Preliminary response to the GNU Taler security audit in Q2/Q3 2020

Christian Grothoff Florian Dold

July 2, 2020

1 Abstract

This is the preliminary response to the source code audit report CodeBlau created for GNU Taler in Q2/Q3 2020. A final response with more details is expected later this year.

2 Management Summary

We thank CodeBlau for their detailed report and thorough analysis. We are particularly impressed that they reported issues against components that were not even in-scope, and also that they found an *interesting* new corner case we had not previously considered. Finally, we also find several of their architectural recommendations to strengthen security to be worthwhile, and while some were already on our long-term roadmap, we will reprioritize our roadmap given their recommendations.

Given our extensive discussions with CodeBlau, we also have the impression that they really took the time to understand the system, and look forward to working with CodeBlau as a competent auditor for GNU Taler in the future.

3 Issues in the exchange

We agree with the issues CodeBlau discovered and both parties believe that they have all been addressed.

4 Issues in the auditor

We appreciate CodeBlau's extensive list of checks the Taler auditor performs, which was previously not documented adequately by us. We agree that the auditor still needs more comprehensive documentation.

As for issue #6416, we agree with the analysis and the proposed fix, even if the implications are not fully clear. It has not yet been implemented as we want to carefully review all of the SQL statements implicated in the resolution and ensure we fully understand the implications.

5 Issues in GNUnet

We agree with the issues CodeBlau discovered and both parties believe that they have all been addressed.

6 General remarks on the code

We understand that writing the code in another programming language may make certain checks for the auditor less work to implement. However, our choice of C is based on the advantages that make it superior to contemporary languages for our use case: relatively low complexity of the language (compared to C++); availability of mature compilers, static and dynamic analysis tools; predictable performance; access to stable and battle-tested libraries; and future-proofness due to portability to older systems as well as new platforms.

We believe creating a parallel implementation in other languages would provide advantages, especially with respect to avoiding "the implementation is the specification"-style issues. However, given limited resources will not make this a priority.

We disagree that all modern software development has embraced the idea that memory errors are to be handled in ways other than terminating or restarting the process. Many programming languages (Erlang, Java) hardly offer any other means of handling out-of-memory situations than to terminate the process. We also insist that Taler *does* handle out-of-memory as it does have code that terminates the process (we do *not* simply ignore the return value from `malloc()` or other allocation functions!). We simply consider that terminating the process (which is run by a hypervisor that will restart the service) is the correct way to handle out-of-memory situations. We also have limits in place that should prevent attackers from causing large amounts of memory to be consumed, and also have code to automatically preemptively restart the process to guard against memory exhaustion from memory fragmentation. Finally, a common problem with abrupt termination may be corrupted files. However, the code mostly only reads from files and limits writing to the Postgres database. Hence, there is no possibility of corrupt files being left behind even in the case of abnormal termination.

7 More specs and documentation code

We agree with the recommendation that the documentation should be improved, and will try to improve it along the lines recommended by CodeBlau.

8 Protocol change: API for uniformly distributed seeds

We agree with the suggestion, have made the necessary changes, and both parties believe that the suggestion has been implemented.

9 Reduce code complexity

9.1 Reduce global variables

While we do not disagree with the general goal to have few global variables, we also believe that there are cases where global variables make sense.

We have already tried to minimize the scope of variables. The remaining few global variables are largely "read-only" configuration data. The report does not point out specific instances that would be particularly beneficial to eliminate. As we continue to work on the code, we will of course evaluate whether the removal of a particular global variable would make the code cleaner.

Also, we want to point out that all global variables we introduce in the exchange are indicated with a prefix `TEH_` in the code, so they are easy to identify as such.

9.2 Callbacks, type p(r)unning

We understand that higher order functions in C can be confusing, but this is also a common pattern to enable code re-use and asynchronous execution which is essential for network applications. We do not believe that we use callbacks *excessively*. Rewriting the code in another language may indeed make this part easier to understand, alas would have other disadvantages as pointed out previously.

