zeit mithilfe von Tobias Brunner geschrieben und „fuzz_certs“ benannt. Es lädt die nötigen strongSwan Libraries nach und liefert dem Parser Zertifikate, welche mit einem Corpus übergeben werden. Gut zu sehen ist die von libFuzzer benötigte Funktion „LLVMFuzzerTestOneInput“, welche als Einstiegspunkt für jedes Fuzz Target dient.

```
#include <library.h>
#include <utils/debug.h>

int LLVMFuzzerTestOneInput(const uint8_t *buf, size_t len)
{
        certificate_t *cert;
        chunk_t chunk;

        dbg_default_set_level(-1);
        library_init(NULL, "fuzz_certs");
        plugin_loader_add_plugindirs(PLUGINDIR, PLUGINS);
        if (!lib->plugins->load(lib->plugins, PLUGINS))
        {
                return 1;
        }

        chunk = chunk_create((u_char*)buf, len);
        cert = lib->creds->create(lib->creds, CRED_CERTIFICATE, CERT_X509,
                                                        BUILD_BLOB, chunk, BUILD_END);
        DESTROY_IF(cert);

        lib->plugins->unload(lib->plugins);
        library_deinit();
        return 0;
}
```

Abbildung 8 Fuzz Target für Zertifikatsparser (fuzz_certs)

4.2 Seed Corpus

Wie gesagt arbeitet libFuzzer am besten mit gültigen Eingabewerte, welche mutiert werden. Das Ziel war also ein Corpus aufzubauen, welcher mit korrekten X.509 Zertifikaten für den Parser angereichert ist. Idealerweise handelt es sich dabei um möglichst verschiedene Zertifikate mit unterschiedlichen Feldern und sogar fremden Zeichensätzen. Ein effizientes Fuzzing baut auf einem guten Seed Corpus auf, aber woher diese Zertifikate zu bekommen sind war zunächst unklar.

4.2.1 Zertifikate selbst generieren

Die erste Idee war mittels Python möglichst viele eigene Zertifikate zu generieren. Dieser Ansatz wurde aus verschiedenen Gründen schnell wieder verworfen:

1. Wie können möglichst viele verschiedene Variationen einfach generiert werden?
2. Die Zertifikate entsprechen nicht einem realen Szenario, weil sie nicht „echt“ sind.
3. Einarbeitung in korrekte ASN.1 Struktur wäre nötig.

4.2.2 OpenSSL / BoringSSL

OpenSSL und BoringSSL verwenden beide Fuzzing von OSS-Fuzz und prüfen ebenfalls ihren Zertifikatsparser. Es war naheliegend, nach zu schauen was sich in deren Corpora verbirgt und ob diese in gewissen Teilen eventuell sogar identisch sind.
Eine Analyse hat gezeigt, dass OpenSSL 1'240 Dateien und BoringSSL 687 Zertifikate bereithält. Alle Dateien wurden mithilfe ihres SHA1 Hashs gegenübergestellt und es hat sich herausgestellt, dass nur 46 Files identisch sind. Die Frage nach dem Ursprung war somit noch ungeklärt.

Als nächstes wurde versucht die Entwickler der Projekte zu kontaktieren. OpenSSL war via Email zu erreichen und die Antwort folgte zügig. Das Corpus darf unter Einhaltung der OpenSSL Lizenz genutzt werden. Die Zertifikate sind echt und aus diversen Quellen bezogen. (z.B. Certificate Transparency)

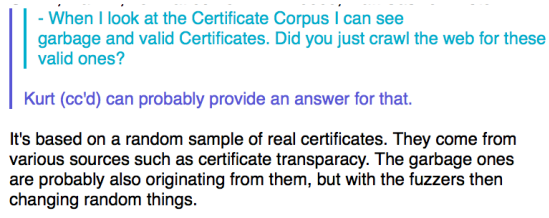


Abbildung 9 OpenSSL Antwort zu den Zertifikaten

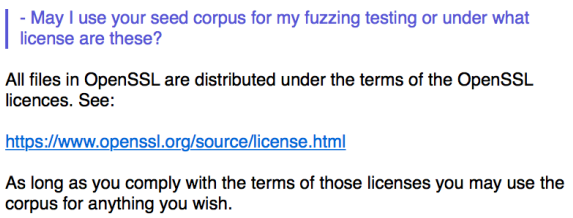


Abbildung 10 OpenSSL Antwort zum Corpus

Bei BoringSSL wurde ein Entwickler via Twitter kontaktiert. Die Antworten waren deckungsgleich mit denen von OpenSSL. Das Corpus darf verwendet werden und die Zertifikate wurden ebenfalls über Certificate Transparency bezogen.

Abbildung 11 Antwort des BoringSSL Entwicklers

Schlussfolgernd kann gesagt werden, dass Certificate Transparency eine geeignete Quelle für reale Zertifikate zu sein scheint. Aus diesem Grund soll auch das Corpus von strongSwan damit erstellt werden.

4.2.3 Google CT

Certificate Transparency[18] ist ein Projekt von Google und beschreibt einen Prozess, mit welchem die Prüfung von digitalen Zertifikaten ermöglicht wird. Zertifikate, welche eine CA ausstellt, werden in einen „Append Only“ Log geschrieben, welcher öffentlich zugänglich sind. Gleichzeitig wird den Zertifikaten ein SCT (Signed Certificate Timestamp) angefügt, welcher von den Logservern ausgestellt wird.
Ein Webbrowser kontrolliert in periodischen Abständen, ob die SCTs der Zertifikate auch in einem Log stehen und somit verifiziert sind. So soll verhindert werden, dass Hacker bei einer CA einbrechen und sich dort selbst Zertifikate für beliebige Domains ausstellen.

Seit dem Beginn des Projektes wurden mehr als 350 Millionen Zertifikate eingetragen und bilden eine exzellente Basis für einen Corpus. Die Daten können mittels einer REST API angefragt werden. Genauer dokumentiert ist dieses Verfahren unter im dazugehörigen RFC.[19]

Damit die Zertifikate heruntergeladen werden können, wurde ein Python Script geschrieben, welches im Anhang zu finden ist. Die Schwierigkeit war die Abbildung der Datenstruktur von CT, der sogenannte Merkle Baum ist aber ebenfalls im RFC genauestens dokumentiert. Das Script unterstützt folgende Optionen:

Option	Bedeutung
url	URL des CT Logservers welcher abgefragt werden soll. Eine Liste von Servern ist auf der offiziellen Website zu finden.[20]
start	Startindex des ersten herunterzuladenden Zertifikates.
end	Schlussindex des letzten herunterzuladenden Zertifikates.
folder	Ordner, in welchen die Zertifikate gespeichert werden, relativ zum Script.
type	In welchem Format sollen die Zertifikate gespeichert werden. „DER“ (Binary) oder „PEM“ (Base64)

Tabelle 3 Command Line Options Python Script

Dieses Beispiel lädt die Zertifikate 1-500 vom Server „ct.googleapis.com/pilot“ im „PEM“ Format in den Ordner „output“:

```
./download_certs.py http://ct.googleapis.com/pilot 1 500 ./output PEM
```

Der Wunsch, Zertifikate im „.PEM“ oder „.DER“ Format zu speichern entsteht aus der Überlegung, die Coverage des Parsers zu steigern. Verschiedene Formate lösen unterschiedliche Codepfade aus.

[18] (https://www.certificate-transparency.org/)
[19] (https://tools.ietf.org/html/rfc6962)
[20] (https://www.certificate-transparency.org/known-logs, 2017)

4.3 Lokales Fuzzing

Die ersten Erfahrungen mit libFuzzer wurden mit lokalem Fuzzing erarbeitet.

4.3.1 strongSwan Build Anpassungen

Um das Fuzz Target (fuzz_certs) erfolgreich erstellen zu können, mussten einige Anpassungen an den bestehenden Konfigurationsdateien gemacht werden. Für eine flexible Steuerung braucht es für den „./configure“ Vorgang ein Flag Namens „--enable-fuzzing“. Dies aktiviert die Ausführung des neuen, für die Fuzz Targets bestimmten Makefiles und erstellt die Fuzzer. Weiter ist es nötig, alle notwendigen strongSwan Plugins, welche vom Fuzz Target gebraucht werden, in „configure.ac“ einzutragen.

```
1  AM_CPPFLAGS = \
2          -I$(top_srcdir)/src/libstrongswan \
3          -DPLUGINDIR=\""$(abs_top_builddir)/src/libstrongswan/plugins\"" \
4          -DPLUGINS="\"${fuzz_plugins}\""
5
6  noinst_PROGRAMS = fuzz_certs
7
8  fuzz_certs_SOURCES = fuzz_certs.c
9
10 fuzz_certs_LDFLAGS = ${libfuzzer} -stdlib=libc++ -lstdc++ -no-install
11
12 fuzz_certs_LDADD = $(top_builddir)/src/libstrongswan/libstrongswan.la
```

Abbildung 12 Makefile für lokales Fuzzing

4.3.2 Erzeugen des ersten Fuzzers

Als erstes müssen alle von strongSwan benötigten Abhängigkeiten auf dem System installiert sein, welche in der Datei „Hacking“ gelistet sind:

```
apt-get install -y automake autoconf libtool pkg-config gettext perl python
  flex bison gperf lcov libgmp3-dev
```

Anschliessend wird strongSwan wie gewohnt kompiliert:

```
./autogen.sh && CC=clang ./configure --enable-fuzzing && make
```

Nach erfolgreichem Vorgang findet sich im Ordner „fuzz“ neu eine ausführbare Datei. Dies ist der Fuzzer für das Fuzz Target „fuzz_certs“.

