RUB

RUHR-UNIVERSITÄT BOCHUM

FUZZING MIT AFL

Martin Mielke

Aktuelle Themen der IT-Sicherheit – 2. März 2018.
Lehrstuhl für Systemsicherheit.

Supervisor: Prof. Dr. Thorsten Holz

Zusammenfassung

Sicherheitsrelevante Softwarefehler treten immer häufiger auf [1]. Um diese Fehler zu erkennen, bedient man sich verschiedener Techniken, unter anderem der dynamischen Programmanalyse mit dem Teilgebiet Fuzzing. In dieser Arbeit geht es um Fuzzing mit dem 2013 von Michael Zalewski entwickelten Tool American Fuzzy Lop (AFL). Zunächst werden grundlegende Thematiken wie Fuzzing, Arten von Fuzzing und Fuzzing-Strategien betrachtet und anschließend untersucht, wie AFL im Detail funktioniert. Danach werden zwei Beispielprogramme mit AFL praktisch untersucht, die Ergebnisse ausgewertet und ein Ausblick für die Zukunft gegeben, wobei auf hybride Ansätze eingegangen wird.

Inhaltsverzeichnis

1 Einleitung

Die Suche nach Fehlern in Software ist ein wichtiges Forschungsgebiet. Es treten immer häufiger sicherheitskritische Probleme in Software auf, die dann von Angreifern für ihre Zwecke ausgenutzt werden. So gibt Kaspersky an [1], dass im Vergleich zu 2015, in 2016 24,54% häufiger versucht wurde Exploits auszuführen. Insgesamt wurde dies 702 Millionen Mal weltweit versucht, wobei die häufigsten Arten Banking Trojaner oder Ransomware waren. Die häufigsten dabei auftretenden Fehlerarten sind Speicherzugriffsfehler, die oftmals dazu führen, dass der Programmfluss durch den Angreifer umgeleitet wird und dieser eigenen Code auf dem System ausführen kann.

1.1 Motivation

Aufgrund der Relevanz von Softwarefehlern und den oft kritischen Auswirkungen, ist die (automatisierte) Suche nach solchen Fehlern von besonderem Interesse. Auf breiter Front wird an Lösungen zum Finden von Softwareschwachstellen geforscht. Gängige Techniken sind hier statische Analyse, dynamische Analyse und symbolische Ausführung. American Fuzzy Lop (AFL) ist ein Tool zur dynamischen Analyse von Programmen. Es sucht automatisiert, mittels Fuzzing, nach solchen Schwachstellen und hat seit seiner Veröffentlichung im Jahr 2013 eine Vielzahl von kritischen Fehlern in über 200 Programmen gefunden und sich zum de facto Standardprogramm für Fuzzing entwickelt. Die besondere Relevanz von AFL wird deutlich, wenn man die im August 2016 stattgefundene Cyber Grand Challenge (CGC) [2] betrachtet. Hier traten verschiedene Teams an, automatisiert Softwareschwachstellen zu finden, zu beheben und Exploits für diese zu entwickeln. Interessant ist, dass viele Teams hierbei auf AFL und Weiterentwicklungen von AFL setzten, um Schwachstellen zu identifizieren. So setzte unter anderem das Gewinnerteam ForAllSecure mit ihrem Tool Mayhem auf eine Kombination von AFL und einer Symbolic-Execution-Engine. Ebenso setzte das zweitplatzierte Team Shellphish mit ihrem Tool Mechanical Phish auf eine solche Kombination.

1.2 Verwandte Arbeiten

Andere Arbeiten die sich mit AFL beschäftigen und als Basis für diese Arbeit dienen, sind unter anderem das offizielle Whitepaper zu AFL [3], der “Fuzzing with AFL“ Talk von

Adam DC949 auf der Circle City Con 2017 [4] und der Talk von Craig Young "Fuzz Smarter Not Harder - an AFL Fuzz Primer" auf der BSides 2017 [5]. Weitergehende interessante Arbeiten zu hybriden Ansätzen sind u.a. das Whitepaper zu Driller [6], einem hybriden Ansatz, der die Symbolic-Execution-Engine Angr mit AFL verbindet. Driller wurde u.a. auch von dem Shellphish Team in der CGC benutzt.

1.3 Aufbau

Im Grundlagen Teil wird zunächst ein Überblick über Arten von Softwarefehlern gegeben. Anschließend wird als Methode zur dynamischen Softwareanalyse Fuzzing vorgestellt und auf verschiedene Strategien, die dabei verwendet werden, eingegangen. Im darauffolgenden Kapitel wird ein Überblick über AFL und seine Funktionsweise gegeben. Anschließend wird in einem praktischen Teil AFL auf Beispielprogramme angewendet. Dabei werden Benutzung und Resultate untersucht und die Ergebnisse ausgewertet. Schließlich wird eine Zusammenfassung über AFL gegeben.

2 Grundlagen

Um die Funktionsweise und das Design von AFL nachzuvollziehen, werden in diesem Kapitel zunächst die zugrunde liegenden Arten von Schwachstellen und Techniken zur Suche derselbigen untersucht.

2.1 Schwachstellen und Schwachstellensuche

Es gibt viele verschiedene Arten von Schwachstellen in Software. Je nachdem um welche Arten von Programmen es sich handelt, zum Beispiel Webanwendungen, kompilierte C-Programme oder Skripte, gibt es typische Schwachstellen, die häufiger auftreten als andere. Typische Schwachstellen in kompilierten Programmen sind unter anderem Speicherfehler wie Buffer-Overflows, Use-After-Free-Zugriffe, Out-of-Bounds-Zugriffe, mangelhafte Eingabevalidierung wie zum Beispiel bei Formatstring-Schwachstellen und Type-Confusion-Schwachstellen.

Buffer-Overflow Schwachstellen entstehen, wenn bei Kopiervorgängen über die Grenzen eines Puffers hinweg geschrieben wird. Hier fehlt oftmals eine vorhergehende Prüfung, ob der zu kopierende Inhalt in den allozierten Puffer hineinpasst. Im Folgenden Beispiel 2.1 ist ein möglicher Stack-Buffer-Overflow in C dargestellt.

```
/* ... */
char buf[8]; // puffer fuer 8 zeichen
gets(buf); // liest beliebig viele zeichen, ueber puffergrenzen hinweg
printf("%s\n", buf);
/* ... */
```

Listing 2.1: Buffer-Overflow-Schwachstelle

Use-After-Free-Schwachstellen entstehen, wenn Speicherbereiche zunächst benutzt, dann freigegeben und anschließend fälschlicherweise noch einmal benutzt werden. Es kann nicht davon ausgegangen werden, dass der Inhalt des Speicherbereiches dann noch vorhanden ist, da der Speicherbereich anderweitig beschrieben sein kann. Ein Beispiel hierfür ist in 2.2 dargestellt.

```
char* ptr = (char*)malloc (SIZE);
...
if (err) {
  abrt = 1;
```

```
  free(ptr);
}
...
if (abrt) {
  logError("operation aborted before commit", ptr);
}
```

