Technische Universität Berlin

Institut für Softwaretechnik und Theoretische Informatik
Chair for Security in Telecommunications

Fakultät IV
Ernst-Reuter-Platz 7
10587 Berlin
https://www.tu.berlin/sect

Bachelor Thesis

Automating Linux Kernel Fuzzing Utilizing Static Code Analysis

Michael Rumler

Matriculation number:
341102

Supervisor:
Vincent Quentin Ulitzsch

Eingereicht am:
27.05.2024

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenständige Anfertigung versichert an Eides statt:

Berlin, den 27.05.2024

.....

Unterschrift

Abstract

English:

This work demonstrates how fuzzing the Linux kernel can be optimized using static code analysis. Through the developed tool-chain, the necessary preparation time is significantly reduced. Steps in code analysis and the generation of snapshots required for the fuzzer, which previously had to be done manually, have been automated. First the topic is briefly introduced, then related projects and works are referenced, followed by a detailed exploration of the areas of Static Code Analysis, Snapshot Generation, and Fuzzing on both theoretical and practical levels. Finally, the achieved results are critically evaluated, conclusions are drawn, and potential directions for future work are proposed.

Deutsch:

Diese Arbeit zeigt, wie das Fuzzen des Linux Kernels unter Zuhilfenahme statischer Code-Analyse optimiert werden kann. Durch die Anwendung dere entwickelten Tool-Chain wird die nötige Vorbereitungszeit signifikant reduziert. Bisher manuell vorzunehmende Schritte in der Code-Analyse und beim Generieren der für den Fuzzer notwendigen Snapshots konnten automatisiert werden. Dazu wird kurz in das Thema eingeführt, auf tangierte Projekte und Arbeiten verwiesen und dann in die Themengebiete Statische Code-Analyse, Snapshot-Generierung und Fuzzing auf theoretischer wie auf praktischer Ebene eingegangen. Abschließend werden die erzielten Ergebnisse kritisch evaluiert, ein Fazit gezogen und Weiterentwicklungsmöglichkeiten für zukünftige Arbeiten vorgeschlagen.

Contents

List of Figures

1 Introduction

1.1 The Emergence of Software Bugs and Fuzzers

With the introduction of the first software the first software bugs appeared. With more and more software projects with a growing code base being present and taking a more and more relevant role in our society the number of bugs increased and became also more and more relevant. To find these increasing number of bugs automated software testing has become an useful practice. One of these automated software test methods is fuzzing. Although originally introduced under this term in 1988 the idea of doing tests by firing random input on software is even older. [1] [2]

The Linux kernel, as the backbone of numerous operating systems, is one of these larger software projects. As a critical component it is necessary to do rigorous testing and analysis to ensure its reliability and security. Given its complexity and size, traditional manual testing methods are insufficient to uncover all potential vulnerabilities. This thesis explores an innovative approach to enhancing the robustness of the Linux kernel by automating fuzz testing through the use of static code analysis techniques.

Fuzzing is a powerful testing methodology that involves providing random or semi-random inputs to software programs to identify unexpected behaviors, crashes, and security vulnerabilities. Despite its effectiveness, applying fuzzing to a project as large and complex as the Linux kernel presents significant challenges, including the need for extensive computational resources and the difficulty of achieving comprehensive code coverage. Also kernel-level fuzzing is an challenge itself.

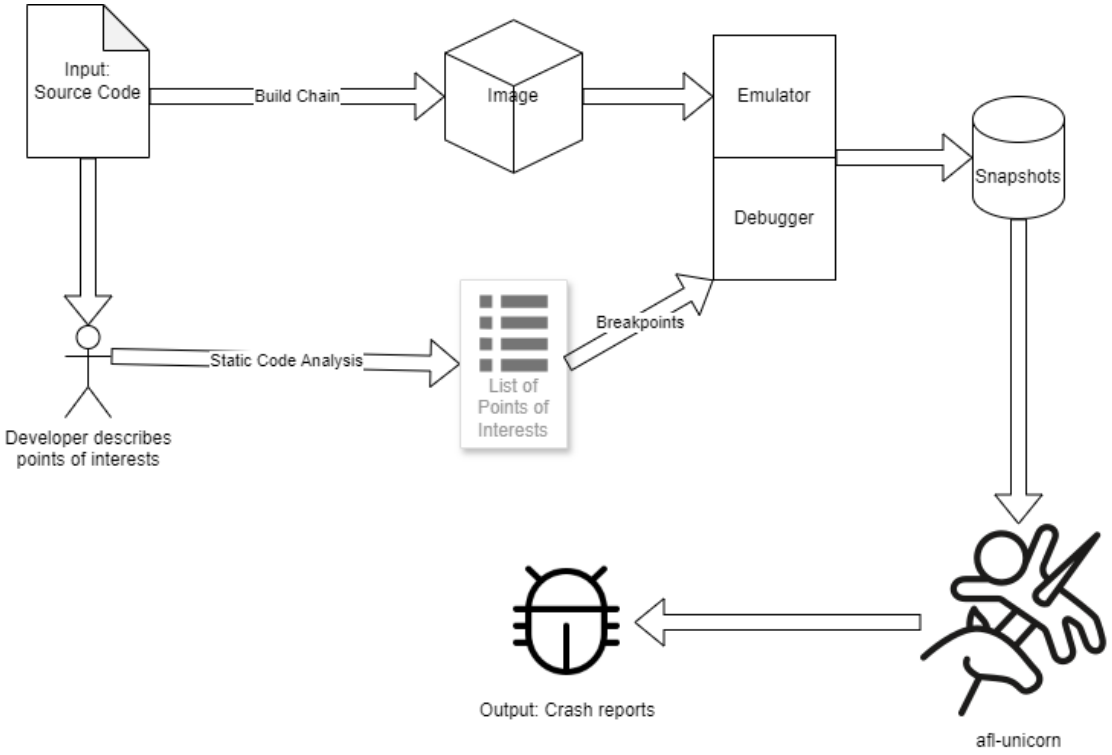
Static code analysis offers a solution to these challenges by enabling the examination of source code without executing it. This technique can identify potential vulnerabilities and problematic code paths early in the development cycle, guiding the fuzzing process to focus on high-risk areas. By integrating static code analysis with automated fuzzing tools, it is possible to enhance the efficiency and effectiveness of kernel testing, leading to more robust and secure software.

