Fine penetration tests for fine websites

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

Security-Review Report SPIRE 01-02.2021

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. R. Peraglie, MSc. N. Krein

Index

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

Introduction

“SPIRE (the SPIFFE Runtime Environment) is a toolchain of APIs for establishing trust between software systems across a wide variety of hosting platforms. SPIRE exposes the SPIFFE Workload API, which can attest running software systems and issue SPIFFE IDs and SVIDs to them. This in turn allows two workloads to establish trust between each other, for example by establishing an mTLS connection or by signing and verifying a JWT token. SPIRE can also enable workloads to securely authenticate to a secret store, a database, or a cloud provider service.”

From https://github.com/spiffe/spire

This report describes the results of a security-centered assessment of the SPIRE complex. Carried out by Cure53 at the beginning of 2021, the project included a penetration test, a source code audit, as well as a broader security posture check of the SPIRE software compound.

It should be clarified that SPIRE, a.k.a. the SPIFFE Runtime Environment, is a toolchain of APIs for establishing trust between software systems across a wide variety of hosting platforms. The work detailed here was requested by the Cloud Native Computing Foundation (CNCF) in late 2020 and carried out by Cure53 in the second half of January 2021. A total of thirty-two days were invested into the project, given the objectives and expected coverage.

To respond to the priorities set by CNCF/SPIRE, three work packages (WPs) were delineated. In WP1, Cure53 reviewed the security posture of the SPIRE project more broadly, moving on to source code audit of the SPIRE code base in WP2. Finally, a penetration test against SPIRE deployment was executed in WP3. Notably, this structure of the WPs, which go beyond standard pentesting and auditing and extend to checking the general perimeter, typically characterized CNCF-related work.

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

White-box methods were used in this project, given that the source code is publicly available as an OSS project. More generally, the approach is consistent with CNCF-commissioned projects that Cure53 has been involved in. Furthermore, Cure53 had access to a pre-configured environment set up by the SPIFFE team. Said deployment offered a production-like test-surface for Cure53 to work with.

All preparations were done on time in January 2021, namely in CW02 and CW03. Consequently, Cure53 could have a smooth start without any roadblocks in CW04. Communications during the test were done using a channel on the SPIFFE Slack workspace. Members of the Cure53 team participating in this assignment could join the discussions and keep the SPIRE/SPIFFE team appraised of new developments in the test. Besides frequent status, live-reporting was done so that the SPIFFE team could address all findings pertinent to SPIRE as quickly as possible.

Overall, the SPIFFE team did a fantastic job in making things available and accessible for the test. In connection to this, Cure53 achieved a very good coverage of the test-targets across WP1-WP3 in the time-frame available for this exercise. Eight security-relevant discoveries were made. Three items were classified to be security vulnerabilities and five belong to a broader array of general weaknesses with lower exploitation potential. Note that one of the findings was ranked as a *High*-level risk; other issues - beyond two *Medium*-level threats - did not call for immediate action or remediation. It can be argued that the evidence indicates a rather praiseworthy result for SPIRE.

In the following sections, the report will first shed light on the scope and key test parameters, as well as the structure and content of the WPs. In order to maximize gains from the project, Cure53 then offers a series of methodology and coverage chapters, so as to highlight what has been done and with which results, especially in the area of security posture checks. Next, all findings will be discussed in grouped vulnerability and miscellaneous categories, then following a chronological order in each grouping. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable. Finally, the report will close with broader conclusions about this January 2021 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team’s observations and collected evidence. Tailored hardening recommendations for the SPIRE complex are also incorporated into the final section.

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

Scope

- **Penetration Tests & Security Reviews against SPIRE**
  - **WP1**: Security posture of the SPIRE project
  - **WP2**: Source code audit of SPIRE
    - *Main focus* https://github.com/spiffe/spire/releases/tag/v0.12.0
    - *Misc focus* https://github.com/spiffe/go-spiffe/releases/tag/v2.0.0-beta.4
  - **WP3**: Penetration test against the SPIRE deployment
    - 34.214.21.89 granola-global # Root Server
    - 54.149.236.250 granola-regional-1 # Nested Server
    - 54.184.34.192 granola-workload # Agent / Workload
    - 18.223.247.8 acme-regional # Agent / Workload
    - 3.131.153.201 acme-workload # Standalone Server
    - 34.220.90.50 granola-regional-2 # Nested Server

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

Test Methodology

This section zooms in on the metrics and methodologies used to evaluate security characteristics of the SPIRE project and codebase. In addition, it includes results pertinent to individual areas of the project's security properties that were either selected by Cure53 or singled out by other involved parties as calling for a closer inspection.

Similarly to previous tests for CNCF, this assignment was also divided into two phases. The general security posture and maturity of the audited codebase of SPIRE has been examined in *Phase 1.* The usage of external dependencies has been audited, security constraints for SPIRE configurations were examined and the documentation had been studied in depth in order to get a general idea of security awareness' levels at SPIRE.

This was followed by research into how security reports and vulnerabilities are handled and whether a healthily secure infrastructure is seen as a serious matter. The latter phase covered actual tests and audits against the SPIRE codebase, with the code quality and its hardening evaluated.

Phase 1: General security posture checks

Because SPIRE is a relatively complex software architecture, Cure53 was pleased to see extensive documentation and additional material, such as the provided Security Self-Assessment. This greatly helped with *Phase 1* of this project in which the general security posture and overall code quality of the SPIRE project was inspected from a high-level perspective.

This encompasses also the management processes such as the handling of vulnerability disclosures, threat modeling approaches and general measures for code hardening. All this gives a meta-level perception of the maturity and robustness that is not solely bound to the code quality itself.

Phase 2: Manual code audits and penetration tests

In this Phase, Cure53 conducted an extensive source code analysis across the different components of the SPIRE software stack. Since SPIRE has well-defined attacker models containing risk-assessments, it is quite clear what issues to look out for and what to concentrate on. As such, identification of security-relevant areas of the project's code base was close-to-unnecessary and, in effect, Cure53 could quickly start with targeted audits on sensitive parts of the system.

