

# Falco Security Audit

---

## Report



**Quarkslab**

**Reference** 23-01-1097-LIV  
**Version** 1.0  
**Date** 2023/01/16

**Quarkslab SAS**  
10 boulevard Haussman  
75009 Paris  
France

# Contents

<b>1</b>	<b>Project Information</b>	<b>1</b>
<b>2</b>	<b>Executive summary</b>	<b>2</b>
2.1	Disclaimer . . . . .	2
2.2	Findings summary . . . . .	2
<b>3</b>	<b>Context and scope</b>	<b>4</b>
3.1	Context . . . . .	4
3.2	Scope . . . . .	4
3.3	Audit settings . . . . .	5
<b>4</b>	<b>Discovery and state of the art</b>	<b>6</b>
4.1	Discovery . . . . .	6
4.2	State of the art . . . . .	6
<b>5</b>	<b>Threat model</b>	<b>7</b>
5.1	A note on threat actors . . . . .	8
<b>6</b>	<b>Static analysis</b>	<b>9</b>
6.1	Automated static analyzers . . . . .	9
6.1.1	Cppcheck . . . . .	9
6.1.2	Infer . . . . .	12
6.1.3	CodeQL . . . . .	14
6.1.4	Scan-Build . . . . .	15
6.1.5	Conclusion . . . . .	17
6.2	Manual review . . . . .	18
6.2.1	Issues with readlink . . . . .	18
6.2.2	Checks on sensitive functions . . . . .	23
6.2.3	Third-party dependencies version . . . . .	24
6.2.4	Conclusion . . . . .	24
<b>7</b>	<b>Dynamic analysis</b>	<b>25</b>
7.1	Fuzzing the rules parser . . . . .	25
7.2	Fuzzing the event processor . . . . .	28
7.2.1	Using libprotobuf-mutator . . . . .	29
7.2.2	Using syzkaller . . . . .	35
7.2.3	Using the scap file format . . . . .	41
7.3	Conclusion . . . . .	44
<b>8</b>	<b>Conclusion</b>	<b>45</b>
	<b>Glossary</b>	<b>46</b>
	<b>Bibliography</b>	<b>47</b>

<b>A</b>	<b>Severity Classification</b>	<b>48</b>
<b>B</b>	<b>Infer report extracts</b>	<b>49</b>
<b>C</b>	<b>Scan-Build experimentations</b>	<b>54</b>
<b>D</b>	<b>Exploiting and debugging readlink issues</b>	<b>56</b>
<b>E</b>	<b>AFL++ persistent mode boilerplate</b>	<b>58</b>
<b>F</b>	<b>Fuzzing with libprotobuf-mutator</b>	<b>59</b>

# 1 Project Information

Document history			
Version	Date	Details	Authors
1.0	2023/01/16	Initial Version	Victor Houal & Laurent Laubin & Mahé Tardy

Quarkslab		
Contact	Role	Contact Address
Frédéric Raynal	CEO	fraynal@quarkslab.com
Ramtine Tofighi Shirazi	Project Manager	mrtofighishirazi@quarkslab.com
Victor Houal	R&D Engineer Apprentice	vhoul@quarkslab.com
Laurent Laubin	R&D Engineer	llaubin@quarkslab.com
Mahé Tardy	R&D Engineer	mtardy@quarkslab.com

Falco maintainers		
Contact	Company	Contact Address
Frederico Araujo	IBM	frederico.araujo@ibm.com
Jason Dellaluce	Sysdig	jasondellaluce@gmail.com
Mauro Ezequiel Moltrasio	Red Hat	mmoltras@redhat.com
Leonardo Grasso	Sysdig	me@leonardograsso.com
Luca Guerra	Sysdig	luca.guerra@sysdig.com
Teryl Taylor	IBM	terylt@ibm.com
Michele Zuccala	Sysdig	michele@zuccala.com

OSTIF		
Contact	Role	Contact Address
Derek Zimmer	President and Executive Director	derek@ostif.org
Amir Montazery	Managing Director	amir@ostif.org

## 2 Executive summary

The goal of the audit was to assist the Falco maintainers to increase their security posture using static and dynamic analysis. Falco maintainers required an emphasis on fuzzing. To that end, Quarkslab's engineers researched multiple topics to provide recommendations and relevant advice. In addition, Quarkslab's engineers assessed the code in order to find some issues, using automated testing tools, fuzzing, or just by manually reviewing the code base.

This report describes the steps and research conducted by Quarkslab's engineers on static analysis and fuzzing. In addition, readers can refer to Section 2.2 for the summary of findings that were found during the audit.

The report starts with introductory sections, the next Section 3, describes the context and scope that were decided for the audit, then Section 4 presents how the auditors got familiar with the project and the state-of-the-art research that was conducted, accompanied by a bibliography. Afterwards, Section 5 presents the threat model that was created, with Section 6 and Section 7 presenting the static and dynamic analysis resulting from the threat modeling. The section on static analysis presents static analyzers that were tested on the project with their results, while the section on dynamic analysis illustrates and explains the research conducted on how fuzz-testing was applied to various parts of the project.

### 2.1 Disclaimer

This report reflects the work and results obtained within the duration of the audit on the specified scope (see Section 3.2) and as agreed between the OSTIF, Falco maintainers, and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure the code is bug or vulnerability free.

### 2.2 Findings summary

The following table synthesizes the various findings that were uncovered during the audit. The severity classification given as informative, low, and medium, reflects a relative hierarchy between the various findings of this report (see the table in Appendix A). It depends on the threat model and security properties considered.

ID	Description	Category	Severity
MEDIUM 1	Potential buffer overflow due to not null terminated output of readlink in lib-scrap/scap_proc_file_root	Buffer overflow	Medium
LOW 1	Memory leak on error structure in lib-scrap/engine/bpf/scap_bpf.c:513	Memory Leak	Low
LOW 2	Resource leak on pfile in lib-scrap/engine/kmod/scap_kmod.c:67	Resource Leak	Low

ID	Description	Category	Severity
LOW 3	Memory leak on handle structure in libscap/scap.c:146	Memory Leak	Low
LOW 4	Memory leak on pAdapterInfo structure in libscap/windows_hal.c:342	Memory Leak	Low
LOW 5	Multiple unchecked return value from malloc, calloc and realloc	Null dereference	Low
LOW 6	Multiple unchecked return value from localtime, sinsp_threadinfo::get_fd_table and scap_write_proclist_begin that can return Nullptr	Null dereference	Low
LOW 7	Double free in libscap/scap.c function scap_open	Double free	Low
LOW 8	Garbage return value from stack in lib-sinsp/sinsp.cpp	Garbage return value	Low
LOW 9	Four null terminations of buffers written out of range by one in libscap/scan_fds.c	Buffer overflow	Low
INFO 1	Multiple bad handling of realloc return value in libsinsp/filterchecks.cpp	Memory Leak	Info
INFO 2	Missing va_end() after va_copy() in test file sinsp_with_test_input.h	Incorrect handling of variadic macros	Info
INFO 3	Returned heap allocated buffer resolved after being deallocated	Return pointer on freed memory	Info
INFO 4	Multiple potential Null pointer dereferences in macro HASH_ADD_INT64	Null dereference	Info
INFO 5	Resource leak in libsinsp example lib-sinsp/examples/test.cpp	Resource leak	Info
INFO 6	Dangerous construct in a vforked process in libscap/engine/gvisor/runsc.cpp	Incorrect use of vfork	Info
INFO 7	Buffer overflow with not null terminated output of readlink in debug code in lib-sinsp/parsers.cpp	Buffer overflow	Info
INFO 8	Return value in various usage of readlink not checked which could lead to write to a different file	Unchecked return value	Info
INFO 9	Multiple crashes in the parsing of scap files and event buffer with malformed files	Null dereference	Info

## 3 Context and scope

### 3.1 Context

Falco's README introduces the project like that:

The Falco Project, originally created by Sysdig, is an incubating [CNCF](#) open-source cloud-native runtime security tool. Falco makes it easy to consume kernel events and enrich those events with information from Kubernetes and the rest of the cloud-native stack. Falco can also be extended to other data sources by using plugins. Falco has a rich set of security rules specifically built for Kubernetes, Linux, and cloud-native. If a rule is violated in a system, Falco will send an alert notifying the user of the violation and its severity.

Falco monitors system calls to secure a system, by:

- Parsing the Linux system calls from the kernel at runtime.
- Asserting the stream against a powerful rules engine.
- Alerting when a rule is violated.

The project ships with a default set of rules ready to be consumed by end-users to secure their Kubernetes clusters. However, users can write their own rules using a syntax created by the project for specific needs.

Historically, Falco was the first runtime security project to join [CNCF](#) as an incubation-level project.

### 3.2 Scope

The scope of this audit was mainly the userspace part of Falco, distributed in the *falcosecurity/falco* and *falcosecurity/libs* repositories on GitHub. Refer to Table 3.1 and 3.2 for URLs and commit hashes. The kernel module was investigated although it was not the focus of the audit. The dependencies and the plugins of Falco were left out of scope. The eBPF drivers of Falco also were not investigated.

The Falco team requested assistance on:

- building fuzzers for security-relevant areas;
- building a threat model;
- using and adding automatic static analysis in their pipeline;
- looking for improper usage of cryptography;
- manually searching for vulnerability, with an emphasis on memory safety.

Quarkslab proposed a multi-step approach:

- discovery of the project, its documentation, build system, architecture and codebase;

- threat modeling;
- static code review, including automatic and manual reviews;
- dynamic testing and fuzzing.

### 3.3 Audit settings

See below Table 3.1 and 3.2 for the versions used to conduct this audit. Most of the work was conducted on the libs repository, thus falco repository is needed to compile the final binary.

<b>Project</b>	falco
<b>Repository</b>	<a href="https://github.com/falcosecurity/falco">https://github.com/falcosecurity/falco</a>
<b>Commit hash</b>	44d1c1eb65031b8895b154139a1cc7bb545df60a
<b>Commit date</b>	2022/10/19
<b>Tag</b>	0.33.0

Table 3.1: falco version references

<b>Project</b>	libs
<b>Repository</b>	<a href="https://github.com/falcosecurity/libs">https://github.com/falcosecurity/libs</a>
<b>Commit hash</b>	74eec76d2c1dbba66a37db227d25a6b41987c1a6
<b>Commit date</b>	2022/10/13
<b>Tag</b>	0.9.0

Table 3.2: libs version references



## 4 Discovery and state of the art

### 4.1 Discovery

Falco is overall well-documented and easy to integrate. The documentation is pretty comprehensive, well-written, and easy to follow. For example, to build and run Falco from the source, the official documentation is great [1]. A tutorial to run Falco on a minikube cluster is also available.<sup>1</sup>

We had no major difficulty to get familiar with the project and understand the cmake setup, the only difficulty could be the high number of dependencies that Falco relies on.

### 4.2 State of the art

To begin, an audit of Falco was conducted by Cure53 in mid-2019 [2]. They performed a manual analysis of the source, some dynamic analysis, and tried to bypass the default set of rules. They found some interesting bypass, configuration issues and crashes.

We noted that almost all the issues from the audit and security advisories were fixed except for a symlink file bypass. Cure53 discussed this topic in their report, giving the example of using `/proc/self/root` to bypass detection on a specific path, like `/etc/shadow`. Open discussions are still happening on how to prevent these issues [3].

Specifically on bypasses, Mark Manning, at the time at NCC Group, wrote about it at the end of 2019 [4]. Then, Brad Geesaman, at the time founding Darkbit, published a bypass article a year after [5], September 2020. Then a Falco maintainer from Sysdig, Leonardo Di Donato, did a presentation at KubeCon Europe 2021 Virtual on the subject [6]. Finally a researcher at Blackberry, Shay Berkovich presented a talk virtually at KubeCon Europe 2022 on new bypasses [7].

Then, some articles are great to understand the inner workings of Falco, for example this official blog post about “Monitoring new syscalls with Falco” [8]. Or articles on personal blogs that try to go deep into the code like this series on Falco design and the source code of Sysdig [9] [10].

The previous CVEs on Falco were investigated, first a [CVE-2019-8339](#), a capacity-related vulnerability in which you can overflow the kernel with events to drop the control on sensitive ones [11]. The GitHub security advisory pages on Falco and libs repository [12], containing the recent security issues were also looked into.

---

<sup>1</sup><https://falco.org/docs/getting-started/source/>

## 5 Threat model

Falco maintainers explicitly asked for a threat model, see Figure 5.1 for the threat model proposed by Quarkslab's engineers.

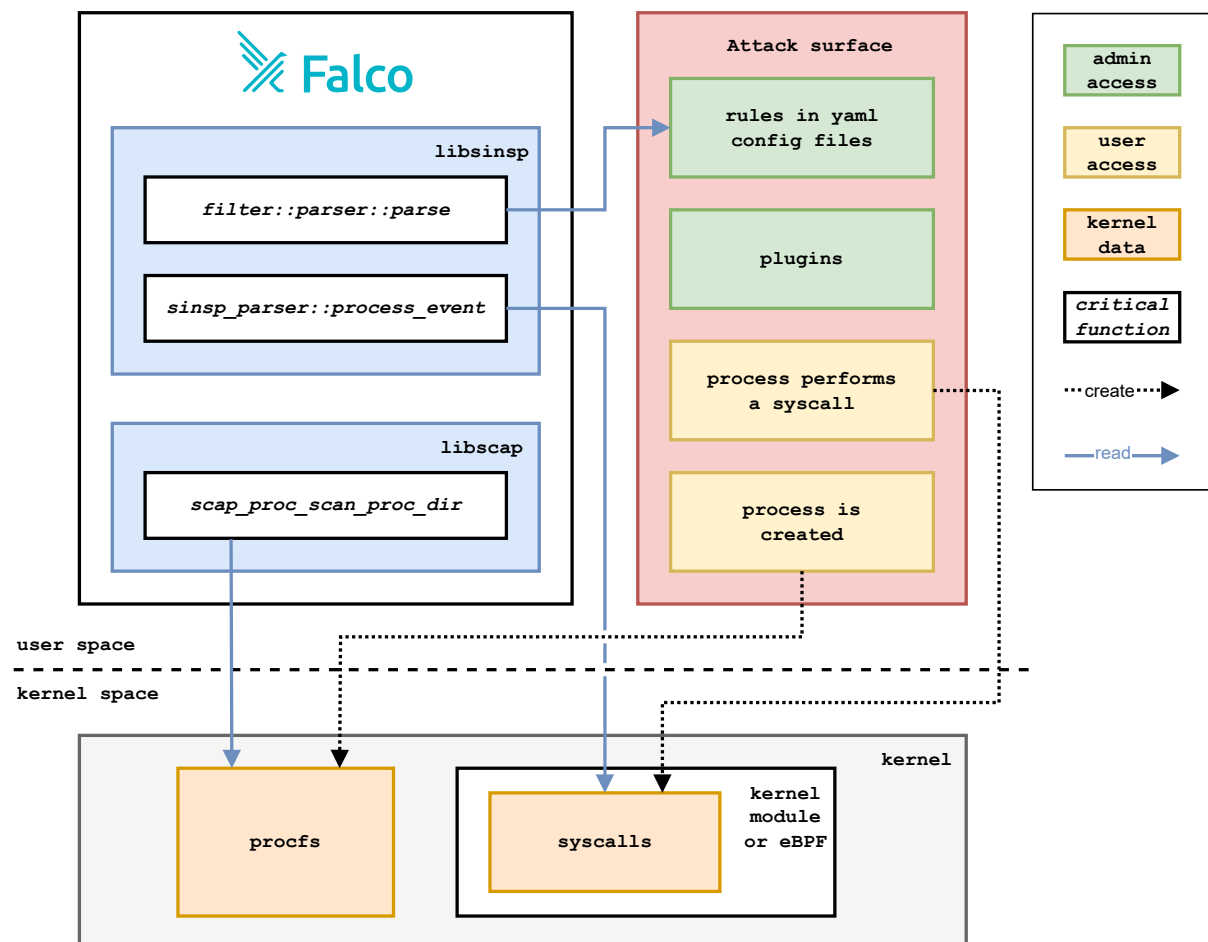


Figure 5.1: Falco threat model

This threat model is a simplified view of the security threats on Falco and is not technically comprehensive. For example, a process creation will also generate a syscall event, which is not represented on the diagram. The goal of this figure was to identify the security-relevant areas of Falco to head the audit in the most interesting direction.

