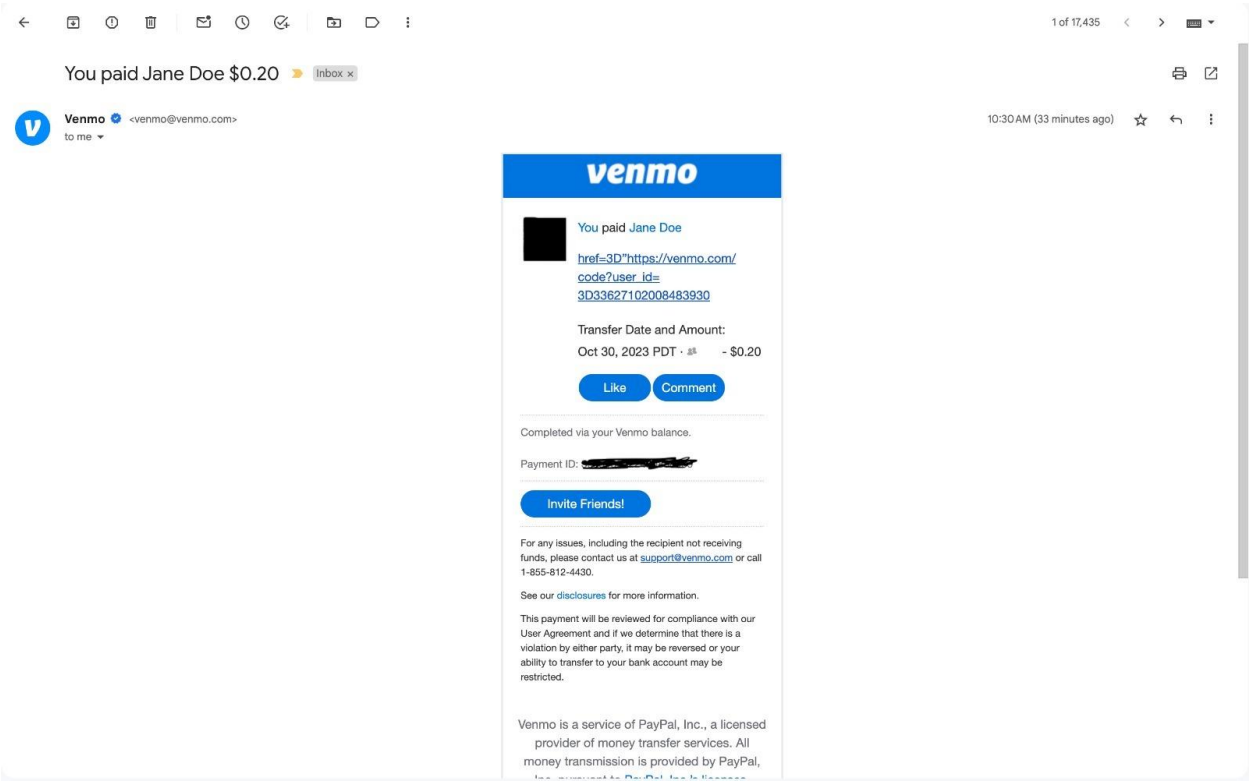
Hey guys, we took a look at your codebase after your announcement on Twitter of the upcoming launch. We noticed quite a few severe vulnerabilities in your circuits and, in general, were quite surprised by the approach of using a regular expression to extract key information from the email. As explained below, with various input manipulation attacks that break the circuits, we would strongly recommend moving away from using regexes and taking a different approach. The attacks below can probably be imposing rules on the ordering of components and regexing additional components. However, in general, regular expressions are very fragile, and the slightest change in Venmo's email templates, which, as we have noticed, happens every couple of months, can introduce attack vectors that can be used to drain the fund. As a simple example, in an email subject, Venmo writes dollar amounts as $1 instead of $1.00 (as does CashApp), then someone could write an amount in the note of their payment and claim an arbitrary amount of USDC. We think that it's very important that any slightest modification in an email template will cause circuits to always fail rather than risk new attack vectors. This opinion was strongly corroborated during our Trail of Bits during the audit of our circuits.

