ATREDIS PARTNERS

Interchain
ICS20 v2 New Features Assessment
Security Assessment Report

Prepared for Interchain Foundation
September 13, 2024 (version 1.0)

Project Team:
Technical Testing: Bryan Geraghty and Stephen Haywood
Technical Editing: Darren Kemp
Project Management: Sara Bettes

Table of Contents

Engagement Overview

Assessment Components and Objectives

Interchain Foundation ("Interchain") recently engaged Atredis Partners ("Atredis") to perform an assessment of the new features introduced by ICS20 v2 in `ibc-go`. Objectives included validation that Interchain infrastructure and services were developed and deployed with security best practices in mind, and to obtain third party validation that any significant vulnerabilities were identified for remediation.

Testing was performed from August 11 through September 3, 2024, by Bryan Geraghty and Stephen Haywood of the Atredis Partners team, with Sara Bettes providing project management and delivery oversight. For Atredis Partners' assessment methodology, please see Appendix I of this document, and for team biographies, please see Appendix II. Specific testing components and testing tasks are included below.

COMPONENT	ENGAGEMENT TASKS
INTERCHAIN ICS20 V2 NEW FEATURES ASSESSMENT	

	▪ Multiple denomination transfers
	  ○ Test handling of transaction version field
	  ○ Test handling of token array
	  ○ Test construction of single token transfers from a multiple token transfer (Support for v1 packets)
	▪ Path Forwarding
	  ○ Test authorization controls around forwarding a given token
	  ○ Test validation of token issuer
	  ○ Test construction of new path from issuer
	  ○ Test escrow account behavior related to a forward
	  ▪ Test that the tokens either reach the final destination, or if there is an error at any of the hops, they are refunded to the sender
	▪ Protobuf encoding
	  ○ Test unmarshalling and handling of FungibleTokenPacketData and data type

REPORTING AND ANALYSIS	
Analysis and Deliverables	▪ Status Reporting and Realtime Communication ▪ Comprehensive Engagement Deliverable ▪ Engagement Outbrief and Remediation Review

The ultimate goal of the assessment was to provide a clear picture of risks, vulnerabilities, and exposures as they relate to accepted security best practices, such as those created by the National Institute of Standards and Technology (NIST), Open Web Application Security Project (OWASP), or the Center for Internet Security (CIS). Augmenting these, Atredis Partners also draws on its extensive experience in secure development and in testing high-criticality applications and advanced exploitation.

Engagement Tasks

Atredis Partners performed the following tasks, at a high level, for in-scope targets during the engagement.

Application Penetration Testing

For relevant web applications, APIs, and web services, Atredis performed automated and manual application penetration testing of these components, applying generally accepted testing best practices as derived from OWASP and the Web Application Security Consortium (WASC).

Testing was performed from the perspective of an anonymous intruder, identifying scenarios from the perspective of an opportunistic, Internet-based threat actor with no knowledge of the environment, as well as, from the perspective a user working to laterally move through the environment to bypass security restrictions and user access levels.

Where relevant, Atredis Partners utilized both automated fuzzing and fault injection frameworks as well as purpose-built, task-specific testing tools tailored to the application and platforms under review.

Network Protocol Analysis

With the objective of identifying scenarios where the integrity of trusted communications can be diminished or reduced, Atredis Partners reviewed network traffic using various packet flow analysis and packet capture tools to observe in-scope network traffic. Network communications were analyzed for the presence of cleartext communications or scenarios where the integrity of cryptographic communications can be diminished, and Atredis attempted to identify means to bypass or circumvent network authentication or replay communications, as well as other case-dependent means to abuse the environment to disrupt, intercept, or otherwise negatively affect in-scope targets and communications.

Source Code Analysis

Atredis reviewed the in-scope application source code, with an eye for security-relevant software defects. To aid in vulnerability discovery, application components were mapped out and modeled until a thorough understanding of execution flow, code paths, and application design and architecture is obtained. To aid in this process, the assessment team engaged key stakeholders and members of the development team where possible to provide structured walkthroughs and interviews, helping the team rapidly gain an understanding of the application's design and development lifecycle.

Executive Summary

The ICS (inter-chain standard) 20 protocol[1] is used by the Cosmos IBC (inter-blockchain communication) protocol[2] to transfer fungible tokens between chains. This assessment focused on the features introduced by version 2 of the transfer protocol, denoted by the version string, `ics20-2`. Specifically, this included the addition of support for multi-denomination transfers, path forwarding, and protobuf[3] encoding for `FungibleTokenPacketData` (and `FungibleTokenPacketDataV2`) packet data in the `ibc-go`[4] package.

To facilitate testing, the Interchain team supplied Atredis with links to the public GitHub repository, documentation on the new functionality, and instructions for how to run the test suites on local chains.

To test these features, Atredis started by reviewing the source code responsible for boostrapping the `transfer` module and handling the `MsgTransfer` call chain. While reviewing the code for potential security issues, Atredis also checked for coverage of each validation in the unit and end-to-end test suites, and executed them to verify that they were functional. In cases where the reviewed behavior and test coverage was not clear, Atredis discussed the behavior with the Interchain team and created additional test cases to validate the behavior.

Key Conclusions

Ultimately, Atredis did not identify any security issues in the ICS-20 v2 implementation in `ibc-go`. The features were well-designed and implemented, and documentation and test coverage were comprehensive. In cases where the test suites did not cover specific validations, the implementation ultimately behaved as expected.

[1] https://github.com/cosmos/ibc/blob/main/spec/app/ics-020-fungible-token-transfer/README.md
[2] https://github.com/cosmos/ibc
[3] https://protobuf.dev/
[4] https://github.com/cosmos/ibc-go

Platform Overview

The purpose of ICS20 is to transfer fungible tokens between blockchains. Each chain has a native coin (ATOM, ETH, BTC, etc.) and supports any number of IBC coins. An IBC coin is a coin that has traversed chains and contains its chain pathway encoded in a hash. To transfer coins between chains, one or more channels must be established between each chain.

A channel is a connection between two chains and consists of a `PortID` and `ChannelID` that indicates the chain it is going to and a `Counterparty.PortID` and `CounterParty.ChannelID` that indicates the chain it is coming from. To support multi-denomination transfers or unwinding, the channel must be version `ics20-2`. To transfer across multiple channels, a series of hops must be built.

A hop consists of a `PortID` and `ChannelID`, and indicates the next channel to which tokens should be transferred on their way to the final chain. Hops are used to build both a transfer path to the final chain and the trace denomination.

A trace is the pathway back to the original chain and is used for unwinding IBC tokens. To build a trace, a pathway back to the original chain must be built using the `CounterParty.PortID` and `CounterParty.ChannelID` of the hops. A trace is needed to unwind coins from a channel back to their original channels.

Transfer Overview

To conduct a transfer, a `MsgTransfer` object is created and then broadcast onto a compatible chain. The `MsgTransfer` protobuf definition contains the fields for both `ics20-1` and `ics20-2`, but not an explicit version field. How the message is parsed is determined by the version of the channel that the transfer is initiated on. When forwarding tokens across multiple chains the `MsgTransfer` object must contain a `Forwarding` object that lists the hops (channels) the transfer should take on the way to the final chain. If the `MsgTransfer` is unwinding, the `Forwarding` object will have the `Unwind` field set to `true` and there will be no `SourcePort` and `SourceChannel` in the object. The path used for unwinding will be built based on the hops in the `Forwarding` object.