Listing 2.2: User-After-Free-Schwachstelle

Out-of-Bounds-Zugriffe, wie in 2.3 dargestellt, entstehen oftmals, wenn bei Array-Zugriffen ein dynamischer Index benutzt wird und dieser einen unerwarteten Wert erhält. So kann über das Ende eines Arrays hinaus zugegriffen werden oder über negative Indizes auf davor liegenden Speicher.

```
char *a = "12345"; // puffer fuer 5 elemente
printf("%s",a[5]); // zugriff auf 6. element
```

Listing 2.3: Out-Of-Bounds-Zugriff

Bei Formatstring-Schwachstellen wird ausgenutzt, dass Funktionen wie scanf oder printf Steuerzeichen wie %x oder %n interpretieren. Enthält eine Benutzereingabe solche Steuerzeichen und wird ungefiltert, an beispielsweise printf, übergeben, kann ein Angreifer Speicher auslesen und schreiben. Die Schwachstelle ist exemplarisch in dem vorgestellten Buffer-Overflow-Beispielcode ebenfalls enthalten.

Type-Confusion-Schwachstellen entstehen, wenn ein Datentyp für einen anderen Datentyp gehalten wird. Dies ist oft bei impliziten Umwandlungen von Datentypen der Fall. Besonders häufig treten diese Fehler bei der Umwandlung zwischen signed und unsigned Werten auf. Unter anderem ist dieser Fehler bei einem bekannten Exploit, der auf CVE-2013-2094 basiert, aufgetreten. Hier wurde ein unsigned 64-Bit Integer in ein signed 32-Bit Integer konvertiert. Das führt dazu, dass die Prüfung “`>= PERF_COUNT_SW_MAX`“ mit negativen Werten umgangen werden kann und so ein Inkrement an einer Nutzer kontrollierten Adresse erfolgen kann, was in 2.4 zu sehen ist.

```
struct perf_event_attr {
  /* ... */
  __u64                 config;
  /* ... */
static int perf_swevent_init(struct perf_event *event)
{
  int event_id = event->attr.config;
  /* ... */
  if (event_id >= PERF_COUNT_SW_MAX)
    return -ENOENT;
  /* ... */
  atomic_inc(&perf_swevent_enabled[event_id]);
  /* ... */
```

Listing 2.4: Type-Confusion-Schwachstelle

Es gibt verschiedene Techniken solche Schwachstellen in Programmen zu finden. Häufig wird dabei zwischen statischer und dynamischer Programmanalyse unterschieden.

Bei der statischen Programmanalyse wird das Zielprogramm in Ruhe analysiert, das heißt ohne ausgeführt zu werden. Dabei wird, falls vorhanden, Quelltext sowohl manuell als auch automatisiert untersucht, mit dem Ziel Fehler im selbigen zu finden. Typische Fehler die hier gefunden werden können, sind Formatstring-Schwachstellen, Buffer-Overflows, Memory Leaks, Out-of-Bounds-Zugriffe und vieles mehr. Falls der Quellcode nicht vorhanden ist, kann der Binärcode, zum Beispiel in einem Dissassembler, untersucht werden. Um mittels statischer Analyse Fehler finden zu können, werden umfassende Kenntnisse über die verwendete Programmiersprache beziehungsweise Architektur benötigt, sowie weitreichende Kenntnisse über potentielle Fehler die enthalten sein können. Dies erfordert hohe Fachkompetenz beim Durchführenden. Die Schwierigkeit bei der statischen Analyse liegt zudem darin, dass sie zum einen sehr aufwendig ist, da das Verhalten des Programms abgeleitet und schrittweise nachvollzogen werden muss, und zum anderen darin, dass Abhängigkeiten mit der Ausführungsumgebung (Library-Versionen, Betriebssystem, und vieles mehr) nicht betrachtet werden können.

Um diese Nachteile zu vermeiden, werden oft in Ergänzung zur statischen Analyse dynamische Verfahren zur Programmanalyse verwendet. Dabei wird das zu untersuchende Programm ausgeführt und sein Verhalten beobachtet. Durch die reale Ausführung erhält man so ein realistisches Ausführungsszenario. Um hier Fehler zu finden wird unter anderem Fuzzing verwendet, welches in der Regel deutlich weniger Detailkenntnisse vom Durchführenden verlangt.

2.2 Fuzzing

Fuzzing wird oftmals in klassisches Fuzzing, Template-basiertes Fuzzing und Smart-Fuzzing unterteilt. Ursprünglich bestand die Idee von Fuzzing darin ein Programm, das Benutzereingaben in irgendeiner Form erwartet, zum Beispiel per Stdin oder als Datei, mit zufälligen Eingaben auszuführen und dabei dynamisch das Programmverhalten zu beobachten. Führen bestimmte Eingaben zu Programmabstürzen kann davon ausgegangen werden, dass ein Fehler beziehungsweise eine Schwachstelle existiert. Vorteil dieses Verfahrens ist, dass es, verglichen mit moderneren Verfahren, sehr schnell ist und dass es leicht durchzuführen ist. Häufig werden hier selbst geschriebene Tools verwendet oder beispielsweise das von Sam Hocevar 2002 entwickelte “zzuf“. Da diese Art des Fuzzens sehr zufallsabhängig ist und viele sinnlose Testfälle erzeugt, wurden Template basierte Fuzzer entwickelt. Der wesentliche Unterschied besteht hier darin, dass a priori Wissen über Dateiformate beziehungsweise Strukturen genutzt wird, um die Testfälle auf die Zielprogramme zuzuschneidern. Wir nehmen an, dass ein Zielprogramm den Dateinamen einer ELF-Datei per Kommandozeile erwartet und diese dann ausführt. ELF Dateien haben zu Beginn die Bytes “0x7FELF“. Sind diese Bytes nicht

vorhanden bricht das Zielprogramm mit einer Fehlermeldung ab, wie beispielhaft in 2.5 dargestellt.

```
assert(magic == b'\x7fELF', 'No ELF-File')
```

Listing 2.5: ELF Magic-Number

Ein zufallsbasierter, klassicher Fuzzer hätte große Schwierigkeiten diese Bedingung zu erfüllen. Selbst wenn einmal die Bytefolge erzeugt würde, würde der Zustand nicht gespeichert und weiterverwendet. Template-basierte Fuzzer erlauben es entsprechende Vorlagen zu erstellen, in die Wissen aus Vorabanalysen eingehen. Hier exemplarisch ein Template des beliebten Fuzzing-Tools "spike" [7], welches das Problem verhindert, indem es den Anfang der Datei vorgibt und erst danach variablen Inhalt erzeugt. Ein Spike-Template ist in 2.6 dargestellt.

```
s_string("0x7fELF ");
s_string_variable("0");
```

Listing 2.6: Spike Beispiel-Template

Der Vorteil ist hier, dass die Testfälle so eingeschränkt werden, dass eine große Anzahl unsinniger Fälle von vornherein ausgeschlossen werden kann. Nachteil ist aber, dass eine umgehende Analyse des zu fuzzenden Programmes und seiner erwarteten Eingaben vorab erfolgen müssen.