I think that as pioneers in the space, the main hurdle we face is getting users to trust ZK proofs and p2p payment receipts as a way of onramping. Therefore, if any of our protocols get hacked, it will hurt everyone building towards this decentralized future. So, we urge that you move away from using regexes to ensure the complete security of your circuits. We underwent a thorough security audit and hope all other teams building, including ZK Email, do the same. We will be open-sourcing our circuits for email verification and Venmo/CashApp/GooglePay verification shortly so that other teams can build on top of them. As an alternative to using regular expressions, we constrain large sections of HTML to ensure that the slightest change in the template will cause our circuits to fail. This approach was deemed secure by TOB but there may be other approaches to this problem.

We present below a full attack on the current ZK P2P (Ramp) protocol, which allows an attacker to drain arbitrary funds from a seller. We will also highlight below additional attacks we found that break account registration and various individual regex components. We also stress that the circuits are only safe when used with client-side proofs, and we believe the protocol is vulnerable to a man-in-the-middle attack (i.e. if performed by someone's email client or admin) as the intent hash is not included in the email itself.

Attack Example

Consider the scenario where a seller places a large sell order on Ramp. An attacker, through numerous wallets, places intents to collect the entire sell order. The attacker will first learn the user's Venmo ID. The attacker can do this easily as they are shown the user's Venmo handle, furthermore, the seller had to pass the pre-image of the Poseidon hash of their handle when setting up the sell order so the attacker can easily decode the packed ID. The attacker now sends multiple Venmo payments, for each intent, to their friend, for the correct amount, and with note of this form:’ href=3D\"https://venmo.com/code\\?user_id=3D(0|1|2|3|4|5|6|7|8|9|\r|\n|=)+` as seen in an example below:



We notice that when the proof is generated, the circuit will run the following:

```
// VENMO SEND PAYEE ID REGEX
var max_payee_packed_bytes = count_packed(max_payee_len, pack_size); // ceil(max_num_bytes / 7)

signal input venmo_payee_id_idx;
signal reveal_payee_packed[max_payee_packed_bytes];

signal (payee_regex_out, payee_regex_reveal[max_body_bytes]) <== VenmoPayeeId(max_body_bytes)(in_body_padded);
signal is_found_payee <== IsZero()(payee_regex_out);
is_found_payee === 0;

// PACKING
// Special packing to skip over `=\r\n` only for Venmo payee ids
reveal_payee_packed <== ShiftAndPackVenmoPayeeId(max_body_bytes, max_payee_len, pack_size)(payee_regex_reveal, venmo_payee_id_idx);
```

Now, the VenmoPayeeId template will extract the ID inserted in the note, amongst the other ones in the email. A malicious attacker will now use a malicious input for venmo_payee_id_idx to point to the one in the note. This means the attacker can manipulate the circuit to output an arbitrary VenmoIDHash. The attacker will then submit the proof on-chain, the extracted VenmIDHash will be found to be correct and the seller's funds will be drained. Note that a coordinated attack of this kind could be used to drain the entire liquidity from the protocol.

Other Attacks

- Using the Venmo Amount Regex to extract the amount paid is very fragile. For example, any request email will contain the amount in the subject. For example, we can get an email subject of the form: Jane Doe requests $30.00 - help. Firstly, the same attack as above can be used to exploit this, where we put an arbitrary VenmoID in the request message.
- Secondly, and perhaps more importantly, if a seller with an active order: cancels a request, sends a request, pays a request, sends a payment (and perhaps other actions). The recipient of the email can use it to drain funds as each of these emails has the user_id string corresponding to the seller's VenmoID. Since it is very difficult to open two Venmo accounts, this effectively means that a seller will not be able to use their Venmo without risking funds being drained.
- A third attack we found was to do with account registration. Similarly to the above, it's quite easy to get an email with an arbitrary person's VenmoID as the actor_id. Therefore, an attacker can break the venmo_registration circuit and can set an arbitrary venmoIdHash on the account map of the Ramp contract: accounts[msg.sender].venmoIdHash = venmoIdHash; We are not sure how this third attack can be exploited but it does render the account registration trivial and provides a further example of why using regular expressions is unsafe.