To describe further this diagram:

- the attack surface is represented in red, with items requiring admin access, in green, and items accessible from a normal user using the system, in yellow.
- In the boxes in blue, you can find the parts of the code that interact with sensitive information, via reading data represented with the blue arrows.

- The sensitive parts of the code are reading directly from attack surface items, for the YAML config files for example, or indirectly from the system, with procs or the syscalls events.
- The dotted arrows just represent the relation between the attack surface and the system for the indirect links.

## 5.1 A note on threat actors

On the attack surface of Falco, it was important to separate what can be targeted by a superuser (i.e., root or admin) and a regular user on the system. Some of the work during this audit targeted components that should be accessible only to superusers, see Section 7.1 for an example. This fuzzer was built, on one side, to get more familiar with the Falco project and on the other side because the rules files could theoretically be accessible to unprivileged users if the Linux file rights were badly configured.

An emphasis should be put on the fact that finding security issues that absolutely require to be a superuser (i.e., root) is less interesting because having those rights on the host already allows the superuser to disable Falco completely. Indeed, a superuser could stop the userland program of Falco or remove the kernel module, or eBPF probe directly. That's why the threat model and this audit insisted on the attack surface that could be reached by a non-admin user of the system.

Considering Falco might be installed on Kubernetes clusters nodes, it's safe to assume that most users will be non-trusted unprivileged users, or diminished root users in containers. Indeed by default, as root in Kubernetes pods, the default set of Linux capabilities is restricted, which means root doesn't have all capabilities, like `CAP_SYS_ADMIN` or the ones allowing to reboot or remove and add kernel modules. In addition, other security mechanisms limit the access to `/proc` or `/sys` as root.

The plurality of "root status" in containers environment creates some confusion. Linux capabilities, namespace and the various security modules allow to create more complex combination of rights. However, considering the most used configuration it should be clarified that regular users on the system, i.e., unprivileged users or containerized root users in Kubernetes pods are the main threat. Root users on the host and root users in privileged containers should be out of the scope of the threat actors. As a matter of fact, privileged containers with root users are almost equivalent to root processes on the host: all except the process Linux namespace are disabled, all capabilities are granted, virtual file systems are unmasked and LSMs are disabled. It's trivial to pivot from a privilege container process to "complete root" on the host.

## 6 Static analysis

### 6.1 Automated static analyzers

Some linters and static checkers were selected and tested on the project to find immediate issues but also to see if they could be easily integrated to the project workflows. These tools were mainly selected because engineers at Quarkslab were familiar with them and they are (most of them) free and open-source.

- **Cppcheck**: a static analysis tool for C/C++ code.
- **CodeQL**: a semantic code analysis engine to discover vulnerabilities across a codebase.
- **Infer**: a static analysis tool for Java, C++, Objective-C, and C.
- **Scan-Build**: a command line utility that enables a user to run the clang static analyzer over their codebase as part of performing a regular build.

These tools are different some of them are linters that can be directly run against the source code, while others need the project to be built to create databases against which the analyzer can run. Some already propose a set of embedded checks, while others are only engines and need a curated selection of queries. **Semgrep** was also considered but the support of C/C++ is still experimental.

#### 6.1.1 Cppcheck

Note that despite the name, Cppcheck is designed for both C and C++. It can provide good information on potential memory and resource leaks for example and is really easy to run. It runs directly on source code and thus could be quickly integrated into a CI step. The version used was Cppcheck 1.90.

Cppcheck was run on the codebase with the following command:

```
cppcheck -j 32 -q --force <folder>
```

The `-j` flag is used to start 32 threads to do the checking work, the `-q` to only print something when there is an error and avoid progression message, `--force` for checking files that have a lot of configurations (using a lot of different combinations depending on C macro). To enable more rules, is it possible to pass the `--enable` flag, for example `--enable=all` will output a lot of errors that could be associated with style, performance, portability, etc.

For this output, the userspace codebase was scanned with the basic checkers enabled.

**libscap** The first issue, on line 1, is a memory leak on the heap allocated error structure, indeed `free(error)` should be added just before line 513 in `engine/bpf/scap_bpf.c`. The second one, line 4, is a resource leak, in the case `fscanf(pfile, "%PRIu32, &max)` returns 0, the function will return without calling `fclose(3)`. Then, line 7, is a memory leak similar to the first issue, on

line 146 of `scap.c`, a `free(handle)` should be added just before. For the issues line 10, 13 and 16, it seems that the `pAdapterInfo` is allocated and then not used nor freed, which could lead to a memory leak.

```

1  engine/bpf/scap_bpf.c:513:3: error: Memory leak: error [memleak]
2      return SCAP_FAILURE;
3      ^
4  engine/kmod/scap_kmod.c:67:4: error: Resource leak: pfile [resourceLeak]
5      return 0;
6      ^
7  scap.c:146:3: error: Memory leak: handle [memleak]
8      return NULL;
9      ^
10 windows_hal.c:325:3: error: Memory leak: pAdapterInfo [memleak]
11     return SCAP_FAILURE;
12     ^
13 windows_hal.c:331:3: error: Memory leak: pAdapterInfo [memleak]
14     return SCAP_FAILURE;
15     ^
16 windows_hal.c:342:3: error: Memory leak: pAdapterInfo [memleak]
17     return SCAP_FAILURE;
18     ^

```

<b>LOW 1</b>	Memory leak on error structure in libscap/engine/bpf/scap_bpf.c:513	
<b>Category</b>	Memory Leak	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

<b>LOW 2</b>	Resource leak on pfile in libscap/engine/kmod/scap_kmod.c:67	
<b>Category</b>	Resource Leak	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

<b>LOW 3</b>	Memory leak on handle structure in libscap/scap.c:146	
<b>Category</b>	Memory Leak	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

<b>LOW 4</b>	Memory leak on pAdapterInfo structure in libscap/windows_hal.c:342	
<b>Category</b>	Memory Leak	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

**libsinsp** The first issue, on line 1, with the `uninitvar` is certainly a false positive. However, the `memleakOnRealloc` issues, on lines 4, 7, 10 are real programming mistakes<sup>1</sup>. The last one, on line 13, is also a real mistake but in the test files, so less problematic.

```

1  dns_manager.cpp:111:9: error: Uninitialized variable: dinfo [uninitvar]
2      return dinfo;
3      ^
4  filterchecks.cpp:5174:3: error: Common realloc mistake: 'm_storage' nulled but not
   ↪ freed upon failure [memleakOnRealloc]
5      m_storage = (char*)realloc(m_storage, encoded_args_len);
6      ^
7  filterchecks.cpp:5326:5: error: Common realloc mistake: 'm_storage' nulled but not
   ↪ freed upon failure [memleakOnRealloc]
8      m_storage = (char*)realloc(m_storage, encoded_tags_len);
9      ^
10 filterchecks.cpp:5810:4: error: Common realloc mistake: 'm_storage' nulled but not
   ↪ freed upon failure [memleakOnRealloc]
11     m_storage = (char*)realloc(m_storage, encoded_tags_len);
12     ^
13 test/sinsp_with_test_input.h:95:117: error: va_list 'args2' was opened but not
   ↪ closed by va_end(). [va_end_missing]
14     throw std::runtime_error("the test framework does not currently support equal
   ↪ timestamps or out of order events");

```

<b>INFO 1</b>	Multiple bad handling of <code>realloc</code> return value in <code>libsinsp/filterchecks.cpp</code>		
<b>Category</b>	Memory Leak		
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None	

<b>INFO 2</b>	Missing <code>va_end()</code> after <code>va_copy()</code> in test file <code>sinsp_with_test_input.h</code>		
<b>Category</b>	Incorrect handling of variadic macros		
<b>Rating</b>	<b>Impact:</b> Integrity	<b>Exploitability:</b> None	

**chisel** These three issues are the same, the function `realpath_ex` in `userspace/chisel/chisel_utils.cpp` returns a pointer on freed memory. It seems to be a mistake because a variable is created just before, named `ret`, on line 93, that might be the variable that should be returned.

```

1  userspace/chisel/chisel_utils.cpp:95:2: error: Returning/dereferencing 'resolved'
   ↪ after it is deallocated / released [deallocret]
2      return resolved;
3      ^

```

<sup>1</sup>More details on how to properly check the result of `realloc`: <https://stackoverflow.com/a/27589881/4561420>.

```

4  userspace/chisel/chisel_utils.cpp:94:2: note: Returning/dereferencing 'resolved'
   ↪ after it is deallocated / released
5  free(resolved);
6  ^
7  userspace/chisel/chisel_utils.cpp:95:2: note: Returning/dereferencing 'resolved'
   ↪ after it is deallocated / released
8  return resolved;
9  ^

```

<b>INFO 3</b>	Returned heap allocated buffer <code>resolved</code> after being deallocated	
<b>Category</b>	Return pointer on freed memory	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

### 6.1.2 Infer

Infer is an open-source static analysis tool designed for C, C++, Java and Objective-C code by Facebook. Similarly to Scan-Build or CodeQL, Infer uses the compilation step to create an intermediate representation of the program on which it can then run an analysis. The version used for Infer was 1.1.0.

Infer 1.1.0 is using clang 11 and compiling the whole Falco project with dependencies with it will cause errors during the build process because of tbb<sup>2</sup>. However, it is possible to build the problematic dependency beforehand to avoid the issue. In addition, some implementations are missing when compiling with the Infer toolchain, however, adding `add_link_options(-latomic)` fixes the issue.

Like other tools that create databases during the building process, you need to explicitly exclude files from the included ones otherwise you end up with a lot of results coming from the dependencies code.

We managed to make Infer analyze the project with the following setup.

```

vim CMakeLists.txt # add 'add_link_options(-latomic)'
mkdir build-infer && cd build-infer
infer compile -- cmake -DUSE_BUNDLED_DEPS=ON ..
make -j `nproc` tbb

```

<sup>2</sup>It seems that `-flifetime-dse=1`, is not supported on clang 11 with Infer.

```
infer capture --skip-analysis-in-path b64-prefix --skip-analysis-in-path
↳ c-ares-prefix --skip-analysis-in-path catch2-prefix --skip-analysis-in-path
↳ cloudtrail-plugin-prefix --skip-analysis-in-path cloudtrail-rules-prefix
↳ --skip-analysis-in-path cpp-http-lib-prefix --skip-analysis-in-path curl-prefix
↳ --skip-analysis-in-path cxxopts-prefix --skip-analysis-in-path fakeit-prefix
↳ --skip-analysis-in-path grpc-prefix --skip-analysis-in-path jq-prefix
↳ --skip-analysis-in-path json-plugin-prefix --skip-analysis-in-path
↳ k8saudit-plugin-prefix --skip-analysis-in-path k8saudit-rules-prefix
↳ --skip-analysis-in-path njson-prefix --skip-analysis-in-path openssl-prefix
↳ --skip-analysis-in-path protobuf-prefix --skip-analysis-in-path re2-prefix
↳ --skip-analysis-in-path string-view-lite-prefix --skip-analysis-in-path
↳ tbb-prefix --skip-analysis-in-path valjson-prefix --skip-analysis-in-path
↳ yamlcpp-prefix --skip-analysis-in-path zlib-prefix -- make -j `nproc`
infer analyze
```

## Results

During our scan, it found 105 issues. See the following list for comments on the results, you will need Infer report to follow, you can find the report under the name `infer-report.txt` or an abbreviated version in the Appendix B containing only the errors mentioned in the following comments.

- **27 Null Dereferences** - 13 of these errors come from unchecked return value from `malloc`, `calloc` and `realloc` that could lead to Null dereferences, it is errors number 9, 15, 16, 25, 28, 29, 42, 45, 52, 57, 59, 74 and 80.

Error number 17 is similar, on error `localtime(3)` returns Null which is not checked in the code. Errors 54, 56 and 69 are very similar, `sinsp_threadinfo::get_fd_table` can return Null and the value is not checked, it's even immediately dereferenced for case 69. For error 79, potentially `scap_write_proclist_begin` can return Null and the returned pointer is dereferenced in `scap_write_proclist_end` later.

Errors 43, 51, 53, 67, 68, 70, 71, 87 and 88 are issues of dereferences in `hashmap` macro `HASH_ADD_INT64`.

- **7 Resource Leaks** - interestingly, the first resource leak, number 13 is the same as the one detected by Cppcheck in `libscape/engine/kmod/scap_kmod.c` and is valid. See low Recommendation 2.

Error number 24 is a false positive, it's not considering `sinsp_logger` destructor that is properly closing the related files.

Error number 38 is a real issue but in `libsinsp/examples/test.cpp` which is not sensitive.

Errors 60, 61 and 62 are false positives since these files descriptors are saved and the files are closed by calling `scap_dump_close`.

The last error 78 seems to be a false positive because the normal flow actually `closedir` on the `taskdir_p` properly and the error flow as well.

- **26 Dead Stores** - 9 of these errors are raised from the driver folder. While dead stores do not have security implications by themselves, the compiler optimizing them out can lead to



security issues if the underlying memory should be rewritten. However, it does not seem that these dead stores are because of memory reset for sensitive information. They can still be checked for program correctness because they might imply a programming mistake (a var assigned in the place of another).

- **17 Static Initialization Order Fiascos** - all these errors come from detection on GTest TEST\_CASE macro. We can consider them as false positives and due to the use of GTest.
- **28 Uninitialized Values** - these errors seems to be false positive of access to uninitialized variables.

<b>LOW 5</b>	Multiple unchecked return value from malloc, calloc and realloc	
<b>Category</b>	Null dereference	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

<b>LOW 6</b>	Multiple unchecked return value from localtime, sinsp_threadinfo::get_fd_table and scap_write_proclist_begin that can return Nullptr	
<b>Category</b>	Null dereference	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

<b>INFO 4</b>	Multiple potential Null pointer dereferences in macro HASH_ADD_INT64	
<b>Category</b>	Null dereference	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

<b>INFO 5</b>	Resource leak in libsinsp example libsinsp/examples/test.cpp	
<b>Category</b>	Resource leak	
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> None

### 6.1.3 CodeQL

First, contrary to the other tools mentioned here, CodeQL is not entirely open source. It's free for research and open source, the queries are open source but the engine is closed source and published as a binary. It was originally built by a company named Semmle that hosted the product on [lgtm.com](https://lgtm.com) with the idea of massively scanning open-source projects. Semmle was acquired by GitHub (itself owned by Microsoft) and is now integrating CodeQL into [github.com](https://github.com) and deprecating [lgtm.com](https://lgtm.com). The versions used were CodeQL command-line toolchain release 2.11.2 and the Visual Studio Code extension v1.7.5.

To use CodeQL, an SQL database of the source code and data flow must be built prior to analysis.

```
codeql database create --language=cpp -j 32 --ram=25000 --command="make -j 32"
↪ falco-db
```

Note that the `--language=cpp` flag includes C code as well, the `-j` and `--ram` flags are to adjust the process of creation to our resources available, the `--command` flag allows to pass a custom build command to trigger the compilation and the last argument is the name of the database folder.

### Warning

Because CodeQL is creating a database based on everything that was compiled during the build the process, it results in a pretty large database, including information about the dependencies like OpenSSL, protobuf, curl, etc.

This is the main issue with using CodeQL on Falco at the moment because executing any queries on the resulting database can lead to a lot of noise from dependencies and thus make the results difficult to browse.