Smart-Fuzzing wiederum hat diesen Nachteil nicht. Bei Smart-Fuzzing basierten Verfahren wird die Programmausführung dynamisch verfolgt. Dies wird auch als Tracing bezeichnet. Dies hat den Vorteil, dass der Fuzzer speichert in welche Bereiche des Programms bestimmte Eingaben geführt haben. Die Technik, wie sie auch von AFL verwendet wird, wird im nächsten Kapitel detailliert vorgestellt. Der Vorteil hierbei ist es, das man als Metrik für die Qualität des Fuzzings verwenden kann, wieviel Code durch die generierten Eingaben in Form von Testfällen abgedeckt wird. Dies wird auch als Coverage-Based-Fuzzing bezeichnet.

2.3 Fuzzing Strategien

In diesem Abschnitt werden verschiedene Strategien betrachtet, die Fuzzer verwenden um Eingaben zu generieren. Fuzzer starten in der Regel mit sogenannten "Seeds", das sind Dateien oder Bytefolgen die vom Programm als reguläre Eingaben erwartet werden. Ausgehend von diesen Seeds verändert der Fuzzer bestimmte Bereiche der Eingabe, führt das Programm erneut aus und beobachtet das Verhalten. Strategien um die Eingaben zu verändern, die auch von AFL benutzt werden, sind Bit-Flipping, Byte-Flipping, arithmetische Veränderungen, und spezielle Werte. Bit-Flipping ist dabei ein Verfahren,

bei dem einzelne Bits der Eingabe zufällig geändert werden. Beim Byte-Flipping werden komplette Bytes verändert und umgekehrt. Bei arithmetischen Veränderung werden Werte, in verschiedenen Speicherbreiten (oftmals 8-Bit, 16-Bit, 32-Bit, 64-Bit), addiert und subtrahiert. Schließlich werden spezielle Werte als Teil der Eingabe ausprobiert, wie zum Beispiel maximal Werte von Integern, der Wert 1 oder 0, oder Werte aus speziellen Wörterbüchern. Smart-Fuzzer benutzen zusätzlich Strategien, anhand derer sie entscheiden welche Pfade innerhalb des untersuchten Programmes sie durchlaufen. Betrachtet man ein Programm als Kontrollflussgraph, können verschiedene bekannte Strategien zum Traversieren von Graphen benutzt werden. Einfache Verfahren sind zum Beispiel die Breitensuche (BFS - breadth-first-search) und die Tiefensuche (DFS - depth-first-search).

2.4 Programmfluss

Eine Darstellungsart für Programme ist die Darstellung als Kontrollflussgraph. Dabei stellen die Knoten Mengen von Instruktionen dar, sogenannte Basic Blöcke und die Kanten mögliche Übergänge zwischen diesen Basic Blöcken. Basic Blöcke sind dabei eine Menge von Instruktion die bestimmte Kriterien erfüllt, die im wesentlichen Aussagen, das Basic Blöcke nur einen definierten Eingang und Ausgang haben und Instruktionen innerhalb einzelner Basic Blöcke keine Ziele von Sprüngen innerhalb des Programmes sein können. Das Fuzzen eines Programmes im Coverage-Based-Ansatz lässt sich so beschreiben, dass möglichst viele Pfade durch das Programm gefunden werden und möglichst viele Basic Blöcke dabei erreicht werden sollen.

3 AFL - Überblick

In diesem Kapitel wird ein Überblick über American Fuzzy Lop (AFL) und seine Funktionsweise gegeben. AFL ist ein von Michael Zalewski entwickelter Smart-Fuzzer, der sich durch seine hohe Erfolgsquote beim Auffinden von Schwachstellen, hohe Geschwindigkeit, Zuverlässigkeit und einer besonders einfachen Bedienbarkeit auszeichnet. AFL hat seit seiner Veröffentlichung unzählige Bugs in Programmen gefunden, unter anderem in OpenSSL, Firefox, Internet Explorer, FFmpeg, PHP, PuTTY und vielen mehr. Insgesamt sind auf der Webseite von AFL (http://lcamtuf.coreduimp.cx/afl/) über 200 der wichtigsten Programme gelistet, in denen AFL Fehler gefunden hat. Bei der Cyber Grand Challenge 2016 war AFL zudem ein integraler Bestandteil der Software vieler Teams. So haben die ersten beiden Plätze, Mayhem von ForAllSecure und Mechanical Phish von Shellphish, Kombinationen von AFL mit symbolischen Solvern benutzt.

3.1 Funktionsweise

AFL ist ein Smart-Fuzzer. Wie im Grundlagenkapitel beschrieben, wird hierbei der Programmablauf verfolgt und anhand der genommen Pfade durch den Code der Fuzzing-Vorgang gelenkt. Um den Programmablauf zu verfolgen, verwendet AFL standardmäßig sogenannte Compile-Time-Instrumentierung. Das bedeutet, dass das Zielprogramm mit einem speziellen Compiler (afl-gcc, afl-clang) gebaut werden muss. Beim Kompilieren werden von AFL dabei zusätzliche Instruktionen in das Programm eingefügt, die es AFL ermöglichen, zu verfolgen, in welchem Basic Block sich die Ausführung zu jedem Zeitpunkt befindet und welche Pfade genommen wurden. Die eingefügten Instruktionen gestalten sich wie in 3.1 dargestellt.