4.3.2.1 Problem mit lstdc++

Zu Beginn tauchte beim linken gegen libFuzzer im „Make Prozess“ ein Fehler auf:

```
libFuzzer.a(FuzzerUtilPosix.o): In function `__gnu_cxx::new_allocator<char>::deallocate(char*, unsigned long)':
/usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../include/c++/5.4.0/ext/new_allocator.h:110: undefined reference
libFuzzer.a(FuzzerUtilPosix.o): In function `std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
 unsigned long, char const*) const':
/usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../include/c++/5.4.0/bits/basic_string.h:271: undefined reference
libFuzzer.a(FuzzerUtilPosix.o): In function `__gnu_cxx::new_allocator<char>::deallocate(char*, unsigned long)':
/usr/lib/gcc/x86_64-linux-gnu/5.4.0/../../../../include/c++/5.4.0/ext/new_allocator.h:110: undefined reference
libFuzzer.a(FuzzerUtilPosix.o):(.eh_frame+0x143): undefined reference to `__gxx_personality_v0'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Abbildung 13 Fehler beim kompilieren ohne lstdc++

StrongSwan ist in C und libFuzzer in C++ implementiert. Der Fehler lässt auf Referenzen zur GNU C++ Standardbibliothek schliessen, welche nicht aufgelöst werden können. Beim Aufruf von Clang wird standardmässig ohne C++ Bibliothek kompiliert. Ist diese notwendig, hier aufgrund von libFuzzer, kann die C++ Standardbibliothek „-lstdc++“ als LDFLAG im Makefile von „fuzz_certs“ eingebunden werden.

4.3.3 Erste Durchführung

Der erstellter Fuzzer kann nun unter Angabe eines Corpus gestartet werden:

```
./fuzz_certs PATH_TO_CORPUS_DIR
```

4.3.3.1 Unübersichtlicher Output

Beim starten des Fuzzer wurde die Konsole dermassen mit Warnungen überflutet, dass es sich dabei um ein Problem handeln musste:

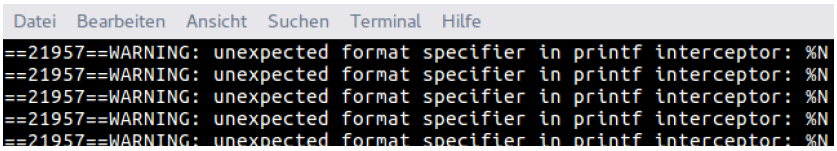


Abbildung 14 ASAN kennt den Custom Specifier lokal nicht

ASAN fängt normalerweise aktiv „printf()“ Aufrufe ab. StrongSwan nutzt an diversen Stellen die Möglichkeit von C um eigene Format Specifier für printf() zu nutzen. Die eigenen Specifier sind ASAN unbekannt und somit wird der Fehler bezüglich „enexpected format specifier“ tausende Male ausgegeben. Der Output wird dadurch schnell unübersichtlich und riesig.

Als Workaround lässt sich der Fuzzer mit der Option „ASAN_OPTIONS=check_printf=0“ starten, was die Überprüfung von „printf()“ Aufrufen deaktiviert:

```
ASAN_OPTIONS=check_printf=0 ./fuzz_certs
```

4.4 OSS-Fuzz

Für das Fuzzing auf der Google Infrastruktur waren zu den jetzigen Änderungen weitere notwendig.

4.4.1 Dockerfile

Die Fuzzer, der Build Prozess und alles andere werden bei Google in Docker Images ausgeführt. Darum muss im Projektverzeichnis des „geforkten“ OSS-Fuzz Repository eine Docker Definitionsdatei erstellt werden. Die Docker Images werden anhand dieser Definition aufgesetzt.

Wichtig ist, dass alle Abhängigkeiten welche für den Build benötigt werden, im Docker installiert werden. Diese sind aus Kapitel 4.3.2 zu entnehmen. Zudem müssen alle Repositories geklont werden, welche im Zusammenhang mit Fuzzing stehen. Für strongSwan sind dies zwei verschiedene, zum einen eines für den Source Code und das zweite für die Corpora. Eine Trennung macht durchaus Sinn, da ein Corpus problemlos auf mehrere tausend Files anwachsen kann.

```
17  FROM gcr.io/oss-fuzz-base/base-builder
18  MAINTAINER tobias@strongswan.org
19  RUN apt-get update && apt-get install -y automake autoconf libtool pkg-config gettext perl python flex bison gperf lcov libgmp3-dev
20  RUN git clone -b fuzzing --depth 1 https://github.com/strongswan/strongswan.git strongswan
21  RUN git clone --depth 1 https://github.com/strongswan/fuzzing-corpora.git strongswan/fuzzing-corpora
22  WORKDIR strongswan
23  COPY build.sh $SRC/
```

Abbildung 15 Docker Definitionen für strongSwan bei OSS-Fuzz

4.4.2 Build.sh

Sobald der Docker ausgeführt wird, startet das Build.sh Script, welches dafür verantwortlich ist, alle Fuzzer zu erstellen. Das Script ist so geschrieben, dass es den normalen Build Prozess anstösst und am Ende alle erstellten Fuzzer und Corpora wegkopiert, sodass diese anschliessend automatisch in einem anderen Docker genutzt werden können.

```
18  ./autogen.sh
19
20  ./configure CFLAGS="$CFLAGS -DNO_CHECK_MEMWIPE" --enable-fuzzing --with-libfuzzer=$LIB_FUZZING_ENGINE --enable-monolithic
21   --disable-shared --enable-static
22  make -j$(nproc)
23
24  fuzzers=$(find fuzz -maxdepth 1 -executable -type f -name \fuzz_*)
25  for f in $fuzzers; do
26          fuzzer=$(basename $f)
27          cp $f $OUT/
28          corpus=${fuzzer#fuzz_}
29          corpus=${corpus%%_*}
30          if [ -d "fuzzing-corpora/${corpus}" ]; then
31                  zip -rj $OUT/${fuzzer}_seed_corpus.zip fuzzing-corpora/${corpus}
32          fi
33  done
```

Abbildung 16 Build.sh für strongSwan OSS-Fuzz

4.4.3 Lokale Tests mit Docker

Damit später bei Google keine Fehler auftauchen, werden Docker Images und ein Hilfsscript angeboten. Die Docker Images bilden dieselbe Umgebung ab, auf welcher online das Fuzzing stattfindet. So wird sichergestellt, dass nichts hochgeladen wird, was nicht funktioniert und so Zeit verschwendet.

Erstellt das Docker Image für strongSwan:

```
./helper.py build_image strongswan
```

Erstellt alle Fuzzer mit dem Sanitizer "ASAN" für strongSwan:

```
./helper.py build_fuzzers –sanitizer address strongswan
```

Startet einen spezifischen Fuzzer. Der Name hängt vom Makefile für die Fuzz Targets ab.

```
./helper.py run_fuzzer strongSwan fuzz_certs
```

Wurden alle Befehle ohne Probleme durchgeführt, können die Dateien zu OSS-Fuzz geladen werden.

4.4.4 Einschränkungen und Probleme

Beim Testvorgang mit Docker sind Fehler aufgetreten, welche im lokalen Fuzzing nie erschienen sind. Diese sind auf die unterschiedliche Umgebung im Vergleich zu den Dockern zurückzuführen.

4.4.4.1 Fuzzing Engine als Variable

Auf Empfehlung von Google soll die verwendete Fuzzing Engine nicht fest einprogrammiert werden, da mit der Zeit weitere unterstützt werden. Darum wurde in „configure.ac" eine neue Variable definiert, mit welcher dem „./configure" Script der Pfad zur Engine (libFuzzer.a) übergeben wird. Auf den OSS-Fuzz Systemen wird zu diesem Zweck eine Umgebungsvariable mit dem Namen „$LIB_FUZZING_ENGINE" bereitgehalten, welche im „Build.sh" Script Verwendung findet.

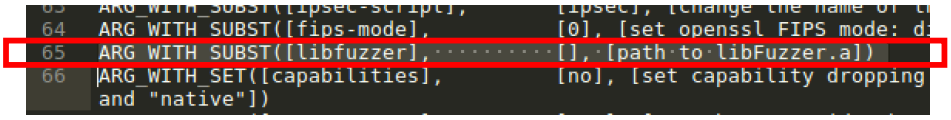


Abbildung 17 Auszug aus configure.ac

4.4.4.2 Asan printf() Interception

Das Problem ist dasselbe wie bei den lokalen Tests. Der Unterschied liegt darin, dass es sich nicht mit „ASAN_OPTIONS=check_printf=0" beheben lässt. Dies aus demselben Grund wie bei „detect_stack_use_after_return", es kann kein Einfluss auf Umgebungsvariablen bei Google genommen werden.

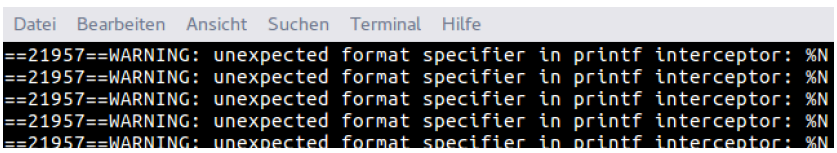


Abbildung 18 ASAN kennt den Custom Specifier im Docker nicht

Als Lösung wurde von Tobias Brunner für dieses Problem ein Workaround implementiert, welcher zum einen alternative Codeblöcke aktiviert, falls mit „--enable-fuzzing" kompiliert wird. Die zweite

Änderung deaktiviert die Evaluation eines anderen Custom Specifiers, falls der strongSwan Thread Pool nicht verwendet wird. Beim Fuzzing von Zertifikaten ist das der Fall ist.

4.4.4.3 memwipe()

Die ASAN Option „detect_stack_use_after_return=1“ hat zur Folge, dass der Fuzzer sofort nach dem Start mit einem Fehler abbricht. Die Option prüft, ob ein Stack Objekt nochmals genutzt wird nachdem die Funktion, welche das Objekt auf den Stack gelegt hat, bereits „returned“ ist. Standardmässig ist die Prüfung inaktiv[21], in der Umgebung von Google aber aktiviert. Somit besteht kein Einfluss auf dieses Verhalten. Die Prüfung an sich ist sinnvoll, in diesem Fall aber eine False-Positive Meldung.

