Report for the GNU Taler security audit in Q2/Q3 2020

Version 1.0 from June 29, 2020

Code Blau GmbH
Eitel-Fritz-Str. 22
14129 Berlin

Abstract

This is the report for our source code audit of GNU Taler, specifically the exchange and auditor components. The first part of the udit was performed in April 2020 and focused on coding bugs. The second half of the audit went from end of May till end of June 2020 and focused on the mathematics and crypto soundness and the correctness of the auditor.

contact:	Code Blau GmbH
eMail:	contact@codeblau.de
fon:	+49.30.65004524
fax:	+49.30.55145804
adress:	Eitel-Fritz-Str. 22
loc:	14129 Berlin
url:	http://www.codeblau.de

Contents

1 Management Summary

Code Blau was invited to do a source code audit and design review on the exchange and auditor parts of implementation of the GNU Taler payment system. The scope of the audit was the exchange and auditor component. Third party libraries originally were not in scope, but it was necessary to add some of the core libraries such as GNUNET to it.

The design and implementation of the software looked sound to us. We found several coding issues, but they were easily rectified and present no long-term obstacle.

The code is generally well written and follows deliberate security decisions that were adhered to throughout the code base. While we have some suggestions for improvements, the general code quality leaves little to be desired.

The code complexity is driven up by the ambitions design goals of GNU Taler, including anonymous payments and taxability, that go beyond the design goals of previously attempted systems. We applaud these goals and believe the code complexity to be a necessary evil to achieve them.

1.1 General impressions

We generally tell our customers: Be innovative in what you do, not how you do it. GNU Taler is a shining example for this approach.

GNU Taler relies on proven technology only. It is very innovative in what it tries to achieve, but conservative in how it achieves it. The interactions are done via simple web services, the database is a PostgreSQL, the cryptographic instruments are Chaum blind signatures, Diffie-Hellman and ed25519. All of these are well-understood, in particular the risks associated with them.

Several decisions were a trade-off, however, and an argument could be made for choosing differently. We list those in the recommendations section of this report.

We were impressed by the responsiveness of the developers. Our bugs were generally fixed within 30 minutes or at most a few hours.

Compared to other projects of comparable ambition, the component selection of GNU Taler was done very conservatively. The selected components are either developed in-house or selected deliberately as to introduce as few other dependencies. All of the selected dependencies have a good security and quality track record.

Part I

Code audit results

The following sections contain a selection of the issues that we have found during the code audit[1]. They reflect the kind of issues we have have found in the code base: mostly small issues which are easy to fix (and have been fixed).

2 Issues in the exchange

2.1 6147: buffer too small in `TALER_amount2s`

Note, this issue 6147 has already been resolved.

`TALER_amount2s` prints a monetary amount into a string buffer, including the value itself, an optional fraction part, and the name of the currency. The buffer size is large enough to handle 32-bit values, but we are actually printing 64-bit values. It should be increased.

In src/util/amount.c:

```
624       const char *
625       TALER_amount2s (const struct TALER_Amount *amount)
626       {
627         /* 12 is sufficient for a uint32_t value in decimal; 3 is for ":.\0" */
628         static GNUNET_THREAD_LOCAL char result[TALER_AMOUNT_FRAC_LEN
629                                                + TALER_CURRENCY_LEN + 3 + 12];
```

We are not printing a `uint32_t`, we are printing a 64-bit value.

```
639       char tail[TALER_AMOUNT_FRAC_LEN + 1];
640
641       amount_to_tail (&norm,
642                       tail);
643       GNUNET_snprintf (result,
644                        sizeof (result),
645                        "%s:%llu.%s",
646                        norm.currency,
647                        (unsigned long long) norm.value,
648                        tail);
```

The printing itself will abort if the value does not fit into the buffer, so no buffer overflow vulnerability here.

However, a utility function like this should be able to print the full value range.

[1] The full list given in the Appendix A

2.2 6193: memset for clearing key material

Note, this issue 6193 has already been resolved.