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location^prev_location]++;
prev_location=cur_location>>1;
```

Listing 3.1: AFL Instrumentierung

Die durchlaufenden Pfade beziehungsweise Basic-Block-Übergänge werden dabei in einer Shared-Memory-Region gespeichert. Es gibt einige auf AFL aufbauende Projekte, die für die Ablaufverfolgung andere Wege gewählt haben. So verwendet eine Windows-Variante von AFL [8] zum Beispiel DynamoRIO als Tracer.

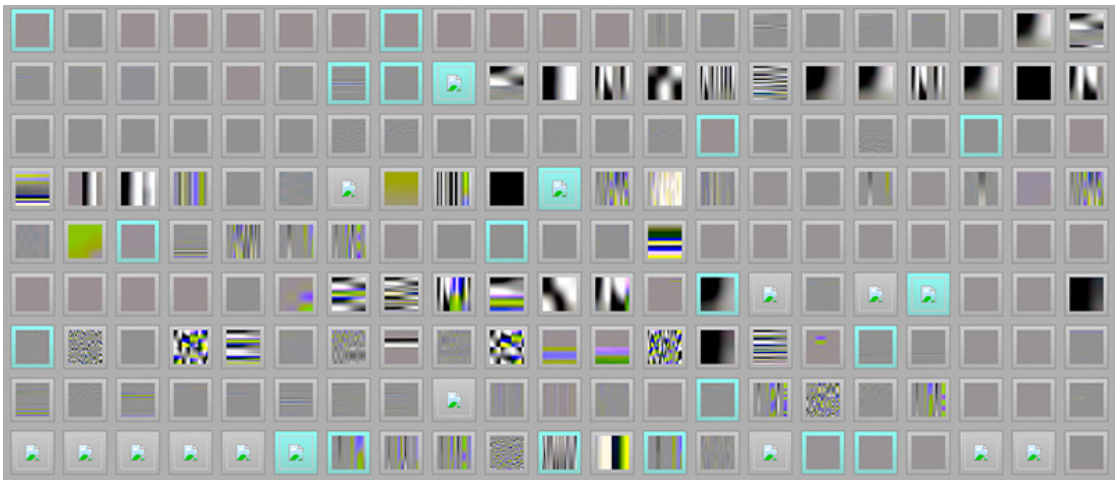

Abbildung 3.2: AFL generiert Bilder

Um das Zielprogramm zu fuzzen verwendet AFL Seed-Dateien. Diese Dateien sind Eingaben, die das Programm normalerweise erwartet und verarbeiten kann. Ausgehend davon werden beim Fuzzing-Vorgang zufällige Varianten der Seeds als Eingaben verwendet. Zum Mutieren der Eingaben verwendet AFL dabei Bit-Flipping, Byte-Flipping, Arithmetische Operationen und Spezielle Werte wie Wörterbucheinträge, maximale und minimale Werte von Datentypen und vieles mehr. Seed-Dateien sind grundsätzlich jedoch nicht nötig für das Fuzzing mit AFL. Die Abbildung 3.2 zeigt einen Bildbetrachter, der mit einer Textdatei, die nur "hello" enthielt gefuzzed wurde. AFL hat dabei selbstständig Bilddateien im JPEG-Format erzeugen können.

Das Verwenden von guten Seed-Dateien beschleunigt allerdings erheblich den Fuzzing-Vorgang, da die Ausführung schneller in interessante Regionen gelenkt wird. Dabei ist zu beachten, dass die Testfälle möglichst kleine Dateien sein sollten, um Mehraufwand zu verhindern.

Der Ablauf des Fuzzing Vorgangs ist dabei wie folgt. Zunächst wird das Zielprogramm bis zur Main-Funktion ausgeführt. AFL speichert den Zustand und forked ab hier in die einzelnen Testdurchläufe. Dies hat den Vorteil, dass Mehraufwand durch häufiges erzeugen neuer Prozesse verhindert wird. AFL lädt nun die vom Benutzer vorgegeben Testfälle und führt das Programm damit aus. Anschließend minimiert es diese Testfälle soweit es möglich ist, ohne dass der Ausführungspfad sich ändert. Anschließend verändert AFL, mithilfe der Fuzzing Strategien, diese Eingaben und speichert welche Eingaben in welche neuen Pfade im Programm führen. Dies wird für alle implementierten Strategien durchgeführt. Eingaben, die nicht zu neuen Pfaden führen, werden hier sofort verworfen. Nachdem der Zyklus für alle Strategien durchlaufen ist, werden die gespeicherten Testeingaben, die nun alle einzigartige Pfade erzeugen, als Seed für einen neuen Durchlauf verwendet. Dies wird endlos durchgeführt bis der User das Fuzzing be-

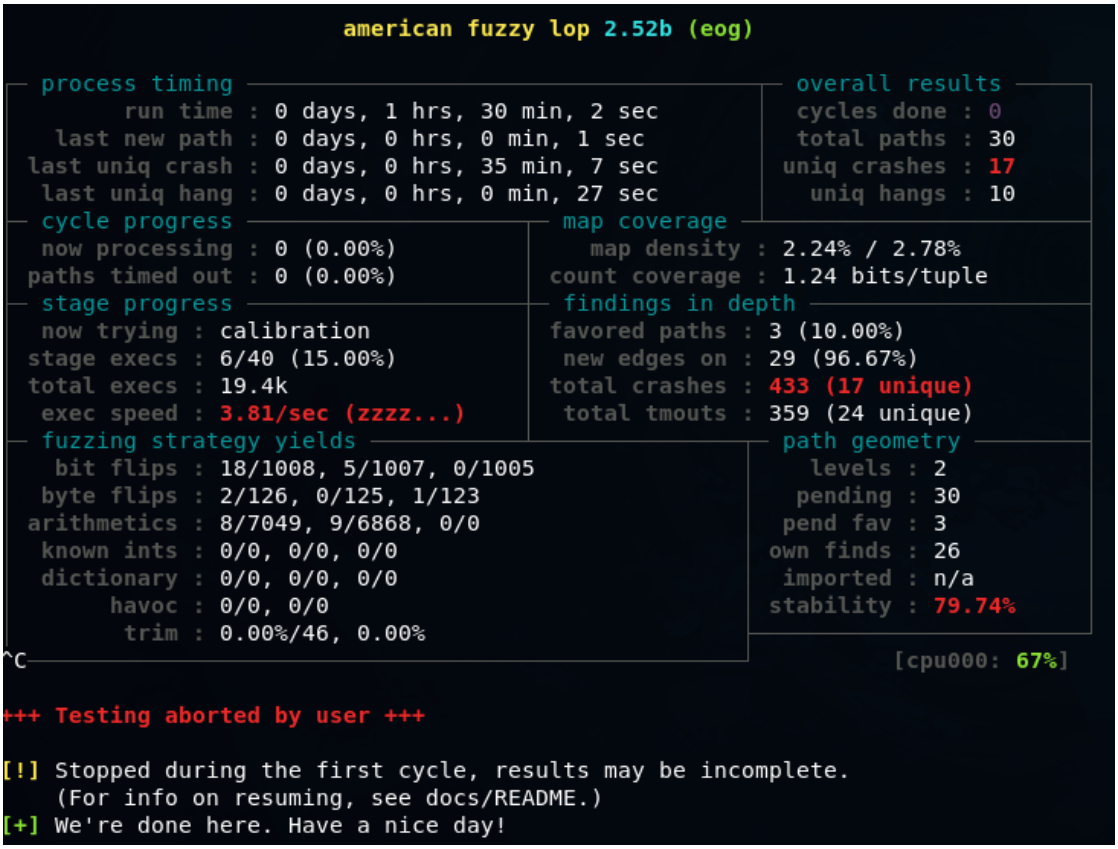

Abbildung 3.3: Fuzzing SoX

endet. Was AFL hier so effektiv macht, ist das nur neue Pfade weiter verfolgt werden und viele Testfälle die zu gleichen Ergebnissen führen würden, vermieden werden. Man bezeichnet diese Art des Fuzzings auf Basis des Programmflusses als Coverage-Based-Fuzzing. Während des Fuzzings gestaltet sich die Ausgabe von AFL wie in Abbildung 3.3 dargestellt.

Besonders interessant sind hier vor allem die “overall results“, welche anzeigen, wie viele komplette Zyklen AFL durchlaufen hat, wie viele Pfade es identifiziert hat, wie viele Eingaben zu Abstürzen in eigenen Pfaden geführt haben und wieviel “hangs“ es gab. Hangs sind dabei Ausführungen, die in das festgelegte Timeout gelaufen sind. Interessant sind zudem bei “stage progress“ in welcher Phase sich der Fuzzer aktuell befindet und wie viele Ausführung pro Sekunde, beziehungsweise wie viele Ausführungen insgesamt von AFL durchgeführt werden. In diesem Beispiel ist zu sehen, dass die Anzahl der Ausführungen pro Sekunde sehr gering ist (3.81/s), was in der Regel auf einen Fehler hindeutet. In diesem Fall liegt es jedoch daran, dass eine GUI-Applikation gefuzzed wird und dies erheblichen zusätzlichen Aufwand erzeugt. Das Feld “now trying“ kann die Werte calibration, trim L/S, bitflip L/S, arith L/8, intereset L/8, extras, havoc, splice

und sync annehmen. In der Phase Calibration befindet sich der Fuzzer in einer Vorbereitungsphase, in der der Ausführungspfad analysiert wird, die Ausführungsgeschwindigkeit und gegebenenfalls Anomalien. Erreicht der Fuzzer die Phase trim L/S, werden Testcases so minimiert, dass sie immer noch den selben Ausführungspfad nehmen, aber an Komplexität abnehmen. In der Phase Bitflip L/S werden deterministisch einzelne Bits umgekehrt. Kommt der Fuzzer zur Phase arith L/8, versucht er 8-,16- und 32-Bit Werte zu subtrahieren. Interest L/8 bedeutet, dass spezielle 8-,16- und 32-Bit Werte in die Testeingabe geschrieben werden. In der Phase extras werden spezielle Wörterbucheinträge, die manuell oder automatisch erzeugt wurden, in die Testeingabe geschrieben. Havoc führt verschiedene zufällige Mutationen durch und kombiniert andere bereits vorgestellte Methoden. Splice ist schließlich die letzte Fuzzing-Technik, die AFL anwendet, wenn nach einem vollen Durchgang keine neuen Pfade gewunden werden. Dabei werden die gleichen Methoden wie in Havoc angewendet, jedoch werden vorher 2 verschiedene Eingaben an einem zufälligen Punkt zusammengeschlossen, um eine neue Eingabe zu erhalten. Sync ist ein Zusatzschritt, der ausgeführt wird, wenn die Parallelisierung von AFL genutzt wird. Dabei werden Zwischenergebnisse von verschiedenen AFL-Instanzen zwischen diesen hin und her kopiert.

Identifiziert AFL Crashes werden diese mit einem speziellen Dateinamen abgelegt in der Form “id:000001,sig:11,src:000015,op:int8,pos:11,val:+0“. Dies bedeutet, dass die Eingabe im Programm zur Auslösung von Signal 11 geführt hat, also einem Segmentation Fault. Die anderen Bestandteile geben die Identifikationsnummer der Eingabe an und an welchen Stellen Veränderungen gegenüber der Seed-Datei durchgeführt wurden.

3.1.1 Parallelisierung