1.2 Architecture

This thesis aims to bridge the gap between static code analysis and fuzzing. It investigates the use of advanced static code analysis tools, such as Joern, to automate the identification of critical code segments to prepare for the fuzzing process. The result is a tool chain which automates many steps that would have to be done manually otherwise.

To solve the task we have divided it into smaller sub-tasks like illustrated in figure 1.1.

Figure 1.1: Architecture



At the beginning there is the provision of source code. In principle this can be any code that can be compiled and executed in an emulator. For our mission it will be the linux kernel. Having the sources available, the developer describes what his points of interests are. This is done by configuring the first part of the tool chain, the scripts for static code analysis. For the practical part of this work we define the `struct sk_buff` as our point of interest, but basically the developer is free to choose anything that can be targeted by static code analyzers. The static code analysis will create source code fitting list of occurrences of the defined points of interest. This is worked out in chapter 3.

For input creation for the fuzzer we build the sources and run it in an emulator of our choice. For us this is done utilizing Buildroot and emulation by QEMU. We attach a script-able debugger to steer the emulator, this will be the GDB. The occurrences of points of interest from chapter 3 are taken as breakpoints to trigger the necessary snapshot creation. This is elaborated in chapter 4.

Finally preparing the input for the fuzzer and the fuzzer itself has do be done. This work will demonstrate how to make use of UnicornAFL this for; see chapter 5.

2 Related Work

While fuzzing user space programs has comparatively low entry barriers and became more common, fuzzing kernel space programs has higher ones and is less widespread. Nevertheless over the last decades various successful kernel fuzzers and related works appeared.

2.1 AFL

American Fuzzy Lop (AFL) is a open source fuzz testing tool designed to find security vulnerabilities and bugs in software applications. AFL came up with a novel approach to automate the creation of inputs for software programs to trigger unexpected behaviors, crashes, and other issues that might indicate security flaws. It is using coverage-guided fuzzing. During execution it collects information about which parts of the code are executed and based on that AFL generates new input to explore different execution pathes for the upcoming iterations. Starting with an initial set of valid inputs it mutates them. The newly generated input queue entries are based on coverage feedback, deterministic mutations (like bit flips, adding small integers, replacement with interesting integers like 0, 1, `INT_MAX`) and random non-deterministic mutations. [3] This approach has lead to a remarkable amount of bug-finds[1]. [3]

AFL's QEMU mode allows fuzzing of binary applications without source code by using QEMUs binary translation. This enables AFL to collect coverage information and guide the fuzzing process. Unlike kAFL and TriforceAFL, which leverage hardware-assisted virtualization and full-system emulation for more complex and kernel-level fuzzing, AFL's QEMU mode focuses on user-space applications, making it simpler but potentially less powerful for kernel fuzzing scenarios. [4]

To get AFL working, the program to be analyzed is compiled with the AFL compiler, which adds special instructions used to monitor the execution flow. The so instrumented binary is then run under AFL's control. Meanwhile the AFL user interface displays the progress and intermediate results. In case of a crash, the input used for that crash is logged.

2.2 libFuzzer

LibFuzzer is a library for coverage-guided fuzz testing, integrated with LLVM. It is designed to be used as a part of the LLVM project and targets individual functions or libraries. Although libFuzzer is widely used, it does not support kernel space fuzzing. [5]

[1] `https://github.com/mrash/afl-cve`

2.3 Honggfuzz

Honggfuzz is a modern, general-purpose fuzzer that emphasizes performance, ease of use, and a rich set of features. It can be used for both user-space applications and kernel-space components. Similar to AFL it is performing coverage-guided exploration of the target. To retrieve the necessary information, the libFuzzers instrumentation to keep track of coverage and GCC plugins that place trace information between the instructions are used. It comes with native kernel fuzzing support. Honggfuzz has lead to the discovery of many interesting security problems. [6]

2.4 TriforceAFL

TriforceAFL is an extension of AFL, designed to enhance its capabilities to become a full-system fuzzer. It extends the traditional AFL's fuzzing approach to include kernel fuzzing, which allows it to test entire operating systems and their kernels, rather than just user-space applications. Therefore the target operating system and kernel are set up within a virtualized or emulated environment like QEMU. TriforceAFL is available in a Dockerfile containing everything necessary for a quick start. [7]

2.5 Trinity

Trinity is a Linux system call fuzzer designed to test the Linux kernel by generating and executing random system calls with various inputs. Trinity aims to uncover bugs and vulnerabilities within the kernel by systematically exploring the vast number of possible system call sequences and input parameters. [8]

2.6 DIFUZE

DIFUZE is a specialized fuzzing tool designed to uncover security vulnerabilities in kernel device drivers. It automatically analyzes the provided kernel sources to extract driver interface information. Via ioctl commands DIFUZE tries to trigger crashes in the device drivers. [9]

2.7 DifFuzz

DifFuzz utilized a technique known as differential fuzzing to compare the behavior of different implementations of the same interface, such as different versions of a device driver, or device drivers from different vendors implementing the same hardware interface. In that process DIFUZE searches for input that leads to an unnormal resource usage, such as time consumption, response size and used memory. [10]

2.8 syzkaller

Syzkaller is an open-source, coverage-guided fuzzer specifically designed for fuzzing the Linux kernel. Syzkaller automates the process of generating and executing system calls with random or semi-random inputs to discover vulnerabilities and bugs in the kernel. It uses a sophisticated feedback mechanism to maximize code coverage and uncover subtle and hard-to-find issues which makes it superior to Trinity for large software projects in terms of coverage. Collecting code coverage information is based on two mechanisms: First a kernel patch for linux (and some other operating systems) which allows, if the kernel was compiled with the `CONFIG_KCOV` flag, to trace kernel space code execution. And second utilizing sanitizer coverage for the compiler like gcc, which adds trace information during compile time. [11]