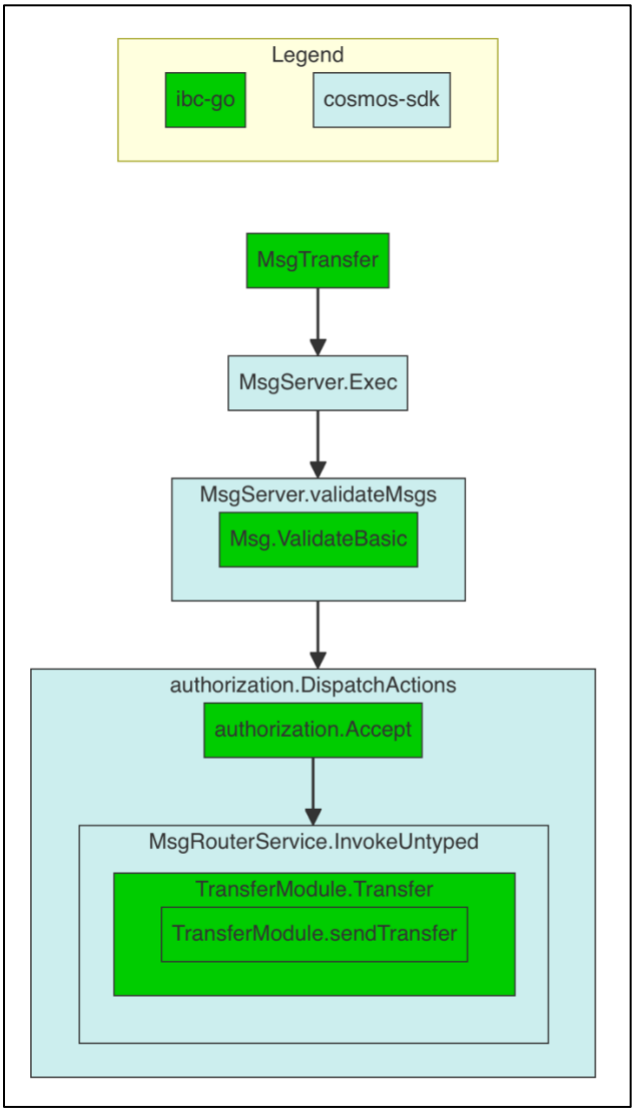
```
// MsgTransfer defines a msg to transfer fungible tokens (i.e Coins) between
// ICS20 enabled chains. See ICS Spec here:
// https://github.com/cosmos/ibc/tree/master/spec/app/ics-020-fungible-token-
transfer#data-structures
 type MsgTransfer struct {
    // the port on which the packet will be sent
    SourcePort string
`protobuf:"bytes,1,opt,name=source_port,json=sourcePort,proto3"
json:"source_port,omitempty"`
```

```
    // the channel by which the packet will be sent
    SourceChannel string
`protobuf:"bytes,2,opt,name=source_channel,json=sourceChannel,proto3"
json:"source_channel,omitempty"`
    // the token to be transferred. this field has been replaced by the tokens
field.
    Token types.Coin `protobuf:"bytes,3,opt,name=token,proto3" json:"token"` //
Deprecated: Do not use.
    // the sender address
    Sender string `protobuf:"bytes,4,opt,name=sender,proto3"
json:"sender,omitempty"`
    // the recipient address on the destination chain
    Receiver string `protobuf:"bytes,5,opt,name=receiver,proto3"
json:"receiver,omitempty"`
    // Timeout height relative to the current block height.
    // The timeout is disabled when set to 0.
    TimeoutHeight types1.Height
`protobuf:"bytes,6,opt,name=timeout_height,json=timeoutHeight,proto3"
json:"timeout_height"`
    // Timeout timestamp in absolute nanoseconds since unix epoch.
    // The timeout is disabled when set to 0.
    TimeoutTimestamp uint64
`protobuf:"varint,7,opt,name=timeout_timestamp,json=timeoutTimestamp,proto3"
json:"timeout_timestamp,omitempty"`
    // optional memo
    Memo string `protobuf:"bytes,8,opt,name=memo,proto3" json:"memo,omitempty"`
    // tokens to be transferred
    Tokens []types.Coin `protobuf:"bytes,9,rep,name=tokens,proto3" json:"tokens"`
    // optional forwarding information
     Forwarding *Forwarding `protobuf:"bytes,10,opt,name=forwarding,proto3"
json:"forwarding,omitempty"`
}
```

`MsgTransfer` object definition

When a transfer is initiated, a chain of interconnected calls in `ibc-go` and `cosmos-sdk`[5] are triggered. The Cosmos SDK provides the base `MsgServer` that performs callbacks into the `ibc-go` module and triggers the necessary authorization checks when processing messages, as shown in the diagram below.

[5] https://github.com/cosmos/cosmos-sdk



Transfer Message Call Chain

When the `MsgServer.Exec` RPC method is called, it first calls `MsgServer.validateMsgs` which calls `ValidateBasic` on each message. `MsgTransfer.ValidateBasic` validates path forwarding, channel identifiers, tokens, the sender address, the receiver address, and the memo field, as shown in the code sample below.

```
func (msg MsgTransfer) ValidateBasic() error {
    if err := msg.validateForwarding(); err != nil {
        return err
    }

    if err := msg.validateIdentifiers(); err != nil {
        return err
    }
```

```
    if len(msg.Tokens) == 0 && !isValidIBCCoin(msg.Token) {
        return errorsmod.Wrap(ibcerrors.ErrInvalidCoins, "either token or token
array must be filled")
    }

    if len(msg.Tokens) != 0 && isValidIBCCoin(msg.Token) {
        return errorsmod.Wrap(ibcerrors.ErrInvalidCoins, "cannot fill both token
and token array")
    }

    if len(msg.Tokens) > MaximumTokensLength {
        return errorsmod.Wrapf(ibcerrors.ErrInvalidCoins, "number of tokens must
not exceed %d", MaximumTokensLength)
    }

    _, err := sdk.AccAddressFromBech32(msg.Sender)
    if err != nil {
        return errorsmod.Wrapf(ibcerrors.ErrInvalidAddress, "string could not be
parsed as address: %v", err)
    }
    if strings.TrimSpace(msg.Receiver) == "" {
        return errorsmod.Wrap(ibcerrors.ErrInvalidAddress, "missing recipient
address")
    }
    if len(msg.Receiver) > MaximumReceiverLength {
        return errorsmod.Wrapf(ibcerrors.ErrInvalidAddress, "recipient address
must not exceed %d bytes", MaximumReceiverLength)
    }
    if len(msg.Memo) > MaximumMemoLength {
        return errorsmod.Wrapf(ErrInvalidMemo, "memo must not exceed %d bytes",
MaximumMemoLength)
    }

    for _, coin := range msg.GetCoins() {
        if err := validateIBCCoin(coin); err != nil {
            return errorsmod.Wrapf(ibcerrors.ErrInvalidCoins, "%s: %s",
err.Error(), coin.String())
        }
    }

    return nil
}
```

`ValidateBasic` function in `ibc-go`

The values for `MaximumReceiverLength`, `MaximumMemoLength`, and `MaximumTokensLength` seem to have been chosen arbitrarily and sending separate message with each of these values set at their maximum was successful. Sending a large number of messages in parallel with all of these values at the maximum length was successful as well, without any noticeable impact on performance.

Of particular concern during this assessment was how forwarding and unwinding was handled and could potentially be tampered with by attackers. Unwind packets can be built and broadcast standalone, so Atredis investigated whether it might be possible to unwind a coin to an arbitrary chain. To do this, a valid packet must be sent to the relayer that specifies how to route to the arbitrary chain and the type of coin that should be moved. However, the message validation checks in ensure that a source port and channel cannot be set in an unwind packet.

```
func (msg MsgTransfer) validateIdentifiers() error {
    if msg.Forwarding.GetUnwind() {
        if msg.SourcePort != "" {
            return errorsmod.Wrapf(ErrInvalidForwarding, "source port must be
empty when unwind is set, got %s instead", msg.SourcePort)
        }
        if msg.SourceChannel != "" {
            return errorsmod.Wrapf(ErrInvalidForwarding, "source channel must be
empty when unwind is set, got %s instead", msg.SourceChannel)
        }

        return nil
    }
```

validateIdentifiers in ibc-go

The only way to specify the path to an arbitrary chain is by configuring the hops in the `Forwarding` object. When the system processes a transfer message, it calculates the denomination of the tokens to transfer based on the trace of the chains in the `token.Denom.Trace`, which won't match the hops in the `Forwarding` object, since the transfer message is built to unwind to an arbitrary chain. When the transfer message is sent to the next hop, there will not be any coins of the correct denomination to forward to the next hop.

