Wormhole Foundation EVM-CCTP Integration Audit Report

Prepared by Cyfrin
Version 2.1

Lead Auditors

Giovanni Di Siena

0kage

Assisting Auditors

Hans

April 9, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

4.1 Wormhole CCTP Integration

Wormhole is a generic message passing protocol that enables communication between blockchains. CCTP is a permissionless on-chain utility that enables supported Circle assets (USDC/EURC) to move securely between supported blockchain networks. Together, the Wormhole CCTP integration allows USDC/EURC to be transferred between supported CCTP domains along with a generic Wormhole payload.

4.1.1 Wormhole

Verifiable Action Approvals (VAAs) are the core messaging primitive in Wormhole and are emitted any time a cross-chain application contract initiates a cross-chain transfer via the Core Wormhole contract (and, by extension, the CCTP integration contract). Guardian signatures are aggregated and combined into a VAA which is subsequently relayed to the target chain and processed by a receiving contract. VAAs are multicast by default, meaning they specify no destination address and simply attest that something happened on some source chain; however, this does not mean an application cannot specify a destination address or chain to be verified on the destination chain. In this case, the source and destination CCTP domains are encoded in the VAA and Governance maintains a list of registered emitter addresses for each foreign chain. Note that Wormhole chain identifiers do not map directly to expected blockchain network identifiers.

Core Contract There is one Core Contract on each blockchain. All cross chain applications either interact directly with the Core Contract or interact with another contract that does. The Guardian Network picks up messages emitted by the Core Contract as Observations (the data structure that describes a message that was emitted by the Core Contract and noticed by the Guardian node).

Core Contracts can be broken down to a sending and receiving side:

- Sending:
  - `Implementation::publishMessage` is called, posting the `emitterAddress` (the contract which called `publishMessage`), `sequenceNumber`, and `consistencyLevel` into the blockchain logs.

    - Once the desired `consistencyLevel` has been reached and the message passes all of the Guardians' optional checks, the Guardian Network will produce the requested VAAs.
    - Function parameters:
        * `nonce` – A free integer field that can be used however the developer would like. Note that a different nonce will result in a different digest.
        * `payload` – The content of the emitted message, an arbitrary byte array. It may be capped to a certain maximum length due to the constraints of individual blockchains.
        * `consistencyLevel` – A numeric `enum` data type that allows advanced integrators chain-specific flexibility to get messages before finality. Different chains use different consensus mechanisms, so there are different finality assumptions with each one.
    - Returns:
        * `sequenceNumber` – A unique number that increments for every message for a given emitter (and implicitly chain). This combined with the emitter address and emitter chain ID allows the VAA for this message to be queried from the APIs.
- Receiving:
    - `Messages::parseAndVerifyVM` is called, either returning the payload and associated metadata for the VAA or throwing an exception.
    - An exception should only ever throw if the VAA fails signature verification, indicating the VAA is invalid or inauthentic in some form.
    - This method is called during the execution of a transaction where a VAA is passed to ensure the signatures are checked and verified.
    - Function parameters:
        * `encodedVM` – The VAA to be parsed and verified.
    - Returns:
        * `vm` – The parsed and verified VAA.
        * `valid` – A boolean indicating whether the VAA is valid or not.
        * `reason` – An error message if the VAA is invalid.

Developer Considerations When receiving a VAA as an integrating contract (Wormhole CCTP or otherwise), the following points should be considered:

- Receiving a message from relayer:
    - The VAA should originate from an expected emitter address.
    - The VAA should originate from an expected chain id.
    - The core Wormhole `Messages::parseAndVerifyVM` function should be called on any additional VAAs.
- Whether replay protection has been implemented.
- Message ordering:
    - The VAA should have the expected sequence number.
    - There are no guarantees on the order of messages delivered - whether out of order deliveries have been handled.
- Finality:
    - The consistency level should be enough to guarantee the transaction won't be reverted due to block reorgs.
- Forwarding/Call Chaining.

- Refunding overpayment of `gasLimit`.
- Refunding overpayment of `msg.value` sent.

Outside of body of the VAA, but also relevant, is the digest of the VAA which can be used for replay protection by checking if the digest has already been seen. Since the payload itself is application specific, there may be other elements to check to ensure safety.

Governance Before any Wormhole transaction or upgrade can be completed, it must pass through 2/3+ of 19 Guardians (13/19), each of whom conducts their own independent validation process prior to verifying and validating that transaction. Governance also allows Wormhole Guardians to provide optional value movement protections to token bridges built on Wormhole. This protection allows Wormhole Guardians to govern (or effectively rate-limit) the notional flow of assets from any given token bridge chain. This safety feature allows Guardians to limit the impact of any security issue any given chain may have from affecting other connected chains. In the case of Wormhole CCTP, Governance is responsible for the registration of emitter addresses for each foreign chain/CCTP domain and performing contract upgrades.

4.1.2 CCTP

CCTP solves the issues of liquidity fragmentation and poor user experiences caused by unofficial, bridged versions of USDC/EURC used within the DeFi ecosystem; however, CCTP has no direct impact upon existing bridged versions of these assets. To enable Circle assets to be minted on the destination chain, a call must be made to `MessageTransmitter::receiveMessage`. First, the asset is burned on the source chain, then a signed attestation is fetched from Circle, and finally the asset is minted on the destination chain along with the execution of additional logic.

CCTP is available on mainnet for many of the blockchains where USDC is natively issued. Chains supported by the Wormhole CCTP integration include:

Domain	Name
0	Ethereum
1	Avalanche
2	OP (Optimism)
3	Arbitrum
4	Noble
6	Base
7	Polygon PoS

Contract Responsibilities

- `TokenMessenger`: Entrypoint for cross-chain USDC transfer. Routes messages to burn USDC on a source chain, and mint USDC on a destination chain.
- `MessageTransmitter`: Generic message passing. Sends all messages on the source chain, and receives all messages on the destination chain.
- `TokenMinter`: Responsible for minting and burning USDC. Contains chain-specific settings used by burners and minters.

Attestation Service & API

- Circle listens for the `{Burn}` event emitted by calls to `TokenMessenger::depositForBurn` and signs an attestation which provides authorization to mint the specified amount of USDC on the destination chain.

- This attestation is retrieved by calling an API endpoint with the `keccak256` hash of the `messageBytes` emitted by the `{MessageSent}` event.
- An unresponsive attestation service would temporarily preclude new burn messages from being signed.
- The attestation is signed after a given number of block confirmations, configured per chain, to ensure that the burn transaction is final. Additional documentation can be found here.

This public API provides signed attestations used to transmit cross-chain messages. For more information, see the API reference.

Environment	URL
Testnet	https://iris-api-sandbox.circle.com
Mainnet	https://iris-api.circle.com

4.1.3 Architecture

The architecture of this protocol primarily consists of the `ICirleIntegration` contract, composed of the `Governance` and `Logic/WormholeCctpTokenMessenger` APIs. This contract interfaces with the external Wormhole Core Bridge contract (with functions defined in `Messages/Implementation`) and the CCTP `TokenMessenger/MessageTransmitter` contracts. Summaries of the architecture, including important function calls and their return values, are shown below:

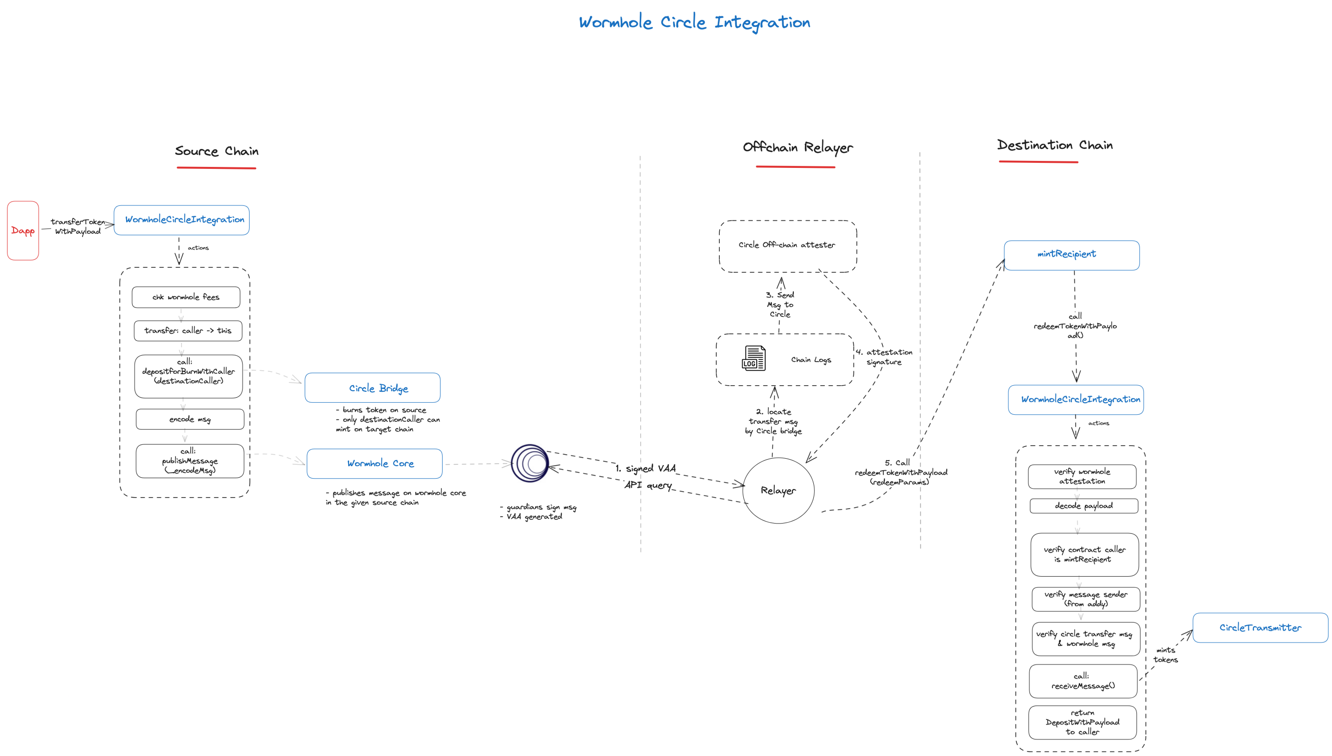

Figure 1: Architecture Diagram 1

4.1.4 Threat Model

Governance API

`registerEmitterAndDomain`

- Misconfiguration of chains/domains (intentional & unintentional).

`upgradeContract`

- Upgrade to invalid/malicious implementation.

Shared

- Reentrancy (via Wormhole Core Bridge, unlikely).
- Replayed VAA.
- Fraudulent VAA.
- Cross-chain replay.
- Invalid message/chain/source/action.
- Impersonation of governance module.
- Execution for invalid target chain.

Logic/WormholeCctpTokenMessenger API

`transferTokensWithPayload`

- Unable to transfer tokens.
- Transfer without burning tokens.
- Circumvent fee.
- Malicious payload.
- CCTP nonce issues.
- Wormhole sequence issues.

`redeemTokensWithPayload`

- Unable to redeem tokens.
- Redeem someone else's tokens.
- Mismatch between VAA and CCTP message.

Shared

- Reentrancy (via CCTP TokenMessenger/Wormhole Core Bridge, unlikely).
- Replayed VAA.
- Fraudulent VAA.
- Cross-chain replay.
- Incorrect encoding/decoding.

5 Audit Scope

Cyfrin conducted an audit of the Wormhole CCTP integration contracts based on the code present in the repository commit hash f7df33b.

The following directories were included in the scope of the audit:

- evm/src/contracts/*
- evm/src/libraries/*

No existing Wormhole/CCTP smart contracts or their respective off-chain components were included.

6 Executive Summary

Over the course of 21 days, the Cyfrin team conducted an audit on the Wormhole Foundation EVM-CCTP Integration smart contracts provided by Wormhole Foundation. In this period, a total of 13 issues were found.

This review of the Wormhole CCTP integration contracts yielded two medium-severity findings, with one relating to the potential malicious application of the otherwise relatively opaque Circle blacklist and another that raises an oversight in validating the target recipient address, which fails to consider aliased L2 address, thereby potentially affecting protocol uptime in the event of L2 sequencer downtime. A handful of low-severity issues have also been identified where it may be possible for core assumptions to be broken under certain circumstances; however, these do not appear to pose an immediate threat to the functioning of the protocol. A number of additional informational findings have been raised where there is no immediate impact but potential for issues to occur in future, especially where there is responsibility on integrators to correctly handle certain behaviors which should be clearly documented and communicated, for example related to the deprecation of Wormhole Guardian sets which should be addressed to ensure valid VAAs containing CCTP messages can be correctly reconstructed/re-validated at scale. Overall, this is an incredibly well-architected and well-tested system with a strong focus on security and reliability.

Summary

Project Name	Wormhole Foundation EVM-CCTP Integration
Repository	wormhole-circle-integration
Commit	f7df33b159a7...
Audit Timeline	Jan 15th - Feb 12th
Methods	Manual Review, Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	3
Informational	8
Gas Optimizations	0
Total Issues	13

Summary of Findings

[M-1] Redemptions are blocked when L2 sequencers are down	Acknowledged
[M-2] Loss of funds due to malicious forcing of `mintRecipient` onto Circle blacklist when CCTP message is in-flight	Acknowledged
[L-1] Potentially dangerous out-of-bounds memory access in `BytesParsing::sliceUnchecked`	Acknowledged
[L-2] A given CCTP domain can be registered for multiple foreign chains due to insufficient validation in `Governance::registerEmitterAndDomain`	Acknowledged
[L-3] Lack of Governance action to update registered emitters	Acknowledged
[I-1] Use `SafeERC20::safeIncreaseAllowance` in the place of `IERC20::approve` in `WormholeCctpTokenMessenger::setTokenMessengerApproval`	Resolved
[I-2] Potential accounting error when the decimals of bridged assets differ between CCTP domains	Acknowledged
[I-3] `Setup` unnecessarily inherits OpenZeppelin `Context`	Resolved
[I-4] Potential dangers for inheriting applications executing the Wormhole payload	Acknowledged
[I-5] Sequencing considerations should be clearly documented and communicated to integrators	Acknowledged
[I-6] Calldata restriction on Wormhole payload should not be modified	Acknowledged
[I-7] Temporary denial-of-service when in-flight messages are not executed before a deprecated Wormhole Guardian set expires	Acknowledged
[I-8] The `mintRecipient` address should be required to indicate interface support to prevent potential loss of funds	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Redemptions are blocked when L2 sequencers are down

Description: Given that rollups such as Optimism and Arbitrum offer methods for forced transaction inclusion, it is important that the aliased sender address is also checked within `Logic::redeemTokensWithPayload` when verifying the sender is the specified `mintRecipient` to allow for maximum uptime in the event of sequencer downtime.

```
// Confirm that the caller is the `mintRecipient` to ensure atomic execution.
require(
    msg.sender.toUniversalAddress() == deposit.mintRecipient, "caller must be mintRecipient"
);
```

Impact: Failure to consider the aliased `mintRecipient` address prevents the execution of valid VAAs on a target CCTP domain where transactions are batched by a centralized L2 sequencer. Since this VAA could carry a time-sensitive payload, such as the urgent cross-chain liquidity infusion to a protocol, this issue has the potential to have a high impact with reasonable likelihood.

Proof of Concept:

1. Protocol X attempts to transfer 10,000 USDC from CCTP Domain A to CCTP Domain B.
2. CCTP Domain B is an L2 rollup that batches transactions for publishing onto the L1 chain via a centralized sequencer.
3. The L2 sequencer goes down; however, transactions can still be executed via forced inclusion on the L1 chain.
4. Protocol X implements the relevant functionality and attempts to redeem 10,000 USDC via forced inclusion.
5. The Wormhole CCTP integration does not consider the contract's aliased address when validating the `mintRecipient`, so the redemption fails.
6. Cross-chain transfer of this liquidity will remain blocked so long as the sequencer is down.

Recommended Mitigation: Validation of the sender address against the `mintRecipient` should also consider the aliased `mintRecipient` address to allow for maximum uptime when `Logic::redeemTokensWithPayload` is called via forced inclusion.

Wormhole Foundation: Since CCTP doesn't deal with this aliasing, we don't feel strongly that we should either.

Cyfrin: Acknowledged.

7.1.2 Loss of funds due to malicious forcing of `mintRecipient` onto Circle blacklist when CCTP message is in-flight

Description: A scenario has been identified in which it may not be possible for the `mintRecipient` to execute redemption on the target domain due to the actions of a bad actor while an otherwise valid CCTP message is in-flight. It is ostensibly the responsibility of the user to correctly configure the `mintRecipient`; however, one could reasonably assume the case where an attacker dusts the `mintRecipient` address with funds stolen in a recent exploit, that may have been deposited to and subsequently withdrawn from an external protocol, or an OFAC-sanctioned token such as TORN, to force this address to become blacklisted by Circle on the target domain while the message is in-flight, thereby causing both the original sender and their intended target recipient to lose access to the tokens.

In the current design, it is not possible to update the `mintRecipient` for a given deposit due to the multicast nature of VAAs. CCTP exposes `MessageTransmitter::replaceMessage` which allows the original source caller to update the destination caller for a given message and its corresponding attestation; however, the Wormhole CCTP integration currently provides no access to this function and has no similar functionality of its own to allow updates to the target `mintRecipient` of the VAA. Without any method for replacing potentially affected VAAs with new VAAs specifying an updated `mintRecipient`, this could result in permanent denial-of-service on the `mintRecipient` receiving tokens on the target domain – the source USDC/EURC will be burnt, but it may be very unlikely that the legitimate recipient is ever able to mint the funds on the destination domain, and once the tokens are burned, there is no path to recovery on the source domain.