2.9 Filesystem Fuzzer for AFL

To fuzz filesystem implementations, Nossum and Casasnovas have added functionality to AFL for kernel space fuzzing. They also utilize the GCC's sanitizer coverage information and use shared memory via `mmap` and `vmalloc` to provide the filesystem memory from kernel to AFL in user space. [12]

2.10 kAFL

The kAFL fuzzer is a kernel space fuzzer that is using a modified QEMU/KVM version. For execution control and coverage feedback it utilizes the Intel VT (Virtualization Technology)[2], Intel PML (Page Modification Logging)[3] and Intel PT(Processor Trace)[4] technologies. [13]

2.11 Unicorn

The Unicorn engine is a QEMU fork. Everything but CPU emulation is stripped from QEMU. Unicorn wraps around the remaining part and is adding an API offering instrumentation at various levels and various other improvements. [14]

2.12 AFL-Unicorn

AFL-Unicorn is an AFL extension to fuzz software utilizing the Unicorn Engine for emulation. It works by making AFL's QEMU mode placing it's block-edge instrumentation for code coverage feedback into the Unicorn Engine instead. [15]

[2] `https://cdrdv2-public.intel.com/671081/vt-directed-io-spec.pdf`

[3] `https://cdrdv2-public.intel.com/671378/335279-performance-monitoring-events-guide.pdf`

[4] `https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/004/intel-processor-trace/`

2.13 AFL++

AFL++ is a fork of AFL with modifications to make it faster, improve the mutation generator, make instrumentation easier and other features. This project offers a docker container that comes with everything necessary including UnicornAFL. [16]

2.14 UnicornAFL

Similar to AFL-Unicorn for AFL and Unicorn, this project builds a bridge between AFL++ and Unicorn. It requires a Unicorn Engine installation and is an AFL++ extension. [17]

2.15 Unicorefuzz

Unicorefuzz is a project demonstrating the abilities of AFL-Unicorn by fuzzing arbitrary parsers in the linux kernel space. [18] It was recently upgraded to be based on UnicornAFL and therefore AFL++. [19]

3 Static Code Analysis

3.1 Introduction

Static code analysis describes a method for investigating and debugging a software by inspecting its source code. In contrast to dynamic code analysis there is no program or script execution involved. Like almost all parts of debugging the static code analysis can be done with or without tool usage and be done manually or partly or completely automated. A large variety of free, open source and commercial tools is available to support developers. [20]

3.1.1 Manual Code Analysis

Manual code analysis, or manual code review, is performed by developers or dedicated reviewers who meticulously inspect the code line by line. Compared to Automated Code Analysis this has a couple of advantages and disadvantages. Key advantages are the superiority of human judgment and expertise, the opportunity to move the focus dynamically to code parts suspected to have issues and to find problems that automated tools might overlook or simply cannot identify. The biggest disadvantage is that the analysis can be very time-consuming. Also humans tend to fail to notice issues, and the larger the software code base becomes the more issues might be overlooked. [21]

3.1.2 Automated Code Analysis

Automated code analysis uses specialized tools to systematically examine the codebase for errors, vulnerabilities, and coding standard violations. These tools apply predefined rules and algorithms to identify issues quickly and consistently. Main advantage of automated code analysis is, once the tools are set up, a decrease in human time effort. This scales excellent with growing code size. Also in most cases the coverage becomes better compared to human investigation. In terms of quantity the automated code analysis wins over manual code analysis. Nonetheless, today's tools lack human intelligence and therefore they lose in terms of quality. [21]

3.2 Features

The primary goals of static code analysis are to detect misbehavior, security vulnerabilities which might lead to abuse or crashes, style issues, verification for following best practices and finding redundant or dead code. [22].

While each of the aforementioned features is useful in the field of software development, most of them are irrelevant for the process of fuzzing. For our purposes, it is sufficient to make use of the lexical analysis capability that each static code analysis tool provides to be able to work on software code.

3.3 Determining a Target

For this project we want to fuzz the Linux kernel. Our focus shall be kernel functions parameterized with `struct sk_buff`. This structure is omnipresent in the Linux network stack and used to manage all kind of network packets. Therefore it is an interesting target for security analysis. Nevertheless `struct sk_buff` is only chosen for demonstration purposes and can be replaced by any other element the developer wants to fuzz.

3.4 Joern

3.4.1 Introduction

Joern is static code analysis tool specifically designed to facilitate the discovery of security vulnerabilities in source code. Core features are a robust parser, the generation of code property graphs, taint analysis and it offers a query language for searching the code. It currently supports 13 languages, including C and C++. [23] Joern offers prebuilt binaries at github[1] and a bash script for easy installation. It is also available in a docker container. [23]

3.4.2 Command-line Interpreter Mode and Script Interpreter Mode

Joern offers flexibility in its operation by allowing users to run it in two distinct modes. Either as a command-line interpreter or as a script interpreter.

The command-line interpreter mode enables users to interact with Joern directly from the terminal, providing an interactive environment for executing ad-hoc queries and analyzing code on-the-fly. This mode is particularly useful for quick inspections and exploratory analysis. The script interpreter mode allows users to write and execute scripts, making it ideal for automating complex analysis tasks and repetitive processes. [24]

For our purposes, the script interpreter mode is very suitable.