After the messages have been validated, `MsgServer.Exec` calls `Authorization.DispatchActions` and returns the results as a `MsgExecResponse` protobuf object. `Authorization.DispatchActions` iterates through the messages, checks the message signers, checks the authorization grants, and if the signer is not the grantee, calls `Authorization.Accept`.

```
func (k Keeper) DispatchActions(ctx context.Context, grantee sdk.AccAddress, msgs
[]sdk.Msg) ([][]byte, error) {
  results := make([][]byte, len(msgs))
  now := k.Environment.HeaderService.HeaderInfo(ctx).Time
  for i, msg := range msgs {
       signers, _, err := k.cdc.GetMsgSigners(msg)
      if err != nil {
          return nil, err
      }
      if len(signers) != 1 {
```

```
            return nil, authz.ErrAuthorizationNumOfSigners
        }
        granter := signers[0]
        // If granter != grantee then check authorization.Accept, otherwise we
        // implicitly accept.
         if !bytes.Equal(granter, grantee) {
            skey := grantStoreKey(grantee, granter, sdk.MsgTypeURL(msg))
            grant, found := k.getGrant(ctx, skey)
            if !found {
                return nil, errorsmod.Wrapf(authz.ErrNoAuthorizationFound,
                    "failed to get grant with given granter: %s, grantee: %s &
msgType: %s ", sdk.AccAddress(granter), grantee, sdk.MsgTypeURL(msg
            }
            if grant.Expiration != nil && grant.Expiration.Before(now) {
                return nil, authz.ErrAuthorizationExpired
            }
             authorization, err := grant.GetAuthorization()
            if err != nil {
                return nil, err
            }
            // pass the environment in the context
            // users on server/v2 are expected to unwrap the environment from the
context
            // users on baseapp can still unwrap the sdk context
             resp, err := authorization.Accept(context.WithValue(ctx,
corecontext.EnvironmentContextKey, k.Environment), msg)
            if err != nil {
                return nil, err
            }
            if resp.Delete {
                err = k.DeleteGrant(ctx, grantee, granter, sdk.MsgTypeURL(msg))
            } else if resp.Updated != nil {
                updated, ok := resp.Updated.(authz.Authorization)
                if !ok {
                    return nil, fmt.Errorf("expected authz.Authorization but got
%T", resp.Updated)
                }
                err = k.updateGrant(ctx, grantee, granter, updated)
            }
            if err != nil {
                return nil, err
            }
            if !resp.Accept {
                return nil, sdkerrors.ErrUnauthorized
            }
        }
```

`DispatchActions` function in `cosmos-sdk`

`TransferAuthorization.Accept` validates the `TransferAuthorization` grant allocations, path forwarding definitions, whether the receiving address is allow-listed, the memo field, and whether the transfer exceeds the grant spend limit, as shown in the code sample below.

```
func (a TransferAuthorization) Accept(goCtx context.Context, msg proto.Message)
(authz.AcceptResponse, error) {
    msgTransfer, ok := msg.(*MsgTransfer)
    if !ok {
        return authz.AcceptResponse{}, errorsmod.Wrap(ibcerrors.ErrInvalidType,
"type mismatch")
    }

     index := getAllocationIndex(*msgTransfer, a.Allocations)
     if index == allocationNotFound {
        return authz.AcceptResponse{}, errorsmod.Wrap(ibcerrors.ErrNotFound,
"requested port and channel allocation does not exist")
    }

     if err := validateForwarding(msgTransfer.Forwarding,
a.Allocations[index].AllowedForwarding); err != nil {
        return authz.AcceptResponse{}, err
    }

    ctx := sdk.UnwrapSDKContext(goCtx)

     if !isAllowedAddress(ctx, msgTransfer.Receiver,
a.Allocations[index].AllowList) {
        return authz.AcceptResponse{}, errorsmod.Wrap(ibcerrors.ErrInvalidAddress,
"not allowed receiver address for transfer")
    }

     if err := validateMemo(ctx, msgTransfer.Memo,
a.Allocations[index].AllowedPacketData); err != nil {
        return authz.AcceptResponse{}, err
    }

    // bool flag to see if we have updated any of the allocations
    allocationModified := false

    // update spend limit for each token in the MsgTransfer
    for _, coin := range msgTransfer.GetCoins() {
        // If the spend limit is set to the MaxUint256 sentinel value, do not
subtract the amount from the spend limit.
        // if there is no unlimited spend, then we need to subtract the amount
from the spend limit to get the limit left
        if
a.Allocations[index].SpendLimit.AmountOf(coin.Denom).Equal(UnboundedSpendLimit())
{
            continue
        }

        limitLeft, isNegative := a.Allocations[index].SpendLimit.SafeSub(coin)
        if isNegative {
            return authz.AcceptResponse{},
errorsmod.Wrapf(ibcerrors.ErrInsufficientFunds, "requested amount of token %s is
more than spend limit"
        }
```

```
        allocationModified = true

        // modify the spend limit with the reduced amount.
        a.Allocations[index].SpendLimit = limitLeft
    }

    // if the spend limit is zero of the associated allocation then we delete it.
    // NOTE: SpendLimit is an array of coins, with each one representing the
remaining spend limit for an
    // individual denomination.
    if a.Allocations[index].SpendLimit.IsZero() {
        a.Allocations = append(a.Allocations[:index], a.Allocations[index+1:]...)
    }

    if len(a.Allocations) == 0 {
        return authz.AcceptResponse{Accept: true, Delete: true}, nil
    }

    if !allocationModified {
        return authz.AcceptResponse{Accept: true, Delete: false, Updated: nil},
nil
    }

    return authz.AcceptResponse{Accept: true, Delete: false, Updated:
&TransferAuthorization{
        Allocations: a.Allocations,
    }}, nil
}
```

Authorization checks in Accept function

Then `Authorization.DispatchActions` calls `MsgRouterService.InvokeUntyped` and returns the response. `MsgRouterService.InvokeUntyped` validates that the protobuf URL and response object type can be created, calls `InvokeTyped`, and returns the response back to `Authorization.DispatchActions`.

```
func (m *msgRouterService) InvokeUntyped(ctx context.Context, msg
gogoproto.Message) (gogoproto.Message, error) {
    messageName := msgTypeURL(msg)
    respName := m.router.ResponseNameByMsgName(messageName)
    if respName == "" {
        return nil, fmt.Errorf("could not find response type for message %s (%T)",
messageName, msg)
    }

    // get response type
    typ := gogoproto.MessageType(respName)
    if typ == nil {
        return nil, fmt.Errorf("no message type found for %s", respName)
    }
    msgResp, ok := reflect.New(typ.Elem()).Interface().(gogoproto.Message)
    if !ok {
        return nil, fmt.Errorf("could not create response message %s", respName)
    }

     return msgResp, m.InvokeTyped(ctx, msg, msgResp)
}
```

InvokeUntyped in cosmos-sdk

`MsgRouterService.InvokeTyped` retrieves the protobuf URL for the message type, calls `MsgServiceRouter.HybridHandlerByMsgName`, invokes the handler, and returns the results back to `MsgRouterService.InvokeUntyped`.

```
// InvokeTyped execute a message and fill-in a response.
// The response must be known and passed as a parameter.
// Use InvokeUntyped if the response type is not known.
func (m *msgRouterService) InvokeTyped(ctx context.Context, msg, resp
gogoproto.Message) error {
     messageName := msgTypeURL(msg)
     handler := m.router.HybridHandlerByMsgName(messageName)
     if handler == nil {
        return fmt.Errorf("unknown message: %s", messageName)
     }

     return handler(ctx, msg, resp)
}
```