AFL unterstützt Parallelisierung und kann auf mehreren Prozessorkernen ausgeführt werden. Dazu definiert man beim Start eine Master-Instanz, ein gemeinsames Synchronisierungs-Verzeichnis, und eine Anzahl von Slave-Instanzen. Das Fuzzing wird so auf mehrere Kerne verteilt und die Ergebnisse in der Master-Instanz aggregiert.

3.1.2 Address Sanitizer

Der Address Sanitizer (ASAN) ist keine direkte Funktion von AFL, sondern ein Feature, dass moderne Compiler integriert haben. ASAN sorgt dafür, dass Speicherfehler bei der Ausführung erkannt werden, das Programm beim Auftreten eines solchen Fehlers sofort terminiert wird und Informationen zum Fehler anzeigt werden. Typische Fehler die ASAN findet sind Buffer-Overflow- und Use-After-Free-Fehler. In Kombination mit AFL ist ASAN besonders effektiv, da viele verschiedene Programmpfade durch das Fuzzing durchlaufen werden und so die Speicherprüfungen den Code großflächig abdecken.

3.1.3 Netzwerk Fuzzing

Grundsätzlich unterstützt AFL nur Fuzzing über Stdin und Eingabedateien. Es gibt mit preeny [9] jedoch eine Möglichkeit, auch Programme die Netzwerkeingaben erwarten mit AFL zu fuzzen. Preeny benutzt LD_Preload um relevante Systemcalls zu hooken. Statt echten Netzwerkverkehr zu erzeugen kann AFL so wie üblich Dateien erzeugen und die Lese-Operation vom Socket werden von preeny auf Dateien umgebogen.

3.1.4 AFL Qemu Modus

AFL besitzt für den Fall, dass kein Quellcode vorliegt, den sogenannten Qemu-Modus. Dabei wird anstatt der Compile-Time-Instrumentierung, das Zielprogramm mit einer modifizierten Version des Emulators Qemu ausgeführt und darüber getraced. Der Nachteil hier ist, dass das Tracing durch Qemu um ein vielfaches langsamer ist, als das Tracing mittels Compiler.

3.1.5 AFL für Windows

Es gibt einen Windows-Fork von AFL [8], der grundsätzlich wie die Linux Version funktioniert, jedoch auf DynamoRIO als Tracer setzt. Die Geschwindigkeit unter Windows ist deutlich langsamer, zum einen durch DynamoRIO selbst und zum anderen da der Mechanismus des Forkens an der Main Funktion auf Windows nicht funktioniert (Windows hat keinen Fork-Systemcall). AFL kann zudem nur sehr langsam arbeiten unter Windows, da die meisten getesteten Programme GUI-Applikationen sind. Während unter Windows damit nur wenigen Ausführungen pro Sekunde möglich sind, sind es unter Linux mehrere tausend Ausführungen pro Sekunde. Netzwerk-Fuzzing mit preeny ist für Windows ebenfalls nicht möglich.

3.1.6 AFL für Python

Desweiteren existiert eine Version von AFL, mit der Programme, die in Python geschrieben sind gefuzzed werden können [10]. Als Tracer verwendet das Projekt die Python-Funktion “settrace“. Die Ausführungsgeschwindigkeit ist dadurch deutlich geringer als beim Standard-AFL.

4 AFL - Nutzung

In diesem Kapitel wird die praktische Nutzung von AFL vorgestellt. Es werden Testprogramme ausgewählt, vorbereitet und mit AFL gefuzzed. Anschließend werden die Ergebnisse ausgewertet.

4.1 Auswahl von Programmen

Einer der wichtigsten Schritte beim Fuzzing ist die Auswahl der zu untersuchenden Programme. Damit AFL effektiv verwendet werden kann, müssen bestimmte Kriterien erfüllt sein. Zum einen muss das Zielprogramm Eingaben von der Kommandozeile akzeptieren und über Stdin oder in Dateiform Daten akzeptieren und verarbeiten. Anschließend muss das Programm sich selbst beenden. Bei Programmen wo dies nicht der Fall ist, muss ein sogenannter Test-Harness geschrieben werden, der dafür sorgt, dass diese Bedingungen erfüllt werden oder der Quellcode des Testprogrammes muss so angepasst werden, dass sich das Programm nach erreichen eines definierten Punktes beendet.

Für diese Arbeit wurden die Programme Eye of Gnome (eog) und Sound eXchange (SoX) ausgewählt.

Das Programm eog ist ein quelloffener Bildbetrachter, der Teil des Gnome Projektes ist. Programme, die für Medienkonvertierung oder Darstellung genutzt werden, eignen sich grundsätzlich gut für Fuzzing, da die Angriffsfläche durch viele unterschiedliche Formate und deren Eigenheiten relativ groß ist. Der Nachteil bei GUI-Anwendungen ist jedoch die geringe Ausführungsgeschwindigkeit. Eog wurde ausgewählt, um exemplarisch eine GUI-Anwendung zu testen. Zudem kann hier gezeigt werden, wie Programme modifiziert werden können um mit AFL getestet zu werden.

Als zweites Programm wurde SoX ausgewählt. Es handelt sich um ein Programm welches zwischen verschiedenen Soundformaten konvertiert und dazu eine Vielzahl von (teilweise exotischen) Formaten implementiert. Deshalb wird hier auch eine relativ große Angriffsfläche vermutet. Da SoX auf der Kommandozeile funktioniert und Input in Dateiform erwartet, eignet es sich zudem gut für den Test.

4.2 Vorbereitung und Durchführung

4.2.1 Eye of Gnome

Beim Betrachten der Dokumentation von eog wird klar, dass das Programm von der Kommandozeile gestartet werden kann und als Parameter eine Bilddatei in einem der folgende Formate erwartet: ani, bmp, gif.ico, jpg, pcx, png, pnm, ras, svg, tga, tiff, wbmp, xbm und xpm. Um das Programm mit AFL zu fuzzen, ist es nötig, dass nach erfolgreicher Darstellung eines Bildes, das Programm beendet wird. Um dies zu erreichen, muss der Quellcode gelesen beziehungsweise der Programmablauf nachvollzogen werden. Beim Studium des Quellcodes konnte zunächst keine offensichtliche Stelle nach dem vollständigen Laden eines Bildes gefunden werden, so dass der Ablauf dynamisch in gdb nachvollzogen werden musste. Dazu wurde das Programm in gdb gestartet und auf alle Funktionen die mit "eog_" beginnen, ein Breakpoint gesetzt. Anschließend wurde solange durch den Code gestepped, bis das Bild angezeigt wurde. Es zeigt sich, dass eog ein Bild fertig geladen hat, wenn "display_draw" in "eog-scroll-view.c" zum zweiten Mal aufgerufen wird. Daher wird der Code in 4.1 am Anfang der Funktion eingefügt.