This type of scenario is likely to occur primarily where a bad actor intentionally attempts to sabotage a cross-chain transfer of funds that the source caller otherwise expects to be successful. A rational actor would not knowingly attempt a cross-chain transfer to a known blacklisted address, especially if the intended recipient is not a widely-used protocol, which tend to be exempt from sanctions even when receiving funds from a known attacker, but rather an independent EOA. In this case, the destination call to `Logic::redeemTokensWithPayload` will fail when the CCTP contracts attempt to mint the tokens and can only be retried if the `mintRecipient` address somehow comes back off the Circle blacklist, the mechanics of which are not overly clear. It is also possible that request(s) made by law-enforcement agencies for the blacklisting of an entire protocol X, as the mint recipient on target domain Y, will cause innocent users to also lose access to their bridged funds.

It is understood that the motivation for restricting message replacement functionality is due to the additional complexity in handling this edge case and ensuring that the VAA of the original message cannot be redeemed with the replaced CCTP attestation, given the additional attack surface. Given that it is not entirely clear how the Circle blacklisting policy would apply in this case, it would be best for someone with the relevant context to aid in making the decision based on this cost/benefit analysis. If it is the case that a victim can be forced onto the blacklist without a clear path to resolution, then this clearly is not ideal. Even if they are eventually able to have this issue resolved, the impact could be time-sensitive in nature, thinking in the context of cross-chain actions that may need to perform some rebalancing/liquidation function, plus a sufficiently motivated attacker could potentially repeatedly front-run any subsequent attempts at minting on the target domain. It is not entirely clear how likely this final point is in practice, once the messages are no longer in-flight and simply ready for execution on the destination, since it is assumed the blacklist would not likely be updated that quickly. In any case, it is agreed that allowing message replacement will add a non-trivial amount of complexity and does indeed increase the attack surface, as previously identified. So depending on how the blacklist is intended to function, it may be worth allowing message replacement, but it is not possible to say with certainty whether this issue is worth addressing.

Impact: There is only a single address that is permitted to execute a given VAA on the target domain; however, there exists a scenario in which this `mintReceipient` may be permanently unable to perform redemption due to the malicious addition of this address to the Circle blacklist. In this case, there is a material loss of funds with reasonable likelihood.

Proof of Concept:

1. Alice burns 10,000 USDC on CCTP Domain A to be transferred to her EOA on CCTP Domain B.
2. While this CCTP message is in-flight, an attacker withdraws a non-trivial amount of USDC, that was previously obtained from a recent exploit, from protocol X to Alice's EOA on CCTP domain B.
3. Law enforcement notifies Circle to blacklist Alice's EOA, which now holds stolen funds.
4. Alice attempts to redeem 10,000 USDC on CCTP Domain B, but minting fails because her EOA is now blacklisted on the USDC contract.
5. The 10,000 USDC remains burnt and cannot be minted on the target domain since the VAA containing the attested CCTP message can never be executed without the USDC mint reverting.

Recommended Mitigation: Consider allowing VAAs to be replaced by new VAAs for a given CCTP message and corresponding attestation, so long as they have not already been consumed on the target domain. Alternatively, consider adding an additional Governance action dedicated to the purpose of recovering the USDC burnt by a VAA that has not yet been consumed on the target domain due to malicious blacklisting.

Wormhole Foundation: Although CCTP has the ability to replace messages, it is also subject to this same issue since the original message recipient can't be changed.

Cyfrin: Acknowledged.

7.2 Low Risk

7.2.1 Potentially dangerous out-of-bounds memory access in `BytesParsing::sliceUnchecked`

Description: `BytesParsing::sliceUnchecked` currentlybails early for the degenerate case when the slice length is zero; however, there is no validation on the length of the encoded bytes parameter `encoded` itself. If the length of `encoded` is less than the slice `length`, then it is possible to access memory out-of-bounds.

```
function sliceUnchecked(bytes memory encoded, uint256 offset, uint256 length)
    internal
    pure
    returns (bytes memory ret, uint256 nextOffset)
{
    //bail early for degenerate case
    if (length == 0) {
        return (new bytes(0), offset);
    }

    assembly ("memory-safe") {
        nextOffset := add(offset, length)
        ret := mload(freeMemoryPtr)

        /* snip: inline dev comments */

        let shift := and(length, 31) //equivalent to `mod(length, 32)` but 2 gas cheaper
        if iszero(shift) { shift := wordSize }

        let dest := add(ret, shift)
        let end := add(dest, length)
        for { let src := add(add(encoded, shift), offset) } lt(dest, end) {
            src := add(src, wordSize)
            dest := add(dest, wordSize)
        } { mstore(dest, mload(src)) }

        mstore(ret, length)
        //When compiling with --via-ir then normally allocated memory (i.e. via new) will have 32 byte
        //  memory alignment and so we enforce the same memory alignment here.
        mstore(freeMemoryPtr, and(add(dest, 31), not(31)))
    }
}
```

Since the `for` loop begins at the offset of `encoded` in memory, accounting for its length and accompanying `shift` calculation depending on the `length` supplied, and execution continues so long as `dest` is less than `end`, it is possible to continue loading additional words out of bounds simply by passing larger `length` values. Therefore, regardless of the length of the original bytes, the output slice will always have a size defined by the `length` parameter.

It is understood that this is known behavior due to the unchecked nature of this function and the accompanying checked version, which performs validation on the `nextOffset` return value compared with the length of the encoded bytes.

```
function slice(bytes memory encoded, uint256 offset, uint256 length)
    internal
    pure
    returns (bytes memory ret, uint256 nextOffset)
{
    (ret, nextOffset) = sliceUnchecked(encoded, offset, length);
    checkBound(nextOffset, encoded.length);
}
```

It has not been possible within the constraints of this review to identify a valid scenario in which malicious calldata can make use of this behavior to launch a successful exploit; however, this is not a guarantee that the usage of this library function is bug-free since there do exist certain quirks related to the loading of calldata.

Impact: The impact is limited in the context of the library function's usage in the scope of this review; however, it is advisable to check any other usage elsewhere and in the future to ensure that this behavior cannot be weaponized. `BytesParsing::sliceUnchecked` is currently only used in `WormholeCctpMessages::_decodeBytes`, which itself is called in `WormholeCctpMessages::decodeDeposit`. This latter function is utilized in two places:

1. `Logic::decodeDepositWithPayload`: here, any issues in slicing the encoded bytes would impact users' ability to decode payloads, potentially stopping them from correctly retrieving the necessary information for redemptions.

2. `WormholeCctpTokenMessenger::verifyVaaAndMint`/`WormholeCctpTokenMessenger::verifyVaaAndMintLegacy`: these functions verify and reconcile CCTP and Wormhole messages in order to mint tokens for the encoded mint recipient. Fortunately, for a malicious calldata payload, Wormhole itself will revert when `IWormhole::parseAndVerifyVM` is called via `WormholeCctpTokenMessenger::_parseAndVerifyVaa` since it will be unable to retrieve a valid version number when casting to `uint8`.