3.4.3 Performing the Static Code Analysis

When importing source code as a new project, Joern generates a code property graph. Depending on the software's size and the hosts computing power this varies between some seconds and many days. The graphs are stored in the Joern workspace to be reusable. For examination of the input, the code property graph is available as a global

[1] `https://github.com/ShiftLeftSecurity/joern/releases/`

data structure. Starting from the root element the user is able to mine through the code-base. [25]

To find all occurrences of `struct sk_buff` we created the following query:

```
cpg.method.parameter.typeFullNameExact("sk_buff*").method
```

3.5 Stripping the Input with Grep

While exploring the possibilities of the Joern query language, it quickly became apparent that analyzing large software projects like the Linux kernel is resource-intensive and can overwhelm workstations. In our case, importing the complete source code was running for hours and failed to complete the creation of the code property graph due to insufficient heap memory.

We managed to increase the performance enormous by reducing the Linux source code to files that actually contain the desired target, in our case `struct sk_buff`.[2] Since we do not need to have a code property graph containing the whole kernel or dependencies between multiple files this does not restrict us. We need to utilize a lexical analyzer to break down the code to functions and its parameters. Occurrences in comments or as local variables for example are not of interest. Joern can be run on single source code files ignoring all other references an dependencies. Thus we run Joern on every file containing our needle once and achieve an acceptable runtime duration.

The figure 3.1 shows an excerpt of the results of grep's preparatory work.

Figure 3.1: Grep Output

```
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:545
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:548
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:549
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:607
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:622
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:626
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:655
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:672
/home/thunder/linux-source-6.8.0/linux-source-6.8.0/net/ipv4/udp_offload.c:710
```

3.5.1 Automating the Static Code Analysis

For the automated static code analysis we come back to Joerns script interpreter mode. Our tool-chain[3] imports the results grep has preprocessed. Afterwards the Joern component `c2cpg.sh`, a bash script that generates code property graphs for the interim results, is called. Finally we execute Joern and hand it a parameterized Scala script[4] which utilized the query introduced in subsection 3.4.3. The Scala script will analyze

[2] `https://github.com/mrumler/BApublic/1_grep.sh`
[3] `https://github.com/mrumler/BApublic/2_joern.sh`
[4] `https://github.com/mrumler/BApublic/search.sc`

the all functions found by the query for its parameters and dump all fuzzing related information into the output file.
The figure 3.2 shows an excerpt of the results of Joern's work. We introduce a well-readable json format which is also easy to parse for later purposes. Each entry contains the function to be fuzzed, its location, the line number of the function definition and the names and types of the parameters. Additionally the points of interests — a subset or the function's parameters — are stored.

Figure 3.2: Joern Output


```
{
        "function": "udp4_gro_receive",
        "bp_target": "net/ipv4/udp_offload.c:622",
        "args": [
                ["head", "struct list_head*"],
                ["skb", "struct sk_buff *"]
        ],
        "poi_args": [
                "skb"
        ]
}
```

4 Snapshot Acquisition

To aquire input for our fuzzing runs we run a linux kernel in an emulator of our choice, attach a debugger break at all points of interest (in our case at all function definitions that contain `sk_buff` parameters) and dump all required information. This includes the guests' main memory, all the guest's register values and the base address of the `sk_buff` parameters.

4.1 Creating a Debuggable Kernel

To obtain a debuggable kernel image, we make use of Buildroot. Buildroot is an open-source and easy to use tool which allows building (embedded) Linux systems. It automates the process of generating a customizable and optimized root filesystem, kernel and bootloader.

4.1.1 Configuring Buildroot

Before building the Linux kernel we preconfigure Buildroot using `make qemu_x86_64_defconfig`. To add additional debugging capabilities, to define the output formats and to choose a kernel version we do further configuration via `make menuconfig`. Subsequently we utilize `make linux-menuconfig` to activate all helpful kernel hacking settings and `CONFIG_PVH=y` to be able to boot in QEMU.

4.1.2 Buildroot Output

Finally Buildroot stores the output in the `output` subdirectory. In `output/build/linux-`*`versionnumber`* the downloaded kernel source is placed. The following files are primarily of interest to us:

- `output/images/bzImage` — the compressed kernel image
- `output/images/rootfs.ext4` — the rootfs file
- `output/build/linux-`*`versionnumber`*`/vmlinux` — the raw kernel image, useful for the debugger

4.2 Setting up Emulator and Debugger

4.2.1 Emulating

For this thesis we choose QEMU for emulation. When running QEMU, for scaling purposes we parameterize it to use a minimum amount of main memory. Furthermore QEMU is told to start a GDB Server — a program that facilitates remote debugging by acting as an intermediary between the GNU Debugger and the target, allowing to control the execution of emulated program — and to suspend the boot process.

4.2.2 Attaching a Debugger

The GDB (GNU Debugger) is a powerful debugging tool supporting various programming languages, including C. It is capable of setting breakpoints, inspecting memory and registers. GDB is handed the uncompressed kernel image to be supplied with debug information.

4.2.3 Orchestrating GDB

With GDB script we can automate the process of setting breakpoints and inspecting memory and registers. Our GDB script has do break on every point of interest we have localized before. Breaking will be done only once per point of interest, but could also be done for example at the 10th visit or at every nth visit. For each break GDB script has do solve the following tasks:

- Emit the address of the function parameter that should be the targeted by the fuzzers input queue.
- Emit the guest machines register values.
- Create a snapshot of the guest machines main memory.

One way to automate GDB with a GDB script is placing a `.gdbinit` file in the current directory. In our first approach we used the Joern output as input and created a script with fitting instructions for every single breakpoint. This, however, contained more than 1 million lines of script code. Therefore we dropped that approach. GDB has a very comprehensive python API [26] which we utilized[1] to place breakpoints and breakpoint handlers in a well scaling way.