```
static int count = 0;
/* ... */
if (count == 1)
  exit(0);
count++;
```

Listing 4.1: Eog - Anpassung

Mit diesen Änderungen beendet sich eog nach der Darstellung eines Bildes sofort und ist für das Fuzzing mit AFL bereit.

Als Kandidaten für mögliche Seed-Werte bietet es sich an, ein Exemplar für jedes Bildformat zu finden oder zu generieren. Dabei ist zu beachten, dass möglichst kleine Bilddateien gewählt werden. Die Veränderung von Werten einzelner Pixel im Bild ist in der Regel nicht sehr zielführend (man möchte eher Metainformationen treffen) und die Performance des Fuzzvorgangs verschlechtert sich mit zunehmender Seed-Dateigröße deutlich. Zum anderen sollte möglichst alle Bildformate gefuzzed werden, um ein umfassendes Ergebnis zu erhalten, welches möglichst viele Fehler abdeckt.

AFL wurde mit "`afl-fuzz -m 100000 -i in -o -t 3000 out eog @@`" und einer Beispielbilddatei für jedes Format gestartet. Die Parameter sind bis auf den Timeout-Wert Standardwerte. Da die Bilddarstellung einige Sekunden brauchen kann, je nach Größe der mutierten Bilder, ist ein Timeout angepasst auf die Rechenleistung des Systems nötig. Hier wurden daher drei Sekunden als Timeout-Wert gewählt, um eine Balance zwischen Performance und Vermeiden von Fehlern zu finden.

4.2.2 Sound eXchange

SoX konvertiert Audiodateien zwischen verschiedenen Formaten. Dazu wird auf der Kommandozeile im einfachsten Fall "`sox src dst.wav`" benutzt, um eine Quelldatei in eine Wave-Dateien zu konvertieren. Nach erfolgter Konvertierung beendet sich das Programm. Die Voraussetzungen für AFL sind also gegeben, ohne dass Modifikation am Quellcode oder ein spezieller Test-Harness nötig sind.

Da es sich nicht um eine GUI-Anwendung handelt, bietet es sich an die Ausführung zu parallelisieren. Dazu wird AFL mehrfach gestartet. Eine Instanz verwendet dabei den Parameter "`-M Name`" als Master-Instanz und alle anderen "`-S Name'`" als Slave-Instanzen. Wichtig ist dabei, dass alle Instanzen dasselbe Ausgabeverzeichnis angeben, da AFL den Fuzzing Vorgang so zwischen den Instanzen synchronisieren kann.

4.3 Fuzzing Ergebnisse und Auswertung

Die Testprogramme wurden jeweils mindestens 90 Minuten gefuzzed und die Testvorgänge danach manuell terminiert. Insgesamt wurde mehrere Testfälle gefunden, die die Programme zum Absturz bringen. Obwohl AFL nach eigener Aussage keine doppelten Fehlercases erzeugen sollte, wurde beobachtet, dass viele Testfälle zu Abstürzen an denselben Codestellen führen. Dies liegt vermutlich daran, dass andere Pfade genommen wurden, aber letztendlich der Absturz dennoch an derselben Stelle erfolgte. Die Testfälle wurden zur Verifikation ausgeführt und beobachtet, ob die Abstürze tatsächlich auftreten. Dies war bei allen Testfällen der Fall. Im folgenden werden ausgesuchte Crashes untersucht und die Programmfehler nachvollzogen.

4.3.1 Eye of Gnome

Am Ende des Fuzzing-Vorganges hatte AFL die in 4.2 zu sehende Ausgabe erzeugt.

Es ist zu sehen, dass das Programm 19400 mal ausgeführt wurde und dabei 30 verschiedene Pfade durch das Programm genommen wurden. In 17 Pfaden wurden Crashes beobachtet. Ein Blick auf die Dateinamen der Crashes zeigt, dass alle Abstürze mit Signal 5, SIGTRAP, erfolgt sind. Um die Ursache des Crashes zu ermitteln, wird eog in gdb mit der speziellen Eingabe gestartet, was zu dem Fehler in 4.3 führt.