Proof of Concept: Apply the following git diff to differential test against a Python implementation:

```
diff --git a/evm/.gitignore b/evm/.gitignore
--- a/evm/.gitignore
+++ b/evm/.gitignore
@@ -7,3 +7,4 @@ lib
 node_modules
 out
 ts/src/ethers-contracts
+venv/
diff --git a/evm/forge/tests/differential/BytesParsing.t.sol
↪  b/evm/forge/tests/differential/BytesParsing.t.sol
new file mode 100644
--- /dev/null
+++ b/evm/forge/tests/differential/BytesParsing.t.sol
@@ -0,0 +1,72 @@
+// SPDX-License-Identifier: Apache 2
+pragma solidity ^0.8.19;
+
+import "forge-std/Test.sol";
+import "forge-std/console.sol";
+
+import {BytesParsing} from "src/libraries/BytesParsing.sol";
+
+contract BytesParsingTest is Test {
+    using BytesParsing for bytes;
+
+    function setUp() public {}
+
+    function test_sliceUncheckedFuzz(bytes memory encoded, uint256 offset, uint256 length) public {
+        bound(offset, 0, type(uint8).max);
+        bound(length, 0, type(uint8).max);
+        if (offset > encoded.length || length > encoded.length || offset + length > encoded.length) {
+            return;
+        }
+
+        sliceUncheckedBase(encoded, offset, length);
+    }
+
+    function test_sliceUncheckedConcreteReadOOB() public {
+        bytes memory encoded = bytes("");
+        bytes32 dirty = 0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef;
+        assembly {
```

```
+            mstore(add(encoded, 0x20), dirty)
+        }
+        uint256 offset = 0;
+        uint256 length = 32;
+
+        sliceUncheckedBase(encoded, offset, length);
+    }
+
+    function sliceUncheckedBase(bytes memory encoded, uint256 offset, uint256 length)
+        internal
+        returns (
+            bytes memory soliditySlice,
+            uint256 solidityNextOffset,
+            bytes memory pythonSlice,
+            uint256 pythonNextOffset
+        )
+    {
+        (soliditySlice, solidityNextOffset) = encoded.sliceUnchecked(offset, length);
+        assertEq(soliditySlice.length, length, "wrong length");
+
+        string[] memory inputs = new string[](9);
+        inputs[0] = "python";
+        inputs[1] = "forge/tests/differential/python/bytes_parsing.py";
+        inputs[2] = "slice_unchecked";
+        inputs[3] = "--encoded";
+        inputs[4] = vm.toString(encoded);
+        inputs[5] = "--offset";
+        inputs[6] = vm.toString(offset);
+        inputs[7] = "--length";
+        inputs[8] = vm.toString(length);
+
+        (pythonSlice, pythonNextOffset) = abi.decode(vm.ffi(inputs), (bytes, uint256));
+
+        emit log_named_uint("soliditySlice.length", soliditySlice.length);
+        emit log_named_uint("pythonSlice.length", pythonSlice.length);
+
+        emit log_named_bytes("soliditySlice", soliditySlice);
+        emit log_named_bytes("pythonSlice", pythonSlice);
+        emit log_named_uint("solidityNextOffset", solidityNextOffset);
+        emit log_named_uint("pythonNextOffset", pythonNextOffset);
+
+        assertEq(soliditySlice, pythonSlice, "wrong slice");
+        assertEq(solidityNextOffset, pythonNextOffset, "wrong next offset");
+    }
+}
diff --git a/evm/forge/tests/differential/python/bytes_parsing.py
↪  b/evm/forge/tests/differential/python/bytes_parsing.py
new file mode 100644
--- /dev/null
+++ b/evm/forge/tests/differential/python/bytes_parsing.py
@@ -0,0 +1,42 @@
+from eth_abi import encode
+import argparse
+
+
+def main(args):
+    if args.function == "slice_unchecked":
+        slice, next_offset = slice_unchecked(args)
+        encode_and_print(slice, next_offset)
+
+
+def slice_unchecked(args):
```

```
+    if args.length == 0:
+        return (b"", args.offset)
+
+    next_offset = args.offset + args.length
+
+    encoded_bytes = (
+        bytes.fromhex(args.encoded[2:])
+        if args.encoded.startswith("0x")
+        else bytes.fromhex(args.encoded)
+    )
+    return (encoded_bytes[args.offset : next_offset], next_offset)
+
+
+def encode_and_print(slice, next_offset):
+    encoded_output = encode(["bytes", "uint256"], (slice, next_offset))
+    ## append 0x for FFI parsing
+    print("0x" + encoded_output.hex())
+
+
+def parse_args():
+    parser = argparse.ArgumentParser()
+    parser.add_argument("function", choices=["slice_unchecked"])
+    parser.add_argument("--encoded", type=str)
+    parser.add_argument("--offset", type=int)
+    parser.add_argument("--length", type=int)
+    return parser.parse_args()
+
+
+if __name__ == "__main__":
+    args = parse_args()
+    main(args)
diff --git a/evm/forge/tests/differential/python/requirements.txt
↪  b/evm/forge/tests/differential/python/requirements.txt
new file mode 100644
--- /dev/null
+++ b/evm/forge/tests/differential/python/requirements.txt
@@ -0,0 +1 @@
+eth_abi==5.0.0
\ No newline at end of file
diff --git a/evm/foundry.toml b/evm/foundry.toml
--- a/evm/foundry.toml
+++ b/evm/foundry.toml
@@ -31,4 +31,7 @@ gas_reports = ["*"]

 gas_limit = "18446744073709551615"

+[profile.ffi]
+ffi = true
+
```

Recommended Mitigation: Consider bailing early if the length of the bytes from which to construct a slice is zero, and always ensure the resultant offset is correctly validated against the length when using the unchecked version of the function.

Wormhole Foundation: The slice method does this checking for us. Since we're controlling the length specified in the wire format, we can safely use the unchecked variant.

Cyfrin: Acknowledged.

7.2.2 A given CCTP domain can be registered for multiple foreign chains due to insufficient validation in `Governance::registerEmitterAndDomain`

Description: `Governance::registerEmitterAndDomain` is a Governance action that is used to register the emitter address and corresponding CCTP domain for a given foreign chain. Validation is currently performed to ensure that the registered CCTP domain of the foreign chain is not equal to that of the local chain; however, there is no such check to ensure that the given CCTP domain has not already been registered for a different foreign chain. In this case, where the CCTP domain of an existing foreign chain is mistakenly used in the registration of a new foreign chain, the `getDomainToChain` mapping of an existing CCTP domain will be overwritten to the most recently registered foreign chain. Given the validation that prevents foreign chains from being registered again, without a method for updating an already registered emitter, it will not be possible to correct this corruption of state.

```
function registerEmitterAndDomain(bytes memory encodedVaa) public {
    /* snip: parsing of Governance VAA payload */

    // For now, ensure that we cannot register the same foreign chain again.
    require(registeredEmitters[foreignChain] == 0, "chain already registered");

    /* snip: additional parsing of Governance VAA payload */

    // Set the registeredEmitters state variable.
    registeredEmitters[foreignChain] = foreignAddress;

    // update the chainId to domain (and domain to chainId) mappings
    getChainToDomain()[foreignChain] = cctpDomain;
    getDomainToChain()[cctpDomain] = foreignChain;
}
```

Impact: The impact of this issue in the current scope is limited since the corrupted state is only ever queried in a public view function; however, if it is important for third-party integrators, then this has the potential to cause downstream issues.

Proof of Concept:

1. CCTP Domain A is registered for foreign chain identifier X.
2. CCTP Domain A is again registered, this time for foreign chain identifier Y.
3. The `getDomainToChain` mapping for CCTP Domain A now points to foreign chain identifier Y, while the `getChainToDomain` mapping for both X and Y now points to CCTP domain A.

Recommended Mitigation: Consider adding the following validation when registering a CCTP domain for a foreign chain:

```
+ require (getDomainToChain()[cctpDomain] == 0, "CCTP domain already registered for a different foreign
↪  chain");
```

Wormhole Foundation: We are comfortable that governance messages are sufficiently validated before being signed by the guardians and submitted on-chain.

Cyfrin: Acknowledged.

7.2.3 Lack of Governance action to update registered emitters

Description: The Wormhole CCTP integration contract currently exposes a function `Governance::registerEmitterAndDomain` to register an emitter address and its corresponding CCTP domain on the given foreign chain; however, no such function currently exists to update this state. Any mistake made when registering the emitter and CCTP domain is irreversible unless an upgrade is performed on the entirety of the integration contract itself. Deployment of protocol upgrades comes with its own risks and should not be performed as a necessary fix for trivial human errors. Having a separate governance action to update the emitter address, foreign chain identifier, and CCTP domain is a preferable pre-emptive measure against any potential human errors.

```
function registerEmitterAndDomain(bytes memory encodedVaa) public {
    /* snip: parsing of Governance VAA payload */

    // Set the registeredEmitters state variable.
    registeredEmitters[foreignChain] = foreignAddress;

    // update the chainId to domain (and domain to chainId) mappings
    getChainToDomain()[foreignChain] = cctpDomain;
    getDomainToChain()[cctpDomain] = foreignChain;
}
```

Impact: In the event an emitter is registered with an incorrect foreign chain identifier or CCTP domain, then a protocol upgrade will be required to mitigate this issue. As such, the risks associated with the deployment of protocol upgrades and the potential time-sensitive nature of this issue designate a low severity issue.

Proof of Concept:

1. A Governance VAA erroneously registers an emitter with the incorrect foreign chain identifier.
2. A Governance upgrade is now required to re-initialize this state so that the correct foreign chain identifier can be associated with the given emitter address.

Recommended Mitigation: The addition of a `Governance::updateEmitterAndDomain` function is recommended to allow Governance to more easily respond to any issues with the registered emitter state.

Wormhole Foundation: Allowing existing emitters to be updated comes with similar impacts of admin mistakes. But allowing updates is indeed easier than coordinating a whole contract upgrade. However we won't change this since we can't easily enforce that governance messages to perform these updates are played in sequence.

Cyfrin: Acknowledged.