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

This also underlines the well-thought-out development process and self-reflection of the developers that have the attacker's perspective in mind. While Cure53's goal was to reach good coverage across the scope, such large-scale audits are always limited by the budget and require a more isolated focus on the particularly sensitive parts of the code. Consequently, this phase also determines which parts of the project's scope deserve more focus in future audits.

Later chapters and especially the findings in the Identified Vulnerabilities and the Miscellaneous Issues sections of this report highlight issues that were found during this audit. Their implications for the SPIRE software complex are discussed there.

Phase 1: General security posture checks

This Phase is meant to provide a more detailed overview of the SPIRE project's security properties that are seen as somewhat separate from both the code and the SPIRE software. The first few subsections of the posture audit focus on more abstract components instead of judging the code quality itself.

Later subsections look at elements that are linked more strongly to the organizational and team aspects of SPIRE. In addition to the items presented below, the Cure53 team also focused on tasks that fostered a cross-comparative analysis of all observations.

- The documentation was examined to understand all provided functionality and acquire examples of what a real-world deployment of SPIRE could look like.
- The extensive architectural design documentation as well as several parts of the self-security assessments were reviewed.
- The network topology and connected parts of the overall architecture were examined. This also included consideration of relevant configurations that are necessary to deploy SPIRE. Some potential weaknesses were already spotted there and are highlighted in the Configuration Concerns section of this report.
- The given code-base was reviewed for structural design, documentation and comments. High-level code audits and common pitfalls in the Go programming language were looked out for. For example, general issues such as incorrect usage of the *unsafe* keyword can quickly be enumerated.
- Code issue reports from *gosec* were studied to check if there are any remaining low-hanging fruit inside that simply have been overlooked during the codebase development.
- External libraries and frameworks that are referenced in the code were studied and checked to make sure they do not contain any publicly known vulnerabilities. As highlighted in the External Libraries & Frameworks and the connected finding under SPI-01-005, Cure53 proposed some improvements.

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

- Normally, past vulnerability reports for SPIRE would have been checked out to spot further interesting areas and generally monitor the disclosure process, yet SPIRE has not received impactful vulnerability reports thus far.
- The security posture checks phase is concluded with an analysis of organizational specifics, such as making sure there are good guidelines for security contacts. Soundness of the bug tracking and review process was also verified.
- Drawing on the evidence from the steps above, the project's maturity was evaluated; specific questions about the software were compiled from a general catalogue according to applicability.

Application/Service/Project Specifics

In this section, Cure53 will share insights on the application-specific aspects which lead to a good security posture. These include the choice of programming language, selection and oversight of external third-party libraries, as well as other technical aspects like logging, monitoring, test coverage and access control.

Language Specifics

Programming languages can provide functions that pose an inherent security risk and their use is either deprecated or discouraged. For example, *strcpy()* in C has led to many security issues in the past and should be avoided altogether. Another example would be the manual construction of SQL queries versus the usage of prepared statements. The choice of language and enforcing the usage of proper API functions are, therefore, crucial for the overall security of the project.

Since SPIRE seamlessly integrates with software such as Envoy and can be built on top of Kubernetes clusters, it comes as no surprise that Go has been chosen as a programming language. Go has proven to offer higher levels of memory safety compared to other languages that compile to native code. It is quite rare to spot direct memory safety issues that other languages such as C and C++ suffer from.

Issues like buffer overflows, type confusions or Use-After-Free vulnerabilities are directly taken care of by Go's internal memory management system. The compiler equally makes sure that memory bounds are automatically verified by placing checkpoints into the generated assembly. Although it is still possible to write unsafe Go code, SPIRE completely refrains from doing so.

While memory safety issues are of lesser concern in software stacks that build on Go, Cure53 still focused on shortcomings represented by integer overflows and race conditions through incorrectly placed mutexes. Although Go's garbage collector is the

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

saving grace in many situations that lead to Denial-of-Service, leaky memory stemming from open file descriptors can signify threats in the form of bugs or input validation issues.

External Libraries & Frameworks

External libraries and frameworks can also contain vulnerabilities but their benefits outweigh the possible risks. Relying on sophisticated libraries is advised instead of reinventing the wheel with every project. This is especially true for cryptographic implementations, which are known to be prone to errors.

SPIRE mostly imports well-tested crypto packages of the Go language framework. This includes *crypto/rand*, *crypto/tls* or *crypto/x509*. For transport mechanisms and the different APIs, SPIRE mainly uses *google.golang.org/grpc* derived from *protobufs* (*github.com/golang/protobuf*). While some of the direct dependencies are mostly up to date, Cure53 noticed a significant list of outdated modules that are linked in the *master* branch of *spiffe/spire*. This is additionally documented as a miscellaneous finding in SPI-01-005. Only one outdated module was found to be suffering from a security vulnerability but it should generally be considered to tidy up the dependency list. This generally prevents shipping a product that includes vulnerable modules which are actually mitigated in more recent versions.

With a significant number of third-party modules, dependency tracking and vulnerability checking becomes no easy task. Consequently, it is recommended to automate this process and have a build-system that takes major version changes for third-party modules into consideration. For software written in Go, Cure53 had good experience with *go-mod-outdated*[1], which lists version changes for direct dependencies. Other hand tools like *OWASP Dependency-Check*[2] work with any language and pull additional data from the NIST National Vulnerability Database[3]. This additionally helps in finding outdated modules affected by issues and requiring immediate attention/ mitigation.

Configuration Concerns

Complex and adaptable software systems usually have many variable options which can be configured accordingly to what the actually deployed application necessitates. While this is a very flexible approach, it also leaves immense room for mistakes. As such, it often creates the need for additional and detailed documentation, in particular when it comes to security.

[1] https://github.com/psampaz/go-mod-outdated
[2] https://owasp.org/www-project-dependency-check/
[3] https://nvd.nist.gov/vuln/data-feeds

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

The attested agent path for many built-in node attestor plugins may be a risk due to user-input embedded within the attested agent path, as seen in SPI-01-003 and SPI-01-007. It is advised to consider this threat in the security section of the documentation.