```
(eog:3843): Gdk-ERROR **: The program 'eog' received an X Window System error.
This probably reflects a bug in the program.
The error was 'BadAlloc (insufficient resources for operation)'.
...
Thread 1 "eog" received signal SIGTRAP, Trace/breakpoint trap.
```

Listing 4.3: Eog - SIGTRAP

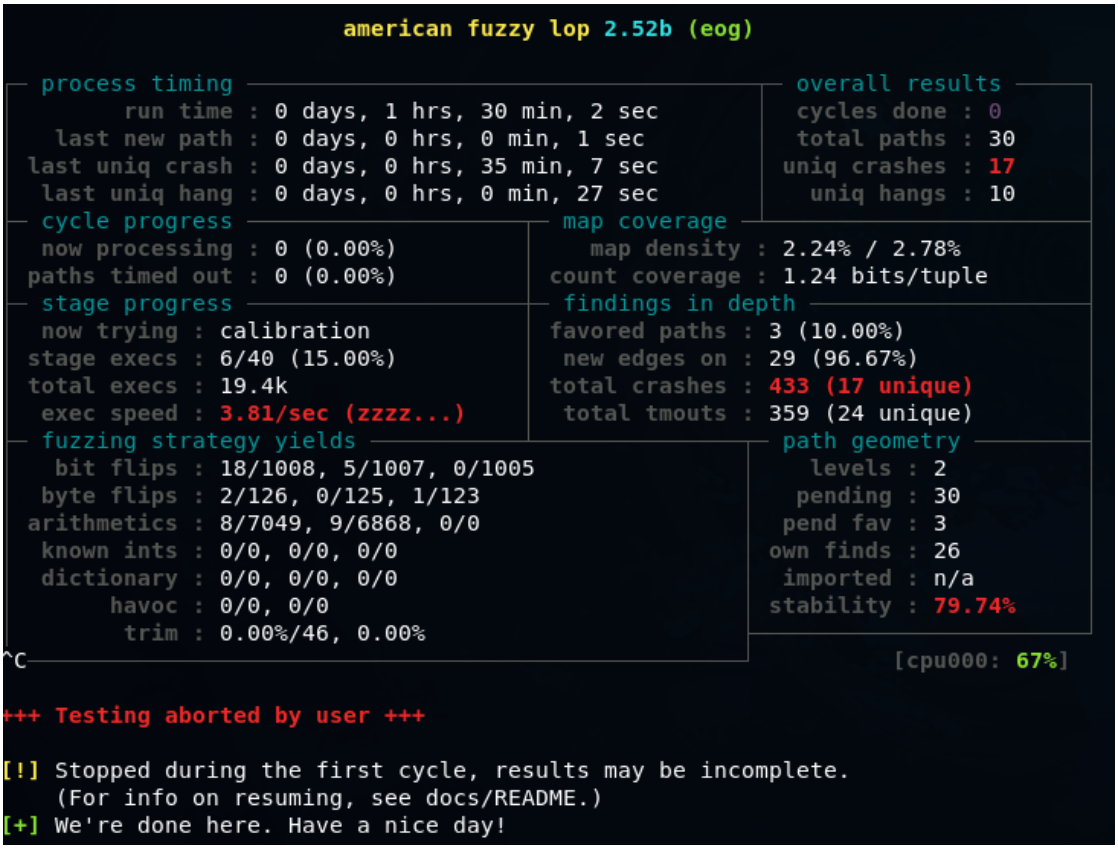

Abbildung 4.2: Fuzzing EOG

Dies zeigt, dass eog einen Fehler im X-Window-System ausgelöst hat. Die Eingabe, die von AFL erzeugt wurde, ist ein JPEG-Datei mit 200x32912 Pixeln.

4.3.2 Sound eXchange

Am Ende des Fuzzing-Vorganges von sox hatte AFL die in 4.4 dargestellte Ausgabe.

Es ist zu sehen, das das Programm über eine Million Mal ausgeführt wurde und dabei 105 verschiedene Pfade durch das Programm genommen wurden. In 7 Pfaden wurden Crashes beobachtet. Eine der Eingaben die zum Absturz führen trägt den Namen “id:000001,sig:11,src:000015,op:int8,pos:11,val:+0“, daraus lässt sich ableiten, dass die Datei zu einem Segmentation Fault geführt hat. Um die Ursachen des Crashes zu ermitteln, wird das Programm im Debugger, hier GNU Debugger (gdb), mit der speziellen Eingabe gestartet. Wie erwartet wird das Programm mit einem Segmentation Fault beendet, die Ausgabe ist in 4.5 dargestellt.

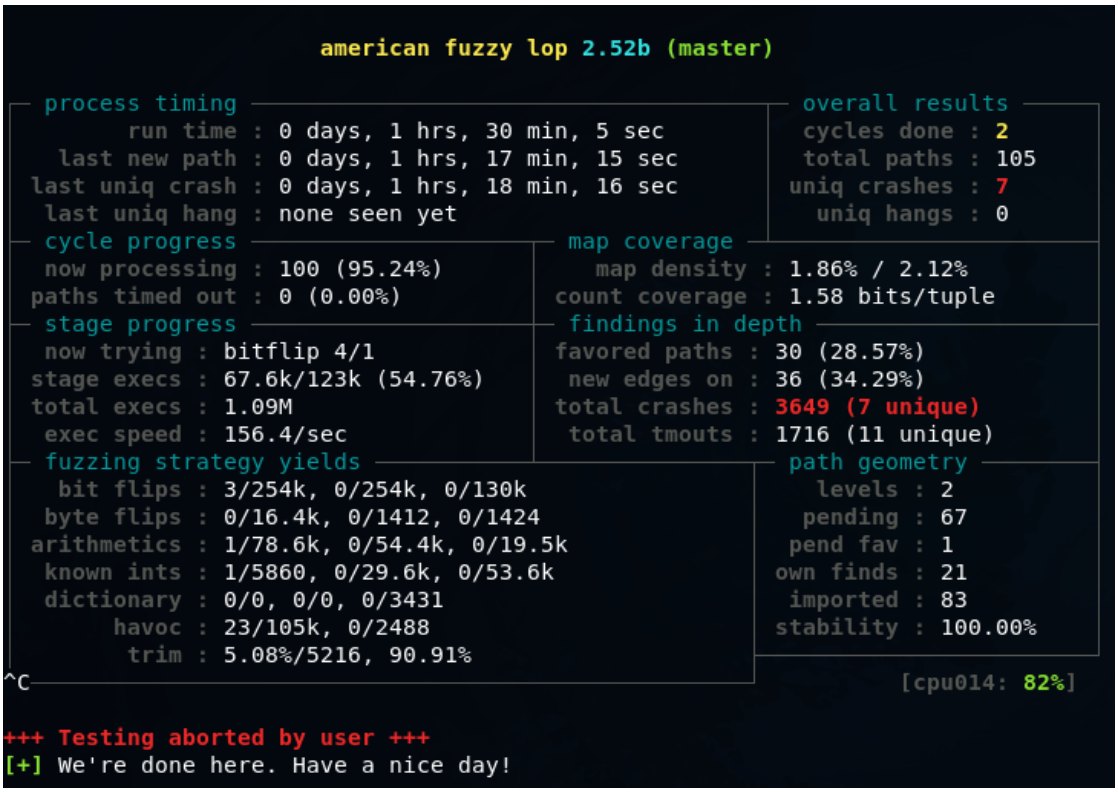

Abbildung 4.4: Fuzzing SoX