```
INFO: Seed: 3900257796
INFO: Loaded 2 modules (8042 guards): [0x7f75361c8360, 0x7f75361d00f4), [0x74cf30, 0x74cf44),
INFO: -max_len is not provided, using 64
INFO: A corpus is not provided, starting from an empty corpus
#0      READ units: 1
================================================================
==27123==ERROR: AddressSanitizer: stack-use-after-return on address 0x7f7530b81720 at pc 0x7f7535ce015e bp 0x7ffd711b1250 sp 0x7ffd711b1248
READ of size 4 at 0x7f7530b81720 thread T0
    #0 0x7f7535ce015d in check_memwipe /home/sven/git-work/strongswan/src/libstrongswan/library.c:274:7
    #1 0x7f7535cded87 in library_init /home/sven/git-work/strongswan/src/libstrongswan/library.c:390:7
    #2 0x4f41e8 in LLVMFuzzerTestOneInput /home/sven/git-work/strongswan/fuzz/fuzz_certs.c:25:2
```

Abbildung 19 stack-use-after-return Fehler

Verursacht wird das Problem bei der Initialisierung von libstrongswan. Dabei wird die Funktion „memwipe()“ auf korrektes Verhalten überprüft. „Memwipe()“ wird unter anderem genutzt, um kryptografisch sensitives Material aus dem Speicher zu löschen. Dazu wird von „do_magic()“ der Wert „CAFEBABE“ in ein Buffer Element „buf“ gespeichert und wieder gelöscht. Die Funktion „check_memwipe()“ greift in Zeile 274 aber nochmals auf „buf“ zu, welches von „do_magic()“ auf den Stack gelegt wurde und daher kommt es zu dem Fehler.

```
247  __attribute__((noinline))
248  static void do_magic(int *magic, int **out)
249  {
250          int buf[MEMWIPE_WIPE_WORDS], i;
251
252          *out = buf;
253          for (i = 0; i < countof(buf); i++)
254          {
255                  buf[i] = *magic;
256          }
259          dbg(DBG_LIB, 3, "memwipe() pre: %b", buf, sizeof(buf));
260          memwipe(buf, sizeof(buf));
261  }
266  static bool check_memwipe()
267  {
268          int magic = 0xCAFEBABE, *buf, i;
269
270          do_magic(&magic, &buf);
271
272          for (i = 0; i < MEMWIPE_WIPE_WORDS; i++)
273          {
274                  if (buf[i] == magic)
275                  {
276                          DBG1(DBG_LIB, "memwipe() check failed: stackdir:
     %b",
277                                  buf, MEMWIPE_WIPE_WORDS * sizeof(int));
278                          return FALSE;
279                  }
280          }
281          return TRUE;
282  }
```

```
249  __attribute__((noinline))
250  static void do_magic(int *magic, int **out)
251  {
252          int buf[MEMWIPE_WIPE_WORDS], i;
253
254          *out = buf;
255          for (i = 0; i < countof(buf); i++)
256          {
257                  buf[i] = *magic;
258          }
261          dbg(DBG_LIB, 3, "memwipe() pre: %b", buf, sizeof(buf));
262          memwipe(buf, sizeof(buf));
263  }
268  static bool check_memwipe()
269  {
270          int magic = 0xCAFEBABE, *buf, i;
271
272          do_magic(&magic, &buf);
273
274          for (i = 0; i < MEMWIPE_WIPE_WORDS; i++)
275          {
276                  if (buf[i] == magic)
277                  {
278                          DBG1(DBG_LIB, "memwipe() check failed: stackdir:
     %b",
279                                  buf, MEMWIPE_WIPE_WORDS * sizeof(int));
280                          return FALSE;
281                  }
282          }
283          return TRUE;
284  }
```

Abbildung 20 Problem mit memwipe()

Eine Lösung dafür wurde von Tobias Brunner in libstrongswan eingeführt. Der verantwortliche Codeblock der Prüffunktion in der Initialisierungsphase wurde mit „#ifndef“ ausgeklammert und eine neue Compiler Option mit dem Namen „-DNO_CHECK_MEMWIPE“ angelegt.

[21] (https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterReturn)

4.4.4.4 Statischer Build

Um das grösste Problem zu verstehen, ist es wichtig zu wissen, wie die OSS-Fuzz Infrastruktur funktioniert. Wie beschrieben laufen alle Prozesse von der Kompilierung bis zur Ausführung des Fuzzers auf dem sogenannten ClusterFuzz. Alle Prozesse laufen dabei in Dockern, welche nur für die Lebensdauer der einen Operation existieren.

Die Kompilate eines Prozesses (z.B. Building) sind wiederrum die Voraussetzung für einen anderen Prozess (Ausführen der Fuzzer). Die einzige Möglichkeit der Interaktion zwischen den verschiedenen Dockern sind 3 Ordner auf der Infrastruktur, welche über Umgebungsvariablen erreichbar sind.

Location	Env	Description
`/out/`	`$OUT`	Directory to store build artifacts (fuzz targets, dictionaries, options files, seed corpus archives).
`/src/`	`$SRC`	Directory to checkout source files
`/work/`	`$WORK`	Directory for storing intermediate files

Abbildung 21 Verfügbare OSS-Fuzz Verzeichnisse

Was in die Ordner gelangt wird zum einen in der Docker Definitionsdatei, wie auch im „Build.sh“ definiert:

„$SRC“ Ordner

Alle Repositories und das „Build.sh“ Script werden hierhin kopiert. Dies steht im Docker Definitionsfile.

```
root@21ac0c82cf9c:/out# ls -la $SRC
total 24
drwxrwxrwx  5 root root 4096 May 25 13:26 .
drwxr-xr-x 76 root root 4096 May 25 13:26 ..
drwxr-xr-x 10 root root 4096 May 24 16:11 afl
-rw-rw-r--  1 root root 1154 Apr 28 12:02 build.sh
drwxr-xr-x  5 root root 4096 May 24 16:11 libfuzzer
drwxr-xr-x 14 root root 4096 May 25 13:26 strongswan
```

Abbildung 22 Inhalt von "$SRC" während des Build Vorgang

„$OUT“ Ordner

Die vom „Build.sh“ Script erstellten Fuzzer Executables werden nach „$OUT“ kopiert, damit diese im nächsten Docker ausgeführt werden können.

```
root@21ac0c82cf9c:/out# ls -la $OUT
total 18156
drwxrwxrwx  3 root root     4096 May 25 13:18 .
drwxr-xr-x 76 root root     4096 May 25 13:26 ..
```

Abbildung 23 Inhalt von "$OUT" vor dem Build Vorgang

Eigentliches Problem

Wird strongSwan normal kompiliert, finden sich nach Abschluss diverse „shared“ Bibliotheken auf dem System. Das Fuzz Target (fuzz_certs) kann bei der Ausführung auf diese zugreifen und seine Referenzen auflösen. Beim lokalen Fuzzing waren alle Dateien auf demselben System vorhanden und somit kam es zu keinen Komplikationen.

Da Online ausser dem Fuzzer keinerlei Dateien den Build Vorgang überstehen, schlägt die Ausführung des Fuzzers im „base-runner“ Docker fehl, weil benötigte Bibliotheken nicht gefunden werden.

```
/out/fuzz_certs: error while loading shared libraries: libstrongswan.so.0: cannot open
shared object file: No such file or directory
```

Abbildung 24 Fehler: libstrongswan ist nicht im Executable

Gelöst werden kann das Problem wie beim linken der libFuzzer.a Bibliothek mit dem Fuzz Target. StrongSwan und die nötigen Abhängigkeiten müssen statisch gebaut und mit dem Executable gelinkt werden, sodass alle nötigen Bibliotheken in einer ausführbaren Datei vorhanden sind.

Eine Lösung ist es, beim „./configure“ Vorgang das Flag „--enable-monolitic“ mitzugeben, was libstrongswan monolithisch erstellt und alle aktivierten Plugins in einer Datei vereint. Der erneute Start zeigt, dass die Bibliothek gefunden wird, dafür nun dasselbe Problem mit der GMP Bibliothek besteht.

```
/out/fuzz_certs: error while loading shared libraries: libgmp.so.10: cannot open
shared object file: No such file or directory
```

Abbildung 25 Fehler: GMP Library ist nicht im Executable

Indem im Makefile des Fuzz Targets GMP explizit mittels LDFLAGS statisch angegeben wird, kann dieses Problem ebenfalls umgangen werden.

```
    -Wl,-Bstatic -lgmp -Wl,-Bdynamic \
```

Abbildung 26 GMP statisch einbinden

Ein erneuter Start des Fuzzers funktionierte nach den Anpassungen, doch leider lieferte das Programm keinerlei Ergebnisse. Zu sehen waren nur „pulse“ Events und die Meldung, dass beim parsen kein „Builder“ verwendet wurde. Debugging mit Tobias Brunner hat gezeigt, dass dem Fuzzer Zertifikate übergeben werden, doch keines davon vom Parser verarbeitet wird, weil die Plugins nicht korrekt geladen sind.