InvokeTyped in cosmos-sdk

The `Transfer` handler verifies that sending is enabled for the given coins, that the sender is not blocked, handles coin unwinding, and sends the transfer packet.

```
// Transfer defines an rpc handler method for MsgTransfer.
```

```
func (k Keeper) Transfer(goCtx context.Context, msg *types.MsgTransfer)
(*types.MsgTransferResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    if !k.GetParams(ctx).SendEnabled {
        return nil, types.ErrSendDisabled
    }

    sender, err := sdk.AccAddressFromBech32(msg.Sender)
    if err != nil {
        return nil, err
    }

    coins := msg.GetCoins()

    if err := k.bankKeeper.IsSendEnabledCoins(ctx, coins...); err != nil {
        return nil, errorsmod.Wrapf(types.ErrSendDisabled, err.Error())
    }

     if k.isBlockedAddr(sender) {
        return nil, errorsmod.Wrapf(ibcerrors.ErrUnauthorized, "%s is not allowed
to send funds", sender)
    }

     if msg.Forwarding.GetUnwind() {
        msg, err = k.unwindHops(ctx, msg)
        if err != nil {
            return nil, err
        }
    }

     sequence, err := k.sendTransfer(
        ctx, msg.SourcePort, msg.SourceChannel, coins, sender, msg.Receiver,
msg.TimeoutHeight, msg.TimeoutTimestamp,
        msg.Memo, msg.Forwarding.GetHops())
    if err != nil {
        return nil, err
    }

    k.Logger(ctx).Info("IBC fungible token transfer", "tokens", coins, "sender",
msg.Sender, "receiver", msg.Receiver)

    return &types.MsgTransferResponse{Sequence: sequence}, nil
}
```

`Transfer` function in `ibc-go`

When unwinding tokens, the tokens are sent back to the previous chain as defined in the forwarding hops. Each chain will remove the IBC tokens created for its chain and will send the packet back to the previous hop. When the packet gets back to the original chain, it will return the escrowed native

tokens back to the sender. The coin denomination on the final chain is the SHA256 hash of the native denomination of the sending chain and the transfer path string.

```
// unwindHops unwinds the hops present in the tokens denomination and returns the
message modified to reflect
// the unwound path to take. It assumes that only a single token is present (as
this is verified in ValidateBasic)
// in the tokens list and ensures that the token is not native to the chain.
func (k Keeper) unwindHops(ctx sdk.Context, msg *types.MsgTransfer)
(*types.MsgTransfer, error) {
  unwindHops, err := k.getUnwindHops(ctx, msg.GetCoins())
  if err != nil {
      return nil, err
  }

  // Update message fields.
  msg.SourcePort, msg.SourceChannel = unwindHops[0].PortId,
unwindHops[0].ChannelId
  msg.Forwarding.Hops = append(unwindHops[1:], msg.Forwarding.Hops...)
  msg.Forwarding.Unwind = false

  // Message is validate again, this would only fail if hops now exceeds maximum
allowed.
  if err := msg.ValidateBasic(); err != nil {
      return nil, err
  }
  return msg, nil
}

// getUnwindHops returns the hops to be used during unwinding. If coins consists
of more than
// one coin, all coins must have the exact same trace, else an error is returned.
getUnwindHops
// also validates that the coins are not native to the chain.
func (k Keeper) getUnwindHops(ctx sdk.Context, coins sdk.Coins) ([]types.Hop,
error) {
  // Sanity: validation for MsgTransfer ensures coins are not empty.
  if len(coins) == 0 {
      return nil, errorsmod.Wrap(types.ErrInvalidForwarding, "coins cannot be
empty")
  }

  token, err := k.tokenFromCoin(ctx, coins[0])
  if err != nil {
      return nil, err
  }

  if token.Denom.IsNative() {
      return nil, errorsmod.Wrap(types.ErrInvalidForwarding, "cannot unwind a
native token")
  }

  unwindTrace := token.Denom.Trace
```

```
  for _, coin := range coins[1:] {
      token, err := k.tokenFromCoin(ctx, coin)
      if err != nil {
          return nil, err
      }

      // Implicitly ensures coin we're iterating over is not native.
      if !slices.Equal(token.Denom.Trace, unwindTrace) {
          return nil, errorsmod.Wrap(types.ErrInvalidForwarding, "cannot unwind
tokens with different traces.")
      }
  }

  return unwindTrace, nil
}
```

`unwindHops` and `getUnwindHops` functions in `ibc-go`

After the unwinding has been performed, the `Transfer` handler calls `sendTransfer`. A transfer can fail because there is an `ics20-1` channel on the forwarding path or because the transfer was not added to the chain within the timeout height or timeout time. When a transfer fails, the relayer sends an error message to the previous chain and unwinds the escrowed tokens to the previous chain. This pattern continues until the tokens are back on their original chain.

```
func (k Keeper) sendTransfer(
  ctx sdk.Context,
  sourcePort,
  sourceChannel string,
  coins sdk.Coins,
  sender sdk.AccAddress,
  receiver string,
  timeoutHeight clienttypes.Height,
  timeoutTimestamp uint64,
  memo string,
  hops []types.Hop,
) (uint64, error) {
  channel, found := k.channelKeeper.GetChannel(ctx, sourcePort, sourceChannel)
  if !found {
      return 0, errorsmod.Wrapf(channeltypes.ErrChannelNotFound, "port ID (%s)
channel ID (%s)", sourcePort, sourceChannel)
  }

  appVersion, found := k.ics4Wrapper.GetAppVersion(ctx, sourcePort, sourceChannel)
  if !found {
      return 0, errorsmod.Wrapf(ibcerrors.ErrInvalidRequest, "application version
not found for source port: %s and source channel: %s", sourcePort, sourceChannel)
  }

  if appVersion == types.V1 {
```

```
        // ics20-1 only supports a single coin, so if that is the current version,
we must only process a single coin.
        if len(coins) > 1 {
            return 0, errorsmod.Wrapf(ibcerrors.ErrInvalidRequest, "cannot transfer
multiple coins with %s", types.V1)
        }

        // ics20-1 does not support forwarding, so if that is the current version,
we must reject the transfer.
        if len(hops) > 0 {
            return 0, errorsmod.Wrapf(ibcerrors.ErrInvalidRequest, "cannot forward
coins with %s", types.V1)
        }
    }

    destinationPort := channel.Counterparty.PortId
    destinationChannel := channel.Counterparty.ChannelId

    // begin createOutgoingPacket logic
    // See spec for this logic:
https://github.com/cosmos/ibc/tree/master/spec/app/ics-020-fungible-token-
transfer#packet-relay

    tokens := make([]types.Token, 0, len(coins))

    for _, coin := range coins {
        // Using types.UnboundedSpendLimit allows us to send the entire balance of a
given denom.
        if coin.Amount.Equal(types.UnboundedSpendLimit()) {
            coin.Amount = k.bankKeeper.GetBalance(ctx, sender, coin.Denom).Amount
        }
        token, err := k.tokenFromCoin(ctx, coin)
        if err != nil {
            return 0, err
        }

        // NOTE: SendTransfer simply sends the denomination as it exists on its own
        // chain inside the packet data. The receiving chain will perform denom
        // prefixing as necessary.

        // if the denom is prefixed by the port and channel on which we are sending
        // the token, then we must be returning the token back to the chain they
originated from
        if token.Denom.HasPrefix(sourcePort, sourceChannel) {
            // transfer the coins to the module account and burn them
            if err := k.bankKeeper.SendCoinsFromAccountToModule(
                ctx, sender, types.ModuleName, sdk.NewCoins(coin),
            ); err != nil {
                return 0, err
            }

            if err := k.bankKeeper.BurnCoins(
                ctx, types.ModuleName, sdk.NewCoins(coin),
            ); err != nil {
```

```
                // NOTE: should not happen as the module account was
                // retrieved on the step above and it has enough balance
                // to burn.
                panic(fmt.Errorf("cannot burn coins after a successful send to a
module account: %v", err))
            }
        } else {
            // obtain the escrow address for the source channel end
            escrowAddress := types.GetEscrowAddress(sourcePort, sourceChannel)
            if err := k.escrowCoin(ctx, sender, escrowAddress, coin); err != nil {
                return 0, err
            }
        }

        tokens = append(tokens, token)
    }

    packetDataBytes, err := createPacketDataBytesFromVersion(appVersion,
sender.String(), receiver, memo, tokens, hops)
    if err != nil {
        return 0, err
    }

    sequence, err := k.ics4Wrapper.SendPacket(ctx, sourcePort, sourceChannel,
timeoutHeight, timeoutTimestamp, packetDataBytes)
    if err != nil {
        return 0, err
    }

    events.EmitTransferEvent(ctx, sender.String(), receiver, tokens, memo, hops)

    telemetry.ReportTransfer(sourcePort, sourceChannel, destinationPort,
destinationChannel, tokens)

    return sequence, nil
}
```