4.3 Acquired Snapshots

For each reached breakpoint the breakpoint handler generates a snapshot. Our snapshots consist of the following parts:

[1] `https://github.com/mrumler/BApublic/break_poi.py`

- Register values — These are stored in separate files, having the register's name as their filename, `.rv` as their file extension and the register content as the file's content.
- `mem.dump` — This file contains the guests physical memory dump.
- `mem.txt` — This file contains the output of the QEMU command `info_mem` which provides us which the virtual memory mapping information.
- *`address`*`.vdump` files — These files contain the guests virtual memory dumps.
- `mtree.txt` — This file contains the output of the QEMU command `info_mtree` which visualizes the memory tree. Primary helpful for debugging purposes.
- `metadata.json` — This file contains all metadata for our point of interest. First the `data` member's value of our `struct sk_buff` (this is the address the fuzzer has to tackle with entropy) next to the function parameter's name (necessary to store that information if a function has more than one `struct sk_buff` parameter). Second the POI description to keep that information hand in hand with the other parts of our snapshot.

The figure 4.1 shows an example `metadata.json` file.

Figure 4.1: metadata.json

```
{
    "data": {
        "skb": 18446612682234034000
    },
    "poi": {
        "function": "udp4_gro_receive",
        "bp_target": "net/ipv4/udp_offload.c:622",
        "args": [
            [
                "head",
                "struct list_head*"
            ],
            [
                "skb",
                "struct sk_buff *"
            ]
        ],
        "poi_args": [
            "skb"
        ]
    }
}
```

5 Fuzzing

5.1 Introduction

Fuzzing is an automated software testing technique used to discover security vulnerabilities and bugs by inputting large amounts of random, malformed, or semi-random data into a program. The goal is to make the program crash, behave unexpectedly, or expose unintended behaviors, which can then be analyzed to identify and fix underlying issues. Fuzzing helps uncover various types of vulnerabilities such as buffer overflows, memory leaks, format string vulnerabilities, use-after-frees and other security issues. It is particularly effective because it tests the program in ways that human testers might not consider, and it can be automated to run continuously or at scale. Furthermore fuzz testing can utilized to run on different implementations for the same API, mass testing behavioral equivalence for them.
To get an overview of the overwhelming number of fuzzers (and their forks) – some of them were introduced in chapter 2 – it is worthwhile to categorize them.

5.2 Categorization

5.2.1 Based on Execution Method

Fuzzing can be categorized based on the execution method into black-box, white-box, and grey-box fuzzing.
Black-box fuzzing operates without any knowledge of the internal workings of the application. It treats the software as a closed box, focusing solely on the inputs and outputs. This approach is simple to implement and widely applicable, making it useful for testing a broad range of software without needing access to the source code. However, its lack of insight into the application's internal state often leads to lower efficiency in finding deep-seated vulnerabilities.
White-box fuzzing, on the other hand, leverages complete knowledge of the application's internal logic and source code. This method allows for more thorough analysis and testing by utilizing techniques like symbolic execution and static analysis to generate inputs that cover more code paths and edge cases. While white-box fuzzing is highly effective at uncovering subtle bugs, it requires significant computational resources and in-depth knowledge of the application, making it enormous complex to set up and run.
Grey-box fuzzing strikes a balance between these two approaches by having partial knowledge of the application's internals, often using instrumentation to gather runtime information and guide the fuzzing process. This method provides a good trade-off

between efficiency and simplicity, making it a popular choice for many fuzzing applications. [27]

5.2.2 Based on Import Format Knowledge and Feedback Mechanism

Fuzzing can also be categorized based on the feedback mechanism used to generate new inputs for the fuzzers input queue.
Dumb fuzzing operates without any knowledge of the input structure or the application's logic and without any feedback from the application under test. It is generating purely random inputs in an attempt to trigger any unexpected behaviors or crashes. Its simplicity and ease of implementation can quickly generate a large volume of inputs. Unfortunately dumb fuzzing often results in lower efficiency, as many of the generated inputs are invalid and quickly discarded by the target application. As a result, dumb fuzzing is likely to miss deeper bugs that require specific input patterns to trigger.
Smart fuzzing, also known as feedback-guided or coverage-guided fuzzing, leverages feedback from the application's execution to improve the generation of new inputs. Tools like AFL utilize that approach to identify which parts of the code have been exercised by previous inputs and focus on generating inputs that explore new or less-covered code paths. This feedback loop significantly enhances the efficiency of the fuzzing process, allowing the discovery of more complex vulnerabilities. By prioritizing inputs that increase code coverage or trigger unique execution paths, smart fuzzing can uncover bugs that dumb fuzzing might miss, making it a more effective approach for thorough security testing. Additionally, smart fuzzing leverages an understanding of the input format, protocols, or the internal state of the application to produce well-formed and semi-random inputs. This approach increases the likelihood of exploring deeper code paths and uncovering subtle bugs, making it significantly more effective for complex and structured input scenarios. However, smart fuzzing requires a more sophisticated setup and domain-specific knowledge, which can limit its immediate applicability and increase the initial overhead. Balancing the trade-offs between these two approaches is crucial for optimizing the fuzzing process and achieving comprehensive software testing. By combining these techniques, smart fuzzing can systematically and intelligently explore the input space, enhancing the overall robustness and security of software systems. The use of feedback mechanisms not only improves the efficiency of input generation but also ensures that the fuzzing process is adaptive and focused, targeting areas of the code that are most likely to yield valuable insights into potential vulnerabilities. [27].