At the same time, SPIRE APIs could be configured to be exposed via TCP instead of a unix domain socket. However, this configuration is discouraged by the documentation for the Agents Workload API. The same applies to turning off the TLS verification within *go-spiffe/v2,* which is also highly discouraged by SPIRE documentation.

Configured workload and node attestor plugins could become a risk as they contain critical and potentially vulnerable code that could allow attackers to impersonate agents and workloads. If SPIFFE Federation is configured with an untrusted bundle endpoint, poisoning the root of the certificate chain could be accomplished.

Access Control

Every access to SPIRE server that could potentially do harm or extract sensitive information is primarily controlled via SSL client authentication and the local socket type. The SSL verification is outsourced to Go's built-in *crypto/tls* module and provides the certificate authority specified by the user. The *authorization* code is redundant in modern API and legacy API in such that the validations and checks performed are the same. Therefore, it is recommended to remove the old legacy node and entry registration APIs the code base to minimize redundancy.

The authorization logic distinguishes a total of five user-roles: unauthenticated, agent, local, admin and other downstream servers. The local role will be assigned to peers that connect locally via the unix domain socket. All other roles will be determined by the presence of a valid client certificate and the associated Spiffe ID extracted from the certificate. Every gRPC method provided by the server was explicitly assigned a set of allowed user-roles required for invocation. Validated information relevant for authorization is stored in a context persistently passed to all handlers.

Access Control on the Agents Workload API is primarily done physically by only exposing the Workload and Secret Service Discovery API over a unix domain socket. It requires workloads to obtain file-access to the UDS file. Additionally, a mandatory security metadata key must be added to every request to the Workload API. This can mitigate the impact of potential SSRF vulnerabilities in the network if access was indeed given via TCP (which is discouraged in most scenarios by the documentation).

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

Logging/Monitoring

Having a good logging/monitoring system in place allows developers and users to identify potential issues more easily or get an idea of what might be going wrong. It can also provide security-relevant information, for example when a verification of a signature fails. Consequently, having such a system in place has a positive influence on the project.

On the one hand the SPIRE agents and the SPIRE server offer centralized logging via the *log_file* flag of the corresponding configuration files with extra levels for verbosity. This default setting here, however, is set to logging to STDOUT only, which should be treated as suboptimal. Default logging to one of the standard */var/log* directories should be considered here.

On the other hand, both agents and servers allow exporting of metrics to external collectors such as *Datadog, M3, Prometheus* and *StatsD*. This can be configured through the *server.conf* and *agent.conf,* respectively. SPIRE offers a wide range of telemetric data to be transferred; all are explained through dedicated documentation[4].

Unit/Regression and Fuzz-Testing

While tests are essential for any project, their importance grows with the scale of the endeavor. Especially for large-scale compounds, testing ensures that functionality is not broken by code changes. Furthermore, it generally facilitates the premise where features function the way they are supposed to. Regression tests also help guarantee that previously disclosed vulnerabilities do not get reintroduced into the codebase. Testing is therefore essential for the overall security of the project.

SPIRE extensively incorporates unit-tests in nearly all of their modules, giving a comprehensive test coverage across the complete codebase. While walking through the commit log, Cure53 also noticed test-cases for regressions that happened in the past, like for bugs such as for #1863[5] or smaller DoS security issues, e.g., #577[6]. Especially thorough testing is included for highly-sensitive code paths for verifying JWT-SVID tokens or TLS certificate generation, as well as in the realm of correct URI parsing for SPIFFE IDs. All in all, the process of unit-testing and its importance is well-understood at SPIRE. There is high expectation that this approach continues throughout development in the future.

[4] https://github.com/spiffe/spire/blob/master/doc/telemetry.md
[5] https://github.com/spiffe/spire/commit/1e5cda99b7d0934908d37dd2e27f039b479d8d4c
[6] https://github.com/spiffe/spire/pull/577

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

Documentation

Good documentation contributes greatly to the overall state of the project. It can ease the workflow and ensure final quality of the code. For example, having a coding guideline which is strictly enforced during the patch review process ensures that the code is readable and can be easily understood by various developers. Following good conventions can also reduce the risk of introducing bugs and vulnerabilities to the code.

The overall quality of the documentation of SPIFFE/SPIRE is praiseworthy. It contains every information that is necessary to understand the overall concept of the architecture, how to deploy it and how to use it. It is structured in a way that step-wise guides help the user to first learn about SPIRE's design and goals, and then explain how to set up agents and servers, for example in regard to Kubernetes clusters. Each step gets a dedicated section in the documentation, inclusive of extensive guides about configurational details and scaling the architecture throughout multiple trust domains.

The content does not leave much room for questions and underlines the overall maturity of the software and its concept. Apart from the linked *CONTRIBUTING.md* inside the GitHub page, the documentation additionally contains a guideline on extending SPIRE, mostly for the different plugin systems it supports. It appropriately advises caution when using third-party plugin code. Extra sections on how to interact with workload APIs is also offered.

What might be a little confusing for some is that documentation pages inside the GitHub repository under https://github.com/spiffe/spire/tree/master/doc do not entirely match the official one. The former often provides a little more detail on certain components of the SPIRE architecture. Given that the *doc* pages on GitHub are regularly updated, it should be considered to sync them with the official documentation more properly.

Organization/Team/Infrastructure Specifics

This section will describe the areas Cure53 looked at to learn more about the security qualities of the SPIRE project that cannot be linked to the code and software but rather encompass handling of incidents. As such, it tackles the level of preparedness for critical bug reports within the SPIRE development team. In addition, Cure53 also investigated the levels of community involvement, i.e. through the use of bug bounty programs. While a good level of code quality is paramount for a good security posture, the processes and implementations around it can also make a difference in the final assessment of the security posture.