Because CodeQL does not come bundled with predefined queries, it's more complicated to exhibit particular results from the analysis. However, the open source queries can be found in the [github/codeql](https://github.com/github/codeql)<sup>3</sup> repository. And especially, C and C++ queries, can be found under `codeql/cpp/ql/src`<sup>4</sup>.

The rule sets used were the ones under `Security/CWE` and `Critical`. There were a lot of outputs, with a lot of noise, coming from dependencies and false positives. The results will not be printed here but they were an important inspiration for the manual review. The queries were run through the Visual Studio Code extension of CodeQL.

## 6.1.4 Scan-Build

Scan-Build is a tool to run the Clang static analyzer<sup>5</sup>, part of the LLVM project, on a codebase. You can also use CodeChecker instead of Scan-Build<sup>6</sup> which might be more adequate for large projects. During a project build, as source files are compiled they are also analyzed in tandem by the static analyzer. Upon completion of the build, results are then presented to the user within a web browser. The version used of Scan-Build is the one packaged with the LLVM 10 clang-tools package on Ubuntu 20.04.

Similarly to Infer and CodeQL, this tool requires compiling the codebase and checking the built code. Thus, it, by default, raises tons of alerts on dependencies that were built along Falco.

At first, we struggled to make Scan-Build work on the project, see Appendix C for more details. Finally, we used the following command to scan and output only the bugs found on Falco's codebase and it worked, resulting in 202 warnings (with 41 "bugs" from the security checkers that are mostly just grepping some sensitive functions, which could be disabled):

<sup>3</sup><https://github.com/github/codeql>

<sup>4</sup><https://github.com/github/codeql/tree/main/cpp/ql/src>

<sup>5</sup><https://clang-analyzer.llvm.org/>

<sup>6</sup><https://clang-analyzer.llvm.org/command-line.html>

```
cmake -DUSE_BUNDLED_DEPS=ON
↳ -DCMAKE_C_COMPILER=/usr/share/clang/scan-build-10/bin/./libexec/ccc-analyzer
↳ -DCMAKE_CXX_COMPILER=/usr/share/clang/scan-build-10/bin/./libexec/c++-analyzer
↳ ..
scan-build -enable-checker unix,nullability,core,cplusplus,security --exclude
↳ b64-prefix --exclude c-ares-prefix --exclude catch2-prefix --exclude
↳ cloudtrail-plugin-prefix --exclude cloudtrail-rules-prefix --exclude
↳ cpp-http-lib-prefix --exclude curl-prefix --exclude cxxopts-prefix --exclude
↳ fakeit-prefix --exclude grpc-prefix --exclude jq-prefix --exclude
↳ json-plugin-prefix --exclude k8saudit-plugin-prefix --exclude
↳ k8saudit-rules-prefix --exclude njson-prefix --exclude openssl-prefix
↳ --exclude protobuf-prefix --exclude re2-prefix --exclude
↳ string-view-lite-prefix --exclude tbb-prefix --exclude valijson-prefix
↳ --exclude yamlcpp-prefix --exclude zlib-prefix make -j 32
```

Note that the `--exclude` flag will not disable checking the code of dependencies but just filter out the results in the report. That's why this scan can take a significant amount of time to process.

Scan-Build generates an HTML report that explains each issue in detail, illustrating the multiple steps and the branches the program must take to be in the situation indicated by the tool. The HTML report generated from the above steps will be included with this report under the file name `scan-build-44d1c1e.tar.gz`, open the `index.html` file for more information.

## Results

Apart from the security checkers (security checker is just issuing warning based on some function name), the clang static analyzer found some potential issues. Here are some comments on what it found, some issues seem real, some are certainly false positives, and some may come from confusion in macros.

- **A double free** - in `libscap` `scap.c` file, from the `scap_open` function, when calling `scap_open_udig_int`, a double free occurs because the function that calls `scap_close(handle)` will free the handle, and on the next line, the handle is freed again with `free(handle)`. Removing line 317 should solve the issue.
- **A garbage return value** - in `libsinsp` `sinsp.cpp`, the function `sinsp::get_read_progress_with_str`, with a specific control flow execution detailed in the report, can return a non-initialized stack variable, which might be garbage and disrupt the execution.
- **A dangerous construct in a vforked process** - in the code related to the gvisor engine in `libscap/engine/gvisor/runsc.cpp`, the vforked process is closing a file descriptor which could be an undefined behavior according to POSIX, and it should not call `exit(3)` as well. Some refactoring could be done considering the restriction and caution on using `vfork(2)`.
- **Ten NULL pointer dereferences** - the path of execution are various in length but it might be interesting to investigate these highlight to fix this potential issues that could result in crashes.
- **Five cases of argument with nonnull attribute passed null** - the static analyzer revealed 41 issues like that, it seems that most of them are not interesting because they highlight a potential null value in a comparison with an int. However, five of these

highlights result in a null pointer being passed to arguments of memcpy, three of them the destination, and two of them the source. Here are their names in the scan-build-44d1c1e.tar.gz archive: report-2dfbca.html, report-9cb243.html, report-760867.html, report-940464.html, report-e46a72.html.

- **Five memory leaks** - in various places of the codebase, they seem to be realistic scenarios of memory leaks. Some involve more steps than others and their path of execution might terminate execution but should be investigated.
- **Four dead store issues** - two dead assignments and two dead increments which are real but might not even be considered as bugs since eliminating them could lead to other bugs in the future by missing these assignments and increments.
- **Use of zero allocated memory** - in function scap\_write\_proc\_fds which is certainly expected, so a false positive.
- **Some division by zero and use-after-free issues** - a total of eighteen issues that seem to be false positives confusions because of the usage of macro for hash maps manipulation.

LOW 7	Double free in libscap/scap.c function scap_open
Category	Double free
Rating	<b>Impact:</b> Integrity, Confidentiality, <b>Exploitability:</b> None Availability

LOW 8	Garbage return value from stack in libsinsp/sinsp.cpp
Category	Garbage return value
Rating	<b>Impact:</b> Availability <b>Exploitability:</b> None

INFO 6	Dangerous construct in a vforked process in libscap/engine/gvisor/runsc.cpp
Category	Incorrect use of vfork
Rating	<b>Impact:</b> Availability <b>Exploitability:</b> None

### 6.1.5 Conclusion

These four static analyzers produced interesting results. Because CodeQL cannot, at the moment, exclude files on database creation or systematically exclude files at the analysis stage, it's a bit difficult to integrate with Falco. Otherwise, including Cppcheck is easy since it's directly running on the source code. Then, integrating more elaborate analyzers like Infer and Scan-Build is more complex, but possible, and can provide more hints on potential issues in the future.

On top of that, some analyzers like Infer have options for differential analysis, thus running only on the addition that was made in a pull request<sup>7</sup>, thus cutting the noise from the rest of the project issues. Cppcheck can also only run on specified files, which can limit the scope of the analysis but provide useful information.

## 6.2 Manual review

The manual review was mainly performed in two ways, trying to find issues in sensitive areas, such as some libscap functions for example, and systematically searching for particular patterns on all code base. The first way guiding the latter. The code reviewed was the code of the libs.

### 6.2.1 Issues with readlink

From the Falco threat model (see Figure 5.1) we identified that libscap function `scap_proc_scan_proc_dir` must be investigated. Indeed, being able to influence `procfs` content is at the reach of every user of the system by starting carefully crafted processes. Such a function, because it is parsing text from the `procfs` file interface is highly specific to Linux and building an automated fuzzing setup, reproducing the whole `procfs` interface, would be extremely time-consuming and quickly out of date. That is why a manual review would be more efficient for this part of the code.

#### Issue in `scap_proc_fill_root` with `readlink`

Most of the code manipulating strings is carefully terminating them with 0 but an issue was spotted in the function that is called to retrieve the root folder path of a process.

```
554 static int32_t scap_proc_fill_root(scap_t *handle, struct scap_threadinfo* tinfo,
    ↪ const char* procdirname)
555 {
556     char root_path[SCAP_MAX_PATH_SIZE];
557     snprintf(root_path, sizeof(root_path), "%sroot", procdirname);
558     if ( readlink(root_path, tinfo->root, sizeof(tinfo->root)) > 0)
559     {
560         return SCAP_SUCCESS;
561     }
562     else
563     {
564         snprintf(handle->m_lasterr, SCAP_LASTERR_SIZE, "readlink %s failed (%s)",
565                 root_path, scap_strerror(handle, errno));
566         return SCAP_FAILURE;
567     }
568 }
```

The issue is that `readlink` does not null-terminate the string that will read and copied into `tinfo->root` and the size of the buffer is `sizeof(tinfo->root)` instead of `sizeof(tinfo->root) - 1` to let space for inserting a zero at the last index. Thus it's possible to create a path exceeding

---

<sup>7</sup><https://fbinfer.com/docs/steps-for-ci>

SCAP\_MAX\_PATH\_SIZE and chroot a program into it in order for this part of the code to write a not null-terminated string into the structure of the size.

See the following listing for the structure in which the root buffer appears, on line 8.

```
1  typedef struct scap_threadinfo
2  {
3      // [...] abbreviated
4      int64_t vtid;
5      int64_t vpid;
6      char cgroups[SCAP_MAX_CGROUPS_SIZE];
7      uint16_t cgroups_len;
8      char root[SCAP_MAX_PATH_SIZE+1];
9      int filtered_out; ///< nonzero if this entry should not be saved to file
10     scap_fdinfo* fdlist; ///< The fd table for this process
11     uint64_t clone_ts;
12     int32_t tty;
13     int32_t loginuid; ///< loginuid (avid)
14
15     UT_hash_handle hh; ///< makes this structure hashable
16 }scap_threadinfo;
```

In theory, because the static size buffer is inserted directly in the structure, the situation could potentially mean that future reads of this string, supposed to be null-terminated, could overflow and read the data of the next structure's fields values. However, the size of the buffer is SCAP\_MAX\_PATH\_SIZE + 1, which translates to 1025, so the compiler will align the next fields, resulting in the next integer to be shifted by some blocks. And because this structure is allocated using calloc, the bytes hole will be filled with zeros. The result will be a string with a size equal to the expected length plus one, which could potentially be an issue with copies trusting that the string is well formed.

It seems that this situation is not present in the codebase, for example, line 433 of userspace/libscap/scap\_savefile.c, the computation of the length is performed using strlen using the constant 1024 as max length rootlen = (uint16\_t)strlen(root, SCAP\_MAX\_PATH\_SIZE);.

Find in Appendix D more information about how to exploit this issue and debug with GDB to understand the situation.

<b>MEDIUM 1</b>	Potential buffer overflow due to not null terminated output of readlink in libscap/scap_proc_file_root	
<b>Category</b>	Buffer overflow	
<b>Rating</b>	<b>Impact:</b> Availability, Integrity	<b>Exploitability:</b> None

## Searching similar issues in libscap

Because a first issue was found with `readlink`, other parts of the codebase were investigated looking for the same programming mistake. We used `ripgrep`<sup>8</sup> to search the codebase.

```
userspace/libscap/scap_procs.c
63:     target_res = readlink(filename, tinfo->cwd, sizeof(tinfo->cwd) - 1);
558:    if ( readlink(root_path, tinfo->root, sizeof(tinfo->root)) > 0)
723:    target_res = readlink(filename, target_name, sizeof(target_name) - 1);
↪ // Getting the target of the exe, i.e. to which binary it points to
756:        // null-terminate target_name (readlink() does not append a null byte)
```

In `userspace/libscap/scap_procs.c`, the issue from the previous section is present on line 558, where `tinfo->root` is not null terminated. On line 63 and line 723, the `bufsize`, third argument of `readlink` is correctly set to the size of the buffer minus one.

```
userspace/libscap/scap_fds.c
418:    r = readlink(fname, link_name, SCAP_MAX_PATH_SIZE);
619:    r = readlink(fname, link_name, SCAP_MAX_PATH_SIZE);
727:    r = readlink(fname, link_name, SCAP_MAX_PATH_SIZE);
1609:   r = readlink(f_name, link_name, sizeof(link_name));
```

In `userspace/libscap/scap_fds.c`, all the `readlink` call are made with `SCAP_MAX_PATH_SIZE` (even in the forth case, because `link_name` is always initialized as `char link_name[SCAP_MAX_PATH_SIZE];`). The issue is that then these buffer are manually null terminated doing `link_name[r] = '\0'`; leading to an overflow by one. Let's see an example, the one line 619:

```
616   char link_name[SCAP_MAX_PATH_SIZE];
617   ssize_t r;
618
619   r = readlink(fname, link_name, SCAP_MAX_PATH_SIZE);
620   if (r <= 0)
621   {
622       return SCAP_SUCCESS;
623   }
624
625   link_name[r] = '\0';
```

Compiled with `-O0`, the compiler lets the locals in the order there are declared on the stack, thus we have on the stack (in the order of the stack growth), the return address, a canary, the frame pointer and the end of the `link_name`. So the line `link_name[r] = '\0'`; will effectively write a zero out of bound on the frame pointer, which might start with zeros.

So in this precise situation, there are no consequences. But in a different setup, where locals could be located before the buffer, the stack overflow could theoretically lead to issues.

It is recommended to always `readlink` with a `bufsize` of the buffer size minus one, and then to correctly null terminate the string using the return value as the index.

---

<sup>8</sup><https://github.com/BurntSushi/ripgrep>

<b>LOW 9</b>	Four null terminations of buffers written out of range by one in libscap/scan_fds.c	
<b>Category</b>	Buffer overflow	
<b>Rating</b>	<b>Impact:</b> Availability, Integrity	<b>Exploitability:</b> Easy

## Searching similar issues in libsinsp

```
userspace/libsinsp/sinsp_auth.cpp
71:         ssize_t sz = readlink(fd_path.c_str(), buf, sizeof(buf));
72-         if(sz != -1 && sz <= static_cast<ssize_t>(sizeof(buf)))
73-         {
```

The bufsize is equal to sizeof(buf) but it's not an issue since the value is later copied using real size of the content of buf using std::string::assign.

```
userspace/libsinsp/threadinfo.cpp
1108:         ret = readlink(proc_path, dirfd_path, sizeof(dirfd_path) - 1);
1109-         if (ret < 0)
1110-         {
```

In threadinfo.cpp, the second finding of the above output, the bufsize is correctly specified.

```
userspace/libsinsp/parsers.cpp
4486:         target_res = readlink((chkstr + "/").c_str(),
4487-         target_name,
4488-         sizeof(target_name) - 1);
```

Finally, in parsers.cpp, the target\_name buffer is not null terminated which could lead to an overflow. However this piece of code is by default not used, you must specify the -A flag to analyze the getcwd syscall. And on top of that, it is only debug code that is not shipped in the release.

<b>INFO 7</b>	Buffer overflow with not null terminated output of readlink in debug code in libsinsp/parsers.cpp	
<b>Category</b>	Buffer overflow	
<b>Rating</b>	<b>Impact:</b> Availability, Integrity, Confidentiality	<b>Exploitability:</b> None

## Checking the return value of readlink

In the previous sections, we searched for potential buffer overflows but, in addition to this issue, most of these usages of readlink are potentially vulnerable to referring to the wrong file. Indeed,



readlink takes the bufsize as a third argument and if the return value is equal to this, it means that “truncation may have occurred”. The return value should be compared against bufsize and increase the buffer or handle the error properly.

In many situations above, especially when reading from files in /proc/%d/fd, this issue is almost impossible to exploit since it would require a legitimate absolute path target longer than 1024 bytes because the path in the fd directory are already canonicals thanks to the kernel.