In exchange/src/lib/exchange_api_refresh_common.c:

```
65        /* Finally, clean up a bit...
66        (NOTE: compilers might optimize this away, so this is
67        not providing any strong assurances that the key material
68        is purged.) */
69        memset (md,
70                0,
71                sizeof (struct MeltData));
```

glibc now has a function called `explicit_bzero` for this (since version 2.25 from Feb 2017).

2.3 6195: integer overflow in deserialize_denomination_key

Note, this issue 6195 has already been resolved.

In exchange/src/lib/exchange_api_refresh_common.c:

```
269       memcpy (&be,
270               buf,
271               sizeof (uint32_t));
272       pbuf_size = ntohl (be);
273       if (size < sizeof (uint32_t) + pbuf_size)
```

If 32-bit platforms are supported, the addition in line 273 can cause arithmetic overflow, leading to the range check not triggering and a potential out of bounds memory read (maybe even convertible into a Heartbleed situation):

```
279       dk->rsa_public_key
280         = GNUNET_CRYPTO_rsa_public_key_decode (&buf[sizeof (uint32_t)],
281                                                pbuf_size);
```

3 Issues in the auditor

3.1 What to check for?

The role of the auditor is to verify the correctness of the transactions performed by the exchange. But what checks should the auditor exactly perform?

There is no reference document available defining and describing the particular checks to be performed by the auditor, other than the C implementation itself. In contrast, for the exchange we have at least the PhD thesis of Florian Dold from 2019 – even when the current version of Taler deviates from it slightly – which defines the algorithms and transactions performed by the exchange, customers and merchants.

Nevertheless, we can verify the presence of a large number of sensible tests in the auditor implemented in seperate programs:

taler-helper-auditor-aggregation

- arithmetic inconsistencies
  - disagreement in fee for deposit between auditor and exchange db
  - disagreement in fee for melt between auditor and exchange db
  - disagreement in fee for refund between auditor and exchange db
  - aggregation of fee is negative
  - aggregation (contribution): Expected coin contrinbutions differ: coin value without fee, total deposit without refunds
  - wire out fee is negative
- coin arithmentic inconsistencies
  - refund (merchant) is negative
  - refund (balance) is negative
  - spend > value
- coin denomination signature invalid
- start date before previous end date
- end date after next start date
- wire out inconsistencies in amount
- row inconsistencies
  - wire account given is malformed
  - h(wire) does not match wire
  - failed to compute hash of given wire data
  - database contains wrong hash code for wire details
  - no transaction history for coin claimed in aggregation
  - could not get coin details for coin claimed in aggregation
  - could not find denomination key for coin claimed in aggregation
  - coin denomination signature invalid
  - target of outgoing wire transfer do not match hash of wire from deposit
  - date given in aggregate does not match wire transfer date
  - wire fee signature invalid at given time
  - specified wire address lacks method
  - wire fee unavailable for given time

taler-helper-auditor-coins

- emergency on denomination over loss
  - value of coins deposited exceed value of coins issued
- emergency on number of coins, num missmatch
- arithmetic inconsistencies
  - melt contribution vs. fee
  - melt (cost)
  - refund fee
- row inconsistencies
  - revocation signature invalid
  - denomination key not found

  - denomination key for fresh coin unknown to auditor
  - denomination key for dirty coin unknown to auditor
  - denomination key for deposited coin unkonwn to auditor
- coin validity in known_coin, by checking denomination signatures
- coin validity in melt, by checking signatures
- refresh hanging, zero reveals (harmless)
- verify deposit signature
- verify refund signature
- recoup, check coin
- recoup, check signature
- recoup, denomination not revoked

taler-helper-auditor-deposits

This program only performs the following check: *deposit confirmation missing*.

In the discussion of issue 6413 it is pointed out by the developers that other, deposit-related checks are performed with other tools, for instance `coin_pub` and `coin_sub` are checked in `taler-helper-auditor-coins`. We recommend to add comments to the code for `taler-helper-auditor-deposits` to point to those locations.