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

Security Contact

To ensure a secure and responsible disclosure of security vulnerabilities, it is important to have a dedicated point of contact. This person/team should be known, meaning that all necessary information - such as an email address and preferably also encryption keys of that contact - should be communicated appropriately.

SPIFFE/SPIRE mentions a security contact multiple times in their GitHub repository. The *SECURITY.md* and the *CONTRIBUTING.md* as well as the *README.md* contain a small section on where to report security vulnerabilities to. Except for the email address, no guidelines on the report format or further details are provided. The sections additionally lack a PGP key which would be beneficial for more severe vulnerability reports. Additionally, the official documentation lacks further mention of a security contact. The mentioned email address is not found outside of the GitHub repository. Since many developers and users start with the documentation under https://spiffe.io/, a dedicated section about a security contact might be helpful there as well.

Security Fix Handling

When fixing vulnerabilities in a public repository, it should not be obvious that a particular commit addresses a security issue. Moreover, the commit message should not give a detailed explanation of the issue. This would allow an attacker to construct an exploit based on the patch and the provided commit message prior to the public disclosure of the vulnerability. This means that there is a window of opportunity for attackers between public disclosure and wide-spread patching or updating of vulnerable systems. Additionally, as part of the public disclosure process, a system should be in place to notify users about fixed vulnerabilities.

Walking through the commit log, SPIRE and *go-spiffe-v2* did not receive fixes for vulnerability reports so far. The only exception here is a security fix that would mean that attackers with read log-files access could obtain *JWT-SVIDs*[7]. This is transparently explained and even highlights the faulty commit that introduced this issue. Still, at the time of writing, there is no real sample set to judge the handling of security fixes. At the same time, SPIRE transparently highlighted security impact of version changes that happened whenever security issues in important libraries were disclosed. There are a few examples - such as issue #690[8] or issue #1204[9] - that fix vulnerabilities in Golang and where the commit log mentions the appropriate CVE to highlight what problems the fix mitigates.

[7] https://github.com/spiffe/spire/pull/1953
[8] https://github.com/spiffe/spire/pull/690
[9] https://github.com/spiffe/spire/pull/1204

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

Bug Bounty

Having a bug bounty program acts as a great incentive in rewarding researchers and getting them interested in projects. Especially for large and complex projects that require a lot of time to get familiar with the codebase, bug bounties work on the basis of the potential reward for efforts.

The SPIRE project does not have a bug bounty program at present, however this should not be strictly viewed in a negative way. This is because bug bounty programs require additional resources and management, which are not always a given for all projects. However, if resources become available, establishing a bug bounty program for SPIRE should be considered. It is believed that such a program could provide a lot of value to the project.

Bug Tracking & Review Process

A system for tracking bug reports or issues is essential for prioritizing and delegating work. Additionally, having a review process ensures that no unintentional code, possibly malicious code, is introduced into the codebase. This makes good tracking and review into two core characteristics of a healthy codebase.

SPIRE's *readme* page[10] explains that bugs should be filed via the GitHub's issue tracker. There is no exact guideline or template on how to report bugs, leading to a rather messy report system that makes it harder to distinguish between valid concerns and generic feature requests.

Apart from that, contributions can be made through pull requests on GitHub. This is thoroughly explained in the *CONTRIBUTING.md*[11] file where the required coding conventions and review process are described. Each pull request needs to be approved by one or two maintainers to make sure all changes comply with the required standards, in turn preventing submission of malicious or dysfunctional code.

Evaluating the Overall Posture

Choosing the Go programming language for the majority of code in this project has been a good decision and almost automatically reduces the potential for introducing memory-safety-related issues. Additionally, the excellent documentation along with the well-documented processes for patches and contributions further reduce the risk of security vulnerabilities being handled badly or remaining undetected.

[10] https://github.com/spiffe/spire#contribute-to-spire
[11] https://github.com/spiffe/spiffe/blob/master/CONTRIBUTING.md

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

Note that a dedicated section in the publicly available documentation material also lists the security audits and assessments that have already been carried out in the past, prior to Cure53's thorough posture review and code audit[12]. A topic worth-mentioning is that of a bug bounty program, although it is understandable that smaller projects are likely unable to secure funds for these. However, with future growth of the project and potentially increased resources, a bug bounty scheme should definitely be considered.

Further, some concerns were raised and documented regarding the configurability and the high-level of flexibility being a possible venue for mistakes and resulting vulnerabilities in rolled-out deployments. Two issues were raised here, hinting towards a more security-aware documentation of the configuration options for additional safety.

Phase 2: Manual code auditing & pentesting

This section comments on the code auditing coverage within areas of special interest and documents the steps undertaken during the second phase of the audit against the SPIRE software complex.

- While inspecting the *JWT-SVID* related parts, it was checked if SPIRE properly permits and enforces the algorithms set by the specification and enforces a single header.
- Relating to the server GRPC implementation, it was checked whether the agent authentication requires a valid and signed certificate with agent attestation, with special look at various plugins that make attestation obtainable.
- In the realm of authorization, diverse checks were executed against the *AuthorizeAnyOf*, *lolcaOrAdmin* and *downStream* implementations. Special attention was paid to the certificate check in the Node *authz*/*authn* parts, where a missing check for the expiration of the certificate in the registration API was spotted.
- Further checks were executed against the *spire-agent* and *spire-server* binaries running on the provided test-systems using *checksec* (verifying binary protection flags); a low severity finding was spotted here and filed as SPI-01-001.
- It was examined whether any outdated or vulnerable third-party software dependencies affect SPIRe, for instance linking against *gorm v1.9.9* which is vulnerable to an SQLi that was fixed in software version *v1.9.10*.
- The team also looked at the user-permissions the *spire-agent* and *spire-server* are running with. It has to be noted that the customer confirmed that the *spire-agent* was running with *root* privileges due to the deployment setup that has been provided, whereas - in real-world scenarios - the *spire-agent* must not necessarily run with *root* privileges.