5.3 A Brief Overview of struct sk_buff

The `struct sk_buff`[1] is a crucial data structure of the Linux kernel's networking stack. It is designed to encapsulate network packets, providing a versatile and efficient mechanism for managing them through the stack layers. Various fields and pointers that

[1] `https://github.com/torvalds/linux/blob/master/include/linux/skbuff.h`

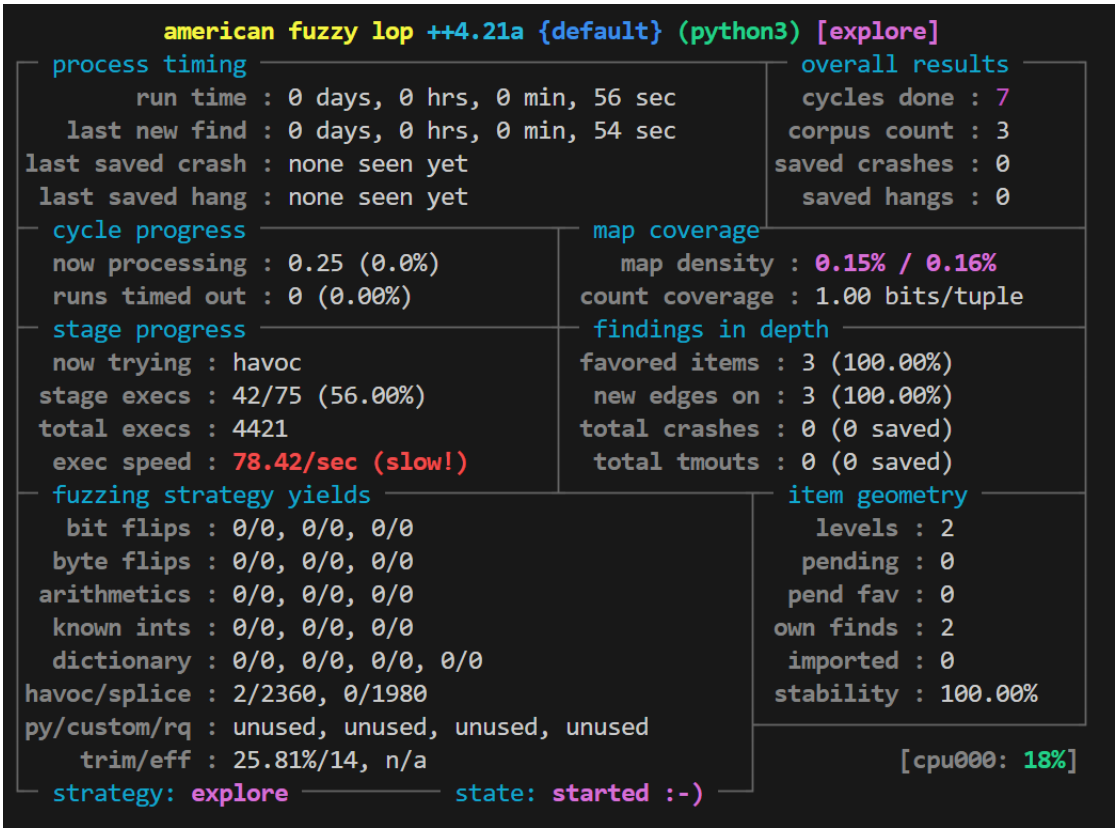
describe the packet data and its metadata. Many sections are `union`s, which makes the structure very versatile.
The structure starts with `struct sk_buff next` and prev pointers allowing it to be linked together in a list, facilitating the management of packet queues. Fields like `mark`, `priority`, and `protocol` store information about the packet's processing priority, protocol type, and other relevant metadata. The header fields include pointers to the packet's head, data, tail, and end. They define the packet's boundaries within the allocated buffer.
The address in the data pointer is a good target for our fuzzers. It is important to consider the buffer size to avoid creating a buffer overflow. Manipulating the linked list or buffer addresses itself will likely result in an instant crash and is not of any value for us. The alteration of some of the metadata could lead to the discovery of vulnerabilities in the kernel, but here a generation based fuzzer might be more successful than a mutation based one. [28]

5.4 Running the fuzzer

Loading snapshots into UnicornAFL with a sufficient degree of accuracy is achieved with a very simple harness[2]: It is a python script which loads all supported memory registers, maps and initializes all exported memory regions, sets the exit address to the address of the `rsp` register (in the case of `x86_64` targets this contains the return address of the fuzzed function), registers the place_input_callback and starts fuzzing operations. No further action is required, UnicornAFL finds new paths as expected (if possible).

Figure 5.1: Fuzzing run on `udp4_gro_receive`.