```
Program received signal SIGSEGV, Segmentation fault.
0x00005555556c0cbc in read_samples (ft=0x5555559d2610, buf=0x5555559d5630, len=8192) at xa.c:219
219             inByte = xa->buf[i];
/* ... */
```

Listing 4.5: SoX - Segmentation Fault

Es kann gesehen werden, dass die Eingabe dazu geführt hat, dass in der Funktion read_samples in der Datei “xa.c“ in Zeile 219 ein ungültiger Speicherzugriff erfolgt ist. Konkret kann im disassemblierten Code der Absturz an der Stelle “movzx r11d,BYTE PTR [rcx]“ beobachtet werden, wobei das RCX-Register zu dem Zeitpunkt Null ist. Es handelt sich also um eine Null-Pointer-Dereferenz. Bei XA-Dateien handelt es sich um das Maxis-Dateiformat. AFL hat hier also eine Maxis-Datei erzeugt, ohne vorheriges Wissen über irgendein von SoX unterstütztes Dateiformat. Der Fehler wurde im SourceForge Repository von SoX reported. Es wurden noch weitere Fehler von AFL in SoX gefunden, für die allerdings bereits öffentliche Fehlerberichte vorhanden waren.

5 Zusammenfassung

In dieser Arbeit wurde das Tool American Fuzzy Lop (AFL) vorgestellt. Zunächst wurde ein Überblick über typische Fehler und Typen von Fehlern in Software gegeben. Anschließend wurde Fuzzing als Technik der dynamischen Softwareanalyse eingeführt. Daraufhin wurde AFL umfassend beleuchtet und zwei Beispielprogramme mit AFL analysiert. Bei beiden Programmen wurden ohne großen Aufwand verschiedene Softwarefehler gefunden. Ein schwerwiegender Fehler in SoX wurde an die zuständigen Entwickler gemeldet.

Zusammenfassend ist zu sagen, dass AFL aktuell eines der erfolgreichsten Fuzzing-Tools ist. Dies verdankt AFL seiner hohen Erfolgsquote und den vielen Fehlern, die es in der kurzen Zeit seit der AFL verfügbar ist gefunden hat und seiner einfachen Bedienbarkeit.

5.1 Ausblick

Wie unter anderem in der Cyber Grand Challenge beobachtet, geht der Trend zu Smart-Fuzzing, welches Symbolic-Execution mit Coverage-Based-Fuzzing, wie AFL es durchführt, verbindet. Besonders interessant ist dabei meiner Meinung nach das Tool "Driller", das von Shellphish, dem zweitplatzierten Team, entwickelt wurde. Driller verbindet die Symbolic-Execution-Engine Angr mit AFL und wechselt immer wenn der Fuzzer feststeckt auf symbolische Ausführung und anschließend wieder zurück. Diese Technik ermöglicht es, mehr Fehler zu finden, die sich hinter Magic-Werten verbergen. In Zukunft werden Tools gewünscht, die solche hybride Ansätze verfolgen und mit der gleichen einfachen Bedienbarkeit und Robustheit wie AFL aufwarten können.

Eine andere interessante Entwicklung ist im Bereich der Tracer zu erwarten. Intel-Prozessoren haben mittlerweile Hardware-Tracing Features, die es ermöglichen mit minimalen Overhead Programme zu verfolgen.

Literatur

[1] Kaspersky. *Era of exploits: number of attacks using software vulnerabilities on the rise*. Kaspersky. 2017. URL: https://www.kaspersky.com/about/press-releases/2017_era-of-exploits-number-of-attacks--using-software-vulnerabilities-on-the-rise.

[2] Darpa. *Cyber Grand Challenge*. Darpa. 2016. URL: http://archive.darpa.mil/cybergrandchallenge/index.htm.

[3] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. http://lcamtuf.coredump.cx/. 2014. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[4] Adam DC949. *Fuzzing with AFL*. Youtube. 2017. URL: https://www.youtube.com/watch?v=vzfhHjnycnE.

[5] Craig Young. *Fuzz Smarter Not Harder An afl fuzz Primer*. Youtube. 2016. URL: https://www.youtube.com/watch?v=29RbO5bftwo.

[6] Nick Stephens u. a. „Driller: Augmenting Fuzzing Through Selective Symbolic Execution“. In: *Network and Distributed System Security Symposium 2016* (2016).

[7] ImmunitySec. *The Advantages of Block-Based Protocol Analysis for Security Testing*. ImmunitySec. 2002. URL: https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html.

[8] ivanfratric. *AFL for fuzzing Windows binaries*. ivanfratric. 2016. URL: https://github.com/ivanfratric/winafl.

[9] zardus. *preeny - Some helpful preload libraries for pwning stuff*. zardus. 2016. URL: https://github.com/zardus/preeny.

[10] jwilk. *American Fuzzy Lop fork server and instrumentation for pure-Python code*. jwilk. 2017. URL: https://github.com/jwilk/python-afl.

[11] Brian S. Pak. „Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution“. In: (2012).