[12] https://github.com/spiffe/spire#security-assessments

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

- Further inspected features included the datastore cache (*dscache*) of the SPIRE server and its locking mechanism using mutexes, as well as the entry cache used for caching registration entries of agents of the SPIRE server and similar processes. No issues have been spotted here.
- Checks were executed against the Trust Bundle Bootstrapping performed by the SPIRE agent for establishing the initial trust. No issues were spotted here.
- Multiple checks were executed against the SPIRE Workload API. The unix domain socket was found to have permissions of *0777*, meaning anyone on the system running the *spire-agent* can *connect()* to the socket and interact with it.
- The developers are also aware that the workload API socket is completely unauthenticated.
- The workload API was found to have no rate-limiting implemented; the developers are aware of the potential risk of a malicious workload attempting to DoS the agent (assuming *discover_workload_path* is set to *true*) by enforcing the agent to calculate the *sha256* checksum of very large binaries during attestation.
- Workload attestation is a crucial component of SPIRE and any ways of subverting workload attestation could result in stealing identities of neighboring workloads. No issues were spotted in this area.
- Special attention was given to the implementation of the *JWT* token verification the software in scope performs; no issues were spotted though.
- The team also inspected the implementation of the node attestators for the agent. Node attestation is best being performed on a case-by-case basis, e.g. leveraging AWS or GCP-based node attestation implies that the computing platform is assumed to be trustworthy, and leveraging Kubernetes for workload attestation implies that the Kubernetes deployment is assumed to be trustworthy.
- The *join_token* is responsible for attesting the agent's identity using a one-time-use pre-shared key. Here, the team checked the generation of the *Join* token, which is using *uuid.newV4().* Its validation on the SPIRE server-side was also reviewed.
- Further checks in this realm included auditing of the code for potential integer overflows / underflows as well as potential race condition vulnerabilities and TOCTOU issues. No discoveries were made.
- Audits were also performed against the *peertracker,* with a particular focus on Linux. One potential file descriptor leak results in a Denial-of-Service situation described in SPI-01-006.
- Cure53 paid attention to the active-active setup between *granola-regional-1* and *granola-regional-2*. Both server instances have the same datastore settings and server configurations. Such a setup is solely achieved through the configuration of all servers in the same trust domain to read and write to the same shared datastore. One of the goals of reviewing such an active-active setup was to

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

identify potential attacks that should only be performed once and how an active-active cluster mitigates the risk of an attacker racing in such scenarios.
- The upstream authority setup between *granola-regional-1*, *granola-regional-2* and *granola-global* has been reviewed but no issues have been identified.
- The team also looked into *go-spiffe*'s particulars in terms of fetching and validating SVID's, but no issues worth-reporting have been spotted.
- Finally, checks were also performed against the Federation features, but no ways of subverting the federation relationship have been spotted.

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. *SPI-01-001*) for the purpose of facilitating any future follow-up correspondence.

SPI-01-003 WP2: Path normalization in Spiffe ID allows impersonation *(Medium)*

It was found that the SPIRE implementation applies unspecified, undocumented and inconsistent path normalization when parsing or constructing the Spiffe ID. The path normalization is occasionally applied by SPIRE and was adopted from the filesystem with respect to anomalies like the current (./) or parent (../) directory.

This allows adversaries to launch a path traversal attack when user-supplied data is embedded within the *path* part of the Spiffe ID. This path traversal attack can be combined with encoded URL entities that are inconsistently decoded. This allows for several bypasses of security checks and could lead to the misidentification or impersonation of another node, agent or server.

Affected File:
support/k8s/k8s-workload-registrar/mode-reconcile/controllers/pod_controller.go

Affected Code:
```
func (r *PodReconciler) makeSpiffeIDForPod(pod *corev1.Pod) *spiretypes.SPIFFEID
{
    var spiffeID *spiretypes.SPIFFEID
    switch r.Mode {
    case PodReconcilerModeServiceAccount:
        spiffeID = r.makeID(path.Join("/ns", pod.Namespace, "sa",
pod.Spec.ServiceAccountName))
    case PodReconcilerModeLabel:
```

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

```
        if val, ok := pod.GetLabels()[r.Value]; ok:
               spiffeID = r.makeID(path.Join("/", val))
    case PodReconcilerModeAnnotation:
        if val, ok := pod.GetAnnotations()[r.Value]; ok:
               spiffeID = r.makeID(path.Join("/", val))
    }
    return spiffeID
}
```

The Kubernetes workload registrar can be configured to use the value of a Kubernetes-specific pod label. If the value of this pod label is partially or fully attacker-controlled, the path normalization applied by *path.Join* can be used to specify an arbitrary identity. A similar approach was used for some node attestator plugins, allowing to configure the template for the attested Spiffe ID to include potentially user-controlled data, such as the AWS instance tags, prior to path normalization.

Affected File:
pkg/common/idutil/spiffeid.go

Affected Code:

```
func AgentURI(trustDomain, p string) *url.URL {
    return &url.URL{
        Scheme: "spiffe",
        Host:   trustDomain,
        Path:   path.Join("spire", "agent", p),
    }
}
```

Attackers could occasionally use URL encoding to the path traversal attack as the path of a Spiffe ID are sometimes URL-decoded by Spiffe with the *Parse* function of Go's built-in *url* package.

Affected File:
pkg/common/idutil/spiffeid.go

Affected Code:

```
func ParseSpiffeID(spiffeID string, mode ValidationMode) (*url.URL, error) {
    u, err := url.Parse(spiffeID)
    [...]

    return normalizeSpiffeIDURL(u), nil
}
```

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

The Spiffe specification leaves the interpretation of the path of a Spiffe ID to the administrator. On the one hand, specifying the Spiffe ID as an URL comes with the comfort and rich presence of many parsers that can be used from all languages. On the other hand, this diversity pairs tightly with differences in the URL parsing, path and character normalization. This can be a covert security pitfall when the verifier or the issuer treats two distinct entities with ambiguous identities.