Similarly, potential checks regarding *refunds* are performed in various places, but there is no particular helper program to check only refund related issues.

taler-helper-auditor-reserves

- report arithmentic inconsistency
  - closing aggregation fee
  - global escrow balance
- denomination key validity withdraw inconsistencies
- bad signature losses in withdraw
- bad signature losses in recoup
- bad signature losses in recoup-master
- reserve balance, insufficient, losses and gains
- reserve balance, summary wrong
- reserve not closed after expiration time
- could not determine closing fee
- denomination key not found for withdraw
- denomination key not in revocation set for recoup
- closing-fee unavailable
- target account not verified, auditor does not know reserve
- target account does not match origin account

taler-helper-auditor-wire

- check pending
- wire missing
- execution date mismatch
- wire out consistency
- wire transfer not made (yet?)
- receiver accound mismatch
- wire amount does not match
- justification for wire transfer not found
- duplicate subject hash
- duplicate wire offset
- incoming wire transfer claimed by exchange not found
- wire subject does not match
- wire amount does not match
- debig account url does not match
- execution date mismatch
- closing fee above total amount

3.2 6416: Same coin_pub with multiple denom_sigs

Note: This issue filed under category "auditor", as it was found while reading the code of the auditor. However, it should be considered as in issue with the exchange.

Taler uses a cache for fast lookups of `coin_pub` → (`denom_pub`, `denom_sig`) — the table `known_coins`. The table is populated via `TEH_DB_know_coin_transaction` before deposit, melt and recoup operations, i.e. independent of the outcome of those operations.

Consider the scenario where the same `coin_pub` is signed with different denomination keys. The first usage of one of those coins would lock the denomination value in the `known_coins` table. However, it is not clear what would happen if the same `coin_pub` than is used later with a *different* (but also validly signed) denomination for any of the operations.

We have not come up with a particular attack to the advantage of a customer (i.e. gain profit). But maybe leaving the exchange in a confused state that the auditor might notice and complain about could lead to DoS?

We suggest to have the pair (`coin_pub`, `denom_h`) as an index for the table `known_coins` and allow multiple entries with the same `coin_pub` in it.

4 Issues in GNUNET

4.1 6146: GNUNET_STRINGS_absolute_time_to_string is not thread-safe

`GNUNET_STRINGS_absolute_time_to_string` is not thread-safe. It returns a pointer to a static buffer, which is the same for all callers.

If two threads call it simultaneously, the two callers will clobber each other's output into the buffer and return a garbled value.

Recommendation: add a `GNUNET_STRINGS_absolute_time_to_string_r` function that additionally takes a buffer and a buffer size as arguments.

Note, this issue 6146 has already been resolved.

4.2 6149: GNUNET_CRYPTO_eddsa_key_create_from_file is a really bad API

```
72        /**
73        * Create a new private key by reading it from a file. If the
74        * files does not exist, create a new key and write it to the
75        * file. Caller must free return value. Note that this function
76        * can not guarantee that another process might not be trying
77        * the same operation on the same file at the same time.
78        * If the contents of the file
79        * are invalid the old file is deleted and a fresh key is
80        * created.
```

From an ops perspective this means that the exchange needs write access to the key. This is not good.

From a reliability perspective it means this code must deal with several potential race conditions, which makes it very complicated and potentially dangerous.

The API is bad, but the implementation is worse: It uses `access()`, which even its own man page warns against because it creates race conditions.

In the discussion about this bug it was mentioned that GNU Taler already comes with a helper program to replace keys. We therefore recommend removing this code altogether and relying on that helper.

Note, this issue 6149 has already been resolved.

4.3 6152: Using GNUNET_memcmp for comparing public keys

The exchange code uses `GNUNET_memcmp`, which is a dangerous API because the size is implicit.

`GNUNET_memcmp` looks like this:

```
1123        #define GNUNET_memcmp(a, b)       \
1124          ({                              \
1125            const typeof (*b) * _a = (a);  \
1126            const typeof (*a) * _b = (b);  \
1127            memcmp (_a, _b, sizeof(*a)); \
1128          })
```

This carries several risks.