sendTransfer function in ibc-go

When a packet is received by the `OnRecvPacket` function, the packet is validated, and then additional logic checks are conducted to make sure the receiver is allowed to receive the transfer and to ensure the token amount being transferred is valid. Any errors received will result in an `Acknowledgement_Error` packet being sent back to the sender.

```
func (k Keeper) OnRecvPacket(ctx sdk.Context, packet channeltypes.Packet, data
types.FungibleTokenPacketDataV2) error {
    // validate packet data upon receiving
    if err := data.ValidateBasic(); err != nil {
        return errorsmod.Wrapf(err, "error validating ICS-20 transfer packet
data")
    }
```

```
    if !k.GetParams(ctx).ReceiveEnabled {
        return types.ErrReceiveDisabled
    }

    receiver, err := k.getReceiverFromPacketData(data)
    if err != nil {
        return err
    }

    if k.isBlockedAddr(receiver) {
        return errorsmod.Wrapf(ibcerrors.ErrUnauthorized, "%s is not allowed to
receive funds", receiver)
    }

    receivedCoins := make(sdk.Coins, 0, len(data.Tokens))
    for _, token := range data.Tokens {
        // parse the transfer amount
        transferAmount, ok := sdkmath.NewIntFromString(token.Amount)
        if !ok {
            return errorsmod.Wrapf(types.ErrInvalidAmount, "unable to parse
transfer amount: %s", token.Amount)
      }

      // This is the prefix that would have been prefixed to the denomination
      // on sender chain IF and only if the token originally came from the
      // receiving chain.
      //
      // NOTE: We use SourcePort and SourceChannel here, because the counterparty
      // chain would have prefixed with DestPort and DestChannel when originally
      // receiving this token.
      if token.Denom.HasPrefix(packet.GetSourcePort(), packet.GetSourceChannel())
{
          // sender chain is not the source, unescrow tokens

          // remove prefix added by sender chain
          token.Denom.Trace = token.Denom.Trace[1:]

          coin := sdk.NewCoin(token.Denom.IBCDenom(), transferAmount)

          escrowAddress := types.GetEscrowAddress(packet.GetDestPort(),
packet.GetDestChannel())
          if err := k.unescrowCoin(ctx, escrowAddress, receiver, coin); err != nil
{
              return err
          }

          // Appending token. The new denom has been computed
          receivedCoins = append(receivedCoins, coin)
      } else {
          // sender chain is the source, mint vouchers

          // since SendPacket did not prefix the denomination, we must add the
destination port and channel to the trace
```

```
            trace := []types.Hop{types.NewHop(packet.DestinationPort,
packet.DestinationChannel)}
            token.Denom.Trace = append(trace, token.Denom.Trace...)

            if !k.HasDenom(ctx, token.Denom.Hash()) {
                k.SetDenom(ctx, token.Denom)
            }

            voucherDenom := token.Denom.IBCDenom()
            if !k.bankKeeper.HasDenomMetaData(ctx, voucherDenom) {
                k.setDenomMetadata(ctx, token.Denom)
            }

            events.EmitDenomEvent(ctx, token)

            voucher := sdk.NewCoin(voucherDenom, transferAmount)

            // mint new tokens if the source of the transfer is the same chain
            if err := k.bankKeeper.MintCoins(
                ctx, types.ModuleName, sdk.NewCoins(voucher),
            ); err != nil {
                return errorsmod.Wrap(err, "failed to mint IBC tokens")
            }

            // send to receiver
            moduleAddr := k.authKeeper.GetModuleAddress(types.ModuleName)
            if err := k.bankKeeper.SendCoins(
                ctx, moduleAddr, receiver, sdk.NewCoins(voucher),
            ); err != nil {
                return errorsmod.Wrapf(err, "failed to send coins to receiver %s",
receiver.String())
            }

            receivedCoins = append(receivedCoins, voucher)
        }
    }

    if data.HasForwarding() {
        // we are now sending from the forward escrow address to the final receiver
address.
        if err := k.forwardPacket(ctx, data, packet, receivedCoins); err != nil {
            return err
        }
    }

    telemetry.ReportOnRecvPacket(packet, data.Tokens)

    // The ibc_module.go module will return the proper ack.
    return nil
}
```

OnRecvPacket in ibc-go

If the message has forwarding configured, it will be forwarded to the next hop through a new `MsgTransfer` message.

```
// forwardPacket forwards a fungible FungibleTokenPacketDataV2 to the next hop in
the forwarding path.
func (k Keeper) forwardPacket(ctx context.Context, data
types.FungibleTokenPacketDataV2, packet channeltypes.Packet, receivedCoins
sdk.Coins) error {
  var nextForwardingPath *types.Forwarding
  if len(data.Forwarding.Hops) > 1 {
      // remove the first hop since we are going to send to the first hop now and
we want to propagate the rest of the hops to the receiver
      nextForwardingPath = types.NewForwarding(false, data.Forwarding.Hops[1:]...)
  }

  // sending from module account (used as a temporary forward escrow) to the
original receiver address.
  sender := k.authKeeper.GetModuleAddress(types.ModuleName)

  msg := types.NewMsgTransfer(
      data.Forwarding.Hops[0].PortId,
      data.Forwarding.Hops[0].ChannelId,
      receivedCoins,
      sender.String(),
      data.Receiver,
      clienttypes.ZeroHeight(),
      packet.TimeoutTimestamp,
      data.Forwarding.DestinationMemo,
      nextForwardingPath,
  )

  resp, err := k.Transfer(ctx, msg)
  if err != nil {
      return err
  }

  k.setForwardedPacket(ctx, data.Forwarding.Hops[0].PortId,
data.Forwarding.Hops[0].ChannelId, resp.Sequence, packet)
  return nil
}
```

`forwardPacket` function in `ibc-go`

Testing Approach

To validate code paths and specific behavior, Atredis added traps to the code and executed end-to-end tests to verify that they failed as expected. These messages also allowed Atredis to verify message values.

```
diff --git a/modules/apps/transfer/types/transfer_authorization.go
b/modules/apps/transfer/types/transfer_authorization.go
index 6e166fe63..218cc4154 100644
--- a/modules/apps/transfer/types/transfer_authorization.go
+++ b/modules/apps/transfer/types/transfer_authorization.go
@@ -37,6 +37,7 @@ func (TransferAuthorization) MsgTypeURL() string {

 // Accept implements Authorization.Accept.
 func (a TransferAuthorization) Accept(goCtx context.Context, msg proto.Message)
(authz.AcceptResponse, error) {
+       return authz.AcceptResponse{}, errorsmod.Wrap(ibcerrors.ErrInvalidType,
"Atredis TransferAuthorization Trap")
        msgTransfer, ok := msg.(*MsgTransfer)
        if !ok {
                return authz.AcceptResponse{},
errorsmod.Wrap(ibcerrors.ErrInvalidType, "type mismatch")
```