It is recommended to mention within the Spiffe specification that the interpretation of the Spiffe ID MUST be consistent across all workloads, agents and servers, especially those including any path normalization like URL decoding, Path Normalization or Unicode Normalization. This could be supported by supplying a reference/default interpretation and, perhaps, supplemented with an implementation of parsing the Spiffe-ID in multiple languages.

SPI-01-004 WP2: Server impersonation through legacy node API *(High)*

It was found that the legacy node API suffers from a logical flaw that allows malicious agents to request and receive a x509 peer certificate for other workloads, agents or servers within the same trust domain. The handler of the *FetchX509SVID* method offered by the server's legacy gRPC node API does not properly validate the Spiffe ID but only validates the entry ID associated with certificate signing requests received from authenticated agents.

This handling signifies the risk of malicious or compromised agents performing identity theft or impersonation attacks against any peers of the trust domain. The issue could be abused to perform Man-in-the-Middle (MitM) attacks intercepting sensitive information.

Affected File:
pkg/server/endpoints/node/handler.go

Affected Code:

```
func (h *Handler) buildSVID(ctx context.Context, id string, csr *CSR, regEntries
map[string]*common.RegistrationEntry) (*node.X509SVID, error) {
    entry, ok := regEntries[id]
    if !ok {
        [...]
        return nil, errors.New("not entitled to sign CSR for given ID type")
    }

    svid, err := h.c.ServerCA.SignX509SVID(ctx, ca.X509SVIDParams{
        SpiffeID:  csr.SpiffeID,
        PublicKey: csr.PublicKey,
        TTL:        time.Duration(entry.Ttl) * time.Second,
        DNSList:   entry.DnsNames,
```

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

```
})
```

Log File Excerpt:

```
Jan 24 13:24:52 granola-regional-2 spire-server[1212]: ... msg="This API is
deprecated and will be removed in a future release" method=/spire.api.node.Node/
FetchX509SVID subsystem_name=api
Jan 24 13:24:52 granola-regional-2 spire-server[1212]: ... msg="Signing SVID"
address="54.184.34.192:56706"
caller_id="spiffe://granola-co/spire/agent/aws_iid/006101183245/us-west-2/i-
00a447038625dd336" spiffe_id="spiffe://granola-co/spire/server"
subsystem_name=node_api
Jan 24 13:24:52 granola-regional-2 spire-server[1212]: ... msg="Signed X509
SVID" expiration="2021-01-24T14:24:52Z"
spiffe_id="spiffe://granola-co/spire/server" subsystem_name=ca
```

Reproduction Steps:

1. Receive the entryID of a workload for a targeted *spire-agent*
2. Build the modified *spire-agent* via `make`
3. Run the modified *spire-agent* via CLI as follows:
   - `spire-agent fetchx509 <targetServerAddress> <spiffeIDToBeSigned> <entryID> <certificateDirectory>`
4. For instance, use the following command as *root* from *granola-workload*:
   - `spire-agent fetchx509 'granola-co.spiffe.me:8081' 'spiffe://granola-co/spire/server' '7de77de9-6897-4f0f-917d-45de852dc2ee' /etc/spire/.data`
5. The response of the gRPC request will now be printed. If successful, it will contain the signed x509 certificate. Additionally, the log file of the *spire-server* will match the excerpt from above.

The Spiffe ID of the certificate signing request should be confirmed to match the Spiffe ID of the registration entry that is identified by the entry ID supplied by the agent. Alternatively, support for the legacy node API could be dropped, mitigating this vulnerability in full.

SPI-01-006 WP1: File-descriptor leak inside Linux *peertracker* (Medium)

During a review of the workload API, it was noticed that the *spire-agent* keeps track of workload processes by having open file descriptors of the workload's process */proc/<pid>* entry. Under specific circumstances, the function *newLinuxWatcher()* does not add the opened file descriptor to the returned *linuxWatcher* object.

This can, for example, occur whenever the function *getStarttime()*, invoked by *newLinuxWatcher()*, returns an error message. An attacker could leverage this flaw and cause the agent's *peertracker* to exercise this code path many, many times. Hitting the

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

maximum open file limit set by the Linux operating system would result in DoS. It has to be noted that this value is configurable and potentially different on different Linux systems, however, the provided test-servers had the limit of open file handles per process set to 1024.

Affected File:
spire/pkg/common/peertracker/tracker_linux.go

Affected Code:

```
func newLinuxWatcher(info CallerInfo) (*linuxWatcher, error) {
          // If PID == 0, something is wrong...
          if info.PID == 0 {
                return nil, errors.New("could not resolve caller information")
          }

          procPath := fmt.Sprintf("/proc/%v", info.PID)

          // Grab a handle to proc first since that's the fastest thing we can
   do
          procfd, err := syscall.Open(procPath, syscall.O_RDONLY, 0)
          if err != nil {
                return nil, fmt.Errorf("could not open caller's proc directory:
   %v", err)
          }

          starttime, err := getStarttime(info.PID)
          if err != nil {
                return nil, err
          }

          return &linuxWatcher{
                gid:       info.GID,
                pid:       info.PID,
                procPath:  procPath,
                procfd:    procfd,
                starttime: starttime,
                uid:       info.UID,
          }, nil
}
```

It is important to properly keep track of open file descriptors. Cure53 recommends closing the opened file handle whenever *getstarttime()* fails, in order to eliminate the risk of DoS.

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.

SPI-01-001 WP1: Build-system lacks security flags *(Low)*

While checking the properties of the compiled *spire-agent* and *spire-server* binaries, it has been identified that the resulting binaries do not have the *compiler* time security hardening flags enabled. The following security hardening options - applicable and executable - are missing:

- *PIE (spire-agent and spire-server)*
- *RELRO (spire-agent and spire-server)*
- *Stack Canaries / Stack-Smashing Protection (spire-agent)*
- *FORTIFY_SOURCE (spire-agent)*

A detailed description of the referred security hardening *compiler* flags can be found online[13].

***Shell* excerpt:**
The following PoC demonstrates the lack of *compile* time security hardening flags by using the *checksec.sh*[14] utility on two of the provided server systems.