1. if a is a pointer to an array, only the first element is compared.
2. the first two lines are apparently meant to assert that a and b are pointers to the same type, but in C pointers to different types are assignable. In some cases you'd get a compiler warning, but not in all.
3. If we are comparing crypto material, we might be introducing a timing side channel because memcmp will abort after the first unmatched byte.

Note, this issue 6152 has already been resolved.

4.4 6154: Integer overflow in GNUNET_STRINGS_buffer_fill

```
65        GNUNET_STRINGS_buffer_fill (char *buffer,
66                                    size_t size,
67                                    unsigned int count, ...)
68        {
69          size_t needed;
70          va_list ap;
71
72          needed = 0;
73          va_start (ap, count);
74          while (count > 0)
75          {
76            const char *s = va_arg (ap, const char *);
77            size_t slen = strlen (s) + 1;
78
79            GNUNET_assert (slen <= size - needed);
```

The arithmetic can overflow here. That could only happen if the caller is trying to trick you, so it's a matter of defense in depth. But maybe the caller was attacked and the attacker gained enough control to trick this

function and not to gain code execution immediately. In that case, this bug would give her code execution.

Note, this issue 6154 has already been resolved.

5 General remarks on the code

We want to close the part about the code audit with some general remarks on the code of GNU Taler.

The programming language of choice for GNU Taler is plain C, but the code uses patterns from more modern languages, like abstractly iterating over the elements in a container.

C type safety

Clearly, having this abstraction is good, and iterating is a common operation. But the code loses what little type safety C provides by passing function pointers and `void*` around.

Resource exhaustion handling

We were also unhappy that the approach to handling out of memory errors was to assume it does not happen and abort the program if it does. That approach was common in the 1990ies but modern software development has embraced the idea that code will handle all errors gracefully, clean up after itself, and signal failure via error codes. In fact, the 3rd party json library the code uses adheres to this standard. But GNU Taler itself does not.

Error handling

With such an error handling the code size will grow, but auditing can become easier – the audit just has to check whether every statement has proper error handling, and statements without error handling are immediately suspect. We also want to point out that other server software, such as PostgresSQL on which GNU Taler relies on, perform full error handling but still restart after a crash (per default).

Global variables

The reliability and auditability of C code has historically suffered from global variables and lack of namespaces. It is hard to reason about code when the state is kept in global variables and all function calls could in theory modify the global state. That means you always have to keep the whole code in mind to make any statements about what the code actually does.

GNU Taler suffers from this problem, too, but to a lesser degree. GNU Taler keeps state in global variables but declares most of them `static`, which limits their visibility to the current compilation unit. The auditor no longer needs to consider the whole source code for side effects, only the current source code file.

Part II
Recommendations

6 More specs and documentation

During the audit of the exchange code it became clear how valuable the PhD thesis of Florian Dold is. It led to our understanding the cryptography and the protocols and served as a guide for the audit of the implementation of the exchange.

However, while there is documentation available of the REST-API's, and guides for installation and administration of the auditor, a real specification document is missing.

We believe that the whole Taler project would benefit, if it could provide a complete specification of all components, including the auditor. This should include - amongst the existing descriptions of the cryptographic algorithms and protocols - a complete state diagram of all states of a coin, the denominations, the transitions based on the protocols etc.

Having such a complete state diagram

- future changes to the protocol can be easier proposed, explained and reasoned about
- implementations in other programming language can be derived,
- invariants between transitions formulated and corresponding checks implemented
- the particular states and transitions can be referred to from within the implementation of any of the components

7 Protocol change: API for uniformly distributed seeds

Note: this is was reported as feature request 6181 and has been implemented in version 0.8 of GNU Taler.

One implicit assumption in the GNU Taler system is the uniformly random distribution of all public keys, for coins and reserves. The likelihood that two customers generate the same ED25519-keypair for their coins must be negligible. In other words, there is an implicit assumption about the quality of the PRNG's on each of the customer's devices.