**Example in userspace/libsinsp/sinsp\_auth.cpp, line 71.** The readlink call could result in a situation where it returns bufsize, its third argument, which here is sizeof(buf). In this situation “truncation may have occurred” and it’s not possible to know if the path was exactly this size or more. This means that the buffer could contain a path to *a different file than expected*. See this extract of the method.

```
1  std::string sinsp_ssl::memorize_file(const std::string& disk_file)
2  {
3      std::string mem_file;
4      // [...]
5      if(fd != -1)
6      {
7          char buf[FILENAME_MAX] = { 0 };
8          std::ifstream ifs(disk_file);
9          std::string fd_path = "/proc/self/fd/" + std::to_string(fd);
10         ssize_t sz = readlink(fd_path.c_str(), buf, sizeof(buf));
11         if(sz != -1 && sz <= static_cast<ssize_t>(sizeof(buf)))
12         {
13             mem_file.assign(buf, sz);
14             std::string str;
15             std::ofstream ofs(mem_file, std::ofstream::out);
16             while(std::getline(ifs, str))
17             {
18                 ofs << str << '\n';
19             }
20         }
21         // [...]
22         return mem_file;
23     }
```

So if the real path behind the file descriptor symlink is too long, readlink returns sizeof(buf) line 10 of the example above, then, the if condition statement is true and the content of the buffer at the maximum size of sizeof(buf) is copied into mem\_file. Later, line 15, an output file stream is created based on this path and written to, line 18. So if the realpath is long enough, it’s possible to write to a different location than expected.

In theory, the program could use the symlink and let the kernel redirect the writing operation to the correct file but it seems that the functions want to return the realpath to the caller, line 22. We tried to exploit this bug but it seems that this method is private and never referenced in the whole codebase. So maybe this can be deleted instead of refactored.

<b>INFO 8</b>	Return value in various usage of <code>readlink</code> is not checked which could lead to write to a different file	
<b>Category</b>	Unchecked return value	
<b>Rating</b>	<b>Impact:</b> Confidentiality	<b>Exploitability:</b> None

## Conclusion on readlink

To conclude on the `readlink` syscall, there are three main issues in its usage:

1. `readlink` does not append a null byte to the buffer which could lead to buffer overflow if not manually written.
2. The return value is often used as an index to write the terminating zero but could go out of bounds if the `bufsize`, the third argument is equal to `sizeof(buf)` and not `sizeof(buf)-1`.
3. The return value should be checked not only against `-1` but also when it's equal to `bufsize`, the third argument. It could imply that truncation may have occurred, ending up with a path to the wrong file.

## 6.2.2 Checks on sensitive functions

Here is a non comprehensive list of common functions known to be unsafe in the sense that they can easily be misused and can lead to crashes or buffer overflow attacks when used with user-controlled inputs. Some are also not mentioned because they don't appear in the codebase.

- `gets` - the "insecure by default" `gets` is not used in the codebase, its more safe counterpart `fgets` is, however.
- `strcpy` - there are 30 occurrences of `strcpy`, which could be replaced with safer `strncpy` or even `strlcpy` which don't assume infinitely long string and can guarantee null termination in some cases.
- `sprintf` - there are 21 occurrences of `sprintf` that could be replaced with `snprintf`, which don't assume infinitely long string.
- `sscanf` - there are 45 occurrences of `sscanf`, mostly to parse structured files in `proafs` that cannot be directly manipulated by users.
- `realloc` - the interface of `realloc` can lead to misuses. `realloc` is called 22 times in the codebase, irregularly, sometimes checking the return value properly and sometimes without. Even if this issue could be theoretical because memory allocation failing can be the symptom of greater issues, it's good practice to properly check the return value.
- `access` - there are 4 occurrences of `access`. It's typically used to know if the effective user has the right to access a file before opening it but present security issues because of potential TOCTOU vulnerability it creates. However, when searching we found two usages in `libscap`, mostly on files in `proafs` that cannot be modified, and two usages in `libinsp`, mostly to checks files, all seeming non-problematic.

- **atoi** - there are 7 occurrences of `atoi`, it seems that `atof`, `atol` or `atoll` are not used in the project. The function is used mostly in example and test code but also in `libscap/scap_procs.c`, in `engine/bpf/scap_bpf.c` and twice in `libsinsp/container_engine/docker/async_source.cpp`. The issue is that `atoi` does not detect errors and just returns 0 for error cases. This function can be replaced with `strtol` or `sscanf` which can handle the error cases properly.
- **realpath** - there are 2 occurrences of `realpath` in `chisel/chisel_utils.cpp`, that can be vulnerable to directory traversal attacks, symlink attacks or resource exhaustion. In those specific cases, `realpath` is called with the second argument as the `nullptr` so the function allocated a buffer automatically with a size up to `PATH_MAX` or return an error.

Most of these functions are not inherently insecure (except maybe for `gets`), but their usage can be problematic in some situations. Some assume that strings can be infinitely long where destination buffers are very finite, others have their return value typically badly handled or can create race conditions. Some have safer alternatives, that are not inherently secure but put an emphasis on what should be correctly taken care of.

### 6.2.3 Third-party dependencies version

Falco being built with various dependencies, we checked for CVE on those libraries, especially those potentially being exposed to the end user. For example, a minimalistic HTTP server based on `cpp-httplib`<sup>9</sup> is embedded in Falco userland process, which just answers to `GET /healthz` with `{"status": "ok"}`, enabling some external monitoring.

If those dependencies are not all up to date, they are however regularly updated and none of the CVEs were applicable to the context of use in Falco.

### 6.2.4 Conclusion

Most of the time dedicated to the manual review of the code was spent on `libscap` because it's a C component that interacts with the system implementation details and interfaces using many syscalls and libc functions. Globally the code quality of Falco is good, but as we saw during our research on specific pain points, the quality can sometimes be heterogeneous (similar syscalls or libc functions called in different ways), which is not surprising in popular open source project.

---

<sup>9</sup><https://github.com/yhirose/cpp-httplib>

## 7 Dynamic analysis

One of the main focuses of this audit was to build fuzzers for security-relevant areas of Falco. The threat model built during this audit was helpful to define which area should be investigated. Although the attack surface considered was mostly the one a random unprivileged user on the system has to interact with Falco, some early investigations were done on components *a priori* only available to super users.

Indeed, a first fuzzer was built for the rules parser because it was quite natural to fuzz and a good first project for Quarkslab's engineers to get familiar with the build system.

### 7.1 Fuzzing the rules parser

First, the diagram on the Figure 7.1 was used to visualize the attack surface of Falco. Please note that this figure is a bit dated at the moment, because of the new plugin mechanism that can handle Kubernetes audit events as any other plugin.

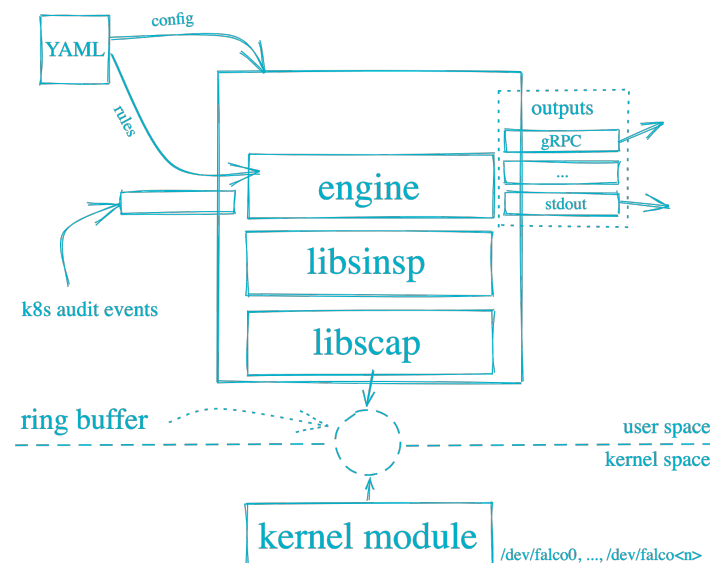


Figure 7.1: Falco architecture from the project documentation

From the userland, there are two entries to the program in YAML, one is the configuration, which should be a simple parsing, and another the rules Falco should use to filter the events. Writing text parsers in C/C++ is notoriously error-prone so it was a good first target for fuzzing although it's not a priority considering the fact that these files should be accessible only by superusers.

Falco maintainers did a rewrite in C++ of their rule parser recently, it was previously written in Lua. This work was done in a PR untitled *refactor(libsinsp): renovate the filter grammar, parser, and compiler*<sup>1</sup> with a proposed grammar in EBNF syntax.

<sup>1</sup><https://github.com/falcosecurity/libs/pull/217>

## Harness

The interface of the parser is ideal to implement fuzzing. A parser object has to be initialized with a string as an input and then `parse()` needs to be called. It's possible to write a basic libfuzzer harness like the following piece of code, directly inspired by test files that use a similar code.

```
#include <filter/parser.h>
#include <string.h>

extern "C" int LLVMFuzzerTestOneInput(uint8_t* data, size_t size)
{
    char* in = (char*)malloc(size + 1);
    memcpy(in, data, size);
    in[size] = '\0';
    libsinsp::filter::parser parser(in);
    try
    {
        parser.parse();
    }
    catch(std::runtime_error& e)
    {
        // do not crash on handled exceptions, ignore.
    }
    free(in);
    return 0;
}
```

Listing 1: Harness for fuzzing the rules parser

To build this harness, the code was put in a dedicated file in the `libsinsp/examples` folder, modifying the local `CMakeLists.txt` adding the following lines.

```
add_executable(fuzz-example
    fuzz.cpp
)
target_link_libraries(fuzz-example
    sinsp
)
```

And adding the following lines at the top `CMakeLists.txt` of `libs` to enable `ASAN`.

```
add_compile_options(-fsanitize=address)
add_link_options(-fsanitize=address)
```

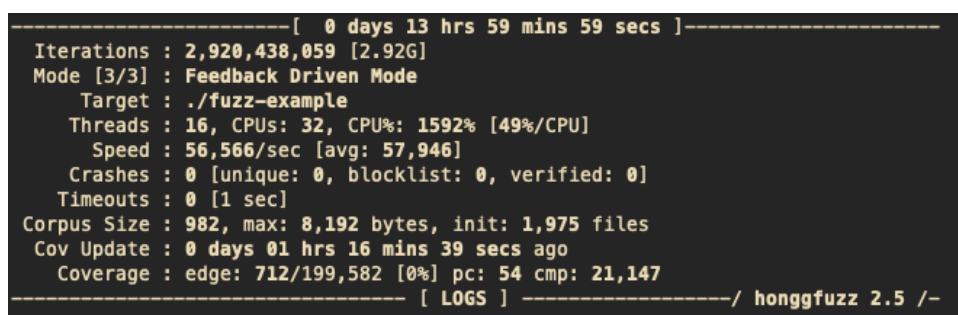
## Corpus

To generate a corpus of rules quickly, test case strings were gathered very quickly with a command similar to the following ones. This is not ideal since it will generate wrong inputs that will be discarded by the fuzzer quickly. Lines were also extracted in a crude way from the default rules files in YAML.

```
grep -v EXPECT filter*.ut.cpp | grep -e '".*"' -o | cut -d "\"" -f 2 | split -l 1  
yq '[][.condition' ../falco_rules.yaml | uniq | split -l 1
```

## Results

The harness on Listing 1 was used for persistent mode fuzzing with Honggfuzz for almost 14 hours, see Figure 7.2, it did not run into any crashes during the whole fuzzing session.



```
-----[ 0 days 13 hrs 59 mins 59 secs ]-----  
Iterations : 2,920,438,059 [2.92G]  
Mode [3/3] : Feedback Driven Mode  
  Target : ./fuzz-example  
  Threads : 16, CPUs: 32, CPU%: 1592% [49%/CPU]  
  Speed : 56,566/sec [avg: 57,946]  
  Crashes : 0 [unique: 0, blocklist: 0, verified: 0]  
  Timeouts : 0 [1 sec]  
Corpus Size : 982, max: 8,192 bytes, init: 1,975 files  
Cov Update : 0 days 01 hrs 16 mins 39 secs ago  
Coverage : edge: 712/199,582 [0%] pc: 54 cmp: 21,147  
----- [ LOGS ] -----/ honggfuzz 2.5 /-
```

Figure 7.2: Results of fuzzing in persistent mode with Honggfuzz for a night

Similarly, AFL++ was run over the harness in the persistent mode for a shorter amount of time to see if it could spot other issues but it did not during our experimentations. Please note that to do the same with AFL++ you need to add some boilerplate code that you can find in Appendix E.

american fuzzy lop ++4.02c {main} (...nsp/examples/fuzz-rules-parser) [fast]		
process timing		overall results
run time : 0 days, 7 hrs, 36 min, 59 sec		cycles done : 249
last new find : 0 days, 1 hrs, 33 min, 16 sec		corpus count : 1995
last saved crash : none seen yet		saved crashes : 0
last saved hang : none seen yet		saved hangs : 0
cycle progress	map coverage	
now processing : 1174.146 (58.8%)	map density : 0.09% / 0.58%	
runs timed out : 0 (0.00%)	count coverage : 6.75 bits/tuple	
stage progress	findings in depth	
now trying : havoc	favorable items : 103 (5.16%)	
stage execs : 2691/3532 (76.19%)	new edges on : 72 (3.61%)	
total execs : 251M	total crashes : 0 (0 saved)	
exec speed : 10.5k/sec	total tmouts : 5913 (0 saved)	
fuzzing strategy yields	item geometry	
bit flips : disabled (default, enable with -D)	levels : 2	
byte flips : disabled (default, enable with -D)	pending : 0	
arithmetics : disabled (default, enable with -D)	pend fav : 0	
known ints : disabled (default, enable with -D)	own finds : 132	
dictionary : n/a	imported : 762	
havoc/splice : 3/68.5M, 3/128M	stability : 99.61%	
py/custom/rq : unused, unused, unused, unused		
trim/eff : disabled, disabled		
		[cpu000: 12%]

Figure 7.3: Results of fuzzing in persistent mode with AFL++ for 8 hours with 4 instances

This does not mean that the parser is bug-free, but at least no bug would produce a crash was found during the fuzzing session, and the harness was compiled with ASAN to produce explicit crashes on memory errors. The coverage produced by the fuzzing session was checked manually and it seemed like the fuzzer went over most of the codebase of the parser successfully.

## 7.2 Fuzzing the event processor

Having the goal of building fuzzers for the userspace part of Falco, the main target was the `sinsp_parser::process_event` function. This function takes a pointer to an event as input and can be seen as a router to the appropriate parser for the encoded event. For example, if the event happens to be a `PPME_SYSCALL_PIPE_X`, which is the exit Falco event for the pipe(2) syscall, this function will redirect the execution to `parse_pipe_exit`. The parser will then retrieve the event parameters in the following manner.

```
// [...]
parinfo = evt->get_param(0);
retval = *(int64_t *)parinfo->m_val;
ASSERT(parinfo->m_len == sizeof(int64_t));
// [...]
parinfo = evt->get_param(1);
ASSERT(parinfo->m_len == sizeof(int64_t));
fd1 = *(int64_t *)parinfo->m_val;
// [...]
```

Fuzzing such big “router” would be interesting since the implementation is tedious and verifying

there is no error in this code might be even harder. The issue is that forming a valid, but random, events (i.e., doing structured fuzzing) for so many different events is difficult.

On top of that, another difficulty is that some parsers do not actually parse all the buffer containing the event's arguments. These arguments are later "lazy loaded" when filtering is happening on these specific events fields to reduce the overall load. That's why it might not be sufficient, in order to reach the code that would trigger an issue, to only ingest the events with the parsers.

That's an illustration of a case in which fuzzing could be more complex and more time consuming than doing a manual review, especially because the target is parsing a highly structured format and might not be built with fuzzing in mind.