7.3 Informational

7.3.1 Use `SafeERC20::safeIncreaseAllowance` in the place of `IERC20::approve` in `WormholeCctpTokenMessenger::setTokenMessengerApproval`

Although the `SafeERC20` library is declared as being used for the `IERC20` interface, `WormholeCctpTokenMessenger::setTokenMessengerApproval` uses `IERC20::approve` directly instead of `SafeERC20::safeApprove`. Whilst the `FiatTokenV2_2` implementation of `IERC20::approve` does return the `true` boolean, reverting otherwise, some tokens can silently fail when this function is called; therefore, it may be necessary to check the return value of this call if the protocol ever intends to work with other ERC20 tokens. Also, note that OpenZeppelin discourages the use of `SafeERC20::safeApprove` (deprecated in v5) and instead recommends the use of `safeERC20::safeIncreaseAllowance`.

Wormhole Foundation: Fixed in PR #52.

Cyfrin: Verified. The direct use of `ERC20::approve` has been modified to instead use `safeERC20::safeIncreaseAllowance`.

7.3.2 Potential accounting error when the decimals of bridged assets differ between CCTP domains

The `FiatTokenV2_2` contract deployed to target CCTP domains typically has 6 decimals; however, on some chains, such as BNB Smart Chain, a decimal value of 18 is used. The Wormhole CCTP integration contract and the core CCTP contracts themselves do not reconcile any differences in the source/destination token decimals, which would cause critical issues in the amount to be minted on the target domain since these contracts are not working with Wormhole x-assets (where this issue is sufficiently mitigated) but rather native USDC/EURC on the respective chains.

For example, assuming both CCTP domains are intended to be supported, burning 20 tokens on BNB Smart Chain where USDC has 18 decimals, encoded as `20e18`, then trying to mint this amount on the destination chain where USDC has 6 decimals (e.g. Ethereum), then there is a problem because the recipient has not, in fact, minted `20e12` tokens instead of 20.

Since BNB Smart Chain is not one of the currently supported domains, and all currently supported CCTP domains use a version of the `FiatTokenV2_2` contract with 6 decimals, this is not an issue at present. If a non-standard domain is ever intended to be supported for cross-chain transfers, then it is important that any differences in the token decimals are correctly reconciled.

Wormhole Foundation: No need to change anything now but will have to make changes if CCTP introduces other chains. One to be aware of and keep and eye on going forwards.

Cyfrin: Acknowledged.

7.3.3 `Setup` unnecessarily inherits OpenZeppelin `Context`

The `Setup` contract currently inherits OpenZeppelin `Context`; however, this is unnecessary as none of its functionality is used anywhere within the logic.

Wormhole Foundation: Fixed in PR #52.

Cyfrin: Acknowledged.

7.3.4 Potential dangers for inheriting applications executing the Wormhole payload

The Wormhole CCTP contracts are written to allow integration by both composition and inheritance. When calling `Logic::transferTokensWithPayload`, users are able to pass an arbitrary Wormhole payload that gets parsed from the VAA on the destination chain. It is our understanding that, if required, execution of this payload is intended to be the responsibility of the integrating application. As such, it has been noted that the behavior of payload execution has not been tested; however, the Wormhole payload does not necessarily need to be executed with an external call, since it could simply contain information that is useful to the inheriting contract.

In the case the payload is used as the input for an arbitrary external call, there is a risk here for the integrator. For applications inheriting the Wormhole CCTP contracts, execution of the payload will occur in the context of these contracts, which could be potentially dangerous. It is, therefore, the responsibility of the integrator to perform sufficient application-specific validation on the payload. This should be clearly documented.

Wormhole Foundation: The existing functionality is as intended.

Cyfrin: Acknowledged.

7.3.5 Sequencing considerations should be clearly documented and communicated to integrators

The Wormhole CCTP integration contracts do not enforce in-sequence message execution by default as a design choice to prevent one message from blocking subsequent messages, instead opting to give integrators the ability to order transactions if they so need. Given it is the responsibility of integrating protocols to execute or otherwise consume the Wormhole payload transmitted by the integration contracts, it is possible for out-of-order executions to cause issues with both high severity and high likelihood if the ordering of message execution is not correctly handled. Wormhole VAAs do not have to be ordered and are effectively multicast, so this does not affect the integration insofar as the contracts in scope for this audit are concerned.

When it comes to handling generic payloads along with token transfers across different chains, corruption of the intended order could have non-trivial consequences for operations that are sensitive to order or timing, such as in lending or derivatives, given how deeply USDC is entrenched within the whole of DeFi. Consider the following scenario:

1. Alice transfers 1000 USDC from CCTP Domain A to Perp X on CCTP Domain B.
2. Alice sends another 100 USDC to Perp X, with a payload to open a 5000 USDC position at 5X leverage.
3. Alice's messages are executed on CCTP Doman B:
    1. If the first message is executed before the second, Alice has a margin of 1100, and the trade is correctly created on X.
    2. If the second message is executed before the first, the trade cannot be opened due to insufficient margin. Factoring in liquidations, auctions, and so on, out-of-sequence execution can have a plethora of unintended consequences.

As noted above, the sender has the ability to specify a Wormhole nonce, and there is also a Wormhole sequence number that is auto-incremented. These are both received on the destination chain in the VAA, so integrators wishing to enforce order can do so either by auto-incrementing the Wormhole nonce on the source domain or by using the Wormhole sequence number and then enforcing ordering on the target domain by checking the source chain, sender address, and nonce/sequence. This should be clearly documented and communicated to users.

Wormhole Foundation: Integrators requiring ordered transactions will have to enforce this themselves, which is intended behavior.

Cyfrin: Acknowledged.

7.3.6 Calldata restriction on Wormhole payload should not be modified

Based on an end-to-end fork test written between Arbitrum and Avalanche C-Chain (15M block gas limit), a gas usage of ~2.5M units has been observed using the maximum allowed payload length of `type(uint16).max`. It is important that this calldata restriction is not modified; otherwise, a scenario could exist where it may not be possible for the `mintRecipient` to execute redemptions on the target domain due to an out-of-gas error caused by an excessively large Wormhole payload. Even in the current state, integrators should be careful to ensure that any additional calls wrapping those to `Logic::redeemTokensWithPayload` cannot be made susceptible to this issue.

Wormhole Foundation: Acknowledged.

Cyfrin: Acknowledged.

7.3.7 Temporary denial-of-service when in-flight messages are not executed before a deprecated Wormhole Guardian set expires

Description: Wormhole exposes a governance action in `Governance::submitNewGuardianSet` to update the Guardian set via Governance VAA.

```
function submitNewGuardianSet(bytes memory _vm) public {
    ...

    // Trigger a time-based expiry of current guardianSet
    expireGuardianSet(getCurrentGuardianSetIndex());

    // Add the new guardianSet to guardianSets
    storeGuardianSet(upgrade.newGuardianSet, upgrade.newGuardianSetIndex);

    // Makes the new guardianSet effective
    updateGuardianSetIndex(upgrade.newGuardianSetIndex);
}
```

When this function is called, `Setters:: expireGuardianSet` initiates a 24-hour timeframe after which the current guardian set expires.

```
function expireGuardianSet(uint32 index) internal {
    _state.guardianSets[index].expirationTime = uint32(block.timestamp) + 86400;
}
```

Hence, any in-flight VAAs that utilize the deprecated Guardian set index will fail to be executed given the validation present in `Messages::verifyVMInternal`.

```
/// @dev Checks if VM guardian set index matches the current index (unless the current set is expired).
if(vm.guardianSetIndex != getCurrentGuardianSetIndex() && guardianSet.expirationTime < block.timestamp){
    return (false, "guardian set has expired");
}
```

Considering there is no automatic relaying of Wormhole CCTP messages, counter to what is specified in the documentation (unless an integrator implements their own relayer), there are no guarantees that an in-flight message which utilizes an old Guardian set index will be executed by the `mintRecipient` on the target domain within its 24-hour expiration period. This could occur, for example, in cases such as:

1. Integrator messages are blocked by their use of the Wormhole nonce/sequence number.
2. CCTP contracts are paused on the target domain, causing all redemptions to revert.
3. L2 sequencer downtime, since the Wormhole CCTP integration contracts do not consider aliased addresses for forced inclusion.
4. The `mintRecipient` is a contract that has been paused following an exploit, temporarily restricting all incoming and outgoing transfers.

In the current design, it is not possible to update the `mintRecipient` for a given deposit due to the multicast nature of VAAs. CCTP exposes `MessageTransmitter::replaceMessage` which allows the original source caller to update the destination caller for a given message and its corresponding attestation; however, the Wormhole CCTP integration currently provides no access to this function and has no similar functionality of its own to allow updates to the target `mintRecipient` of the VAA.