However, history shows that this is assumption is false in reality: F.e., due to a severe bug introduced by Debian into their openssl package in 2008[2],

[2] see https://www.debian.org/security/2008/dsa-1571

customers of GNU Taler back then would very probably have had plenty of collisions of those coins. They would have withdrawn them from the exchange successfully (and thereby moving money from their banks to the reserve and paying fees), but not everybody would be able to use them afterwards because of detected "double-spending". And it seems that there is not much the customer could do about this right now.

This is not per se a security problem for the exchange, but certainly an issue for the acceptance of the GNU Taler system for customers and merchants. And as we boldy assume that 300M Europeans are soon going to use GNU Taler, creating billions of coins a year on systems with unknown entropy of their PRNG's, the key distribution will be unknown and probably non-uniform.

As a potential remedy, an exchange could offer an additional API-endpoint `/seed` (GET) to provide random seeds (32bytes maybe) for the wallet software to use as a seed when generating coins. This would establish two things:

1. The exchange would guarantee (and become responsible) to provide uniformly distributed seeds for all customers.
2. The exchange could then statistically/rightfully argue to not accept any claims of collisions of coins.

Note that, depending on the requirements from the regulator, an auditor might also be able to evaluate the entropy of the PRNG of the exchange.

8 Reduce code complexity

The most important measure of code complexity is how much context you need to see whether a piece of code is correct or not.

The GNU Taler code base presents several opportunities in this regard.

8.1 Reduce global variables

If a piece of code calls a function, and gives it two arguments, and those arguments pose no security risk, the auditor can usually skip looking into the function.

Not so if there is global state!

Therefore we recommend getting rid of global state, and passing state around explicitly, using `const` where possible to indicate read-only access to the state.

Note that more modern programming languages like C++ offer more expressiveness in the form of `mutable`.

8.2 Callbacks, type punning

GNU Taler is based on GNUNET, which uses callbacks extensively.

Since the C language does not support C++ templates, these callbacks are operating on `void *` data types, which negates the type safety mechanisms of C.

Note that we have seen no actual bugs in this area. We still make this recommendation because it took us more time to verify correctness of the code than it otherwise would have.

8.3 Initializing structs with memset

The code uses many structs, which are then initialized field by field in the code. The auditor then needs to look at the struct to see if any field was missed, and look for alignment padding that could leak stack data.

We recommend always `memset`ting structs first.

Note that this applies only to structs allocated on the stack, as `GNUNET_malloc` does include an implicit memset. We recommend renaming the macro to `GNUNET_malloc_init0` or so to make that obvious to the casual code reader.

An argument can be made that not having a memset allows the compiler to warn about referencing uninitialized members. However, we consider it more important to have the code be obviously correct. As a compromise we suggest introducing a memset but allow it to be removed by the preprocessor,

so that a special non-default build mode without memset can be done to get the advantage of the compiler warnings.

If there is a choice between gaining something long-term at the expense of making the code less obvious, we recommend always choosing to make the code obvious. In our experience, the long-term disadvantages of non-obvious code outweigh any other advantages eventually, and it is much harder to make code more obvious retroactively.

8.4 NULL pointer handling

One of the most popular patters to deal with error handling in C is to first initialize all local pointers to NULL, then do the work in the function body, including allocation attempts, and then have unified cleanup code in the end after a label called `error:` or `fail:`

The C programming language helps establish this pattern by explicitly stating that `free(NULL)` is legal and does nothing. That simplifies the cleanup code enormously. A typical example would look like this:

```
short* do_something(char* s,int i) {
  char* ibuf = NULL;        // initialize to NULL
  short* res = NULL;

  if (asprintf(&ibuf, "i is %d\n", i) == -1)
    goto fail;

  if (strlen(ibuf) > 10)
    goto fail;

  res = malloc((strlen(ibuf)+1) * 2);
  if (!res)
    goto fail;

  {
    size_t i;
    for (i=0; ibuf[i]!=0; ++i)
      res[i] = ibuf[i];
    res[i] = 0;
  }

  return res;

fail:                       // fail handles all modes of failure
  free(res);
  free(ibuf);

  return NULL;

}
```

This pattern is easy to write and easy to verify.