34.214.21.89 (granola-global - Root Server)

```
root@granola-global:/home/ubuntu/tmp/checksec.sh# ./checksec --proc-all
[...]
COMMAND PID RELRO         STACK CANARY    SECCOMP    NX/PaX      PIE    FORTIFY
[...]
-server 450 Partial RELRO Canary found    No Seccomp NX enabled No PIE Yes
[...]
```

54.149.236.250 (granola-regional-1 - Nested Server)

```
root@granola-regional-1:/home/ubuntu/tmp/checksec.sh# ./checksec --proc-all
[...]
COMMAND PID RELRO         STACK CANARY    SECCOMP    NX/PaX     PIE    FORTIFY
[...]
-agent  446 No RELRO      No canary found No Seccomp NX enabled No PIE No
-server 449 Partial RELRO Canary found    No Seccomp NX enabled No PIE Yes
[...]
```

[13] https://wiki.archlinux.org/index.php/Arch_package_guidelines/Security#Golang
[14] https://github.com/slimm609/checksec.sh

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

Cure53 encourages the use of all existing *compiler* security features in order to raise the bar for attackers who aim to exploit vulnerabilities within SPIRE. The missing features can be enabled by incorporating the following flags in the build-system:

```
export GOFLAGS='-buildmode=pie'
export CGO_CPPFLAGS="-D_FORTIFY_SOURCE=2"
export CGO_LDFLAGS="-Wl,-z,relro,-z,now"
export CGO_LDFLAGS='-fstack-protector'
```

SPI-01-002 WP1: SPIRE server stores private *key.json* world-accessible *(Medium)*

During an audit of the *spire-server*-related source code, it was noticed that the *key.json* file, holding sensitive information such as *JWT-Signer* and x509-CA private keys, is opened and stored with file permissions 0644.

This insecure default file permissions grants read permissions to anyone. It is important to note that the configured umask of the Linux system, where the *spire-server* binary is invoked, gets applied when creating the referred *key.json* file. Thus, the actual file permissions strongly depend on the configured umask value.

Affected File:
spire/pkg/server/plugin/keymanager/disk/disk.go

Affected Code:

```
func writeEntries(path string, entries []*base.KeyEntry) error {
	data := &entriesData{
		Keys: make(map[string][]byte),
	}
	for _, entry := range entries {
		keyBytes, err := x509.MarshalPKCS8PrivateKey(entry.PrivateKey)
		if err != nil {
			return err
		}
		data.Keys[entry.Id] = keyBytes
	}

	jsonBytes, err := json.MarshalIndent(data, "", "\t")
	if err != nil {
		return newError("unable to marshal entries: %v", err)
	}

	if err := diskutil.AtomicWriteFile(path, jsonBytes, 0644); err != nil {
		return newError("unable to write entries: %v", err)
	}
```

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

```
        return nil
}
```

The following output was captured on the *granola-local* host and shows that sensitive information is stored within the referred *keys.json* file.

***Shell* excerpt:**

```
root@granola-global:/etc/spire# cat keys.json
{
        "keys": {
                "JWT-Signer-A":
"MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgnHN6i3MmltAFIunvAfFCGm65d0sxBOt
Xhny6gqf4xQqhRANCAAQ+u3j23JuqxnRHiuWjhuQ1cItJFluxRjTm+HVXENeq6KAX3QgqpxkarVuG+SZ
jS0A0TzoqJHd1M7yMvdD+zR19",
                "JWT-Signer-B":
"MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgt9/m9kbtMmFmTNSVX1OukhId2B3fBIs
FIKTPs4tUiiqhRANCAAQHP6raGmLbnrIQp5FzRSFTrVSEEbLh3tgXSgIxokHYnb9bVK9mJa+rpUjMw7r
l77w7CWhqRKjEvevBRvhVFPNd",
                "x509-CA-A":
"MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgRKQFwCWb/cWN2XM0gkz9P9Rt7O7LeG4
2eJ7Tm7x/tgWhRANCAASI/
Pl5gMAxim1iD0tn0ILvkwz7N5JuyqluIvM4MZ050g8bWsus47jtzzbHi6VT6Xp/
yDLM8fmWQtXsrvBOgYyA",
                "x509-CA-B":
"MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgHtk+zLvKi4gpH1ILq3ywafZ3i/
c4HOez3HV7IoyCwBWhRANCAAQIEDGWWk/HgBDBA1gRcRH01B/
V+pbNdQVp63RrGQ0PQ7P1+Q5ScSD38qs8S2eKmQ1o9/rFdFTjusWqgGF0A/ru"
        }
```

In the example output depicted above, the configured umask on the provided *granola-local* host prevented read access of the *keys.json* file because of the restrictive umask settings.

Sensitive information, such as private-keys, should never be stored with file permissions 0644. 0600 should be used instead, only granting read and write permissions to the owner of the file.

SPI-01-005 WP1: SPIRE links against outdated third-party modules *(Medium)*

While reviewing all third-party dependencies that are linked within SPIRE-v0.12's codebase, it was noticed that the *go.mod* file (the file that lists all dependency requirements and their versions) contains a list of outdated modules. A few noteworthy mentions are enumerated in the list below.

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

Excerpt with outdated modules:

- *github.com/Azure/azure-sdk-for-go*
  - Linked *v44.0.0* instead of *v50.1.0*
- *github.com/mattn/go-sqlite3*
  - Linked *v1.10.0* instead of *v1.14.6*
- *github.com/sirupsen/logrus*
  - Linked *v1.4.2* instead of *v1.7.0*
- *github.com/jinzhu/gorm*
  - Linked *v1.9.9* instead of *v1.9.16*
- etc…

Especially newer versions of the mentioned *gorm* library contain fixes for one CVE[15] that was reported for versions below *1.9.10*. Although the patch for this CVE was reverted at some point (since it can be considered a non-issue), the current situation with the dependency management is not optimal. At present, especially security fixes that happen during version changes will go unnoticed. This is also mentioned in the maturity section in the beginning of this report where potential automated solutions are highlighted. It is recommended to make sure that version changes for dependencies are included in the build process of SPIRE.