See in the following sections multiple ideas investigated by Quarkslab's engineers. The order does not reflect the chronological reality. Usage of libprotobuf-mutator was investigated in Section 7.2.1 in order to do structured fuzzing targeted directly at particular event parsers. Then usage of syzkaller in Section 7.2.2 was explored, to fuzz the syscalls on a system with an instrumented version of Falco running and getting coverage in the kernel engine. And then, fuzzing the scap file format in Section 7.2.3 was looked into for unstructured fuzzing, it mostly uncovered issues in the file format handling.

## 7.2.1 Using libprotobuf-mutator

For structured fuzzing in C/C++, libprotobuf-mutator<sup>2</sup> is a library to randomly mutate `protobuf` message. It could be used together with guided fuzzing engines, such as libFuzzer. The overall idea is to define a structure in `protobuf` format and libprotobuf-mutator will mutate the fields with type-valid values.

Falco maintainers pointed us to the `driver/event_table.c` files in `libs` that defines all the events supported by Falco. See the following listing for an extract of that structure.

```
// [...]
/* PPME_SYSCALL_CLOSE_E */{"close", EC_IO_OTHER | EC_SYSCALL, EF_DESTROYED_FD |
→ EF_USES_FD | EF_MODIFIES_STATE, 1, {"fd", PT_FD, PF_DEC} },
/* PPME_SYSCALL_CLOSE_X */{"close", EC_IO_OTHER | EC_SYSCALL, EF_DESTROYED_FD |
→ EF_USES_FD | EF_MODIFIES_STATE, 1, {"res", PT_ERRNO, PF_DEC} },
/* PPME_SYSCALL_READ_E */{"read", EC_IO_READ | EC_SYSCALL, EF_USES_FD |
→ EF_READS_FROM_FD, 2, {"fd", PT_FD, PF_DEC}, {"size", PT_UINT32, PF_DEC} },
/* PPME_SYSCALL_READ_X */{"read", EC_IO_READ | EC_SYSCALL, EF_USES_FD |
→ EF_READS_FROM_FD, 2, {"res", PT_ERRNO, PF_DEC}, {"data", PT_BYTEBUF, PF_NA} }
→ },
// [...]
```

In addition of this file, a documentation page is available on the Falco website<sup>3</sup>. More importantly, this page displays which syscalls are monitored by default, for performance reasons, without specifying the `-A` flag that enables everything. Thus, the default list, on top of the syscalls that

<sup>2</sup><https://github.com/google/libprotobuf-mutator>

<sup>3</sup><https://falco.org/docs/rules/supported-events/>



are typically available in Kubernetes containers environment<sup>4</sup>, can be used to make a priority list of syscalls parsers that should be fuzzed with structured fuzzing.

## Compiling libprotobuf-mutator

One of the main difficulties was to link a binary with libprotobuf-mutator which was using a different version of protobuf than Falco. The solution to link without bothering with missing symbols or multiple definitions is to align the protobuf version used by libprotobuf-mutator with the one used by Falco. Fortunately, libprotobuf-mutator v1.1 compiles with an older protobuf version.

```
git clone https://github.com/google/libprotobuf-mutator.git
cd libprotobuf-mutator
git checkout v1.1
```

In the file `cmake/external/protobuf.cmake`, modify line 66 with `GIT_TAG v3.17.3`, which is the version used by Falco at the moment of the audit (see 3.2).

```
mkdir build
cd build
cmake .. -GNinja -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++
↪ -DCMAKE_BUILD_TYPE=Debug -DLIB_PROTO_MUTATOR_DOWNLOAD_PROTOBUF=ON
ninja
```

At this point we successfully compiled a version of libprotobuf-mutator compatible with the protobuf version Falco use.

## Writing the proto file and the harness

The example here will use the events associated with the `open(2)` syscall.

First, a proto file must be written containing the value types needed for filling the arguments of the event we want to generate. Unfortunately, protobuf cannot take expression as input for enum values, so all bitwise flag values must be written directly, instead of using the practical `1 << N` notation. In our situation, the proto file will look like the abbreviated following version (see the full proto definition in the listing in Appendix F).

```
syntax = "proto3";

package sys_open;

message event_args {
    uint64 fd = 1;
    string fspath = 2;
    repeated FILE_FLAGS flags32 = 3;
```

---

<sup>4</sup>Restricted seccomp profiles are still not applied by default in Kubernetes, but Linux capabilities are already restricted, thus reducing the syscalls available even running as root.

```

FILE_MODE mode = 4;
uint32 dev = 5;
uint64 ino = 6;
}

// [...]

```

Then generate the C++ stubs with the protobuf compiler, make sure to use the one compiled along libprotobuf-mutator, it's available in the project build folder libprotobuf-mutator/build/external.protobuf/bin/protoc.

```

protoc --version # this should output "libprotoc 3.17.3"
protoc --cpp_out=. sys_open.proto

```

This command should generate two files, sys\_open.pb.cc and sys\_open.pb. Now, a piece of code must be written, using the generated input to fuzz our parsers: the harness. Thanks to the help of a Falco maintainer, we modified a version of the `sinsp_with_test_input.h`'s class `sinsp_with_test_input` to add our event on an inspector. The resulting harness is similar to the following piece of code.

```

1  #include <parsers.h>
2
3  #include "sinsp_fuzz.h"
4  #include "sys_open.pb.h"
5
6  #include "libprotobuf-mutator/src/libfuzzer/libfuzzer_macro.h" // defines DEFINE_PROTO_FUZZER
7
8  int32_t fuzz_bitwise_flag = 0;
9
10 static protobuf_mutator::libfuzzer::PostProcessorRegistration<sys_open::event_args> reg = {
11     [] (sys_open::event_args* msg, unsigned int seed)
12     {
13         fuzz_bitwise_flag = 0;
14         for(int32_t flag : msg->flags32())
15         {
16             fuzz_bitwise_flag |= flag;
17         }
18     }
19 };
20 DEFINE_PROTO_FUZZER(const sys_open::event_args& msg)
21 {
22     sinsp_fuzz s;
23
24     s.SetUp();
25     s.add_default_init_thread();
26     s.open_inspector();
27
28     auto evt = s.add_event_advance_ts(s.increasing_ts(), 1, PPME_SYSCALL_OPEN_E, 3,
↪ msg.fspath().c_str(), fuzz_bitwise_flag, msg.mode());
29     evt->load_params();
30
31     evt = s.add_event_advance_ts(s.increasing_ts(), 1, PPME_SYSCALL_OPEN_X, 6, msg.fd(),
↪ msg.fspath().c_str(), fuzz_bitwise_flag, msg.mode(), msg.dev(), msg.ino());
32     evt->load_params();
33
34     s.TearDown();
35 }

```

The `DEFINE_PROTO_FUZZER` macro and the post processor are part of the libFuzzer integration of libprotobuf-mutator, see the section in the README for more details<sup>5</sup>. To detail the example here, on the line 11 of harness, a lambda is defined to post process the generated protobuf message in order to create a file flags argument with one or more bits set one. From line 20 to line 35 is the part of the code that will be replayed by libFuzzer.

At this point we have a protobuf definition that is used in a libFuzzer harness and the compiled protobuf-mutator libraries.

## Compile with libFuzzer

To compile with LLVM libFuzzer, you mostly need to pass the `-fsanitize=fuzzer` flag to clang. Please note that you *must* compile with clang and not gcc. To compile everything that was presented in the last sections, use the following lines in the `CMakeLists.txt` of `libsinsp/test`, fixing the relative paths, or create a new one under `libsinsp/test/fuzz` with the following content.

```
add_compile_options(-fsanitize=fuzzer,address)
add_link_options(-fsanitize=fuzzer,address)

add_executable(fuzz_sys_open
    sys_open.pb.cc
    ../test_utils.cpp
    fuzz_sys_open.cpp
)

# Set this path to your libprotobuf-mutator installation directory
set(LIBPROTOBUF_MUTATOR_BASEDIR "/home/mahe/falco-security/libprotobuf-mutator/")

target_include_directories(fuzz_sys_open PRIVATE
    "${LIBPROTOBUF_MUTATOR_BASEDIR}"
    # no need for the following, already included with others Falco dependencies
    # "/home/mahe/falco-security/libprotobuf-mutator/build/external.protobuf/include")
)

target_link_libraries(fuzz_sys_open
    "${LIBPROTOBUF_MUTATOR_BASEDIR}/build/src/libfuzzer/libprotobuf-mutator-libfuzzer.a"
    "${LIBPROTOBUF_MUTATOR_BASEDIR}/build/src/libprotobuf-mutator.a"
    # no need for the following, already included with others Falco dependencies
    # "/home/mahe/falco-security/libprotobuf-mutator/build-test/external.protobuf/lib/libprotobufd.a"
    sinsp
)
```

For ASAN to work on the whole code, the following lines must be added at the beginning of the root `CMakeLists.txt`.

```
add_compile_options(-fsanitize=address)
add_link_options(-fsanitize=address)
```

To reproduce exactly this setup, we generated a diff that can be applied to falco libs repo from version described in Table 3.2, tag 0.9.0, see the file named `sys_open_fuzz.diff`. Go to the root of the git libs repository and type the following commands.

---

<sup>5</sup><https://github.com/google/libprotobuf-mutator#integrating-with-libfuzzer>

```
git checkout 0.9.0
git apply sys_open_fuzz.diff
mkdir build
cd build
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ ..
make # try to use -j <nthreads>
```

## Fuzzing with LLVM libFuzzer

Unlike Honggfuzz or AFL++, LLVM libFuzzer adds the fuzzer directly to the compiled binary. Thus to start the fuzzing session, just execute the binary and use libFuzzer's flags. You can find the documentation on the LLVM website<sup>6</sup> to learn about the available flags and understand the output format. It's possible to run the fuzzer with multiple workers and monitor the logs using `tail` on the multiple generated files. If libFuzzer finds a crash, it will stop and generate a file containing the Protobuf message in text format<sup>7</sup>. It's possible to provide an existing corpus with the same format.

Starting the fuzzing, the output for `fuzz_sys_open` should be similar to this.

```
INFO: found LLVMFuzzerCustomMutator (0x934350). Disabling -len_control by default.
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1989839396
INFO: Loaded 1 modules (830 inline 8-bit counters): 830 [0x20899d4, 0x2089d12),
INFO: Loaded 1 PC tables (830 PCs): 830 [0x1d47f60, 0x1d4b340),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 120 ft: 121 corp: 1/1b exec/s: 0 rss: 51Mb
      NEW_FUNC[1/3]: 0x92e630 in sys_open::event_args::~~event_args()
      ↪ /home/mahe/falco-security/libs/userspace/libsinsp/test/fuzz/sys_open.pb.cc:193
      NEW_FUNC[2/3]: 0x931a00 in void google::protobuf::internal::InternalMetadata::DeleteOutOfLin
      ↪ eHelper<google::protobuf::UnknownFieldSet>()
      ↪ /home/mahe/falco-security/libs/build/protobuf-prefix/src/protobuf/target/include/google/
      ↪ protobuf/metadata_lite.h:190
#3      NEW      cov: 131 ft: 136 corp: 2/11b lim: 4096 exec/s: 0 rss: 53Mb L: 10/10 MS: 7
      ↪ EraseBytes-EraseBytes-ShuffleBytes-CMP-CrossOver-ChangeBit-Custom- DE:
      ↪ "\x00\x00\x00\x00\x00\x00\x00\x00"
#5      NEW      cov: 133 ft: 138 corp: 3/47b lim: 4096 exec/s: 0 rss: 54Mb L: 36/36 MS: 3
      ↪ CustomCrossOver-InsertByte-Custom-
#41     REDUCE cov: 133 ft: 138 corp: 3/44b lim: 4096 exec/s: 0 rss: 67Mb L: 33/33 MS: 2
      ↪ ChangeByte-Custom-
#142    REDUCE cov: 133 ft: 138 corp: 3/23b lim: 4096 exec/s: 0 rss: 104Mb L: 12/12 MS: 1 Custom-
#278    NEW      cov: 134 ft: 139 corp: 4/50b lim: 4096 exec/s: 0 rss: 153Mb L: 27/27 MS: 2
      ↪ CMP-Custom- DE: "proto2"-
#380    REDUCE cov: 134 ft: 139 corp: 4/40b lim: 4096 exec/s: 0 rss: 189Mb L: 17/17 MS: 4
      ↪ ShuffleBytes-Custom-EraseBytes-Custom-
#811    REDUCE cov: 134 ft: 139 corp: 4/37b lim: 4096 exec/s: 811 rss: 341Mb L: 7/17 MS: 2
      ↪ CopyPart-Custom-
#4096   pulse cov: 134 ft: 139 corp: 4/37b lim: 4096 exec/s: 2048 rss: 393Mb
#4118   REDUCE cov: 134 ft: 139 corp: 4/32b lim: 4096 exec/s: 2059 rss: 393Mb L: 7/17 MS: 5
      ↪ CustomCrossOver-ShuffleBytes-CopyPart-CopyPart-Custom-
#6326   REDUCE cov: 135 ft: 140 corp: 5/78b lim: 4096 exec/s: 1581 rss: 394Mb L: 46/46 MS: 5
      ↪ PersAutoDict-CustomCrossOver-Custom-CMP-Custom- DE: "\x00\x00\x00\x00\x00\x00\x00\x00"-(NULL)"-
#6457   REDUCE cov: 135 ft: 140 corp: 5/56b lim: 4096 exec/s: 1614 rss: 394Mb L: 24/24 MS: 1 Custom-
```

<sup>6</sup><https://llvm.org/docs/LibFuzzer.html>

<sup>7</sup><https://developers.google.com/protocol-buffers/docs/text-format-spec>

```
#6918 REDUCE cov: 135 ft: 140 corp: 5/49b lim: 4096 exec/s: 1729 rss: 394Mb L: 17/17 MS: 2
↳ CopyPart-Custom-
#8192 pulse cov: 135 ft: 140 corp: 5/49b lim: 4096 exec/s: 1638 rss: 394Mb
#8564 REDUCE cov: 135 ft: 140 corp: 5/48b lim: 4096 exec/s: 1712 rss: 394Mb L: 6/17 MS: 2
↳ CMP-Custom- DE: "\x01\x00\x00\x00\x00\x00\x00\x00"-
#16384 pulse cov: 135 ft: 140 corp: 5/48b lim: 4096 exec/s: 1820 rss: 395Mb
```

## Fuzzing the SYS\_CLONE\_20\_X event handler

After trying on `sys_open`, we selected a syscall that had a more complex handler on the userspace side. `sys_clone_20_x` was chosen, notably because it has a large number of arguments: twenty, and some could be manipulated by a simple user on the system. So we decided to mutate only the fields that could be influenced by an attacker, i.e. the exe path, args, comm, and flags, meaning that in the end mostly string buffers were fuzzed. In the end, the fuzzing was pretty restricted and concerned mostly string buffers, so it was very limited and could have been checked by a manual analysis. Unsurprisingly, in the end, fuzzing the handler with the harness did not triggered crashes.

However, the same method can be used to fuzz the entirety of the arguments, to find bugs that may not be security relevant. Nevertheless, it confirms that writing security relevant harness can be more time consuming than reviewing the code manually, considering as well that it can be harder to detect some issues by just reading the source and manual review should in addition of a dynamic analysis.

Attached to the report, you will find a git diff named, `sys_clone_fuzz.diff` containing both the changes for the open and clone syscalls from the tag 0.9.0 or more precisely, the version specified in Table 3.2.

## Conclusion

For structured fuzzing, `libprotobuf-mutator` is appropriate in order to create random structures, but it's a bit complex and time-consuming to setup. The drawback is mainly that multiple highly targeted harnesses must be written in order to fuzz a significant amount of code, and thus find bugs. Manual review of the function could be more time efficient in certain situations. On top of that, by creating the proto definition and the harness, we need to have a very precise knowledge on what could be the possible inputs and not make potential false assumptions. It should be combined with random fuzzing that make fewer assumptions in order to find potential issues on the format itself and not just on the range of possible values.