Trap added to verify code path

The following sample shows an example of a triggered failure that includes all of the contextual values.

```
=== RUN
TestAuthzTransferTestSuite/TestAuthz_MsgTransfer_Succeeds/broadcast_MsgGrant
=== NAME  TestAuthzTransferTestSuite/TestAuthz_MsgTransfer_Succeeds
    tx.go:78: blocks created on chain chain-1
=== RUN
TestAuthzTransferTestSuite/TestAuthz_MsgTransfer_Succeeds/broadcast_MsgExec_for_ib
c_MsgTransfer
=== NAME  TestAuthzTransferTestSuite/TestAuthz_MsgTransfer_Succeeds
    tx.go:78: blocks created on chain chain-1
    tx.go:118:
                Error Trace:
/root/Documents/Clients/Interchain/ICS20v2Assessment/source/ibc-
go/e2e/testsuite/tx.go:118

/root/Documents/Clients/Interchain/ICS20v2Assessment/source/ibc-
go/e2e/tests/transfer/authz_test.go:149
                Error:          Not equal:
                                expected: 0xc
                                actual  : 0x0
                Test:
TestAuthzTransferTestSuite/TestAuthz_MsgTransfer_Succeeds
                Messages:       code: 12
                                codespace: ibc
                                data: ""
                                events:
                                - attributes:
                                  - index: true
                                    key: fee
                                    value: ""
                                  - index: true
                                    key: fee_payer
```

```
                                    value:
cosmos1x6y0ekzdca3g9nlys0u0cp3ywqlzrpmtxwhmw9
                                  type: tx
                                - attributes:
                                  - index: true
                                    key: acc_seq
                                    value:
cosmos1x6y0ekzdca3g9nlys0u0cp3ywqlzrpmtxwhmw9/0
                                  type: tx
                                - attributes:
                                  - index: true
                                    key: signature
                                    value:
+YyYSTiZEPsoATGPCulE2S2X4ZcDg6vPZ/B+WG/j0vk32da/+YK4Vn7AXINUqAoWJcEQ6TwegTI6YZJ6b3
0kQQ==
                                  type: tx
                                gas_used: "48673"
                                gas_wanted: "5000000000"
                                height: "104"
                                info: ""
                                logs: []
                                raw_log: 'failed to execute message; message
index: 0: Atredis TransferAuthorization
                                  Trap: invalid type'
                                timestamp: "2024-08-22T20:16:31Z"
                                tx:
                                  '@type': /cosmos.tx.v1beta1.Tx
                                  auth_info:
                                    fee:
                                      amount: []
                                      gas_limit: "5000000000"
                                      granter: ""
                                      payer: ""
                                    signer_infos:
                                    - mode_info:
                                        single:
                                          mode: SIGN_MODE_DIRECT
                                      public_key:
                                        '@type': /cosmos.crypto.secp256k1.PubKey
                                        key:
A1ysKghmCHEuzYx2HOHFtzq2Q3kSb8OIi2Ofo26PBJzG
                                      sequence: "0"
                                    tip: null
                                  body:
                                    extension_options: []
                                    memo: interchaintest
                                    messages:
                                    - '@type': /cosmos.authz.v1beta1.MsgExec
                                      grantee:
cosmos1x6y0ekzdca3g9nlys0u0cp3ywqlzrpmtxwhmw9
                                      msgs:
                                      - '@type':
/ibc.applications.transfer.v1.MsgTransfer
                                        forwarding: null
```

```
                                        memo: ""
                                        receiver:
cosmos1u7kk55nnvdpfuhzjrgcy2r26347z663yaxu9cm
                                        sender:
cosmos1s29gch529nav0kh409umqqweveqpspd5vd53p0
                                        source_channel: channel-0
                                        source_port: transfer
                                        timeout_height:
                                          revision_height: "1102"
                                          revision_number: "2"
                                        timeout_timestamp: "0"
                                        token:
                                          amount: "0"
                                          denom: ""
                                        tokens:
                                        - amount: "10000"
                                          denom: atoma
                                      non_critical_extension_options: []
                                      timeout_height: "0"
                                    signatures:
                                    -
+YyYSTiZEPsoATGPCulE2S2X4ZcDg6vPZ/B+WG/j0vk32da/+YK4Vn7AXINUqAoWJcEQ6TwegTI6YZJ6b3
0kQQ==
                                  txhash:
F19E6E208B529A905815DAF659B4A83604C86E2D281DBC79217BEF45616B6643
=== NAME
TestAuthzTransferTestSuite/TestAuthz_MsgTransfer_Succeeds/broadcast_MsgExec_for_ib
c_MsgTransfer
    testing.go:1576: test executed panic(nil) or runtime.Goexit: subtest may have
called FailNow on a parent test
```

Trap triggered by end-to-end test

After the code paths had been verified, Atredis added test cases for security assumptions that weren't explicitly covered. An example that tests whether forward slashes can be included in the Port ID is shown below. Atredis will provide the source code for end-to-end and unit tests that were developed during this engagement in a separate deliverable.

```
diff --git a/modules/core/04-channel/types/msgs_test.go b/modules/core/04-
channel/types/msgs_test.go
index e660cf39b..94697fa0f 100644
--- a/modules/core/04-channel/types/msgs_test.go
+++ b/modules/core/04-channel/types/msgs_test.go
@@ -164,6 +164,26 @@ func (suite *TypesTestSuite)
TestMsgChannelOpenInitValidateBasic() {
                                        host.DefaultMaxPortCharacterLength,
                                    ), "invalid port ID"),
                }, 
+               {
+                       "port id contains forward slash",
```

```
+                       types.NewMsgChannelOpenInit("test/port", version,
types.ORDERED, connHops, cpportid, addr),
+                       errorsmod.Wrap(
+                               errorsmod.Wrapf(
+                                       host.ErrInvalidID,
+                                       "identifier %s cannot contain separator
'/'",
+                                       "test/port",
+                               ), "invalid port ID"),
+               },
+               {
+                       "port id contains url encoded forward slash",
+                       types.NewMsgChannelOpenInit("test%2fport", version,
types.ORDERED, connHops, cpportid, addr),
+                       errorsmod.Wrap(
+                               errorsmod.Wrapf(
+                                       host.ErrInvalidID,
+                                       "identifier %s must contain only
alphanumeric or the following characters: '.', '_', '+', '-', '#', '[', ']', '<',
'>'",
+                                       "test%2fport",
+                               ), "invalid port ID"),
+               },
                {
                        "port id contains non-alpha",
                        types.NewMsgChannelOpenInit(invalidPort, version,
types.ORDERED, connHops, cpportid, addr),
```

New test cases for slashes in port ID

The example below shows this test being executed and passing in the unit test suite.

```
# go test -v -run TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic
=== RUN   TestTypesTestSuite
=== RUN   TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic
=== RUN   TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/success
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/success:_empty_version
=== RUN   TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_short_port_id
=== RUN   TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_long_port_id
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/port_id_contains_forward_sl
ash
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/port_id_contains_url_encode
d_forward_slash
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/port_id_contains_non-alpha
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/invalid_channel_order
```

```
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/connection_hops_more_than_1
_
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_short_connection_id
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_long_connection_id
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/connection_id_contains_non-
alpha
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/invalid_counterparty_port_i
d
=== RUN
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/channel_not_in_INIT_state
--- PASS: TestTypesTestSuite (0.16s)
    --- PASS: TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic (0.15s)
        --- PASS: TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/success
(0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/success:_empty_version
(0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_short_port_id (0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_long_port_id (0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/port_id_contains_forward_sl
ash (0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/port_id_contains_url_encode
d_forward_slash (0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/port_id_contains_non-alpha
(0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/invalid_channel_order
(0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/connection_hops_more_than_1
_ (0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_short_connection_id
(0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/too_long_connection_id
(0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/connection_id_contains_non-
alpha (0.00s)
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/invalid_counterparty_port_i
d (0.00s)
```