Additionally, there is no method for forcibly executing the redemption of USDC/EURC to the `mintRecipient`, which is the only address allowed to execute the VAA on the target domain, as validated in `Logic::redeemTokensWithPayload`.

```
// Confirm that the caller is the `mintRecipient` to ensure atomic execution.
require(
    msg.sender.toUniversalAddress() == deposit.mintRecipient, "caller must be mintRecipient"
);
```

Without any programmatic method for replacing expired VAAs with new VAAs signed by the updated Guardian set, the source USDC/EURC will be burnt, but it will not be possible for the expired VAAs to be executed, leading to denial-of-service on the `mintRecipient` receiving tokens on the target domain. The Wormhole CCTP integration does, however, inherit some mitigations already in place for this type of scenario where the Guardian set is updated, as explained in the Wormhole whitepaper, meaning that it is possible to repair or otherwise replace the expired VAA for execution using signatures from the new Guardian set. In all cases, the original VAA metadata remains intact since the new VAA Guardian signatures refer to an event that has already been emitted, so none of the contents of the VAA payload besides the Guardian set index and associated signatures change on re-observation. This means that the new VAA can be safely paired with the existing Circle attestation for execution on the target domain by the original `mintRecipient`.

Impact: There is only a single address that is permitted to execute a given VAA on the target domain; however, there are several scenarios that have been identified where this `mintReceipient` may be unable to perform redemption for a period in excess of 24 hours following an update to the Guardian set while the VAA is in-flight. Fortunately, Wormhole Governance has a well-defined path to resolution, so the impact is limited.

Proof of Concept:

1. Alice burns 100 USDC to be transferred to dApp X from CCTP Domain A to CCTP Domain B.
2. Wormhole executes a Governance VAA to update the Guardian set.
3. 24 hours pass, causing the previous Guardian set to expire.
4. dApp X attempts to redeem 100 USDC on CCTP Domain B, but VAA verification fails because the message was signed using the expired Guardian set.
5. The 100 USDC remains burnt and cannot be minted on the target domain by executing the attested CCTP message until the expired VAA is reobserved by members of the new Guardian set.

Recommended Mitigation: The practicality of executing the proposed Governance mitigations at scale should be carefully considered, given the extent to which USDC is entrenched within the wider DeFi ecosystem. There is a high likelihood of temporary widespread, high-impact DoS, although this is somewhat limited by the understanding that Guardian set updates are expected to occur relatively infrequently, given there have only been three updates in the lifetime of Wormhole so far. There is also potentially insufficient tooling for the detailed VAA re-observation scenarios, which should handle the recombination of the signed CCTP message with the new VAA and clearly communicate these considerations to integrators.

Wormhole Foundation: This is the same as how the Wormhole token bridge operates.

Cyfrin: Acknowledged.

7.3.8 The `mintRecipient` address should be required to indicate interface support to prevent potential loss of funds

If the destination `mintRecipient` is a smart contract, it should be required to implement `IERC165` and another Wormhole/CCTP-specific interface to ensure that it has the necessary functionality to transfer/approve USDC/EURC tokens. Whilst it is ultimately the responsibility of the integrator to ensure that they correctly handle the receipt of tokens, this recommendation should help to avoid situations where the tokens become irreversibly stuck after calling `Logic::redeemTokenWithPayload`.

Wormhole Foundation: Responsibility lies with the integrator to ensure their code works with the `CircleIntegration` logic.

Cyfrin: Acknowledged.

8 Appendix

8.1 Test Suite Analysis

Environment files contained within the `evm/env` directory are used to store the relevant environment variables for the test suite, including rpc endpoints, contract addresses and their corresponding chain IDs. This allows for the test suite to be run against different networks in an organised manner and without having to change the test scripts themselves due to the use of the `TESTING_FORK_RPC` environment variable in `evm/Makefile`. Here, it is important to note that the default configuration is for the test suite to be run against a forked Polygon Mumbai instance, minting USDC on Avalanche Fuji. This is defined under the `Forge Test` heading of `evm/env/testing.env`. A tree of the `evm/env` directory structure is shown below.

The test files themselves are located in the `evm/forge/tests` directory and can be run by executing the `make test` command from within the `evm` directory. Within this directory, there exists another `helpers` directory which contains a `USDC` interface file and subdirectory `libraries` which itself contains a number of test-specific helper files. A tree of the `evm/forge/tests/helpers` directory structure is shown below.

- `UsdcDeal.sol` contains a single function `dealAndApprove` which is used to mint USDC to the specified `to` address. Its implementation is slightly strange in that the master minter is first pranked to configure `address(this)` of the current context as a minter before minting itself the desired amount of USDC and then approving the recipient.
- The overloaded `slotValueEquals` functions in `SlotCheck.sol` are used as a wrapper around the `vm.load` cheatcode and return a boolean value to indicate whether the value of a specific storage slot is equal to another value passed as argument.
- `WormholeOverride.sol` appears to provide a number of functions to override the default behaviour of the Wormhole contract for ease of testing. For example:
  - The `setUpOverride` function overwrites all but first guardian set to the zero address and subsequently overwrites the first guardian key with the devnet key specified in the function argument before saving it in a specific slot of Wormhole's storage for later retrieval via `guardianPrivateKey`.
  - The `fetchWormholePublishedPayloads` function extracts messages from the `LogMessagePublished` event emitted by Wormhole and returns a bytes array of `payloads`, discarding the decoded `sender`, `sequence`, `nonce` and `consistencyLevel` values.
  - The `craftVaa` function is used to craft a VAA from the specified `emitterChain`, `emitterAddress`, `sequence` and `payload` values, returning the resulting VAA and encoded bytes. It is prescient to note that a handful of VAA fields are hardcoded in the body of this function, including `version`, `nonce` and `consistencyLevel` which are set to `1`, `420` and `1` respectively. A hash of the encoded body is signed using the `guardianPrivateKey` and the resulting signature is appended to the VAA before returning the fully-encoded bytes. One outstanding question concerns the decrementing of the `v` component of the signature by 27, which may be a quirk of the `vm.sign` cheatcode (additional context on the `v` signature component can be found here).
  - The overloaded `craftGovernanceVaa` functions differ in their specification of the Governance chain ID and Governance contract, either using the defined constants or the values passed as arguments. Both functions then call the `craftVaa` function with the specified arguments.
- `CircleIntegrationOverride.sol` appears to contains helper functions for interacting with the Wormhole CCTP integration and related data structures. For example:
  - The `setupOverride` function first invokes the `setUpOverride` function of `WormholeOverride.sol` before pranking the Circle attester manager to overwrite the signature threshold to `1` and enable the guardian key as an attester. The Circle attester is retrieved by calling the `circleAttester` function, which loads the guardian private key from storage.
  - The `fetchCctpMessages` function extracts messages from the `MessageSent` event emitted by the CCTP `MessageTransmitter` contract and returns an array of `CctpMessage` structs.
  - The `decodeCctpMessage` function decodes a given encoded CCTP message, passed as raw bytes, into a `CctpMessage` struct. This logic is similar to the implementation of `WormholeC-`

ctpMessages::decodeDeposit, but differs in that the encodings are distinct between serialized Wormhole message payloads and CCTP messages. It is also prescient to note that the call to _takeRemainingBytes invokes BytesParsing::sliceUnchecked, so any error in this library function could impact the decoding of CCTP messages in the test suite.

  - The overloaded craftCctpTokenBurnMessage functions differ in their specification of the messageSender and destinationCaller values, either using values derived from the circleIntegration contract or those passed as arguments. All functions then call the _craftCctpTokenBurnMessage function with the specified arguments, returning a CctpTokenBurnMessage struct and two bytes arrays representing an encoded CctpTokenBurnMessage and signed CCTP attestation respectively. Here, it is noted that the CCTP version is hardcoded to 0 and the sourceDomain member is assigned the value of the remoteDomain argument while the destinationDomain member is assigned the value of the localDomain returned by the circleIntegration contract. The sender is set to the CCTP Token Messenger returned by the circleIntegration contracts corresponding to the remoteDomain argument, so this appears to be consistent. It is strange that the CCTP version is included in both the CCTPHeader and CctpTokenBurnMessage structs – why is this necessary?
  - The overloaded craftRedeemParameters functions differ in their specification of the messageSender and destinationCaller values, either using values derived from the circleIntegration contract or those passed as arguments. All functions then call the _craftRedeemParameters function with the specified arguments, returning a RedeemParameters struct. Here, it is noted that _craftTokenBurnMessage is first called to craft the token burn message data which is subsequently used in calling WormholeOverride::craftVaa with some additional arguments such as vaaParams, burnSource and payload. The burnSource parameter appears to be decoupled from the CCTP message itself but it seems that usage is consistent with the address present in the DepositWithPayload struct representing the message encoded in Wormhole message payload. Wormhole CCTP ensures that this message is paired with the corresponding CCTP Token Burn message.

The main test directory contains a number of different test files, including unit, integration, fork and gas usage tests. The integrations directory also contains two example contracts which interact with the WormholeCctpTokenMessenger contract via inheritance and direct composition. A tree of the evm/forge/tests directory structure is shown below, collapsing the helpers directory since it is already shown above.