The GNU Taler source code does not use `free`, it uses `GNUNET_free` instead. `GNUNET_free` will abort the program if the pointer argument was NULL. That may seem like a good idea at first, but it basically moves the logic that was in `free` upwards to all callers, which now need to check for NULL before calling `GNUNET_free`. In that sense, it is an anti-abstraction. It does not reduce but actually increase complexity for the caller.

We note that this has lead to error handling code paths being hand written for each potential failure case, and not having a common unified "goto fail" cleanup area that will handle all failures[3]. This drastically increases the work needed by the developer to write the code, and by the auditor to verify correctness of the code, and it creates opportunities for copy and paste bugs.

We recommend against this practice and would like to point out the massive success the "goto fail" pattern has been in practice in other projects.

Some projects have rules against using `goto`. In those projects, a similar pattern can be established using `do { ... } while (0);` and then using `break;` instead of `goto fail;` to leave the pseudo loop.

8.5 Hidden security assumptions

The code has hidden security assumptions, such as that memory allocations will be limited by a hidden maximum allocation size in the abstraction layer around it. However, this only works if the abstraction layer is used.

The json parser is a third party library that does not use the abstraction layer, and therefore can violate this assumption.

We think that hidden security assumptions are a bad idea in general. If you want to limit memory allocations, make that explicit by using memory pools. These can also improve performance because you do not have to deallocate every single allocation, you can just deallocate the whole pool. Memory pools can also help prevent memory leaks.

8.6 Get rid of boolean function arguments

The code currently has several functions like this:

```
get_coin_transactions(foo, bar, buz, GNUNET_OK, buf);
get_coin_transactions(foo, bar, buz, GNUNET_NO, buf);
```

It improves the code readability to instead move the boolean value into the function name:

```
get_coin_transactions_including_recoup(foo, bur, buz, buf);
get_coin_transactions_without_recoup(foo, bur, buz, buf);
```

[3] See `check_transfers_get_response_ok` for an example.

It can also make auditing the code easier, because now you don't need to skip the cases you were not interested in to begin with.

9 Structural Recommendation

9.1 Least privilege

One of the most important principles in security engineering is the Principle of Least Privilege. Code should run only with the permissions it really needs, and an effort should be made to reduce the permissions it needs, so it can be run with the lowest privileges. If necessary, a process can be split up into a part that requires higher privilege but does very little actual work, and a part that requires less privilege but has most of the risky attack surface (like parsing). This is called Privilege Separation.

GNU Taler has had no work done in that direction. For example, we found code that tries to read a crypto key from disk, but if it finds no key, it will generate a new key and write it to disk. That is good from a convenience perspective, but it is bad from a least privilege perspective, because it means the process needs to have write access to the file system. This is not a huge security risk because being able to read the key is already a full compromise, but it illustrates the problem.

Future work about privilege separation could make sure the generic service has neither read nor write access to any crypto keys.

9.2 File system access

The auditor accesses the file system for reading and writing. Since the actual data is in the database, we think it would make sense to remove all remaining file system accesses in the code base.

Then, the auditor itself could run in an isolated sandbox with no filesystem access at all, for example with `chroot` or more modern Linux container technology like filesystem namespaces or Seccomp.

That would reduce the attack surface for an attacker who manages to take over the auditor process.

The biggest risk for such an attacker would be reading the various crypto keys, so this work would not be sufficient.

9.3 Avoid dlopen

This ties in with the previous point about file system access, but it worth mentioning on its own.

We recommend against allowing loading code modules at run time.

It means an attacker who is able to place something in the filesystem might be able to get it loaded as code.

It also means that during the dlopen some internal data structures of the dynamic linker need to be mapped writable to the process, which exposes

unnecessary attack surface. See RELRO ELF hardening for details.

9.4 Reduce reliance on PostgreSQL

The code currently relies on PostgreSQL not just for data storage but also as enforcement mechanism for security properties like preventing race conditions via database transactions.