```
INFO: Seed: 2401402932
INFO: Loaded 1 modules (6116 guards): [0x967c30, 0x96dbc0),
INFO: -max_len is not provided, using 64
INFO: A corpus is not provided, starting from an empty corpus
#0      READ units: 1
#1      INITED cov: 457 ft: 455 corp: 1/1b exec/s: 0 rss: 44Mb
building CRED_CERTIFICATE - X509 failed, tried 0 builders
no files found matching '/usr/local/etc/strongswan.conf'
#8192   pulse  cov: 457 ft: 455 corp: 1/1b exec/s: 4096 rss: 61Mb
building CRED_CERTIFICATE - X509 failed, tried 0 builders
no files found matching '/usr/local/etc/strongswan.conf'
#16384  pulse  cov: 457 ft: 455 corp: 1/1b exec/s: 3276 rss: 61Mb
#32768  pulse  cov: 457 ft: 455 corp: 1/1b exec/s: 3276 rss: 61Mb
#65536  pulse  cov: 457 ft: 455 corp: 1/1b exec/s: 3120 rss: 61Mb
```

Abbildung 27 Nur 1 geladenes Modul und keine Verarbeitung von Zertifikaten

Die Plugins arbeiten nicht, weil der dafür nötige Code nicht vorhanden ist. Die Plugin Konstruktoren werden nie direkt, sondern dynamisch mittels „dlsym()“ aufgelöst. Das hat zur Folge, dass weder aus einer Bibliothek, noch aus dem Fuzzer eine direkte Referenz auf diese Konstruktoren vorhanden ist. Aus diesem Grund sieht es für den Linker so aus, dass die Symbole gar nicht gebraucht werden (sind nicht referenziert) und er entfernt den zugehörigen Code automatisch. Die Konstruktoren sind der einzige Einstiegspunkt in ein Plugin selbst. Somit sind die diversen Plugins (X509, PEM, SHA1 etc.) nicht nutzbar und beim parsen erscheint die Meldung, dass „0 Builder“ verwendet wurden.
Das Problem könnte beim linken mit der Option „--whole-archive“ umgangen werden, sodass auch alle „ungenutzten“ Symbole eingebettet werden. Für den Build Prozess wird aber libtool genutzt, welches diese Option nicht unterstützt.

Nachdem dafür von Tobias Brunner eine Lösung veröffentlicht wurde, funktionierte das Fuzzing ohne weitere Probleme.

5 Ergebnisse

Das Fuzzing hat sowohl lokal, wie auch bei OSS einige Fehler im strongSwan Code hervorgebracht. Die gefundenen Probleme werden hier aufgelistet.

5.1 Lokal gefundene Probleme

Lokal wurde schon nach wenigen versuchen Fehler im Code gefunden, welche zusammen mit Tobias Brunner angeschaut und behoben wurden.

5.1.1 Crashes

5.1.1.1 Crash 1

```
INFO: Seed: 3837808124
INFO: Loaded 2 modules (8042 guards): [0x7f6af6994360, 0x7f6af699c0f4), [0x74cf30, 0x74cf44),
/home/sven/git-work/strongswan/fuzz/.libs/lt-fuzz_certs: Running 1 inputs 1 time(s) each.
Running: ../../CORPUS/SA/crashes/crash1/crash-7f5584ebbc2c41dc7daa3655cd240bbc6a44ed50
ASAN:DEADLYSIGNAL
=================================================================
==19835==ERROR: AddressSanitizer: FPE on unknown address 0x7f6af0e145ba (pc 0x7f6af0e145ba bp 0x7ffcdf2e19d0 sp 0x7ffcdf2e1940 T0)
    #0 0x7f6af0e145b9 in __gmp_exception (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x95b9)
    #1 0x7f6af0e145ed in __gmp_divide_by_zero (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x95ed)
    #2 0x7f6af0e29a64 in __gmpz_powm_sec (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x1ea64)
    #3 0x7f6af10a0b63 in rsaep /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:86:2
    #4 0x7f6af10a076b in rsavp1 /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:106:9
    #5 0x7f6af109eb08 in verify_emsa_pkcs1_signature /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:144:16
    #6 0x7f6af109e1a8 in verify /home/sven/git-work/strongswan/src/libstrongswan/plugins/gmp/gmp_rsa_public_key.c:311:11
    #7 0x7f6af16f9107 in issued_by /home/sven/git-work/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:1627:10
    #8 0x7f6af16e2584 in parse_certificate /home/sven/git-work/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:1491:7
    #9 0x7f6af16de8c6 in x509_cert_load /home/sven/git-work/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:2519:7
    #10 0x7f6af654ddf4 in create /home/sven/git-work/strongswan/src/libstrongswan/credentials/credential_factory.c:129:16
    #11 0x7f6af12adafd in load_from_blob /home/sven/git-work/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:428:10
    #12 0x7f6af12acd0c in pem_load /home/sven/git-work/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:494:10
    #13 0x7f6af12ace8c in pem_certificate_load /home/sven/git-work/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:524:9
    #14 0x7f6af654ddf4 in create /home/sven/git-work/strongswan/src/libstrongswan/credentials/credential_factory.c:129:16
    #15 0x4f4543 in LLVMFuzzerTestOneInput /home/sven/git-work/strongswan/fuzz/fuzz_certs.c:33:9
    #16 0x4fef03 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:451:13
    #17 0x4ff130 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:408:3
    #18 0x4f4c81 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:268:6
    #19 0x4f789e in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:585:9
    #20 0x4f49f0 in main /home/sven/git-work/Fuzzer/./FuzzerMain.cpp:20:10
    #21 0x7f6af551982f in __libc_start_main /build/glibc-9tT8Do/glibc-2.23/csu/../csu/libc-start.c:291
    #22 0x41c4b8 in _start (/home/sven/git-work/strongswan/fuzz/.libs/lt-fuzz_certs+0x41c4b8)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: FPE (/usr/lib/x86_64-linux-gnu/libgmp.so.10+0x95b9) in __gmp_exception
==19835==ABORTING
```

Abbildung 28 Fehler beim Benutzen der GMP Library

Problem

Das Problem befindet sich im GMP Plugin von strongSwan. Bei der Anwendung des RSA Encryption Primitive Algorithmus wird die Funktion mpz_powm() genutzt. Bei der Kompilierung findet die Überprüfung statt, ob mpz_powm_sec(), eine gegen Side-Channel Attacken robustere Version, verfügbar ist. Ist dies der Fall, wird der Aufruf mit der sicheren Variante ersetzt. Mpz_powm_sec() besitzt zwar dieselben Argumente als Input, setzt aber voraus, dass der Exponent grösser als 0 und der Modulus ungerade ist.[22] Ist dies nicht gegeben, wird eine Floating Point Exception ausgelöst, welche das Programm zum Absturz bringt.

Indem ein Angreifer ein Zertifikat bereithält, welches einen entsprechend manipulierten Public Key enthält, könnte dies für eine Denial-of-Service Attacke ausgenutzt werden. Aus diesem Grund wurde diesem Problem der CVE-2017-9022 zugewiesen und ein Patch erstellt.

Lösung

Als Fix für das Problem darf nicht einfach die Funktion mpz_powm() mit mpz_powm_sec() ausgetauscht werden. Es braucht vor dem Aufruf eine weitere Überprüfung der Input Werte.

[22] (https://gmplib.org/manual/Integer-Exponentiation.html)

5.1.1.2 Crash 2

```
INFO: Seed: 269398408
INFO: Loaded 2 modules (7978 guards): [0x7f6a1600ed40, 0x7f6a160169d8), [0x74bf30, 0x74bf40),
/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs: Running 1 inputs 1 time(s) each.
Running: ../../CORPUS/SA/crashes/crash2/dump/crash-f81b82dceec52aa92064312e5d2fd64cc1b0e8ba
===================================================================
==25169==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61c000001790 at pc 0x7f6a10a1250a bp 0x7ffda4b5d930 sp 0x7ffda4b5d928
READ of size 1 at 0x61c000001790 thread T0
    #0 0x7f6a10a12509 in pem_to_bin /home/sven/git-work/strongrepro/src/libstrongswan/plugins/pem/pem_builder.c:309:9
    #1 0x7f6a10a12509 in load_from_blob /home/sven/git-work/strongrepro/src/libstrongswan/plugins/pem/pem_builder.c:402
    #2 0x7f6a15d1ad47 in create /home/sven/git-work/strongrepro/src/libstrongswan/credentials/credential_factory.c:129:16
    #3 0x4f41a4 in LLVMFuzzerTestOneInput /home/sven/git-work/strongrepro/fuzz/fuzz_certs.c:33:9
    #4 0x4fe833 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:451:13
    #5 0x4fea60 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:408:3
    #6 0x4f45b1 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:268:6
    #7 0x4f71ce in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:585:9
    #8 0x4f4320 in main /home/sven/git-work/Fuzzer/./FuzzerMain.cpp:20:10
    #9 0x7f6a14d9c82f in __libc_start_main /build/glibc-9tT8Do/glibc-2.23/csu/../csu/libc-start.c:291
    #10 0x41c4b8 in _start (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x41c4b8)

0x61c000001790 is located 0 bytes to the right of 1808-byte region [0x61c000001080,0x61c000001790)
allocated by thread T0 here:
    #0 0x4c60cc in __interceptor_malloc (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x4c60cc)
    #1 0x7f6a10a113d9 in load_from_blob /home/sven/git-work/strongrepro/src/libstrongswan/plugins/pem/pem_builder.c:399:9
    #2 0x7f6a15d1ad47 in create /home/sven/git-work/strongrepro/src/libstrongswan/credentials/credential_factory.c:129:16
    #3 0x4f41a4 in LLVMFuzzerTestOneInput /home/sven/git-work/strongrepro/fuzz/fuzz_certs.c:33:9
    #4 0x4fea60 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:408:3
    #5 0x4f45b1 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:268:6
    #6 0x4f71ce in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:585:9
    #7 0x4f4320 in main /home/sven/git-work/Fuzzer/./FuzzerMain.cpp:20:10
    #8 0x7f6a14d9c82f in __libc_start_main /build/glibc-9tT8Do/glibc-2.23/csu/../csu/libc-start.c:291

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/sven/git-work/strongrepro/src/libstrongswan/plugins/pem/pem_builder.c:309:9 in pem_to_bin
```

Abbildung 29 Heap Overflow im PEM Builder Plugin

Problem

PEM basierte Zertifikate sind in Base64 kodiert und werden am Anfang und Ende mit einem „Armor“ umschlossen, welcher klar abgrenzt, um was für Inhalt es sich handelt und wo dieser beginnt und endet. In einem Zertifikat lautet dieser Armor „-----BEGIN CERTIFICATE-----“ und „-----END CERTIFICATE-----“.