Because writing and maintaining a large quantity of custom harness could be burdensome, we can even project that the proto definition and C++ harness could be generated from the `event_table.c` automatically. But again creating such program could exceed the cost in time of a manual review.

For the conclusion of the specific examples, on `sys_open` and `sys_clone_20_x`, fuzzing sessions could not trigger crashes with the harnesses presented in this report.

## 7.2.2 Using syzkaller

For fuzzing related to syscalls, the syzkaller<sup>8</sup> project can be interesting. It's technically an unsupervised coverage-guided kernel fuzzer but also a group of binaries and libraries that can be used separately to generate valid syscall programs and mutate them. The advantage of using this project as a base for creating valid event for Falco is that multiple people already took the time to describe syscalls (their arguments, return values) in a specific domain language<sup>9</sup>.

There are two ways to see how syzkaller could be used on Falco.

1. The whole syzkaller stack could be used as intended, fuzzing the entire system on which Falco is installed and running. The coverage will come from the system code and not from Falco's so it can be seen as random fuzzing without guidance.
2. The library in charge of mutating programs could be used to form valid Falco events (at least enter events). It can be used similarly as in the syz-mutate tool<sup>10</sup>. It's still random fuzzing and more should be written to actually create a fuzzer from this idea: interfacing with the C library containing the process\_event function, catching signals as a fuzzer would do, etc.

For the second point, in order to get an idea of what syzkaller is capable, you can clone the repository, compile the binaries and use syz-mutate to generate valid programs. Here is an extract mutating a lot of open(2) syscalls using ./syz-mutate --len 2000 --enable open.

```
[...]
open$dir(&(0x7f0000146fc0)='./file64\x00', 0x8840, 0x1d1)
open$dir(&(0x7f0000147000)='./file4aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
↪ aaaaa\x00', 0x193000,
↪ 0x40)
open(&(0x7f0000147240)='./file60\x00', 0x191180, 0x44)
open(&(0x7f0000147280)='./file7\x00', 0x40100, 0x14)
open$dir(&(0x7f00001472c0)='./file36\x00', 0x420002, 0x32)
open$dir(&(0x7f0000147300)='./file1/file1/file0/file1\x00', 0x240040, 0x20)
```

## Fuzzing the Falco kernel module with syzkaller

For the first point, we mainly followed the syzkaller documentation<sup>11</sup> to setup a QEMU VM on a Ubuntu host. Some small modifications are required to integrate Falco.

<sup>8</sup><https://github.com/google/syzkaller>

<sup>9</sup>[https://github.com/google/syzkaller/blob/master/docs/syscall\\_descriptions.md](https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md)

<sup>10</sup><https://github.com/google/syzkaller/blob/master/tools/syz-mutate/mutate.go>

<sup>11</sup>[https://github.com/google/syzkaller/blob/master/docs/linux/setup\\_ubuntu-host\\_qemu-vm\\_x86-64-kernel.md](https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md)

- First, in the kernel config parameters, in addition to the required config options for syzkaller, we also added `CONFIG_KCOV_ENABLE_COMPARISONS=y`. This is not required but gives better coverage results based on our experiments. A second point, which is required to enable coverage of dynamic loaded module and so to get Falco driver coverage, is to disable KASLR by adding `CONFIG_RANDOMIZE_BASE`.
- Once the kernel was built, we built Falco's driver using the same kernel source code, in order to get instrumentation into the module. To do this, there is just a small modification to do in <https://github.com/falcosecurity/libs/blob/master/driver/Makefile.in> which defines `KERNELDIR` as:

```
KERNELDIR      ?= /lib/modules/$(shell uname -r)/build
```

We just have to fix this with the folder used to build our kernel.

- Then we need to build the Falco userland process inside the target image. To do this, we started the image with a temporary second drive (Falco is quite huge to build), cloned the git repository into it, added the required packages to build everything and then happily got the expected binary.
- Finally, we added the default `falco.yaml` and `falco_rules.yaml` configuration files and configured Falco to start as a systemd service, based on the scripts from Falco repository.

At this point, restart the VM and quickly check that Falco is ready:

```
Linux syzkaller 5.14.0 #4 SMP Tue Nov 22 07:04:02 EST 2022 x86_64
...
root@syzkaller:~# lsmod
Module                Size  Used by
falco                 1081344  2
root@syzkaller:~# ps aux | grep falco
root      207 19.4  1.4 268792 24928 ?        Rsl  22:03   0:17 /usr/bin/falco
↪ --pidfile=/var/run/falco.pid
root      282  8.0  0.0 11116   928 pts/0    S+   22:04   0:00 grep falco
root@syzkaller:~# touch /root/whatever
root@syzkaller:~# tail -1 /var/log/user.log
Nov 23 22:05:16 syzkaller falco: 22:05:16.545542579: Error File below / or /root
↪ opened for writing (user=root user_loginuid=0 command=touch /root/whatever
↪ pid=283 parent=bash file=/root/whatever program=touch container_id=host
↪ image=<NA>)
```

We can now start syzkaller. A manager configuration file is required to specify various parameters, like kernel image, 'sshkey' to log into the vm, number of VM, etc.

But in addition to the documentation sample, we also need to specify two parameters, `module_obj` and `kernel_subsystem` to help syzkaller retrieve information from Falco's coverage. The global configuration we used is similar to:

```
{
  "target": "linux/amd64",
```

```

    "http": "0.0.0.0:8080",
    "workdir": "/home/jdoe/syzkaller/syzkaller/workdir",
    "kernel_obj": "/home/jdoe/syzkaller/linux",
    "module_obj": ["/home/jdoe/syzkaller/falco/build/driver"],
    "kernel_subsystem": [ { "name": "falco", "path":
↪ ["/home/jdoe/syzkaller/falco/build/driver/src"]}],
    "image": "/home/jdoe/syzkaller/image/stretch.img",
    "sshkey": "/home/jdoe/syzkaller/image/stretch.id_rsa",
    "syzkaller": "/home/jdoe/syzkaller/syzkaller",
    "procs": 4,
    "type": "qemu",
    "vm": {
        "count": 2,
        "kernel": "/home/jdoe/syzkaller/linux/arch/x86/boot/bzImage",
        "cpu": 2,
        "mem": 2048,
    }
}

```

Running syzkaller is then as easy as running

```
./bin/syz-manager -config=./stretch.cfg
```

Syzkaller needs to start each fuzzing VM with the `--snapshot` option, which means that after rebooting, any filesystem modification is lost. But this doesn't prevent us to ssh into a working VM and take a look at `/var/log/user.log` to check if Falco has caught some events:

```

root@syzkaller:~# grep falco /var/log/user.log
...
Nov 23 22:37:20 syzkaller falco: Opening capture with Kernel module
Nov 23 22:37:25 syzkaller falco: 22:37:25.448433801: Error File below / or /root
↪ opened for writing (user=root user_loginuid=0 command=scp -t /syz-fuzzer
↪ pid=362 parent=sshd file=/syz-fuzzer program=scp container_id=host image=<NA>)
Nov 23 22:37:26 syzkaller falco: 22:37:26.648266483: Error File below / or /root
↪ opened for writing (user=root user_loginuid=0 command=scp -t /syz-executor
↪ pid=370 parent=sshd file=/syz-executor program=scp container_id=host
↪ image=<NA>)
Nov 23 22:37:39 syzkaller falco: 22:37:39.810731946: Error File created below
↪ /dev by untrusted program (user=root user_loginuid=0 command=syz-executor.0
↪ exec pid=2188 file=/dev/sr0 container_id=host image=<NA>)
Nov 23 22:37:39 syzkaller falco: 22:37:39.832322826: Error File created below
↪ /dev by untrusted program (user=root user_loginuid=0 command=syz-executor.3
↪ exec pid=2222 file=/dev/nvram container_id=host image=<NA>)
...

```

The first two events are just showing some internals from syzkaller, the syz-manager on the host is pushing its binaries into the vm with scp, and then the two following events show that those binaries start playing with devices.



## Gathering information from the userland Falco process

At this point, an attentive reader will have noted that we are just able to get information from the Falco driver; there is no coverage for the Falco userland process, nor even notification if this process crashes.

If getting coverage for a userland process would require more work to adapt Syzkaller, which was not the aim of this assessment, we nevertheless thought that it would not be so complicated to detect crashes. Indeed, when a userland process crashes, it is reported in the local syslog. We've already said that syzkaller starts the VM with the `--snapshot` option, so to keep track of those potential crashes through reboots, we just configured the VM syslog to forward interesting events to the host syslog.

This required very few modifications of our image, and a small fix in the way syzkaller is starting the QEMU VM, due to the `--restrict` option used by syzkaller to isolate the guest and the host network, obviously to prevent bad things to happen to the host. QEMU provides an option to expose a listening socket on the host side inside the guest, thanks to the `guestfwd` options. Be careful, the port must be bound *before* starting QEMU.

- So the code configuring the network stack of QEMU VM in `/vm/qemu/qemu.go` has been fixed by:

```
diff --git a/vm/qemu/qemu.go b/vm/qemu/qemu.go
index d9933fa05..eac312f66 100644
--- a/vm/qemu/qemu.go
+++ b/vm/qemu/qemu.go
@@ -435,7 +435,8 @@ func (inst *instance) boot() error {
    args = append(args, splitArgs(inst.cfg.QemuArgs, templateDir,
    ↪ inst.index)...)
    args = append(args,
    ↪ "-device", inst.cfg.NetDev+",netdev=net0",
    - "-netdev",
    ↪ fmt.Sprintf("user,id=net0,restrict=on,hostfwd=tcp:127.0.0.1:%v-:22",
    ↪ inst.port))
    + "-netdev",
    ↪ fmt.Sprintf("user,id=net0,restrict=on,hostfwd=tcp:127.0.0.1:%v-:22"+
    + ",guestfwd=tcp:10.0.2.1:514-tcp:127.0.0.1:514",
    ↪ inst.port))
    ↪ if inst.image == "9p" {
```

- In the VM image, in `/etc/rsyslog.conf`, two new actions were added to forward kernel and user events to the exposed remote TCP service: `kern.* @10.0.2.1:514` and `user.* @10.0.2.1:514`. Indeed, the only events received with the user facility are Falco events, which enable us to also have a log on the host of all triggered events.
- The syslog startup must be postponed after the network setup, by adding to the `[Unit]` part of `/etc/systemd/system/syslog.service`

```
After=network.target auditd.service
```

- And to try to get some information in case of a crash, at least a stack trace, `kernel.core_pattern` was configured with a custom script which just calls `coredumpctl info $1 | logger -p user.notice` after having executed `systemd-coredump`.
- Finally, Falco was recompiled with [ASAN](#), in order to get potential memory error under heavy syscalls load.

## Results

After a 48 hours run of syzkaller, Falco kernel driver did not cause any crash. See Table 7.1 for more information. Nevertheless, other crashes were found, like a general protection fault in `scsi_queue_rq`. However those bugs have already been caught and reported by syzbot<sup>12</sup>.

uptime	32h345s	
fuzzing	49h41m0s	
corpus	10540	
signal	130782	
coverage	86763	
syscalls	2125	
crash types	10	(0/hour)
crashes	28	(0/hour)
exec total	6557347	(55/sec)

Table 7.1: Syzkaller fuzzing campaign stats results

See Table 7.2 for coverage information on the Falco kernel module.

filename	covergae	basic block
<code>falco/build/driver/src</code>	57% (81%)	of 3660(2574)
<code>main.c</code>	16% (52%)	of 568(175)
<code>ppm_cputime.c</code>		— of 1
<code>ppm_events.c</code>	40% (46%)	of 385(337)
<code>ppm_fillers.c</code>	68% (89%)	of 2705(2062)
<code>ppm_flag_helpers.h</code>	100% (0%)	of 1(0)

Table 7.2: Coverage on Falco kernel module code

Clearly, when syzkaller starts its fuzzing process inside the VM, Falco is already started and the driver loaded, so there is no reason to see coverage, for example, for various parts of `main.c`, as the one which handles driver initialization or the `file_operations` used when a userland process opens the falco driver or memory map it.

<sup>12</sup><https://syzkaller.appspot.com/upstream>

An interesting point is the coverage of `ppm_fillers.c`, which contains a hook for each syscall in order to extract its parameters, and gives a good idea of which syscall has been reached by the fuzzer. See the following Table 7.3, for a small extract of the coverage for the `f_sys_XXX_e|x` handled in this source file. The full HTML coverage report will be included with this report under the file name `syzkaller-kernel-module-coverage.html.gz`.

function name	coverage	basic block
...	...	...
<code>f_sys_accept4_e</code>	100%	of 1
<code>f_sys_accept_x</code>	83%	of 17
<code>f_sys_access_e</code>	—	of 10
<code>f_sys_bpf_x</code>	—	of 1
<code>f_sys_brk_munmap_mmap_x</code>	—	of 7
<code>f_sys_capset_x</code>	98%	of 250
<code>f_sys_chmod_x</code>	94%	of 33
<code>f_sys_connect_e</code>	80%	of 20
<code>f_sys_connect_x</code>	75%	of 16
<code>f_sys_copy_file_range_e</code>	—	of 13
<code>f_sys_copy_file_range_x</code>	—	of 7
<code>f_sys_creat_e</code>	95%	of 34
<code>f_sys_creat_x</code>	96%	of 43
...	...	...

Table 7.3: Coverage on the Falco kernel module fillers

Coverage of `ppm_events.c`, which contains various tool functions used by `ppm_fillers.c` to extract the syscall arguments, is also interesting, see the following Table 7.4.

function name	coverage	basic block
addr_to_kernel	100%	of 5
compat_parse_readv_writev_bufs	—	of 26
compute_snaplen	2%	of 150
dpi_lookahead_init	—	of 1
f_sys_autofill	85%	of 13
fd_to_socktuple	60%	of 61
pack_addr	91%	of 11
parse_readv_writev_bufs	97%	of 26
ppm_copy_from_user	100%	of 4
ppm_strncpy_from_user	100%	of 11
sock_getname	—	of 21
val_to_ring	86%	of 56

Table 7.4: Coverage of function in ppm\_events.c

Most of the missed code comes from the `compute_snaplen`, which has been manually reviewed.

Concerning the Falco userland process, once again, no crash. And yet, we can see it was in high demand thanks to the various `syscall event drop` present in the Falco notifications:

```
Dec  2 07:43:31 syzkaller falco: 07:43:24.933425157: Debug Falco internal: syscall
↪ event drop. 10551 system calls dropped in last second.
(ebpf_enabled=0 n_drops=10551 ... n_drops_buffer_total=10551 ... n_evts=14714)
```

## Conclusion

Syzkaller appeared to be an interesting option for blackbox fuzzing of Falco’s solution, enabling to challenge both the driver and userland application at a high level rate of event, and being able to trigger complex syscall chain. Falco showed no weakness in dealing with this large amount of unusual events, despite long fuzzing sessions.

A way to improve this fuzzing solution, would be to add userland process coverage to this syzkaller based solution in to better guide the fuzzer.

### 7.2.3 Using the scap file format

Falco can, in addition of the live capture from the kernel module or eBPF probes, read files in the scap file format<sup>13</sup> to process and filter the events based on the content. The sysdig tool<sup>14</sup> can capture syscalls event and generate such file in the scap format. Falco maintainers kindly

<sup>13</sup>The scap format is a flavor of the, often used for network, pcap file format. Note that Gerald Combs, the creator and maintainer of Wireshark was hired by Sysdig, the initial parent company of Falco.

<sup>14</sup><https://github.com/draios/sysdig>

directed us toward a repository, *sysflow-telemetry/sf-collector* where some specific scap files are available<sup>15</sup>.

For fuzzing directly the file format, inspiration was taken from the very helpful examples in the *libs/libinsp* folder<sup>16</sup> keeping only the part concerning the SAVEFILE engine. You can find the complete harness in the git diff file named *fuzz\_file.diff* attached to the report. The binary was compiled with *ASAN*.