SPI-01-007 WP1: Path traversal in Spiffe ID via potentially unsafe *join* token *(Info)*

It was found that the *join* token supplied by the user is then used for the attested Spiffe ID. This could be dangerous if the datastore plugin cuts the *join* token[16] or performs mutations on the *join* tokens before looking them up. This could be abused to bypass the allow-list and perform attacks similar to SPI-01-003.

Affected File:
pkg/server/api/agent/v1/service.go

Affected Code:

```
func (s *Service) attestJoinToken(ctx context.Context, token string)
(*nodeattestor.AttestResponse, error) {
    log := rpccontext.Logger(ctx).WithField(telemetry.NodeAttestorType,
"join_token")

    resp, err := s.ds.FetchJoinToken(ctx, &datastore.FetchJoinTokenRequest{
        Token: token,
    })
    [...]
```

[15] https://www.cvedetails.com/cve/CVE-2019-15562/

[16] An example of a similar flaw in MySQL is illustrated here: https://mathiasbynens.be/notes/mysql-utf8mb4

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

```
    tokenPath := path.Join("spire", "agent", "join_token", token)
```

It is recommended to use the *join* token returned by the datastore instead of using the one sent by the agent. By doing so, implementation flaws of the datastore that cut the *join* token at specific characters or internally before looking them up are mitigated by design.

SPI-01-008 WP1: Anti-SSRF hardening not applied for SDS API *(Info)*

It was found that additional hardening gained through the anti-SSRF token is only applied to the workload API. This introduces the risk of software that runs on the workload being prone to a non-blind SSRF vulnerability, potentially granting attackers access to the unhardened Secret Discovery Service (SDS).

Affected File:
pkg/agent/endpoints/middleware.go

Affected Code:

```
func verifySecurityHeader(ctx context.Context, fullMethod string)
(context.Context, error) {
    if isWorkloadAPIMethod(fullMethod) && !hasSecurityHeader(ctx) {
        return nil, status.Error(codes.InvalidArgument, "security header missing
from request")
    }
    return ctx, nil
}
```

It is recommended to accept but consider this risk in the security section of the documentation pertinent to configuring SDS. Relevant endpoints of the workload API should be listed to have the users additionally informed that the SDS API is unprotected against SSRF attacks.

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

Conclusions

This comprehensive report demonstrates that the SPIRE project, which is the SPIFFE Runtime Environment, has been created with security in mind. The involved members of the Cure53 team, who were commissioned by CNCF to complete this broadly-scope security examination in January 2021, could not find any severe (or *Critical)* security flaws within the SPIRE complex.

Despite an in-depth review of various high impact components of SPIRE, the work asserted the maintenance of high security levels. The meanwhile pretty massive codebase of SPIRE made a solid impression, clearly showcasing that the in-house developers are aware of secure programming principles.

The choice of the programming language, GoLang, positively contributes to the general security posture and reduces the exposure to risks. In addition, it eliminates some bug classes significantly, as is the case with memory corruption issues and the likes often found in C/C++. Cure53 must underline that the SPIRE workload attestation API is a very interesting target for attackers, meaning that it should be protected with uttermost scrutiny. It was not possible to subvert the workload attestation process, however, one potential Denial-of-Service issue has been spotted (SPI-01-006).

The examination revealed that the Go build process has not been taking full advantage of existing binary protection flags hardening, which exposes SPIFFE binaries to potential exploitation vectors (SPI-01-001). Due to the provided setup, an audit of some platform specific plugins, e.g. Kubernetes and docker workload attestation, was only possible from a static code analysis perspective. Considering the maturity of the SPIFFE/SPIRE design concept and its architecture, Cure53 was left with a very good impression on the whole.

To reiterate, the codebase, together with its extensive documentation, is very clean, well-structured and easy to follow. Despite the fact that there are some weaknesses here and there, like outdated third-party libraries (see SPI-01-005), or lacking input validation (as in SPI-01-004), the overall quality of the whole project can be judged as quite mature.

Note that issue SPI-01-004 presents an attractive attack-vector for attackers who wish to escalate the identity of a *spire-agent* into a *spire-server*. Its presence stresses the importance of SPIRE's decision to remove the deprecated legacy node API. While this was found to be an anomaly, SPI-01-001 was present throughout the examined code of SPIRE. This seems to be a more fundamental issue that could have arisen from the SPIFFE specification. The security-relevance of building, parsing and interpreting

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

SPIFFE IDs consistently in all environments of all federated Trust Zones is thereby resonated. In addition, the specification could stress the danger of embedding user input into attested SPIFFE IDs while applying Path Normalization as observed in SPI-01-001 or SPI-01-007.

Judging by the rest of the issues and the complexity of the project, it can be argued that the developers of the reference implementation of SPIFFE have proven their awareness of modern vulnerabilities. The mitigations they have crafted and utilized are capable of reducing the attack preponderance of the SPIRE complex on the whole, as evident from a small number and generally limited numbers of findings. This is especially impressive in the face of the sheer size of the codebase, indicating that there really is not that much to improve from the auditors' perspective. With remediation of the issues highlighted by Cure53 above, the already high level of security would be increased even further.

Finally, it is important to stress the good flow and efficiency of the communication between the testers and the SPIFFE team. No questions were left unanswered and the provided self-security assessments fostered understanding the concepts and security model of the whole project. Additional material regarding potential attacker models and greater worries also served the purpose of streamlining testing efforts. To conclude, while several issues were spotted and documented in this January 2021 project, the overall impressions of the state of security and its documentation are positive. It is clear that the SPIRE project maintainers are on the right track regarding security.

Cure53 would like to thank Andres Vega, Agustín Martínez Fayó, Andrew Harding and Evan Gilman from the SPIRE 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 needs to be extended to The Linux Foundation for sponsoring this project.