[2] `https://github.com/mrumler/BApublic/harness.py`

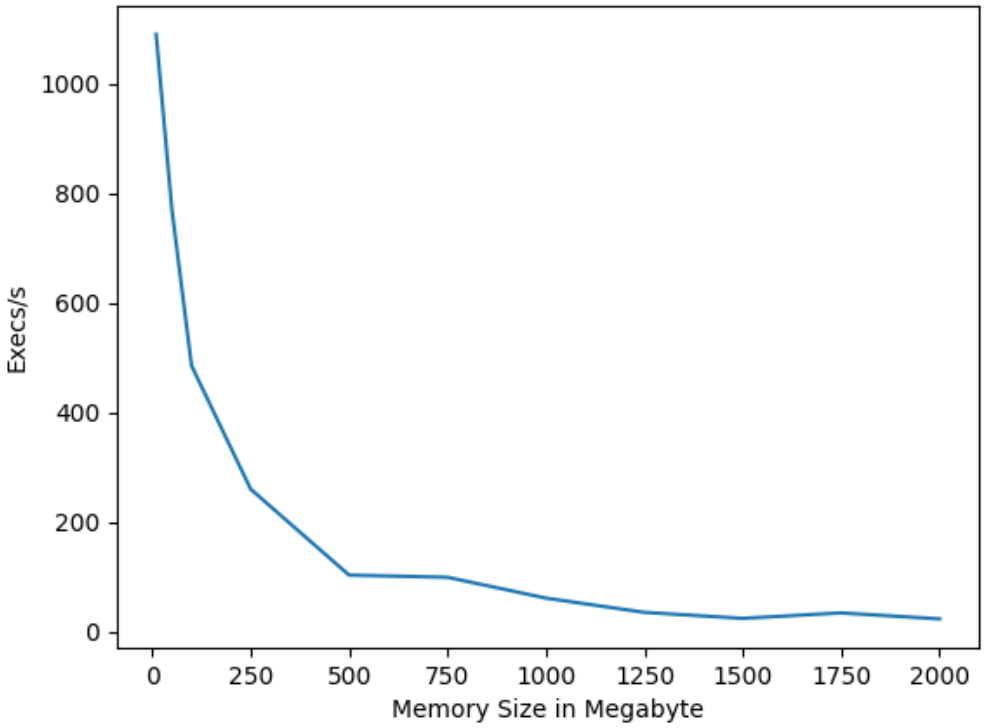

american fuzzy lop ++4.21a {default} (python3) [explore]
process timing
run time : 0 days, 0 hrs, 0 min, 56 sec
last new find : 0 days, 0 hrs, 0 min, 54 sec
last saved crash : none seen yet
last saved hang : none seen yet
overall results
cycles done : 7
corpus count : 3
saved crashes : 0
saved hangs : 0
cycle progress
now processing : 0.25 (0.0%)
runs timed out : 0 (0.00%)
map coverage
map density : 0.15% / 0.16%
count coverage : 1.00 bits/tuple
stage progress
now trying : havoc
stage execs : 42/75 (56.00%)
total execs : 4421
exec speed : 78.42/sec (slow!)
findings in depth
favored items : 3 (100.00%)
new edges on : 3 (100.00%)
total crashes : 0 (0 saved)
total tmouts : 0 (0 saved)
fuzzing strategy yields
bit flips : 0/0, 0/0, 0/0
byte flips : 0/0, 0/0, 0/0
arithmetics : 0/0, 0/0, 0/0
known ints : 0/0, 0/0, 0/0
dictionary : 0/0, 0/0, 0/0, 0/0
havoc/splice : 2/2360, 0/1980
py/custom/rq : unused, unused, unused, unused
trim/eff : 25.81%/14, n/a
item geometry
levels : 2
pending : 0
pend fav : 0
own finds : 2
imported : 0
stability : 100.00%
[cpu000: 18%]
strategy: explore
state: started :-)

6 Evaluation

6.1 Execution Speed

During fuzzing, we quickly noticed that our executions per second were extremely low. Upon investigating this issue, we found that the execution speed is dependent on the amount of memory and decreases exponentially as the mapped main memory increases. A quickly designed script[1] to do some speed tests for different amounts of memory supports this statement. Only a single `NOP` (0x90) instruction per run is done. This measures the overhead performance impact. Results are shown in the figure 6.1.

Figure 6.1: Relation between execution speed and memory size.



[1] `https://github.com/mrumler/BApublic/unicornspeedtest.py`

6.2 Expected False Positives

False positives can occur due to mutated input being placed deep in the call stack. If some functions higher in the call stack sanitize and verify the input but our mutation based fuzzer does not consider this, it is likely that false positives arise.

6.3 No Positives (yet)

No crashes have been found so far. However, considering the resources utilized in other fuzzing projects, this is not surprising. A sensible approach would be to launch a large-scale test run on a cloud provider.

6.4 XMM/CR Registers not Supported

The XMM (SIMD) and CR (dynamic address translation registers) registers are currently unsupported due to issues with the Unicorn engine. Enabling these registers in the harness leads to Unicorn errors. This limitation may be partly because Unicorn is based on a significantly reduced version of QEMU.

7 Conclusion

Our work has demonstrated that the process of fuzzing can be significantly simplified using our introduced toolchain. By incorporating static code analysis and automating the process of snapshotting, a considerable amount of time can be saved, making the fuzzing process more efficient and streamlined. However, there are still numerous opportunities for improvement in various areas.

7.1 Future Work

7.1.1 Inlining, Preprocessors and Optimization Levels

Inline instructions and preprocessor directives have not been considered in this work so far. It may be beneficial to run at least the preprocessor before performing static analysis.
Moreover, different optimization levels can affect the code behavior. Optimizing code during compile time (-O2, -O3) might behave differently than unoptimized code (-O0). While during development often unoptimized code is used, kernels in practice are often optimized, so it's crucial to compile the kernel with the same optimization settings used in the deployment environment to ensure realistic fuzzing results.

7.1.2 Memory Size Optimization

As noted in the evaluation section 6.1, execution speed is significantly impacted by the amount of memory allocated. First of all, the built kernel image should be stripped further to remove everything not necessary for our task. An additional approach is the usage of partial memory mapping, as demonstrated in Unicorefuzz. Mapping only the necessary memory required for the fuzzing run can enhance performance by reducing memory usage, leading to higher execution speeds. [18]

7.1.3 Parallelized Fuzzing

Fuzzing can be scaled efficiently using cloud infrastructure. By creating numerous workers for emulation and snapshot creation and other workers which will continuously retrieve and process new snapshots from a central repository (e.g., S3 buckets) this can be achieved. Currently, only the first breakpoint in each function is fuzzed, but this approach could be expanded to randomize or increase the number of snapshots per point of interest, enhancing coverage and robustness.

7.1.4 Preselection

Handling a large number of snapshots necessitates prioritization. By analyzing code coverage and using concolic fuzzing with a constraint solver, we can prioritize snapshots that are more likely to reveal significant vulnerabilities. Functions that do not interact with `sk_buff->data` can be pre-evaluated using symbolic execution to filter out irrelevant snapshots, streamlining the fuzzing process.

7.1.5 Reducing False Positives

As elaborated in section 6.2, false positives can occur due to mutated input being placed deep in the call stack, while a functions higher in the call stack would sanitize and verify the input. Improving the static code analyzes could detect those scenarios and place the fuzzer input earlier.

7.1.6 Concolic execution

Concolic execution can be employed to enhance our fuzzer by collecting constraints that an execution path encounters. Using these constraints, we can utilize a constraint solver to determine inputs that will lead to new execution paths within our functions.

7.1.7 Spin Lock Detector

Detecting spin locks is crucial, as Unicorn simulates only one core. If another core holds a lock during snapshotting, the lock may never be released, causing the fuzzer to wait indefinitely and therefore halting the progress. Implementing detection mechanisms for raw spin unlock events can help manage and mitigate this issue, ensuring smoother fuzzing operations, especially when not fuzzing manually but doing it large-scaled and unsupervised in a cluster.