- CircleIntegration.t.sol is designed to test the behavior of the Wormhole CCTP integration contract directly. An outline of the test file is shown below:
  - Setup: this test file defines two immutable variables USDC_ADDRESS and FOREIGN_USDC_ADDRESS given by the values loaded from the TESTING_USDC_TOKEN_ADDRESS and TESTING_FOREIGN_USDC_TOKEN_ADDRESS environment variables respectively. Given these correspond to the values in evm/env/testing.env, this means that the test suite is currently configured to use the USDC token on the Polygon Mumbai fork and the USDC token on the Avalanche Fuji testnet. The setupWormhole function instantiates the Wormhole contract as defined by the TESTING_WORMHOLE_ADDRESS environment variable while the setupUSDC function validates that the decimals of the USDC contract are equal to 6. The setupCircleIntegration function deploys an instance of the Setup contract alongside the implementation contract for use with the newly deployed proxy. This proxy is then cast to the ICircleIntegration interface before calling the CircleIntegrationOverride::setUpOverride function and asserting that the the state changes executed as expected. All three functions are invoked in the Forge setUp function.
  - The test_CannotTransferTokensWithPayloadInvalidToken function first registers the foreign emitter and domain for the given chain and sets up the necessary WETH balances and approvals on the test contract for use with the circleIntegration contract. The _expectRevert function is then invoked with an encoded call to ICircleIntegration::transferTokensWithPayload, taking the WETH address as the token argument, the 0xdeadbeef address as the mintRecipient and an arbitrary payload. This call is expected to revert with the Burn token not supported reason string, since the token to be transferred must be registered with the CCTP Token Messenger contract.
  - The test_CannotTransferTokensWithPayloadZeroAmount function first registers the foreign emitter and domain for the given chain before invoking the _expectRevert function with an encoded call to ICircleIntegration::transferTokensWithPayload. This encoded call takes the USDC address as the token argument, the 0xdeadbeef address as the mintRecipient and an arbitrary payload but is

expected to revert with the `Amount must be nonzero` reason string, since the amount to be transferred argument is `0` but must instead be greater than zero.

- The `test_CannotTransferTokensWithPayloadInvalidMintRecipient` function first registers the foreign emitter and domain for the given chain and deals a non-zero USDC amount to the test contract. The `_expectRevert` function is then invoked with an encoded call to `ICircleIntegration::transferTokensWithPayload`, taking the USDC address as the `token` argument, the zero address as the `mintRecipient` and an arbitrary payload. This call is expected to revert with the `Mint recipient must be nonzero` reason string, since the mint recipient must be a valid address.
- The `test_CannotTransferTokensWithPayloadTargetContractNotRegistered` function first deals a non-zero USDC amount to the test contract but fails to register the foreign emitter and domain for the given chain. The `_expectRevert` function is then invoked with an encoded call to `ICircleIntegration::transferTokensWithPayload`, taking the USDC address as the `token` argument, the zero address as the `mintRecipient`, a `tagetChain` of `1` and an arbitrary payload. This call is expected to revert with the `target contract not registered` reason string, since the destination caller is required to exist both on the target chain and in the registered emitters mapping.
- The `test_TransferTokensWithPayload` function first bounds the fuzzed `amount` parameter between 1 and the burn limit while also assuming that the mint receipient is not the zero address. The foreign emitter and domain are registered for the given chain and a bytes array of arbitrary payloads is initialized. Twice the bounded USDC amount is then dealt to the test contract before caching the current balance. The `vm.recordLogs` cheatcode is used to record all transaction logs in the subsequent calls to `ICircleIntegration::transferTokensWithPayload` which are invoked with the distinct arbitrary payloads, the same amount, the same Wormhole nonce of `420`, and have their Wormhole sequence numbers stored in a `uint64` array. The published Wormhole payloads are then extracted from the logs, using `WormholeOverride::fetchWormholePublishedPayloads` and asserting that the length of the array is `2`, corresponding to the earlier calls.

This payloadd are also both asserted to equal the bytes returned by `WormholeCctpMessages::encodeDeposit`, invoked with: the fuzzed `amount` transferred, source CCTP domain `7` (Polygon Mumbai), target CCTP domain `1` (Avalanche Fuji), a nonce determined by the `IMessageTransmitter::nextAvailableNonce` function and decremented by the index in the `sequences` array, the address of the test contract as the burn source, the fuzzed `mintRecipient` and the corresponding payload. Finally, there is an assertion that the USDC balance of the test contract has decreased by twice the `amount`, implying that both cross-chain transfers were successful in that each burn amount has in fact been burned but notably not verifying that the funds were indeed redeemable on the other side.

- The `test_CannotRedeemTokensWithPayloadUnknownEmitter` function constructs its redeem parameters by invoking `CircleIntegrationOverride::craftRedeemParameters` but does not register the foreign emitter and domain for the given chain, so the call to `ICircleIntegration::redeemTokensWithPayload` is expected to revert with the `unknown emitter` reason string since the encoded VAA must come from a registered Wormhole Circle Integration contract.
- The `test_CannotRedeemTokensWithPayloadCallerMustBeMintRecipient` function first assumes that the fuzzed `mintRecipient` parameter is not the zero address or the address of the test contract and registers the foreign emitter and domain for the given chain. The encoded call to `ICircleIntegration::redeemTokensWithPayload` is then expected to revert with the `caller must be mintRecipient` reason string, since the `mintRecipient` argument is listed as the `0xdeadbeef` address but the actual caller is the test contract itself.
- The `test_CannotRedeemTokensWithPayloadMintTokenNotSupported` function first assumes that the fuzzed `remoteToken` parameter is not equal to the `FOREIGN_USDC_ADDRESS` constant and registers the foreign emitter and domain for the given chain. The encoded call to `ICircleIntegration::redeemTokensWithPayload` is then expected to revert with the `Mint token not supported` reason string, since the token to be minted must be registered with the CCTP Token Minter contract (if it is not, then `TokenMinter::getLocalToken` returns the zero address).
- The `test_CannotRedeemTokensWithPayloadInvalidMessagePair` function first registers the

foreign emitter and domain for the given chain and crafts two sets of redeem parameters using `CircleIntegrationOverride::craftRedeemParameters`, differing only in their CCTP nonce and Wormhole sequence values. The encoded call to `ICircleIntegration::redeemTokensWithPayload` is then expected to revert with the `invalid message pair` reason string, since the encoded VAA payloads were previously swapped between the two sets of redeem parameters but VAA CCTP nonce and encoded CCTP message nonce values must be equal.

- The `test_RedeemTokensWithPayload` function first bounds the fuzzed `amount` parameter between 1 and the burn limit and registers the foreign emitter and domain for the given chain. A set of deposit and redeem parameters are crafted using `CircleIntegrationOverride::craftRedeemParameters` and `CircleIntegrationOverride::craftCctpTokenBurnMessage` respectively, using some fields of the former to construct the latter. The USDC balance of the test contract is then cached before invoking the `ICircleIntegration::redeemTokensWithPayload` with the redeem parameters, asserting that the hash of the returned `DepositWithPayload` struct is equal to that expected. Finally, there is an assertion that the USDC balance of the test contract has increased by `amount`, implying that the cross-chain transfer was successful in that the mint amount has in fact been minted but notably not verifying that the funds were indeed burnt on the source chain.
- The `test_RedeemTokensWithFuzzedPayload` function performs the same steps as the previous test but with a fuzzed payload, assuming that its length is between `1` and `type(uint16).max` bytes. This should provide good guarantees that the payload cannot interfere with the cross-chain transfer; however, the impact of a carefully crafted payload should be explored further.
- Internal functions: a number of helper functions are defined for the purposes of code reuse and readability. These include:
  - The overloaded `_getUsdcBalance` functions return the USDC balance of either the specified `owner` address or `address(this)`. Both functions appear to be unused in the current file.
  - The `_expectRevert` function is used to assert that a given call reverts with the specified reason string. Note that to circumvent a quirk of the `vm.expectRevert` cheatcode, the encoded call is instead passed as an argument to this function, within which the actual call is executed, to ensure that the correct context is used.
  - The `_dealAndApproveUsdc` function is used to mint USDC to the `circleIntegration` address by leveraging the `UsdcDeal::dealAndApprove` function. Note that whilst the integration contract now has an approval to spend USDC from the test contract, it does not, at this stage, have any USDC in its balance.
  - The `_cctpBurnLimit` function queries a chain of contracts (`Circle Integration -> CCTP Token Messenger -> CCTP Token Minter`) for the USDC burn limit per message for the specified `domain` and returns the result having validated that it is non-zero to prevent the test from executing on a forked network where Circle has not set a burn limit.
  - The `_cctpMintLimit` function simply returns the result of calling `_cctpBurnLimit`. There is an inline comment that explains that this case of having the two limits equal is not expected to occur in practice since there is a mint allowance that is enforced by the USDC contract per registered minter; however, for the purposes of testing: "We use this out of convenience since inbound transfers can never be greater than outbound transfers (which are managed by the burn limit)".
  - The `_registerEmitterAndDomain` function registers the Avalanche emitter and domain using values hardcoded in the function body in conjunction with the `vm.store` cheatcode. The `foreignEmitter` variable is stored at the slot given by `keccak256(abi.encode(foreignChain, uint256(6)))` of the `circleIntegration` contract while the `cctpDomain` variable is stored at the slot given by `keccak256(abi.encode(foreignChain, uint256(7)))` and the `foreignChain` variable is stored at the slot given by `keccak256(abi.encode(cctpDomain, uint256(8)))`.
  - The `Error` function simply takes and immediately returns a string from/to memory, for use as an encoded call within the `_expectRevert` function.