```
1  -----BEGIN CERTIFICATE-----
2  MIIEwzCCAqugAwIBAgIHBQTV/zNq1jANBgkqhkiG9w0BAQUFADB/MQswCQYDVQQG
3  EwJHQjEPMA0GA1UECAwGTG9uZG9uMRcwFQYDVQQKDA5Hb29nbGUgVUsgTHRkLjEh
4  MB8GA1UECwwYQ2VydGlmaWNhdGUgVHJhbnNwYXJlbmN5MSMwIQYDVQQDDBpNZXJn
5  ZSBEZWxheSBJbnRlcm1lZGlhdGUgMTAeFw0xNDEwMDcxNDI3MTlaFw0xNDEwMDgx
6  NDI3MTlaMDcxCzAJBgNVBAYTAkdCMSgwJgYDVQQKDB9Hb29nbGUgQ2VydGlmaWNh
7  dGUgVHJhbnNwYXJlbmN5MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA
8  zpcF0QnPnQ7nGCXQuxQugdgrM4yZqC0nO6TuEffJ7iobnmsn/lbfZSOyGSCMCDbU
9  qqc8XS8OTq3zehfSRS7go9XlUiB7BMbvQqjliRabH7VxhBgJ7dHZ5yIdIcEUMQdX
10 XpvlIcN3dPzUL6M3TTzyXcSkqogVk++IcsXNqwcLDPgmpRBDL6LhqZSupY1+7z2x
11 f1dm0FZuy1aXJNrTjyBG9N4KN+Qe0vSS/5x9x3B9mlHjix+ozxrl7mpSBTyUTGy3
12 nI7pPeuEyNO7NI1WMLzqTh+qpSlhFspivuoIhoVwPebRWPocd79x6EoukBKAiJt+
13 q2e2fFPmGbdS4m0WWaDpAQIDAQABo4GLMIGIMCMGA1UdEQQcMBqCGGZsb3dlcnMt
14 dG8tdGhlLXdvcmxkLmNvbTAMBgNVHRMBAf8EAjAAMBMGA1UdJQQMMAoGCCsGAQUF
15 BwMBMB8GA1UdIwQYMBaAFOk8BOGAL8KEEy0mcJ7y/RrPqv7GMB0GA1UdDgQWBBTL
16 ZNBsFoQM0bAtv48zcf2j4tHUNDANBgkqhkiG9w0BAQUFAAOCAgEACKAsqMpoNyyf
17 n6w/CEnXIPaMGGeIKfH0kKi4zKwrWrjRK1dzaXKNKdXSUY1BwPrORxgXus/Y+P/R
18 0KKZLKugVGywK0kQM7Rk1QdWd3anN6XGzqxIrqSsVXIVPHiXyL8TOqC1Vm30Ju5l
19 Usq3blyISHbmg/IKbhgmul057rRdS9AzbvhqXOutJQSvowsFoTFZimCQFJMVxJlu
20 ggZUHmSWF45sf7OuYzmgxWfMUG3Mx5LmiYYLFOD9mOAiUfIJYsYOXKX8nlU+orT6
21 ND4rByEeOLVYi5aBSH5CXgzlhx+8+V2qyvEAPA1hsmALcZ36O8UVrIUolNVIXNgC
22 kAXf57hVLJhpg2kTdACEeXLg6cjqmaL4LRmGeWaozjUIvYRp/WSFm9OBKy64uSWY
23 5V3NziNpNf4Xsk3ckxkHdFMBPFPBSosh8WUcRPpKi1ulcg2InDJsD9s+WhzuOxrp
24 8BMil6Ei9cI12NBvJ6RGhvCNkIEMOgM+XlKbKDnYsD/4H/HFzywKCyoJhqHO6HGz
25 QlnHMmqu+o2+ucvJR9au1kULV/mARNzk7MazYWO5XyEPh650R4JZtDZdaqbHK5lf
26 MrFG+r6J+Z7yATe4H+Av21AubIkTTyGWOj+oc0/F4aDEPNY2XrwWjDTvTU6ES1V5
27 spQdvagtZX5ZzMvaUFzhsqFSzJ+9AXw=
28 -----END CERTIFICATE-----
```

Abbildung 30 Beispiel eines Base64 kodierten Zertifikates

Das PEM Plugin analysiert die Zertifikatszeilen und führt Checks auf das aktuelle Zeichen aus, ohne vorher zu überprüfen, ob überhaupt noch ein Zeichen vorhanden ist. Sind nun z.B. nur 4 statt der 5 erwarteten Bindestriche am Ende in „-----END CERTIFICATE-----“ vorhanden, wird über das Zeilenende hinausgelesen und führt zu dem Fehler.

PEM Zertifikate finden bei strongSwan nur lokale Verwendung, darum ist die Tragweite dieses Fehlers nicht gravierend.

Lösung

Im Code muss sichergestellt werden, dass nicht über das Zeilenende hinausgelesen wird.

5.1.2 Leaks

5.1.2.1 Leak 1

```
INFO: Seed: 2704125364
INFO: Loaded 2 modules (7978 guards): [0x7fb044075d40, 0x7fb04407d9d8), [0x74bf30, 0x74bf40),
/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs: Running 1 inputs 1 time(s) each.
Running: ../CORPUS/SA/leakes/leak-a42a05ed4dceef1218d98f3c2e940b768b8d008f

=================================================================
==8737==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1 byte(s) in 1 object(s) allocated from:
    #0 0x4c60cc in __interceptor_malloc (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x4c60cc)
    #1 0x7fb042e597d7 in vasprintf /build/glibc-9tT8Do/glibc-2.23/libio/vasprintf.c:73
    #2 0x4421ed in __interceptor_vasprintf (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x4421ed)
    #3 0x4432db in __interceptor_asprintf (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x4432db)
    #4 0x7fb03ee30a12 in add_cdps /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:824:7
    #5 0x7fb03ee30701 in x509_parse_crlDistributionPoints /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:899:2
    #6 0x7fb03ee34575 in parse_certificate /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:1457:12
    #7 0x7fb03ee34575 in x509_cert_load /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:2595
    #8 0x7fb043d81d47 in create /home/sven/git-work/strongrepro/src/libstrongswan/credentials/credential_factory.c:129:16
    #9 0x7fb03ea11efc in load_from_blob /home/sven/git-work/strongrepro/src/libstrongswan/plugins/pem/pem_builder.c
    #10 0x7fb043d81d47 in create /home/sven/git-work/strongrepro/src/libstrongswan/credentials/credential_factory.c:129:16
    #11 0x4f41a4 in LLVMFuzzerTestOneInput /home/sven/git-work/strongrepro/fuzz/fuzz_certs.c:33:9
    #12 0x4fe833 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:451:13
    #13 0x4fea60 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:408:3
    #14 0x4f45b1 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:268:6
    #15 0x4f71ce in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:585:9
    #16 0x4f4320 in main /home/sven/git-work/Fuzzer/./FuzzerMain.cpp:20:10
    #17 0x7fb042e0382f in __libc_start_main /build/glibc-9tT8Do/glibc-2.23/csu/../csu/libc-start.c:291
    #18 0x41c4b8 in _start (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x41c4b8)

SUMMARY: AddressSanitizer: 1 byte(s) leaked in 1 allocation(s).
```

Abbildung 31 Leak 1 im X509 Parser

Problem

X509 Zertifikate können einen „CRL Distribution Point“ (CDP) enthalten. Dies ist ein Verweis auf einen Ort, an welchem eine CRL hinterlegt ist und kann in Form eines HTTP URI eingetragen sein.

Mittels asprintf() wird eine Identität in einen String umgewandelt. Dieser Aufruf alloziert Speicher für den String und falls dieser grösser als 0 ist, wird der Speicher auch korrekt freigegeben. Hat der String jedoch die Länge 0 wird ein leerer String alloziert, welcher durch die Längenprüfung nicht wieder freigegeben wird. Dies führt dann zu einem Memory Leak.

Lösung

Nach der Prüfung der String Länge auf 0 muss dieser in jedem Fall wieder freigegeben werden und nicht nur falls er grösser als 0 ist.

5.1.2.2 Leak 2

```
INFO: Seed: 1118500476
INFO: Loaded 2 modules (7978 guards): [0x7f3b49be6d40, 0x7f3b49bee9d8), [0x74bf30, 0x74bf40),
/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs: Running 1 inputs 1 time(s) each.
Running: ../../CORPUS/SA/leakes/leak-81235bd74d04924815a67ab49b0e240e54dafc3a

=================================================================
==25824==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 20 byte(s) in 1 object(s) allocated from:
    #0 0x4c60cc in __interceptor_malloc (/home/sven/git-work/strongrepro/fuzz/.libs/lt-fuzz_certs+0x4c60cc)
    #1 0x7f3b44a30202 in x509_parse_authorityKeyIdentifier /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:580:25
    #2 0x7f3b44a33c85 in parse_certificate /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:1464:33
    #3 0x7f3b44a33c85 in x509_cert_load /home/sven/git-work/strongrepro/src/libstrongswan/plugins/x509/x509_cert.c:2595
    #4 0x7f3b498f2d47 in create /home/sven/git-work/strongrepro/src/libstrongswan/credentials/credential_factory.c:129:16
    #5 0x7f3b44611efc in load_from_blob /home/sven/git-work/strongrepro/src/libstrongswan/plugins/pem/pem_builder.c
    #6 0x7f3b498f2d47 in create /home/sven/git-work/strongrepro/src/libstrongswan/credentials/credential_factory.c:129:16
    #7 0x4f41a4 in LLVMFuzzerTestOneInput /home/sven/git-work/strongrepro/fuzz/fuzz_certs.c:33:9
    #8 0x4fea60 in fuzzer::Fuzzer::RunOne(unsigned char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerLoop.cpp:408:3
    #9 0x4f45b1 in fuzzer::RunOneTest(fuzzer::Fuzzer*, char const*, unsigned long) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:268:6
    #10 0x4f71ce in fuzzer::FuzzerDriver(int*, char***, int (*)(unsigned char const*, unsigned long)) /home/sven/git-work/Fuzzer/./FuzzerDriver.cpp:585:9
    #11 0x4f4320 in main /home/sven/git-work/Fuzzer/./FuzzerMain.cpp:20:10
    #12 0x7f3b4897482f in __libc_start_main /build/glibc-9tT8Do/glibc-2.23/csu/../csu/libc-start.c:291