9.3 Initializing structs with memset

Using `memset()` first prevents compiler (or valgrind) warnings about using uninitialized memory, possibly hiding bugs. We also do use struct initialization in many cases.

The GNUnet-wrappers are generally designed to be "safer" or "stricter" variants of the corresponding libc functions, and not merely "the same". Hence we do not believe that renaming `GNUNET_malloc` is indicated.

The argument that `memset()`ing first makes the code inherently more obvious also seems fallacious, as it would commonly result in dead stores, which can confuse developers and produce false-positive warnings from static analysis tools.

9.4 NULL pointer handling

The problem with the "goto fail" style error handling is that it rarely results in specific error handling where diagnostics are created that are specific to the error. Using this style of programming encourages developers to create simplistic error handling, which can result in inappropriate error handling logic and also makes it harder to attribute errors to the specific cause.

However, we have no prohibition on using this style of error handling either: if it is appropriate, develpers should make a case-by-case decision as to how to best handle a specific error.

We have made some first changes to how `GNUNET_free()` works in response to the report, and will discuss further changes with the GNUnet development team.

9.5 Hidden security assumptions

We disagree that the assumptions stated are "hidden", as (1) the Taler code has its own checks to warrant that the requirements of the `GNUNET_malloc()` API are satisfied (so enforcement is not limited to the abstraction layer), and (2) the maximum allocation size limit is quite clearly specified in the GNUnet documentation. Also, the GNUnet-functions are not merely

an abstraction layer for portability, but they provided extended semantics that we rely upon. So it is not like it is possible to swap this layer and expect anything to continue to work.

When we use the libjansson library, it is understood that it does not use the GNUnet operations, and the code is careful about this distinction.

9.6 Get rid of boolean function arguments

We agree that this can make the code more readable, and have in some places already changed the code in this way.

10 Structural Recommendation

10.1 Least privilege

It is wrong to say that GNU Taler has "no work done" on privilege separation. For example, the `taler-exchange-dbinit` tool is the only tool that requires CREATE, ALTER and DROP rights on database tables, thus enusring that the "main" process does not need these rights.

We also already had the `taler-exchange-keyup` tool responsible for initializing keys. In response to the audit, we already changed the GNUnet API to make sure that tools do not create keys as a side-effect of trying to read non-existent key files.

We agree with the recommendation on further privilege separation for access to cryptographic keys, and intend to implement this in the near future.

10.2 File system access

The auditor helpers actually only read from the file system, only the LaTeX invocation to compile the final report to PDF inherently needs write access. We do not predict that we will retool LaTeX. Also, the file system access is completely uncritical, as the auditor by design runs on a system that is separate from the production exchange system.

Because that system will not have *any* crypto keys (not even the one of the auditor!), CodeBlau is wrong to assume that reading from or writing to the file system represents a security threat.

We have started to better document the operational requirements on running the auditor.

10.3 Avoid dlopen

Taler actually uses `ltdlopen()` from GNU libtool, which provides compiler flags to conver the dynamic linkage into static linkage. For development, dynamic linkage has many advantages.

We plan to test and document how to build GNU Taler with only static linkage, and will recommend this style of deployment for the Taler exchange for production.

10.4 Reduce reliance on PostgreSQL

CodeBlau's suggestion to use an append-only transaction logging service in addition to the PostgreSQL database is a reasonable suggestion for a production-grade deployment of GNU Taler, as it would allow partial disaster recovery even in the presence of an attacker that has gained write access to the exchange's database.

We are currently still investigating whether the transaction logging should be implemented directly by the exchange service, or via the database's extensible replication mechanism. Any implementation of such an append-only logging mechanism must be carefully designed to ensure it does not negatively impact the exchange's availability and does not interfere with serializability of database transactions. As such we believe that transaction logging can only be provided on a best-effort basis. Fortunately, even a best-effort append-only transaction log would serve to limit the financial damage incurred by the exchange in an active database compromise scenario.