This is generally good security practice, but it also carries significant risk. If an attacker manages to take over any service that has database access, that service could potentially corrupt the state in the database.

We suggest having a ”plan B” in case an attacker manages to get into a position that would have allowed write access to the database. We suggest implementing a mechanism for append-only logging of all transactions to a separate machine via an API that only allows one operation: append log record. That way even an attacker with full access to the database cannot retroactively corrupt the records up to the moment of intrusion.

The logging service itself could even use a mechanism like hash chaining to implement a tamper-evident log file.

A List of findings

Id	Priority	Severity	Category	Date Submitted	Summary	Status	Resolution
6146	normal	minor	exchange	2020-04-02	Taler exchange is multithreaded yet uses GNUNET_STRINGS_absolute_time_to_string	resolved	fixed
6147	normal	trivial	exchange	2020-04-02	buffer too small in TALER_amount2s	resolved	fixed
6148	normal	trivial	exchange	2020-04-02	TALER_amount_normalize is O(n) where O(1) would have sufficed	resolved	fixed
6161	normal	feature	exchange	2020-04-06	Suggestion: Do some more signature checks	confirmed	open
6164	normal	minor	exchange	2020-04-07	deposit_cb return value (and code) are confusing	resolved	fixed
6170	normal	minor	exchange	2020-04-08	confusing code in verify_reserve_balance	resolved	fixed
6171	normal	text	exchange	2020-04-09	typo in json diagnostic message in wire_out_cb	resolved	fixed
6172	normal	minor	exchange	2020-04-09	Are 32-bit builds supported?	resolved	fixed
6178	normal	tweak	exchange	2020-04-14	Why do we have length fields in fixed-length structures?	feedback	open
6181	normal	feature	exchange	2020-04-15	Exchange should provide API for uniformly distributed seeds	resolved	fixed
6182	low	text	documentation	2020-04-15	Specifiy the details for FDH in GNU Taler	assigned	open
6187	normal	tweak	exchange	2020-04-17	Redundant copy of dki->issue.properties.fee_deposit	resolved	fixed
6188	none	tweak	exchange	2020-04-17	Lift binary arguments into function names	confirmed	open
6193	normal	minor	exchange	2020-04-21	memset for clearing key material	resolved	fixed
6194	normal	text	exchange	2020-04-21	serialize_denomination_key NULL case looks fishy	resolved	fixed
6195	normal	major	exchange	2020-04-21	integer overflow in deserialize_denomination_key	resolved	fixed
6198	normal	minor	exchange	2020-04-22	language_matches appears to be functionally incorrect	resolved	fixed
6199	normal	trivial	exchange	2020-04-22	loading the terms of service files into memory seems wasteful	confirmed	open
6200	normal	tweak	exchange	2020-04-22	postgres_get_coin_transactions does not allow for multiple deposit transactions	resolved	fixed
6213	low	trivial	exchange	2020-04-23	Better description of TALER_amount.fraction	resolved	fixed
6214	low	tweak	exchange	2020-04-23	Suggestion: Introduce invariants check when dealing with TALER_EXCHANGEDB_TransactionList	resolved	fixed
6215	normal	trivial	exchange	2020-04-23	confusing/wrong overflow check in TALER_string_to_amount	resolved	fixed
6218	normal	tweak	exchange	2020-04-23	integer overflow in buffer_write_urlencode	resolved	fixed
6219	normal	minor	exchange	2020-04-23	integer overflow in calculate_argument_length	resolved	fixed
6373	normal	tweak	exchange	2020-06-10	Simplify check_for_denomination_key by making denomination_key_lookup_by_hash DenominationKeyUse	assigned	open
6413	normal	tweak	auditor	2020-06-24	Not enough checks for deposits	assigned	open
6414	normal	text	auditor	2020-06-24	No checks for refunds table implemented	assigned	open
6416	normal	minor	auditor	2020-06-25	Same coin_pub with multiple denom_sigs - a problem?	assigned	open