SUMMARY: AddressSanitizer: 20 byte(s) leaked in 1 allocation(s).
```

Abbildung 32 Leak 2 im X509 Parser

Problem

In X509 Zertifikaten gibt es eine Extension mit dem Namen „authorityKeyIdentifier“, welches die CA identifiziert, die das Zertifikat unterzeichnet hat. Genauer gesagt den passenden Public Key zum Private Key, mit welchem das Zertifikat ausgestellt wurde. Gemäss RFC[23] darf eigentlich nur ein solcher Eintrag vorhanden sein.

StrongSwan setzt in diesem Bereich den Standard nicht vollständig um. Die Software nimmt auch Zertifikate mit mehreren „authorityKeyIdentifier“ an und parst diese der Reihe nach. Dabei wird der vorherige Wert nicht wieder freigegeben, was in einem Memory Leak resultiert.

Lösung

Beim erneuten Durchgang eines „authorityKeyIdentifier“ muss darauf geachtet werden, den genutzten Speicher des alten freizugeben oder alternativ muss die Umsetzung streng nach RFC geschehen und somit das Zertifikat zurückgewiesen werden.

[23] (https://www.ietf.org/rfc/rfc5280.txt)

5.2 Mit OSS-Fuzz gefundene Probleme

Auf der Google Infrastruktur wird das Fuzzing immer wieder gestartet. Der hier gelistete Bug ist einer der interessantesten. Gefunden wurden noch mehr, jedoch würde die Darstellung aller Probleme den Rahmen sprengen.

5.2.1 Crash 1

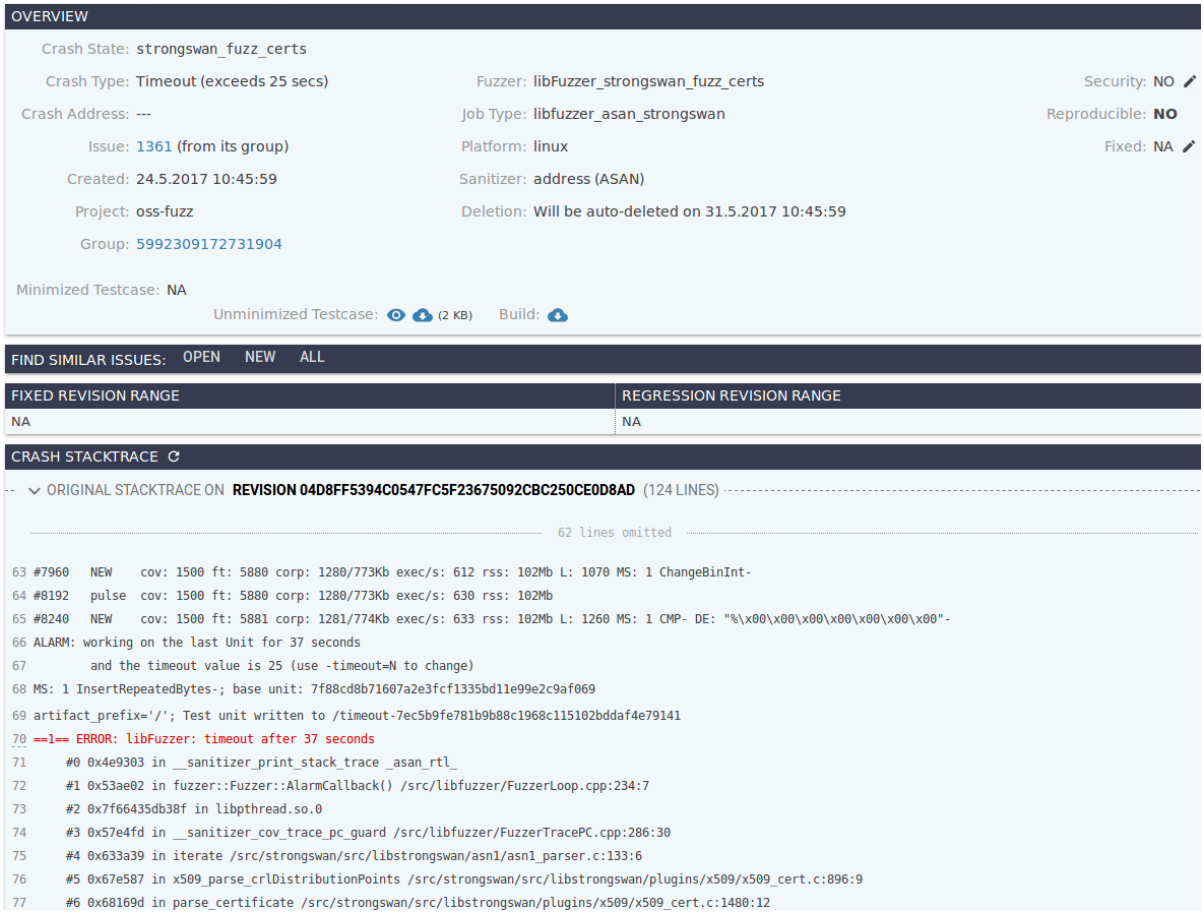


Abbildung 33 Timeout beim CHOICE parsen

Problem

Gemäss ASN.1 Definition gibt es sogenannte CHOICE Typen, welche ähnlich sind wie OPTIONAL Typen. Der Unterschied ist, dass CHOICE keine mehrfache Auswahl zulässt und genau 1 Element ausgewählt werden muss. In X509 Zertifikaten werden solche Typen bei diversen Extensions eingesetzt.

StrongSwan hat bis anhin CHOICE gleich wie OPTIONAL Typen behandelt und dabei nicht darauf geachtet, dass diese überhaupt im Zertifikat vorhanden sind und davon genau einer ausgewählt wird. Kommt es nun zu der Situation, dass keine Option gefunden wird, springt der Parser wieder zurück und landet so in einer Endlosschleife. LibFuzzer bemerkt dieses Timeout und bricht den Vorgang nach einer gewissen Zeit ab.

Dieses Problem könnte für eine Denial-of-Service Attacke ausgenutzt werden, indem ein Angreifer ein entsprechend erstelltes Zertifikat benutzt. Aus diesem Grunde wurde dem Problem der CVE-2017-9023 zugewiesen und ein Patch erstellt.

Lösung

ASN.1 CHOICE Typen müssen so behandelt werden, wie es im Standard vorgesehen ist.

5.2.2 Auswertungen in OSS-Fuzz

5.2.2.1 Darstellung gefundener Probleme

Ein Dashboard stellt übersichtliche die bis jetzt gefundenen Probleme dar und dient als Einstiegspunkt für das weitere Vorgehen. Unter anderem wird ein „Reproducible“ Flag für ein Problem gesetzt, sollte ein Fehler mehrmals reproduzierbar sein. Weiter sind Flags mit der zugewiesenen Issue Nummer sowie der Meldung „Fixed“ zu sehen, sollte OSS-Fuzz dies verifiziert haben.

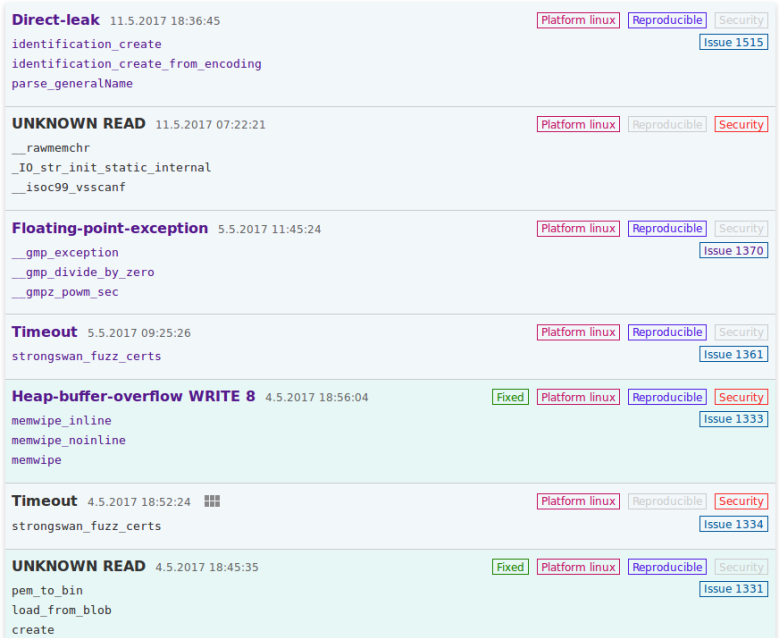


Abbildung 34 Darstellung der Probleme in OSS-Fuzz

5.2.2.2 Detaillierte Darstellung

Ein gefundener Bug wird wie in Kapitel 5.2.1 dargestellt. Nebst den Informationen, welche auch lokal vorhanden sind, werden weitere Angaben angezeigt. Ein Diagramm zeigt an, an welchem Tag der Fehler wie oft gefunden wurde und wie die Regressionstests dazu verlaufen sind.

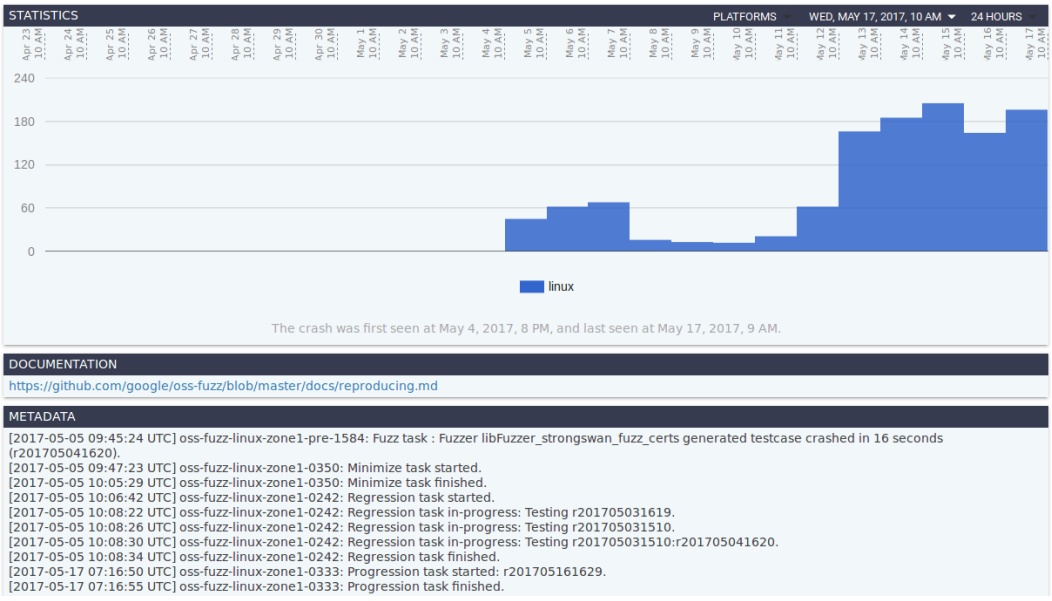
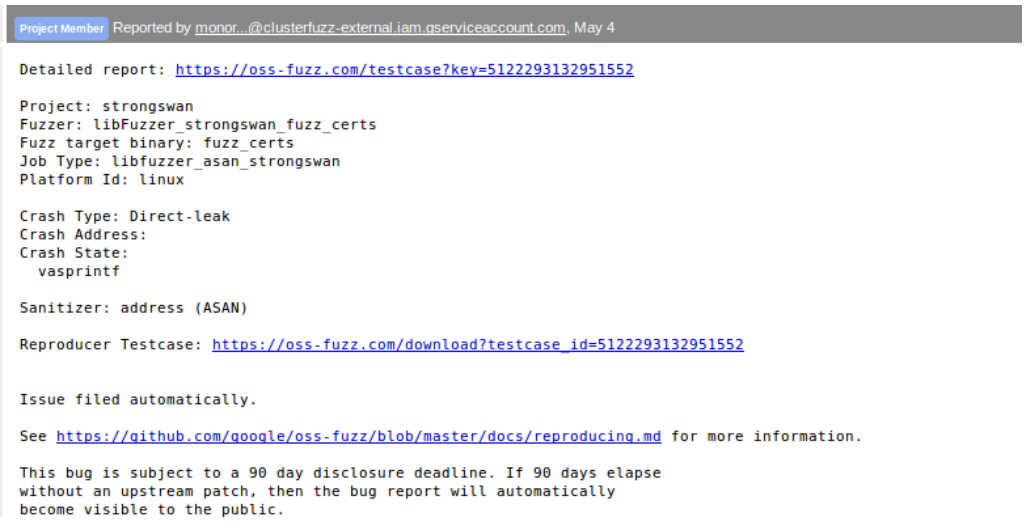


Abbildung 35 Detaillierte Fehlerstatistiken

5.2.2.3 Eröffnetes Issue

Ist ein Fehler aufgetreten wird, wie erwähnt, automatisch ein Issue eröffnet. Dies geschieht in Form einer Forenähnlichen Plattform. Ein zuständiger Entwickler hat die Möglichkeit zu diesen Einträgen Stellung zu beziehen.



Project Member Reported by monor...@clusterfuzz-external.iam.gserviceaccount.com, May 4