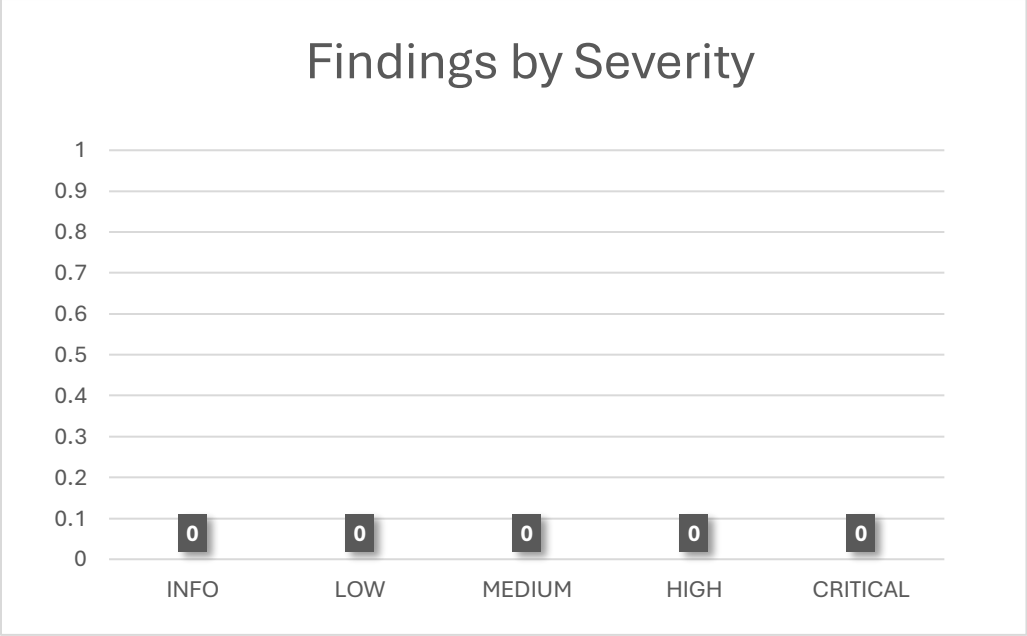
```
        --- PASS:
TestTypesTestSuite/TestMsgChannelOpenInitValidateBasic/channel_not_in_INIT_state
(0.00s)
PASS
ok      github.com/cosmos/ibc-go/v9/modules/core/04-channel/types       0.263s
```

Findings Overview

In performing testing for this assessment, Atredis Partners did not identify any findings.

Atredis defines vulnerability severity ranking as follows:

- **Critical:** These vulnerabilities expose systems and applications to immediate threat of compromise by a dedicated or opportunistic attacker.
- **High:** These vulnerabilities entail greater effort for attackers to exploit and may result in successful network compromise within a relatively short time.
- **Medium:** These vulnerabilities may not lead to network compromise but could be leveraged by attackers to attack other systems or applications components or be chained together with multiple medium findings to constitute a successful compromise.
- **Low:** These vulnerabilities are largely concerned with improper disclosure of information and should be resolved. They may provide attackers with important information that could lead to additional attack vectors or lower the level of effort necessary to exploit a system.



Appendix I: Assessment Methodology

Atredis Partners draws on our extensive experience in penetration testing, reverse engineering, hardware/software exploitation, and embedded systems design to tailor each assessment to the specific targets, attacker profile, and threat scenarios relevant to our client's business drivers and agreed upon rules of engagement.

Where applicable, we also draw on and reference specific industry best practices, regulations, and principles of sound systems and software design to help our clients improve their products while simultaneously making them more stable and secure.

Our team takes guidance from industry-wide standards and practices such as the National Institute of Standards and Technology's (NIST) Special Publications, the Open Web Application Security Project (OWASP), and the Center for Internet Security (CIS).

Throughout the engagement, we communicate findings as they are identified and validated, and schedule ongoing engagement meetings and touchpoints, keeping our process open and transparent and working closely with our clients to focus testing efforts where they provide the most value.

In most engagements, our primary focus is on creating purpose-built test suites and toolchains to evaluate the target, but we do utilize off-the-shelf tools where applicable as well, both for general patch audit and best practice validation as well as to ensure a comprehensive and consistent baseline is obtained.

Research and Profiling Phase

Our research-driven approach to testing begins with a detailed examination of the target, where we model the behavior of the application, network, and software components in their default state. We map out hosts and network services, patch levels, and application versions. We frequently use a number of private and public data sources to collect Open-Source Intelligence about the target and collaborate with client personnel to further inform our testing objectives.

For network and web application assessments, we perform network and host discovery as well as map out all available application interfaces and inputs. For hardware assessments, we study design and implementation, down to a circuit-debugging level. In reviewing source code or compiled application code, we map out application flow and call trees and develop a solid working understanding of how the application behaves, thus helping focus our validation and testing efforts on areas where vulnerabilities might have the highest impact to the application's security or integrity.

Analysis and Instrumentation Phase

Once we have developed a thorough understanding of the target, we use a number of specialized and custom-developed tools to perform vulnerability discovery as well as binary, protocol, and runtime analysis, frequently creating engagement-specific software tools which we share with our clients at the close of any engagement.

We identify and implement means to monitor and instrument the behavior of the target, utilizing debugging, decompilation and runtime analysis, as well as making use of memory and filesystem forensics analysis to create a comprehensive attack modeling testbed. Where they exist, we also use common off-the-shelf, open-source and any extant vendor-proprietary tools to aid in testing and evaluation.

Validation and Attack Phase

Using our understanding of the target, our team creates a series of highly specific attack and fault injection test cases and scenarios. Our selection of test cases and testing viewpoints are based on our understanding of which approaches are most relevant to the target and will gain results in the most efficient manner and built in collaboration with our client during the engagement.

Once our test cases are validated and specific attacks are confirmed, we create proof-of-concept artifacts and pursue confirmed attacks to identify extent of potential damage, risk to the environment, and reliability of each attack scenario. We also gather all the necessary data to confirm vulnerabilities identified and work to identify and document specific root causes and all relevant instances in software, hardware, or firmware where a given issue exists.

Education and Evidentiary Phase

At the conclusion of active testing, our team gathers all raw data, relevant custom tool chains, and applicable testing artifacts, parses and normalizes these results, and presents an initial finding brief to our clients, so that remediation can begin while a more formal document is created. Additionally, our team shares confirmed high-risk findings throughout the engagement so that our clients may begin to address any critical issues as soon as they are identified.

After our brief and initial findings review, we develop a detailed research deliverable report that provides not only our findings and recommendations but also an open and transparent narrative about our testing process, observations and specific challenges in developing attacks against our targets, from the real-world perspective of a skilled, motivated attacker.

Automation and Off-The-Shelf Tools

Where applicable or useful, our team does utilize licensed and open-source software to aid us throughout the evaluation process. These tools and their output are considered secondary to manual human analysis, but nonetheless provide a valuable secondary source of data, after careful validation and reduction of false positives.

For runtime analysis and debugging, we rely extensively on Hopper, IDA Pro and Hex-Rays, as well as platform-specific runtime debuggers, and develop fuzzing, memory analysis, and other testing tools primarily in Ruby and Python.

In source auditing, we typically work in Visual Studio, Xcode and Eclipse IDE, as well as other markup tools. For automated source code analysis, we will typically use the most appropriate tool chain for the target, unless client preference dictates another tool.

Network discovery and exploitation make use of Nessus, Metasploit, and other open source scanning tools, again deferring to client preference where applicable. Web application runtime analysis relies extensively on the Burp Suite, Fuzzer and Scanner, as well as purpose-built automation tools built in Go, Ruby and Python.

Engagement Deliverables

Atredis Partners deliverables include a detailed overview of testing steps and testing dates, as well as our understanding of the specific risk profile developed from performing the objectives of the given engagement.

In the engagement summary we focus on “big picture” recommendations and a high-level overview of shared attributes of vulnerabilities identified and organizational-level recommendations that might address these findings.

In the findings section of the document, we provide detailed information about vulnerabilities identified, provide relevant steps and proof-of-concept code to replicate these findings, and our recommended approach to remediate the issues, developing these recommendations collaboratively with our clients before finalization of the document.

Our team typically makes use of both DREAD and NIST CVE for risk scoring and naming, but as part of our charter as a client-driven and collaborative consultancy, we can vary our scoring model to a given client’s preferred risk model, and in many cases will create our findings using the client’s internal findings templates, if requested.

Sample deliverables can be provided upon request, but due to the highly specific and confidential nature of Atredis Partners' work, these deliverables will be heavily sanitized, and give only a very general sense of the document structure.