7.1.8 Toolchain Optimization

Optimizing the harness by rewriting it in a native language such as C or Rust, instead of Python, can enhance speed due to the performance benefits of compiled languages. Additionally, memory dumps can be compressed, or the built virtual machine can be minimized to reduce memory usage, improving overall efficiency.

7.1.9 Shipment

Packaging the entire tool-chain into a Docker container might be helpful for everyone who is interested in our work. Dockerizing it would simplify setup and usage, facilitating much easier distribution and testing and thus reducing the entry barriers.

7.1.10 Userspace Traffic

Automating the generation of noise to make the kernel run into so far unreached breakpoints, which is currently done manually, would be very useful. The traffic generator should perform tasks that leads to activity in the Linux kernel's network stack.

Bibliography

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] J. W. Duran and S. Ntafos, "A report on random testing," in *Proceedings of the 5th international conference on Software engineering*, IEEE Press, 1981, pp. 179–183.

[3] AFL-Team, *Afl (american fuzzy lop)*, `https://afl-1.readthedocs.io/en/latest/`, Accessed: 2024-05-12.

[4] B. Machiraju, "Internals of afl fuzzer - qemu instrumentation," Accessed: 2024-05-02. [Online]. Available: `https://tunnelshade.in/blog/afl-internals-qemu-instrumentation`.

[5] libFuzzer Team, *Libfuzzer – a library for coverage-guided fuzz testing.* `https://llvm.org/docs/LibFuzzer.html/`, Accessed: 2024-05-12.

[6] Honggfuzz-Team, *Honggfuzz developer page*, `https://honggfuzz.dev/`, Accessed: 2024-04-22.

[7] Triforce-Team, *Readme*, `https://github.com/nccgroup/TriforceAFL`, Accessed: 2024-04-22.

[8] Trinity-Team, *Readme*, `https://github.com/kernelslacker/trinity`, Accessed: 2024-04-22.

[9] Corina, Machiry, Salls, *et al.*, "Difuze: Interface aware fuzzing for kernel drivers," Accessed: 2024-05-22, Dallas, Texas, USA, 2017, pp. 2123–2138. [Online]. Available: `https://dl.acm.org/doi/pdf/10.1145/3133956.3134069`.

[10] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: Differential fuzzing for side-channel analysis," *arXiv preprint arXiv:1811.07005v2*, 2019, Accessed: 2024-05-22. [Online]. Available: `https://arxiv.org/pdf/1811.07005`.

[11] Syzkaller-Team, *Syzkaller coverage documentation*, `https://github.com/google/syzkaller/blob/master/docs/coverage.md`, Accessed: 2024-04-27.

[12] V. Nossum and Q. Casasnovas, *Filesystem fuzzing with american fuzzy lop*, `https://events.static.linuxfound.org/sites/events/files/slides/AFL%20filesystem%20fuzzing,%20Vault%202016_0.pdf`, Accessed: 2024-04-28, 2016.

[13] kAFL Team, *Kafl readme*, `https://github.com/IntelLabs/kAFL`, Accessed: 2024-05-21.

[14] A. Q. Nguyen and H. V. Dang, *Unicorn: Next generation cpu emulator framework*, `https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf`, Accessed: 2024-01-06, 2015.

[15] N. Voss, "Afl-unicorn: Fuzzing arbitrary binary code," 2017, Accessed: 2024-01-07. [Online]. Available: `https://medium.com/hackernoon/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf/`.

[16] AFL++-Team, "American fuzzy lop plus plus (afl++) readme," Accessed: 2024-01-09. [Online]. Available: `https://github.com/AFLplusplus/AFLplusplus`.

[17] ——, "Unicornafl readme," Accessed: 2024-01-09. [Online]. Available: `https://github.com/AFLplusplus/unicornafl`.

[18] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: `https://www.usenix.org/conference/woot19/presentation/maier`.

[19] D. Maier, F. Freyer, R. Linssen, and A. D. Vito, "Unicorefuzz readme," Accessed: 2024-05-11. [Online]. Available: `https://github.com/fgsect/unicorefuzz`.

[20] K. Kuszczyński and M. Walkowski, "Comparative analysis of open-source tools for conducting static code analysis," *Sensors*, vol. 23, no. 18, pp. 6–8, 2023, Accessed: 2024-05-22. DOI: `10.3390/s23187978`. [Online]. Available: `https://www.mdpi.com/1424-8220/23/18/7978`.

[21] K. Malik, "Code review: Manual vs automated," 2022, Accessed: 2024-04-17. [Online]. Available: `https://www.codiga.io/blog/code-review-manual-vs-automated/`.

[22] G. Fan, X. Xie, X. Zheng, Y. Liang, and P. Di, "Static code analysis in the ai era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents," *arxiv*, pp. 2–3, 2023, Accessed: 2024-04-27. [Online]. Available: `https://arxiv.org/abs/2310.08837`.

[23] Joern-Team, *Joern documentation*, Accessed: 2024-04-22. [Online]. Available: `https://docs.joern.io/`.

[24] ——, *Joern documentation*, Accessed: 2024-04-22. [Online]. Available: `https://docs.joern.io/c-syntaxtree/`.

[25] ——, *Joern documentation*, Accessed: 2024-04-22. [Online]. Available: `https://docs.joern.io/interpreter/`.

[26] GDB-Team, "Python api," Accessed: 2024-05-25. [Online]. Available: `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Python-API.html`.

[27] S. Mallissery and Y.-S. Wu, "Demystifying fuzzing methods: A comprehensive survey," Accessed: 2024-05-07, pp. 3–21. [Online]. Available: `https://wcventure.github.io/FuzzingPaper/Paper/ACM_CSUR22_Demystify_Fuzzing.pdf`.

[28] C. Benvenuti, *Understanding Linux Networking Internals*. O'Reilly Media, 2006, ISBN: 978-0596002558.