```
Detailed report: https://oss-fuzz.com/testcase?key=5122293132951552

Project: strongswan
Fuzzer: libFuzzer_strongswan_fuzz_certs
Fuzz target binary: fuzz_certs
Job Type: libfuzzer_asan_strongswan
Platform Id: linux

Crash Type: Direct-leak
Crash Address:
Crash State:
  vasprintf

Sanitizer: address (ASAN)

Reproducer Testcase: https://oss-fuzz.com/download?testcase_id=5122293132951552


Issue filed automatically.

See https://github.com/google/oss-fuzz/blob/master/docs/reproducing.md for more information.

This bug is subject to a 90 day disclosure deadline. If 90 days elapse
without an upstream patch, then the bug report will automatically
become visible to the public.
```

Abbildung 36 Automatisch eröffneter Issue

Interessant ist zu sehen, wie aktiv OSS-Fuzz betreut wird. Im Falle eines etwas speziellen Stack Trace zu einem Memory Leak wurden von einem Google Mitarbeiter Nachforschungen zu dem Problem angestellt und die Ergebnisse gleich mitgeteilt.

Project Member Comment 1 by kcc@google.com, May 4

```
With ASAN_OPTIONS=fast_unwind_on_malloc=0 I get this report:

Direct leak of 1 byte(s) in 1 object(s) allocated from:
    #0 0x4d7ef8 in malloc /src/llvm/projects/compiler-rt/lib/asan/asan_malloc_linux.cc:66
    #1 0x7f31dc4e77d7 in vasprintf (/lib/x86_64-linux-gnu/libc.so.6+0x767d7)
    #2 0x47f7f8 in vasprintf /src/llvm/projects/compiler-rt/lib/asan/../sanitizer_common
/sanitizer_common_interceptors.inc:1458
    #3 0x47ffb2 in __interceptor_asprintf /src/llvm/projects/compiler-rt/lib/asan/../sanitizer_common
/sanitizer_common_interceptors.inc:1492
    #4 0x87fa8f in parse_authorityInfoAccess /src/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:640:9
    #5 0x863d9f in parse_certificate /src/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:1413:7
    #6 0x860d06 in x509_cert_load /src/strongswan/src/libstrongswan/plugins/x509/x509_cert.c:2519:7
    #7 0x5d8b98 in create /src/strongswan/src/libstrongswan/credentials/credential_factory.c:129:16
    #8 0x8ec6e9 in load_from_blob /src/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:428:10
    #9 0x8eb7aa in pem_load /src/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:494:10
    #10 0x8eb95f in pem_certificate_load /src/strongswan/src/libstrongswan/plugins/pem/pem_builder.c:524:9
    #11 0x5d8b98 in create /src/strongswan/src/libstrongswan/credentials/credential_factory.c:129:16
    #12 0x511c6b in LLVMFuzzerTestOneInput /src/strongswan/fuzz/fuzz_certs.c:33:9

This is a pretty unusual situation of leaking a memory allocated from within libc's vasprintf and with default settings
asan fails to properly unwind the stack.
```

Project Member Comment 2 by monor...@clusterfuzz-external.iam.gserviceaccount.com, May 4

Labels: OS-Linux

Comment 3 by tobias@strongswan.org, May 4