Appendix II: Engagement Team Biographies

Shawn Moyer, Founding Partner and CEO

Shawn Moyer scopes, plans, and coordinates security research and consulting projects for the Atredis Partners team, including reverse engineering, binary analysis, advanced penetration testing, and private vulnerability research. As CEO, Shawn works with the Atredis leadership team to build and grow the Atredis culture, making Atredis Partners a home for some of the best minds in information security, and ensuring Atredis continues to deliver research and consulting services that exceed our client's expectations.

Experience

Shawn brings over 25 years of experience in information security, with an extensive background in penetration testing, advanced security research including extensive work in mobile and Smart Grid security, as well as advanced threat modeling and embedded reverse engineering.

Shawn has served as a team lead and consultant in enterprise security for numerous large initiatives in the financial sector and the federal government, including IBM Internet Security Systems' X-Force, MasterCard, a large Federal agency, and Wells Fargo Securities, all focusing on emerging network and application attacks and defenses.

In 2010, Shawn created Accuvant Labs' Applied Research practice, delivering advanced research-driven consulting to numerous clients on mobile platforms, critical infrastructure, medical devices and countless other targets, growing the practice 1800% in its first year.

Prior to Accuvant, Shawn helped develop FishNet Security's penetration testing team as a principal security consultant, growing red team offerings and advanced penetration testing services, while being twice selected as a consulting MVP.

Key Accomplishments

Shawn has written on emerging threats and other topics for Information Security Magazine and ZDNet, and his research has been featured in the Washington Post, BusinessWeek, NPR and the New York Times. Shawn is a twelve-time speaker at the Black Hat Briefings and has been an invited speaker at other notable security conferences around the world.

Shawn is likely best known for delivering the first public research on social network security, pointing out much of the threat landscape still exists on social network platforms today. Shawn also co-authored an analysis of the state of the art in web browser exploit mitigation, creating the first in-depth comparison of browser security models along with Dr. Charlie Miller, Chris Valasek, Ryan Smith, Joshua Drake, and Paul Mehta.

Shawn studied Computer and Network Information Systems at Missouri University and the University of Louisiana at Lafayette, holds numerous information security certifications, and has been a frequent presenter at national and international security industry conferences.

Bryan C. Geraghty, Principal Research Consultant

Bryan leads and executes highly technical application, network, embedded, and physical covert entry assessments. He specializes in cryptography, blockchain technology, reverse engineering, electronics hardware, and covert entry.

Experience

Bryan has over 20 years of experience building and exploiting networks, software, and hardware systems. His background in systems administration, software development, and cryptography has been demonstrably beneficial for security assessments of custom or unique applications in industries such as healthcare, manufacturing, marketing, finance, utilities, telecommunication, and entertainment.

Key Accomplishments

Bryan is a creator and maintainer of several open-source security tools. He is also a nationally recognized speaker who has presenting research on software, hardware, and communications protocol attacks, and participated in offense-oriented panel discussions. Bryan is also an organizing-board member of multiple Kansas City security events, and a staff volunteer and organizer of official events at DEF CON.

Stephen Haywood, Principal Research Consultant

Stephen is a network and application penetration tester with a love for software development. He is fluent in Python and Go and has a working knowledge of a number of other languages. He has a passion for delivering straightforward security solutions and a wide-ranging skillset that allows him to improve the security of technologies both old and new. Prior to joining Atredis Partners, Stephen built and led the Security Engineering team for 1Password with responsibility for application and infrastructure security as well as applied cryptography.

Experience

Stephen has over twenty years of experience in the information technology field working as a software developer, technical trainer, network operations manager, penetration tester, and security engineering leader. He holds a Bachelor of Science in Mathematics and the Offensive Security Certified Professional (OSCP) and the Offensive Security Certified Expert (OSCE) certifications. Over the course of his career, he has helped improve the network and application security of many businesses ranging in size from ten employees to Fortune 100 by offering practical, time-tested information security advice.

Key Accomplishments

Stephen's accomplishments include building the Security Engineering team at 1Password, building and supporting multiple penetration testing teams, speaking at security conferences, conducting training, developing security tools and making contributions to open source security projects such as Metasploit.

Darren Kemp, Research Consulting Director

Darren Kemp leads and executes highly technical software security, network, and web application assessments and advanced red team assessments, as well as complex reverse engineering and exploit development projects.

Experience

Darren brings over a decade of professional experience in the information security space, spanning a variety of roles and responsibilities. Darren's experience includes penetration testing, application security assessments, malware and vulnerability analysis and reverse engineering.

Most recently, Darren was a security researcher for Duo Security's Duo Labs team, publishing externally facing security research and supporting internal security efforts. Prior to Duo, Darren was a Senior Security Consultant for Leviathan Security Group. Prior to Leviathan, Darren worked as a Threat Analyst for Symantec's DeepSight Research Team.

Key Accomplishments

Darren has several widely-reported and publicly credited vulnerability disclosures in a number of major software products, and his research has been featured on MSN, CNET, Wired, and other media outlets. Darren developed advanced crashdump analysis tools as part of DARPA's CINDER program, and has also made many contributions to the Metasploit Project.

Darren studied Computer Technology at the Southern Alberta Institute of Technology.

Sara Bettes, Client Operations Associate

Sara Bettes assists the creation and completion of projects at Atredis Partners, ranging from the full pre-sales process to project design and management, to final delivery and follow-up. Her goals are to ensure all projects are executed in a way that reaches the goals of the client and assists the consultants at every turn.

Experience

Prior to joining Atredis Partners, Sara led a team that planned international sporting competitions, Olympic and national team qualifying events, as well as supported the mission of multiple non-profits. Her experience includes Live Sports Commentating, Staffing Management, Safety Plan Creation, Event Development, Public Relations, and Marketing efforts.

Key Accomplishments

Sara earned a bachelor's degree in Mass Communications with an emphasis in Broadcast and Public Relations from Oklahoma City University.

Appendix III: About Atredis Partners

Atredis Partners was created in 2013 by a team of security industry veterans who wanted to prioritize offering quality and client needs over the pressure to grow rapidly at the expense of delivery and execution. We wanted to build something better, for the long haul.

In six years, Atredis Partners has doubled in size annually, and has been named three times to the Saint Louis Business Journal's "Fifty Fastest Growing Companies" and "Ten Fastest Growing Tech Companies". Consecutively for the past three years, Atredis Partners has been listed on the Inc. 5,000 list of fastest growing private companies in the United States.

The Atredis team is made up of some of the greatest minds in Information Security research and penetration testing, and we've built our business on a reputation for delivering deeper, more advanced assessments than any other firm in our industry.

Atredis Partners team members have presented research over forty times at the BlackHat Briefings conference in Europe, Japan, and the United States, as well as many other notable security conferences, including RSA, ShmooCon, DerbyCon, BSides, and PacSec/CanSec. Most of our team hold one or more advanced degrees in Computer Science or engineering, as well as many other industry certifications and designations. Atredis team members have authored several books, including *The Android Hacker's Handbook*, *The iOS Hacker's Handbook*, *Wicked Cool Shell Scripts*, *Gray Hat C#*, and *Black Hat Go*.

While our client base is by definition confidential and we often operate under strict nondisclosure agreements, Atredis Partners has delivered notable public security research on improving security at Google, Microsoft, The Linux Foundation, Motorola, Samsung and HTC products, and were the first security research firm to be named in Qualcomm's Product Security Hall of Fame. We've received four research grants from the Defense Advanced Research Project Agency (DARPA), participated in research for the CNCF (Cloud Native Computing Foundation) to advance the security of Kubernetes, worked with OSTIF (The Open Source Technology Improvement Fund) and The Linux Foundation on the Core Infrastructure Initiative to improve the security and safety of the Linux Kernel, and have identified entirely new classes of vulnerabilities in hardware, software, and the infrastructure of the World Wide Web.

In 2015, we expanded our services portfolio to include a wide range of advanced risk and security program management consulting, expanding our services reach to extend from the technical trenches into the boardroom. The Atredis Risk and Advisory team has extensive experience building mature security programs, performing risk and readiness assessments, and serving as trusted

partners to our clients to ensure the right people are making informed decisions about risk and risk management.