The issue is that, as expected, the fuzzing mostly uncovered crashes from the file decoding part since these files input are what we could call “corrupted”, for example having a header “lying” about the content of the body. It seems that the file engine and the event processor assume that file should be valid which is reasonable. Indeed these crashes are not critical because they are only concerning a Falco instance that was ran especially for filtering and investigating about the content of the files.

On top of that, fuzzing is particularly slow (around 50/100 execution per second), which might be because the code parsing the file is actually making a lot of *read(2)* syscalls, which is not ideal in terms of performance. But for an unclear reason a lot of the execution timeout after 1s or a bit more which slows everything down (it might be related to *ASAN* but it’s not clear). In order to find out, it would be reasonable to first fix all the previous crash, which would imply to refactor some parts of the parsing more defensively.

An archive with the name *fuzz-scap-file-results.tar.gz* contains the results of a small session of fuzzing, around an hour in total. The file ending in *.fuzz* can be used as input of the harness to provoke the crash their name indicates. Usually a good idea is to build the harness without optimization, using *-O0*, not *-Og*, and debugging with the help of *gdb*, for example, to find the root of the crash. Please note that some inputs only generate crash because of *ASAN* but are silent memory bug without the sanitizer enabled. Inputs starting with *SIGVTALRM* are inputs that somehow created an execution timeout (still unclear why and maybe not the priority).

## Example of crash using malformed scap file

Here is an example to illustrate, let’s arbitrary take the smallest crash input generated for concision reason. The name of the file (see *Honggfuzz* documentation on how to read the name<sup>17</sup>), is *SIGSEGV.PC.5555560369be.STACK.c30617a76.CODE.1.ADDR.55575e149b22.INSTR.mov\_\_-\_\_(%rax),%rax.fuzz*

```
00000000: 0a0d 0d0a 1c00 0000 4d3c 2b1a 0100 0200 ffff ffff .....M<+.....
00000014: ffff ffff 1c00 0000 2102 200a c000 0000 1000 0000 .....!.....
00000028: 00e0 dbb8 fe61 7269 612d 3235 362d 6163 6dff 0f00 .....aria-256-acm...
0000003c: 7377 00ff 0000 0000 0000 0000 3863 6439 6430 6631 3438 sw.....8cd9d0f148
00000050: 3438 0000 0000 0000 0000 0000 0000 0000 0000 0000 48.....
00000064: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000078: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
0000008c: 0000 0000 0000 0000 0000 0000 0000 0063 7075 5f73 .....cpu_s
000000a0: 6800 0000 0000 0000 0000 0000 0000 0000 0000 0000 h.....
```

<sup>15</sup><https://github.com/sysflow-telemetry/sf-collector/tree/master/tests>

<sup>16</sup><https://github.com/falcosecurity/libs/tree/master/userspace/libinsp/examples>

<sup>17</sup><https://github.com/google/honggfuzz/blob/master/docs/USAGE.md#output-files>

```

000000b4: 0000 0000 0000 0000 0000 00f1 0000 0000 0000 0000 .....
000000c8: 0063 6173 7435 2d65 6362 0000 0000 0000 0000 0000 .cast5-ecb.....
000000dc: 0000 0000 0000 0000 ff00 0000 0000 0001 0000 0000 .....
000000f0: 0000 .....

```

This input will cause the program to stop on segmentation fault because of a NULL pointer dereference `sinsp_parser::parse_clone_exit` at `libs/userspace/libsinsp/parsers.cpp:1017`. Indeed the faulty code is illustrated in the following listing.

```

1015  parinfo = evt->get_param(0);
1016  [...] // there is an assert here.
1017  childtid = *(int64_t *)parinfo->m_val;

```

The `parinfo` variable has for type `sinsp_evt_param` which could be simplified to this C structure.

```

struct sinsp_evt_param {
    char* m_val;    ///< Pointer to the event parameter data.
    uint32_t m_len; ///< Length of the parameter pointed by m_val.
};

```

In the case of this malformed scap file, the function behind `get_param` will return a `sinsp_evt_param` instance with `m_val` being a NULL pointer thus provoking the error on the NULL dereference in the next line of the example, line 1017 of the `parsers.cpp` file.

<b>INFO 9</b>	Multiple crashes in the parsing of scap files and event buffer with malformed files		
<b>Category</b>	Null dereference		
<b>Rating</b>	<b>Impact:</b> Availability	<b>Exploitability:</b> Easy	

## Conclusion

Although the results of this fuzzing campaign reveal really concrete bugs, it seems that they can only be triggered locally with malformed scap files, which reduce drastically the criticality of these crashes. The challenge that we encounter in this Section 7.2 is exactly to overcome these bugs, trying to find crashes that can be triggered by real events, encoded by the kernel counterpart of Falco, without breaking the encoding.

Fixing these issues would make Falco more robust but would not significantly make this form of fuzzing better, since most of the input generated by the random fuzzer would still be invalid encoding, and thus discarded by Falco. To illustrate how unstructured fuzzing is inappropriate in this situation, we could see that as fuzzing, for example, a TCP segment without knowing the structure of the headers and not fixing the checksum. Most of the input generated will hit a validation wall and stay at the very early stage of the program execution.

## 7.3 Conclusion

Fuzzing the core of Falco proved to be non-trivial because of the complexity of the events parser inputs. We approached this problem with different strategies. First trying to fuzz specifically some parsers with libprotobuf-mutator and highly structured fuzzing. Then doing black box fuzzing of Falco and its driver via syzkaller. And finally trying to do random fuzzing using the scap file format. These approaches have limitations but the syzkaller and libprotobuf-mutator techniques could be useful to discover real bugs.

## 8 Conclusion

To conclude, Quarkslab provided many leads and strategies on how to implement static and dynamic security analysis of the Falco project in the restricted amount of time. This audit also unveiled some issues in the codebase, thanks to the automated tools and the manual investigations, but nothing critical or exploitable in the end.

Overall, it was a pleasure to work with the Falco maintainers on this audit, they were very helpful and willing to make the project more secure.



# Glossary

**ASAN** Address Sanitizer is a memory error detector for C and C++, find more information in its documentation <https://github.com/google/sanitizers/wiki/AddressSanitizer>.

**CNCF** The Cloud Native Computing Foundation (CNCF) is a Linux Foundation project that was founded in 2015 to help advance container technology and align the tech industry around its evolution.

**persistent mode** In persistent mode, fuzzers fuzzes a target multiple times in a single forked process, instead of forking a new process for each fuzz execution. This is the most effective way to fuzz, as the speed can easily be x10 or x20 times faster without any disadvantages. Persistent mode requires that the target can be called in one or more functions, and that it's state can be completely reset so that multiple calls can be performed without resource leaks, and that earlier runs will have no impact on future runs.

**protobuf** Protocol Buffers (a.k.a., protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. See the documentation for more information <https://developers.google.com/protocol-buffers/>.

# Bibliography

- [1] Falco maintainers. *Build Falco from source*. The Falco documentation. URL: <https://falco.org/docs/getting-started/source/> (visited on Dec. 6, 2022) (cit. on p. 6).
- [2] Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, MSc. D. Weißer, B. Walny, BSc. J. Hector, and J. Larsson. *Pentest-Report Falco*. July 24, 2019. URL: [https://cure53.de/pentest-report\\_falco.pdf](https://cure53.de/pentest-report_falco.pdf) (visited on Dec. 5, 2022) (cit. on p. 6).
- [3] Hi120ki. *Discuss about fundamental solution of detecting symlink file based bypass method #2203*. Sept. 13, 2022. URL: <https://github.com/falcosecurity/falco/issues/2203> (visited on Dec. 6, 2022) (cit. on p. 6).
- [4] Mark Manning. *Container Runtime Security Bypasses on Falco*. Sept. 15, 2019. URL: <https://www.antitree.com/2019/09/container-runtime-security-bypasses-on-falco/> (visited on Dec. 19, 2022) (cit. on p. 6).
- [5] Brad Geesaman. *Falco Default Rule Bypass*. Sept. 11, 2020. URL: <https://web.archive.org/web/20220605221820/https://darkbit.io/blog/falco-rule-bypass> (visited on Dec. 19, 2022) (cit. on p. 6).
- [6] Leonardo Di Donato. *Bypass Falco*. May 4, 2021. URL: <https://www.youtube.com/watch?v=nGqWskXRSmo> (visited on Dec. 19, 2022) (cit. on p. 6).
- [7] Shay Berkovich. *Bypassing Falco - How to compromise a cluster without tripping the SOC*. July 15, 2021. URL: <https://github.com/blackberry/Falco-bypasses> (visited on Dec. 19, 2022) (cit. on p. 6).
- [8] Jason Dellaluce and Federico Di Pierro. *Monitoring new syscalls with Falco*. The Falco Blog. Jan. 17, 2022. URL: <https://falco.org/blog/falco-monitoring-new-syscalls/> (visited on Dec. 5, 2022) (cit. on p. 6).
- [9] inOva. *Falco Design Principle Analysis*. Chinese. Jan. 10, 2021. URL: <https://driverxdw.github.io/2021/01/10/Falco-Design-Principle-Analysis/> (visited on Dec. 5, 2022) (cit. on p. 6).
- [10] inOva. *Sysdig Source Code Analysis*. Chinese. Part 1 <https://driverxdw.github.io/2021/05/29/Sysdig-Source-Code-Analysis/> and part 2 <https://driverxdw.github.io/2021/06/27/Sysdig-Source-Code-Analysis-II/>. May 29, 2021. (Visited on Dec. 5, 2022) (cit. on p. 6).
- [11] Mark Stemm. *CVE-2019-8339, a Falco capacity related vulnerability*. Sysdig blog. May 13, 2019. URL: <https://sysdig.com/blog/cve-2019-8339-falco-vulnerability/> (visited on Dec. 6, 2022) (cit. on p. 6).
- [12] Falco maintainers. *GitHub Security Advisories*. Falco <https://github.com/falcosecurity/falco/security/advisories> and libs <https://github.com/falcosecurity/libs/security/advisories>. (Visited on Dec. 6, 2022) (cit. on p. 6).

# Appendix A

## Severity Classification

Severity	Description
Medium	Medium issues that cannot be directly exploited, such as buffer overflows that could potentially lead to a crash, arbitrary read or arbitrary code execution by unprivileged users if exploitable in the future.
Low	Low issues that cannot be directly exploited, such as memory, resource leak or Null pointer dereference that could lead to a crash of the program by unprivileged users.
Info	Minor issues such as programming mistakes or the above issues happening in less important parts of the code, like debug or test code that makes it unreachable from unprivileged users.

# Appendix B

## Infer report extracts

This is Infer's partial report, errors not mentioned were removed from this extract, see the `infer-report.txt` file for the complete version.

```
#9
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/plugin/plugin_loader.c:73: error: Null Dereference
pointer `ret` last assigned on line 55 could be null and is dereferenced at line 73, column 5.
71.     }
72. #else
73.     ret->handle = dlopen(path, RTLD_LAZY);
74.     if (ret->handle == NULL)
75.     {

#13
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/engine/kmod/scap_kmod.c:67: error: Resource
Leak
↳ resource of type `_IO_FILE` acquired to `return` by call to `fopen()` at line 61, column 16 is not released after line 67, column 4.
65.         if(w == 0)
66.         {
67.             return 0;
68.         }
69.

#15
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/engine/savefile/scap_reader_gzfile.c:72:
error: Null Dereference
↳ pointer `h` last assigned on line 71 could be null and is dereferenced at line 72, column 5.
70.
71.     reader_handle_t* h = (reader_handle_t *) malloc (sizeof (reader_handle_t));
72.     h->m_file = file;
73.
74.     scap_reader_t* r = (scap_reader_t *) malloc (sizeof (scap_reader_t));

#16
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/engine/savefile/scap_reader_gzfile.c:75:
error: Null Dereference
↳ pointer `r` last assigned on line 74 could be null and is dereferenced at line 75, column 5.
73.
74.     scap_reader_t* r = (scap_reader_t *) malloc (sizeof (scap_reader_t));
75.     r->handle = h;
76.     r->read = &gzfile_read;
77.     r->offset = &gzfile_offset;

#17
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libinsp/filterchecks.cpp:80: error: Null
Dereference
↳ pointer `loc` last assigned on line 78 could be null and is dereferenced at line 80, column 8.
78.     loc = localtime(&t);
79.
80.     dt = (loc->tm_hour - gmt->tm_hour) * 60 * 60 + (loc->tm_min - gmt->tm_min) * 60;
81.
82.     dir = loc->tm_year - gmt->tm_year;

#24
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libinsp/logger.cpp:85: error: Resource Leak
resource of type `_IO_FILE` acquired by call to `fopen()` at line 85, column 11 is not released after line 85, column 2.
83.     ASSERT(m_file == nullptr);
84.
85.     m_file = fopen(filename.c_str(), "w");
86.     if(!m_file)
87.     {

#25
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libinsp/parsers.cpp:99: error: Null Dereference
pointer `evt_state->m_piscapvt` last assigned on line 97 could be null and is dereferenced at line 99, column 2.
97.     evt_state.m_piscapvt = (scap_evt*) realloc(evt_state.m_piscapvt, buf_size);
98.     evt_state.m_scap_buf_size = buf_size;
99.     evt_state.m_piscapvt->type = evt_type;
```

```

100.     evt_state.m_metaevt.m_pevt = evt_state.m_piscapevt;
101. }

#28
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/engine/savefile/scap_reader_buffered.c:132:
↳ error: Null Dereference
pointer `h` last assigned on line 131 could be null and is dereferenced at line 132, column 5.
130.
131.     reader_handle_t* h = (reader_handle_t *) calloc (1, sizeof (reader_handle_t));
132.     h->m_close_reader = own_reader;
133.
133.     h->m_reader = reader;
134.     h->m_buffer = (uint8_t*) malloc (sizeof(uint8_t) * bufsize);

#29
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/engine/savefile/scap_reader_buffered.c:138:
↳ error: Null Dereference
pointer `r` last assigned on line 137 could be null and is dereferenced at line 138, column 5.
136.
137.     scap_reader_t* r = (scap_reader_t *) malloc (sizeof (scap_reader_t));
138.     r->handle = h;
139.
139.     r->read = &buffered_read;
140.     r->offset = &buffered_offset;

#38
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/examples/test.cpp:237: error: Resource Leak
resource acquired by call to `open()` at line 229, column 11 is not released after line 237, column 2.
235.     goto error;
236.
237.     atexit(remove_module);

238.
239.     return true;

#42
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/engine/bpf/scap_bpf.c:351: error: Null
Dereference
pointer `sym` last assigned on line 342 could be null and is dereferenced at line 351, column 6.
349.     }
350.
351.     if(sym[*nr_maps].st_shndx != maps_shndx)
352.     {
353.         continue;

#43
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_fds.c:544: error: Null Dereference
pointer `handle->m_dev_list->hh.tbl` last assigned on line 544 could be null and is dereferenced by call to `memset()` at line 544,
column 5.
542.         mountinfo->mount_id = mount_id;
543.         mountinfo->dev = dev;
544.         HASH_ADD_INT64(handle->m_dev_list, mount_id, mountinfo);
545.
545.         if(uth_status != SCAP_SUCCESS)
546.         {

#45
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/threadinfo.cpp:635: error: Null Dereference
pointer `zero` last assigned on line 634 could be null and is dereferenced by call to `memcmp()` at line 635, column 8.
633.         size_t sz = len - offset;
634.         void* zero = calloc(sz, sizeof(char));
635.         if(!memcmp(left, zero, sz))
636.         {
637.             free(zero);

#51
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_fds.c:710: error: Null Dereference
pointer `*sockets_by_ns->hh.tbl` last assigned on line 710 could be null and is dereferenced by call to `memset()` at line 710,
column 4.
708.         char fd_error[SCAP_LASTERR_SIZE];
709.
710.         HASH_ADD_INT64(*sockets_by_ns, net_ns, sockets);
711.
711.         if(uth_status != SCAP_SUCCESS)
712.         {

#52
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_fds.c:706: error: Null Dereference
pointer `sockets` last assigned on line 705 could be null and is dereferenced at line 706, column 4.
704.     {
705.         sockets = malloc(sizeof(struct scap_ns_socket_list));
706.         sockets->net_ns = net_ns;
707.
707.         sockets->sockets = NULL;
708.         char fd_error[SCAP_LASTERR_SIZE];