```
This is caused by an asprintf() call where the return value is checked against > 0 and the case where the allocated string
is empty (i.e. len == 0) is not handled properly (i.e. the string is not freed). A similar bug has already been fixed
earlier, I'll push a fix for this one too.
```

Abbildung 37 Aktiver Dialog zwischen Google und Entwickler

Wird ein Problem behoben, merkt OSS-Fuzz dies. Der aktualisierte Source Code wird neu kompiliert und der resultierende Fuzzer wird mit Eingabewerten, welche vorher zu Problemen geführt haben (Crashfiles), automatisch erneut getestet. Kann der Fehler nicht mehr reproduziert werden gilt die Lösung als verifiziert und das Issue wird geschlossen.

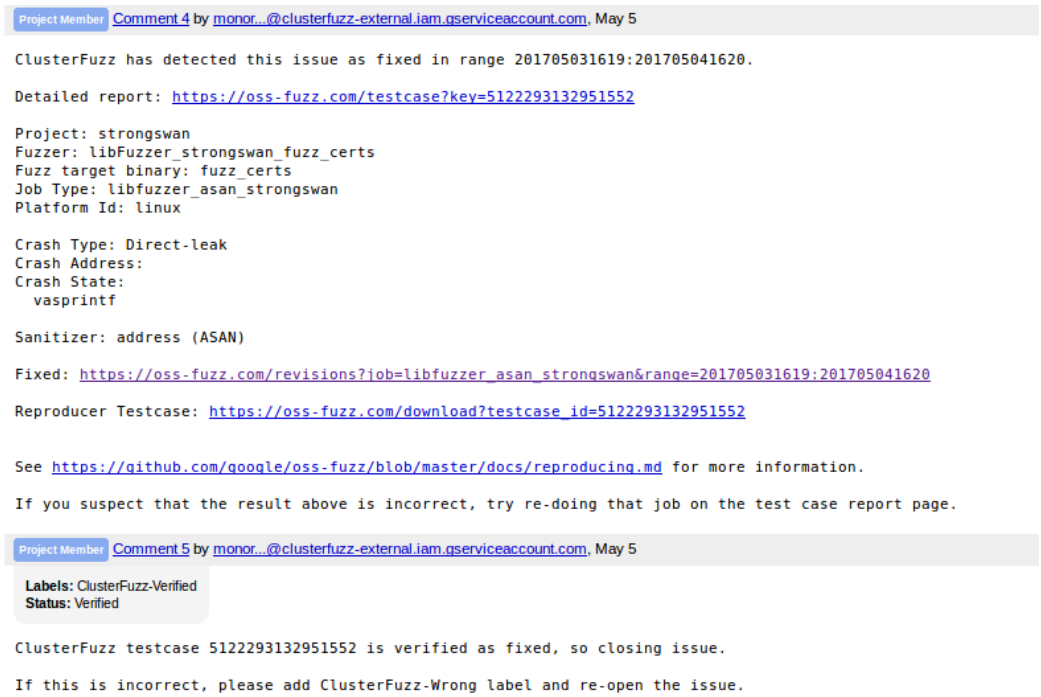
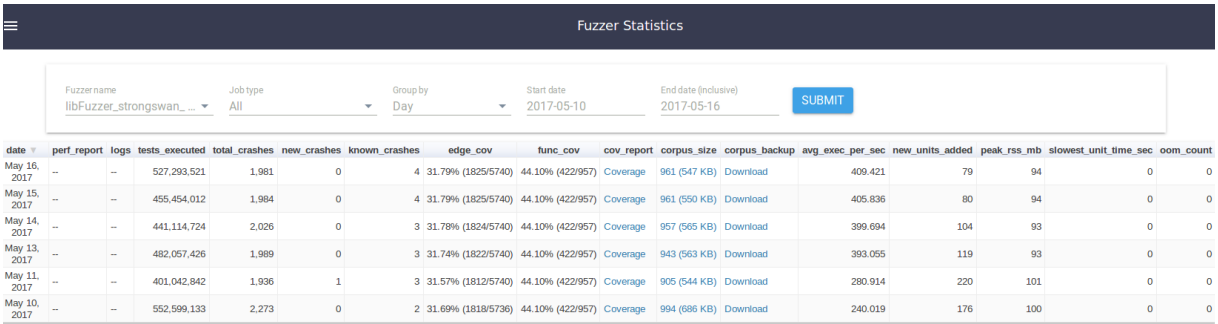


Abbildung 38 Verifikation eines Bugs und Schliessung des Issue

5.2.2.4 Statistiken

Die Plattform bietet ebenfalls eine Übersicht mit allgemeinen Statistiken. Darin sind diverse Informationen abzulesen: Erreichte Coverage des Codes, Anzahl Instruktionen pro Sekunden oder die totale Anzahl an Instruktionen.



date	perf_report	logs	tests_executed	total_crashes	new_crashes	known_crashes	edge_cov	func_cov	cov_report	corpus_size	corpus_backup	avg_exec_per_sec	new_units_added	peak_rss_mb	slowest_unit_time_sec	oom_count
May 16, 2017	--	--	527,293,521	1,981	0	4	31.79% (1825/5740)	44.10% (422/957)	Coverage	961 (547 KB)	Download	409.421	79	94	0	0
May 15, 2017	--	--	455,454,012	1,984	0	4	31.79% (1825/5740)	44.10% (422/957)	Coverage	961 (550 KB)	Download	405.836	80	94	0	0
May 14, 2017	--	--	441,114,724	2,026	0	3	31.78% (1824/5740)	44.10% (422/957)	Coverage	957 (565 KB)	Download	399.694	104	93	0	0
May 13, 2017	--	--	482,057,426	1,989	0	3	31.74% (1822/5740)	44.10% (422/957)	Coverage	943 (563 KB)	Download	393.055	119	93	0	0
May 11, 2017	--	--	401,042,842	1,936	1	3	31.57% (1812/5740)	44.10% (422/957)	Coverage	905 (544 KB)	Download	280.914	220	101	0	0
May 10, 2017	--	--	552,599,133	2,273	0	2	31.69% (1818/5736)	44.10% (422/957)	Coverage	994 (686 KB)	Download	240.019	176	100	0	0

Abbildung 39 Statistiken des OSS-Fuzz

Dank dem Bild wird die enorme Rechenleistung verdeutlicht, welche einem zugeteilt wird. Während bei lokalen Tests auf einem Notebook ca. 500 Instruktionen pro Sekunde erreicht wurden sind es bei Google ca. 400'000, was dem Faktor x800 entspricht.

6 Schlussfolgerung

6.1 Erreichte Ziele

Die Integration von Fuzzing in strongSwan wurde erreicht und die ersten Fuzzings konnten ebenfalls durchgeführt werden. Beim Fuzzing ist es nicht immer klar, ob Fehler entdeckt werden, im Falle der Arbeit wurden „glücklicherweise“ einige Probleme gefunden, welche die Effizienz und den Nutzen von Fuzzing unterstreichen. Diese Fehler wurden bereits behoben und sind danach in Form von Patches oder Fixes wieder in den Code eingeflossen.

6.2 Verbesserungen

Zu verbessern wäre in Zukunft die Art und Weise, wie strongSwan statisch gebaut werden kann. Es funktioniert zum Schluss, doch bei manchen Anpassungen handelt es sich eher um Workarounds, welche noch verbessert werden könnten. Für diese Änderungen sind aber tiefergehende Kenntnisse von strongSwan von Nöten als während der Dauer von 14 Wochen zusätzlich erarbeitet werden konnten.

Mit mehr Arbeitsleistung in Form von Studenten (2er Team) hätten eventuell mehr Fuzz Targets geschrieben werden können, was unter idealen Umständen in gesteigerter Codequalität resultiert hätte.

6.3 Zukunft

Mit dieser Arbeit wurde ein Anfang im Bereich strongSwan Fuzzing gemacht. Nach den gewonnenen Erfahrungen kann die Konzentration darauf gelegt werden, den Code genauer zu analysieren und weitere Teile der Software zu testen. Entsprechende Fuzz Targets lassen sich in Zusammenarbeit mit den Entwicklern finden. Schlussendlich gilt: Je mehr Teile mit Fuzzing abgedeckt werden, desto besser für das ganze Projekt.

Ein besonderer Anreiz dafür bietet Google seit Mai 2017. Das Unternehmen hat bekanntgegeben, dass ab nun für die ideale Integration von OSS-Fuzz in die eigene Software Geld an die Entwickler ausbezahlt wird[24]. Für Fuzzing Abdeckung von 80% des Codes gibt es einen Extra Zuschlag. Im Ganzen ist so eine Auszahlung von bis zu 20'000 Dollar möglich.

[24] (https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html)

7 Literaturverzeichnis

[Online] / Verf. http://llvm.org/docs/LibFuzzer.html#introduction. - 24. Mai 2017.

[Online] / Verf. http://llvm.org/docs/LibFuzzer.html#options. - 22. 05 2017.

[Online] / Verf. http://llvm.org/docs/LibFuzzer.html#output. - 24. 05 2017.

[Online] / Verf. http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf. - 19. 05 2017.

[Online] / Verf. http://www.llvm.org/. - 19. 05 2017.

[Online] / Verf. https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html. - 19. 05 2017.

[Online] / Verf. https://clang.llvm.org/docs/SanitizerCoverage.html#tracing-pcs-with-guards. - 25. 05 2017.

[Online] / Verf. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks. - 23. 05 2017.

[Online] / Verf. https://en.wikipedia.org/wiki/Trust_boundary. - 22. 05 2017.

[Online] / Verf. https://github.com/google/oss-fuzz. - 27. 05 2017.

[Online] / Verf. https://github.com/google/sanitizers. - 27. 05 2017.

[Online] / Verf. https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterReturn. - 27. 05 2017.

[Online] / Verf. https://github.com/mirrorer/afl/blob/master/dictionaries/xml.dict. - 27. 05 2017.

[Online] / Verf. https://github.com/rc0r/afl-fuzz/tree/master/dictionaries. - 27. 05 2017.

[Online] / Verf. https://gmplib.org/manual/Integer-Exponentiation.html. - 25. 05 2017.

[Online] / Verf. https://googleprojectzero.blogspot.ch/2015/02/feedback-and-data-driven-updates-to.html. - 24. 05 2017.

[Online] / Verf. https://msdn.microsoft.com/en-us/library/cc162782.aspx#Fuzzing_topic5. - 25. 05 2017.

[Online] / Verf. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. - 27. 05 2017.

[Online] / Verf. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html. - 26. 05 2017.

[Online] / Verf. https://tools.ietf.org/html/rfc6962. - 22. 05 2017.

[Online] / Verf. https://www.certificate-transparency.org/. - 26. 05 2017.

[Online] / Verf. https://www.certificate-transparency.org/known-logs. - 22. 05 2017.

[Online] / Verf. https://www.ietf.org/rfc/rfc5280.txt. - 25. 05 2017.

[Online] / Verf. https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/. - 22. 05 2017.

8 Abbildungsverzeichnis

9 Tabellenverzeichnis

10 Glossar

Begriff	Erklärung
Fuzz Target	Ein Programm oder eine Funktion welcher mittels Fuzzing getestet werden soll.
Fuzzer Engine	Die eigentliche Fuzzing Logik ist hier implementiert. In diesem Dokument libFuzzer.
Fuzzer	Ausführbare Datei, welche das Fuzz Target und die Fuzz Engine enthalten.
Corpus	Beinhaltet korrekte Daten für ein Fuzz Target welche von der Fuzzing Engine genutzt werden um Mutationen darauf anzuwenden.
Sanitizers	Verschiedene Libraries um Memory Fehler, „Undefined Behavior“ oder andere Bugs im Code zu entdecken. Sind sinnvoll in der Verwendung mit Fuzzing.
Dictionary	Wörterbücher welche das Fuzzing effizienter machen bei Fuzz Targets welche klar strukturierten Input erwarten.
Memory Leak	Speicher wird zwar alloziert aber nicht wieder freigegeben.
OSS-Fuzz	Continuous Fuzz Testing für Open Source Software von Google
CT	Certificate Transparency Projekt von Google.
SCT	Signed Certificate Timestamp wird genutzt um Certificate Transparency zu bewerkstelligen. Anhand dieses Timestamps funktioniert die Verifikation.
GMP	GNU Multiple Precision Arithmetic Library implementiert arithmetische Funktionen.
LDFLAGS	Zusätzliche Flags für den Linker. Nicht für Bibliotheken gedacht.
CFLAGS	Zusätzliche Compiler Flags
LDLIBS	Bibliotheken werden über dieses Flag dem Linker übergeben.
Clang	Compiler-Frontend für C, C++, Objective-C und Objective-C++ für das Compiler-Backend LLVM.