```

```

#53
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_fds.c:876: error: Null Dereference
pointer `_he_new_buckets` last assigned on line 876 could be null and is dereferenced by call to `memset()` at line 876, column 3.
874.     }
875.
876.     HASH_ADD_INT64((*sockets), ino, fdinfo);

877.     if(uth_status != SCAP_SUCCESS)
878.     {

#54
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_savefile.c:1064: error: Resource Leak
resource of type `_IO_FILE` acquired by call to `gzdopen()` at line 1042, column 8 is not released after line 1064, column 9.
790.     sinsp_fdtable* fdt = get_fd_table();
791.
792.     for(it = fdt->m_table.begin(); it != fdt->m_table.end(); ++it)
793.     {
794.         if(it->second.m_type == SCAP_FD_IPV4_SOCKET)

#56
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_savefile.c:1093: error: Resource Leak
resource of type `_IO_FILE` acquired by call to `gzdopen()` at line 1079, column 7 is not released after line 1093, column 9.
817.     sinsp_fdtable* fdt = get_fd_table();
818.
819.     for(it = fdt->m_table.begin();
820.         it != fdt->m_table.end(); ++it)
821.     {

#57
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1418: error: Null Dereference
pointer `_he_new_buckets` last assigned on line 1418 could be null and is dereferenced by call to `memset()` at line 1418, column 2.
912.     {
913.         void* newbuf = malloc(sizes->at(j));
914.         memset(newbuf, 0, sizes->at(j));
915.         m_private_state.push_back(newbuf);
916.     }

#59
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_savefile.c:985: error: Null Dereference
pointer `res` last assigned on line 984 could be null and is dereferenced at line 985, column 2.
983. {
984.     scap_dumper_t* res = (scap_dumper_t*)malloc(sizeof(scap_dumper_t));
985.     res->m_f = gzfile;
986.     res->m_type = DT_FILE;
987.     res->m_targetbuf = NULL;

#60
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_savefile.c:1093: error: Resource Leak
resource of type `_IO_FILE` acquired by call to `gzdopen()` at line 1079, column 7 is not released after line 1093, column 9.
1062.     }
1063.
1064.     return scap_dump_open_gzfile(handle, f, fname, skip_proc_scan);
1065. }
1066.

#61
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_savefile.c:1093: error: Resource Leak
resource of type `_IO_FILE` acquired by call to `gzdopen()` at line 1079, column 7 is not released after line 1093, column 9.
1091.     }
1092.
1093.     return scap_dump_open_gzfile(handle, f, "", skip_proc_scan);
1094. }
1095.

#62
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_savefile.c:1093: error: Resource Leak
resource of type `_IO_FILE` acquired by call to `gzdopen()` at line 1079, column 7 is not released after line 1093, column 9.
1091.     }
1092.
1093.     return scap_dump_open_gzfile(handle, f, "", skip_proc_scan);
1094. }
1095.

#67
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1418: error: Null Dereference
pointer `_he_new_buckets` last assigned on line 1418 could be null and is dereferenced by call to `memset()` at line 1418, column 2.
1416.     int32_t uth_status = SCAP_SUCCESS;
1417.
1418.     HASH_ADD_INT64(handle->m_proclist.m_proclist, tid, tinfo);
1419.     if(uth_status == SCAP_SUCCESS)
1420.     {

```

```

#68
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1418: error: Null Dereference
pointer `handle->m_proclist.m_proclist->hh.tbl` last assigned on line 1418 could be null and is dereferenced by call to `memset()`
↳ at line 1418, column 2.
1416.     int32_t uth_status = SCAP_SUCCESS;
1417.
1418.     HASH_ADD_INT64(handle->m_proclist.m_proclist, tid, tinfo);
1419.
1419.     if(uth_status == SCAP_SUCCESS)
1420.     {

#69
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libsinsp/threadinfo.cpp:1482: error: Null
Dereference
pointer `fdtable` last assigned on line 1474 could be null and is dereferenced at line 1482, column 15.
1480.         eparams.m_ts = m_inspector->m_lastevent_ts;
1481.
1482.         for(fdit = fdtable->begin(); fdit != fdtable->end(); ++fdit)
1483.         {
1484.             eparams.m_fd = fdit->first;

#70
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1434: error: Null Dereference
pointer `_he_new_buckets` last assigned on line 1434 could be null and is dereferenced by call to `memset()` at line 1434, column 2.
1432.     int32_t uth_status = SCAP_SUCCESS;
1433.
1434.     HASH_ADD_INT64(tinfo->fdlist, fd, fdinfo);
1435.
1435.     if(uth_status == SCAP_SUCCESS)
1436.     {

#71
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1434: error: Null Dereference
pointer `tinfo->fdlist->hh.tbl` last assigned on line 1434 could be null and is dereferenced by call to `memset()` at line 1434,
column 2.
1432.     int32_t uth_status = SCAP_SUCCESS;
1433.
1434.     HASH_ADD_INT64(tinfo->fdlist, fd, fdinfo);
1435.
1435.     if(uth_status == SCAP_SUCCESS)
1436.     {

#74
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1501: error: Null Dereference
pointer `stid` last assigned on line 1500 could be null and is dereferenced at line 1501, column 3.
1499.     {
1500.         stid = (scap_tid *) malloc(sizeof(scap_tid));
1501.         stid->tid = tid;
1502.
1502.         int32_t uth_status = SCAP_SUCCESS;
1503.

#78
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1696: error: Resource Leak
resource acquired by call to `opendir()` at line 1660, column 15 is not released after line 1696, column 9.
1694.         if((*procinfo_p)->n_entries == (*procinfo_p)->max_entries)
1695.         {
1696.             if(!scap_alloc_proclist_info(procinfo_p, (*procinfo_p)->n_entries + 256, lasterr))
1697.             {
1698.                 goto error;

#79
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libsinsp/threadinfo.cpp:1686: error: Null
Dereference
pointer `proclist_dumper` last assigned on line 1640 could be null and is dereferenced by call to `scap_write_proclist_end()` at
line 1686, column 5.
1684.     });
1685.
1686.     if(scap_write_proclist_end(m_inspector->m_h, dumper, proclist_dumper, totlen) != SCAP_SUCCESS)
1687.     {
1688.         throw sinsp_exception(scap_getlasterr(m_inspector->m_h));

#80
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap.c:1662: error: Null Dereference
pointer `handle->m_suppressed_comms` last assigned on line 1659 could be null and is dereferenced at line 1662, column 2.
1660.         handle->m_num_suppressed_comms * sizeof(char *));
1661.
1662.     handle->m_suppressed_comms[handle->m_num_suppressed_comms-1] = strdup(comm);
1663.
1664.     return SCAP_SUCCESS;

#87
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1812: error: Null Dereference
pointer `_he_new_buckets` last assigned on line 1812 could be null and is dereferenced by call to `memset()` at line 1812, column 4.

```

```

1810.         {
1811.             int32_t uth_status = SCAP_SUCCESS;
1812.             HASH_ADD_INT64(handle->m_proclist.m_proclist, tid, tinfo);

1813.             if(uth_status != SCAP_SUCCESS)
1814.             {

#88
falcosecurity-libs-repo/falcosecurity-libs-prefix/src/falcosecurity-libs/userspace/libscap/scap_procs.c:1812: error: Null Dereference
pointer `handle->m_proclist.m_proclist->hh.tbl` last assigned on line 1812 could be null and is dereferenced by call to `memset()`
↪ at line 1812, column 4.
1810.         {
1811.             int32_t uth_status = SCAP_SUCCESS;
1812.             HASH_ADD_INT64(handle->m_proclist.m_proclist, tid, tinfo);

1813.             if(uth_status != SCAP_SUCCESS)
1814.             {

```



# Appendix C

## Scan-Build experimentations

First, Scan-Build was ran with no specific options like that, on the falco repository using the `USE_BUNDLED_DEPS=ON` option:

```
scan-build make -j 32
```

It found 595 bugs, all being on Falco's third-party dependencies, none on Falco's libs or direct source code.

It was retried, adding more checkers with the following command:

```
scan-build -enable-checker unix,nullability,core,cplusplus,security make -j 32
```

This scan found 2491 bugs, again all out of Falco's scope.

This was a bit suspicious so we then realized that the analyze “fake compiler” of scan-build was only used on the dependencies and not on Falco's libs and source code, explaining why it was not finding anything on this code.

So we tried to force the use of this “compiler” for the code using (the path is just copy-pasted from the Scan-Build lines of compilation, we could have simplify the `../`):

```
cmake -DUSE_BUNDLED_DEPS=ON
↳ -DCMAKE_C_COMPILER=/usr/share/clang/scan-build-10/bin/./libexec/ccc-analyzer
↳ -DCMAKE_CXX_COMPILER=/usr/share/clang/scan-build-10/bin/./libexec/c++-analyzer
↳ ..
```

Unfortunately, it's possible to run the analyzer like this but it will not generate the report, only the warning during the compilation process. It would have been useful because we could have compiled only the source code we are interested in with the static analyzer.

However, Scan-Build has an `--exclude` flag, that could be used to exclude the source code of dependencies. We tried adding these flags:

```
--exclude b64-prefix --exclude c-ares-prefix --exclude catch2-prefix --exclude  
↳ cloudtrail-plugin-prefix --exclude cloudtrail-rules-prefix --exclude  
↳ cpp-http-lib-prefix --exclude curl-prefix --exclude cxxopts-prefix --exclude  
↳ fakeit-prefix --exclude grpc-prefix --exclude jq-prefix --exclude  
↳ json-plugin-prefix --exclude k8saudit-plugin-prefix --exclude  
↳ k8saudit-rules-prefix --exclude njson-prefix --exclude openssl-prefix  
↳ --exclude protobuf-prefix --exclude re2-prefix --exclude  
↳ string-view-lite-prefix --exclude tbb-prefix --exclude valijson-prefix  
↳ --exclude yamlcpp-prefix --exclude zlib-prefix
```

## Exploiting and debugging readlink issues

```
export NEW_ROOT=$(python3 -c "print('a'*200 + '/' + 'a'*200 + '/' + 'a'*200 + '/' + 'a'*200 + '/' + 'a'*200 + '/' + 'a'*200 + '/' + 'a'*200)")
mkdir -p $NEW_ROOT
mkdir $NEW_ROOT/lib
mkdir $NEW_ROOT/lib64
mkdir $NEW_ROOT/bin
sudo mount -B /lib $NEW_ROOT/lib
sudo mount -B /lib64 $NEW_ROOT/lib64
sudo mount -B /bin $NEW_ROOT/bin
sudo chroot $NEW_ROOT /bin/bash
echo $$
## NOTE; you will have to `sudo umount lib lib64 and bin` before `rm` the folders
```

You don't need to be root to chroot, you can use usernamespace to gain that privilege with `unshare -r bash -c 'chroot root /folder'`.

```
sudo gdb ./userspace/falco/falco
b scap_proc_fill_root
r -c ../falco.yaml -r ../rules/falco_rules.yaml
# `return SCAP_SUCCESS;` was on line 560 of the source
# make sure to change the PID returned by the chrooted bash with echo $$
break 560 if strcmp(procdirname, "/proc/783385/") == 0
r -c ../falco.yaml -r ../rules/falco_rules.yaml
# you can see that the structure with
p *tinfo
# but still the last char on the string array is not \0, it's an 'a'
print tinfo->root[sizeof(tinfo->root)-1]
# with the following command you can see the end of the string array on the
# fourth line, and the rest of the element that starts on the lasts lines (some
# values are at zero so we cannot distinguish them for alignments zeros)
x/128x tinfo->root+1000
```

---

Quarkslab SAS

```

#include <unistd.h>
#include <stdio.h>

#define SCAP_MAX_PATH_SIZE 10

typedef struct scap_threadinfo
{
    char root[SCAP_MAX_PATH_SIZE + 1]
} scap_threadinfo;

static int scap_proc_fill_root(struct scap_threadinfo *tinfo, const char
↪ *procdirname)
{
    if (readlink(procdirname, tinfo->root, sizeof(tinfo->root)) > 0)
    {
        printf("success: %s\n", tinfo->root);
        return 0;
    }
    else
    {
        printf("fail\n");
        return -1;
    }
}

int main()
{
    scap_threadinfo tinfo;
    return scap_proc_fill_root(&tinfo, "input");
}

```

Execute like that, in the same folder:

```

touch aaaaaaaaaaaaaaaaaaaaaa
ln -s aaaaaaaaaaaaaaaaaaaaaa input
cc test.c -fsanitize=address -o test
./test

```

You will get an [ASAN](#) report for a stack buffer overflow in this situation.

# Appendix E

## AFL++ persistent mode boilerplate

On top of the Listing 1, you will need to add this kind of code to do persistent mode fuzzing with AFL++:

```
#include <unistd.h>
__AFL_FUZZ_INIT();

int main()
{
#ifdef __AFL_HAVE_MANUAL_CONTROL
    __AFL_INIT();
#endif
    unsigned char* buf = __AFL_FUZZ_TESTCASE_BUF;

    while(__AFL_LOOP(10000))
    {
        int len = __AFL_FUZZ_TESTCASE_LEN;
        LLVMFuzzerTestOneInput(buf, len);
    }

    return 0;
}
```

You will then need to compile with the AFL++ instrumentation using, for example:

```
cmake -DUSE_BUNDLED_DEPS=ON -DCMAKE_C_COMPILER=afl-gcc-fast
↪ -DCMAKE_CXX_COMPILER=afl-g++-fast ..
```

# Appendix F

## Fuzzing with libprotobuf-mutator

```
syntax = "proto3";

package sys_open;

message event_args {
    uint64 fd = 1;
    string fspath = 2;
    repeated FILE_FLAGS flags32 = 3;
    FILE_MODE mode = 4;
    uint32 dev = 5;
    uint64 ino = 6;
}

enum FILE_FLAGS {
    FUZZ_O_NONE = 0;
    FUZZ_O_RDONLY = 1;
    FUZZ_O_WRONLY = 2;
    FUZZ_O_RDWR = 3;
    FUZZ_O_CREAT = 4;
    FUZZ_O_APPEND = 8;
    FUZZ_O_DSYNC = 16;
    FUZZ_O_EXCL = 32;
    FUZZ_O_NONBLOCK = 64;
    FUZZ_O_SYNC = 128;
    FUZZ_O_TRUNC = 256;
    FUZZ_O_DIRECT = 512;
    FUZZ_O_DIRECTORY = 1024;
    FUZZ_O_LARGEFILE = 2048;
    FUZZ_O_CLOEXEC = 4096;
    FUZZ_O_TMPFILE = 8192;
}

enum FILE_MODE {
    FUZZ_S_NONE = 0;
    FUZZ_S_IXOTH = 1;
    FUZZ_S_IWOTH = 2;
    FUZZ_S_IROTH = 4;
    FUZZ_S_IXGRP = 8;
    FUZZ_S_IWGRP = 16;
    FUZZ_S_IRGRP = 32;
    FUZZ_S_IXUSR = 64;
    FUZZ_S_IWUSR = 128;
    FUZZ_S_IRUSR = 256;
    FUZZ_S_ISVTX = 512;
```

```
FUZZ_S_ISGID = 1024;  
FUZZ_S_ISUID = 2048;  
}
```