- `ForkSlots.t.sol` is written to validate the correctness of state changes that occur as part of the proxy upgrade, namely that slots `0x0 - 0x4` and `0xA` are correctly zeroed while the others remain unchanged.

Originally, this fork test was failing with the mapped slot `0x8` not being equal to the expected value both before and after the upgrade. This was due to Circle migrating their contracts from Goerli to Sepolia, meaning the Circle Integration contracts were recently updated to change registered Wormhole chain IDs to reflect that (Goerli's chain ID to the same as mainnet's, and Sepolia's is `10002`). More details can be found here. Changing `expectedRegisteredChainId` to anything already registered (`6, 10002`) fixes the issue and test now passes. An outline of the fork test file is shown below:

  - Setup: this test file defines two constant values `GOVERNANCE_MODULE` and `FORKED_CIRCLE_INTEGRATION_ADDRESS` for use in the fork test – the `forked` state variable is simply the `FORKED_CIRCLE_INTEGRATION_ADDRESS` value cast to the `ICircleIntegration` interface, with `forkedAddress` somewhat redundantly cast back to an address. State variables corresponding to the expected storage of the proxy are also declared with comments indicating their respective expected slots and initialized in the Forge `setUp` function based on the current state of the forked contract.
  - `test_UpgradeForkAndCheckSlots` performs the proxy upgrade, first deploying the new implementation contract and asserting that the forked contract is not yet initialized with the new implementation. A Governance VAA is crafted for the purpose of performing the contract upgrade, so this behavior is not directly tested; however, it is validated that the VAA has not yet been consumed. Some additional values for the `expectedEmitter` and `expectedCctpDomain` are fetched before performing the upgrade and slot assertions. If the zero slot data is already zeroed, the test checks the remaining zeroed slots; otherwise, it checks that the slots to be zeroed are the expected values from the existing getters. The test always validates the value of the slots that are to remain non-zeroed, ensuring that they equal the expected values both before and after the upgrade. Following the initial assertions, the contract is upgraded to the new implementation and initialized, asserting that this is indeed the case. The Governance VAA is then asserted to have been consumed and the remaining slot values are asserted to be equal to the expected values. The state that has been refactored into immutables is also validated by calling their respective getter functions and asserting that their return values are equal to the expected values from before. Note that while the state of the mapping slots in the proxy contract is validated, it is of course not feasible to validate the state corresponding to the slot of each key in the mapping and so this is not tested or guaranteed to remain unchanged.

- `Governance.t.sol` is designed to test the Governance functionality of the Wormhole CCTP integration contract. An outline of the test file and a summary of its comprehensive tests is shown below:
  - Setup: this test file defines three constant variables `GOVERNANCE_MODULE`, `GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN` and `GOVERNANCE_UPGRADE_CONTRACT` with the values `0x000000000000000000000000000000436972636c65496e746567726174696f6e`, `2` and `3` respectively, for use with the `WormholeOverride::craftGovernanceVaa` function. The `setupWormhole` function instantiates the Wormhole contract as defined by the `TESTING_WORMHOLE_ADDRESS` environment variable while the `setupCircleIntegration` function deploys an instance of the `Setup` contract alongside the implementation contract for use with the newly deployed proxy. This proxy is then cast to the `ICircleIntegration` interface before calling the `CircleIntegrationOverride::setUpOverride` function and asserting that the the state changes executed as expected. These two functions are invoked in the Forge `setUp` function.
  - The `test_CannotConsumeGovernanceMessageInvalidGovernanceChainId` function first assumes that the fuzzed `governanceChainId` parameter does not equal the current Governance chain ID of the Wormhole CCTP integration contract, such that the encoded Governance VAA should be invalid for every input. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded Governance VAA with the fuzzed `governanceChainId` and `action` parameters, along with the derived Governance contract, target chain ID, `GOVERNANCE_MODULE` constant, a Wormhole sequence of `69` and an arbitrary payload. The call to `IGovernance::verifyGovernanceMessage` is finally expected to revert with the `invalid governance chain` reason string, since the Governance emitter chain ID must be equal to `1`.
  - The `test_CannotConsumeGovernanceMessageInvalidGovernanceContract` function first assumes that the fuzzed `governanceContract` parameter does not equal the current Governance contract of the Wormhole CCTP integration contract, such that the encoded Governance VAA should be invalid for every input. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded

Governance VAA with the fuzzed `governanceContract` and `action` parameters, along with the derived Governance chain ID, target chain ID, `GOVERNANCE_MODULE` constant, a Wormhole sequence of `69` and an arbitrary payload. The call to `IGovernance::verifyGovernanceMessage` is finally expected to revert with the `invalid governance contract` reason string, since the Governance emitter contract must be equal to `0x0000000000000000000000000000000000000000000000000000000000000004`.

– The `test_CannotConsumeGovernanceMessageInvalidModule` function first assumes that the fuzzed `governanceModule` parameter does not equal the current Governance module of the Wormhole CCTP integration contract, such that the encoded Governance VAA should be invalid for every input. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded Governance VAA with the fuzzed `governanceModule` and `action` parameters, along with the derived target chain ID, a Wormhole sequence of `69` and an arbitrary payload. The call to `IGovernance::verifyGovernanceMessage` is finally expected to revert with the `invalid governance module` reason string, since the Governance emitter module must be equal to `"CircleIntegration"` (left-padded with zeros).

– The `test_CannotConsumeGovernanceMessageInvalidAction` function first assumes that the fuzzed `action` parameter does not equal the fuzzed `wrongAction`. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded Governance VAA with the fuzzed `action` parameter, along with the derived target chain ID, `GOVERNANCE_MODULE` constant, a Wormhole sequence of `69` and an arbitrary payload. The call to `IGovernance::verifyGovernanceMessage` is finally expected to revert with the `invalid governance action` reason string, since the `wrongAction` parameter is passed in place of `action`.

– The `test_CannotRegisterEmitterAndDomainInvalidLength` function first assumes that the fuzzed `foreignChain` parameter is non-zero and does not equal the current local chain ID of the Wormhole CCTP integration contract. The fuzzed `foreignEmitter` parameter is also assumed to be non-zero, along with the fuzzed `domain` parameter which is also assumed to differ from the current local CCTP domain of the Wormhole CCTP integration contract. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded Governance VAA with the `GOVERNANCE_MODULE` and `GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN` constants, along with the derived target chain ID, a Wormhole sequence of `69` and a payload which in composed of the `foreignChain`, `foreignEmitter`, `domain` and an arbitrary string. The call to `IGovernance::registerEmitterAndDomain` is finally expected to revert with the `invalid governance payload length` reason string, since the length of encoded Governance VAA payload is not consistent with the expected length.

– The `test_CannotRegisterEmitterAndDomainInvalidTargetChain` function first assumes that the fuzzed `targetChain` and `foreignChain` parameters are non-zero and do not equal the current local chain ID of the Wormhole CCTP integration contract. The fuzzed `foreignEmitter` parameter is also assumed to be non-zero, along with the fuzzed `domain` parameter which is also assumed to differ from the current local CCTP domain of the Wormhole CCTP integration contract. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded Governance VAA with the `GOVERNANCE_MODULE` and `GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN` constants, along with the fuzzed `targetChain` parameter, a Wormhole sequence of `69` and a payload which in composed of the fuzzed `foreignChain`, `foreignEmitter` and `domain` parameters. The call to `IGovernance::registerEmitterAndDomain` is finally expected to revert with the `invalid target chain` reason string, since the `foreignChain` encoded within the VAA payload decree is not consistent with the `targetChain`.

– The `test_CannotRegisterEmitterAndDomainInvalidForeignChain` function first assumes that the fuzzed `foreignEmitter` parameter is non-zero, along with the fuzzed `domain` parameter which is also assumed to differ from the current local CCTP domain of the Wormhole CCTP integration contract. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain. The `WormholeOverride::craftGovernanceVaa` function is then invoked to craft an encoded Governance VAA with the `GOVERNANCE_MODULE` and `GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN` constants, along with the derived target chain ID, a Wormhole sequence of 69 and a payload which in composed of the fuzzed foreignChain, foreignEmitter and domain parameters. Note that foreignChain is modified to take one of two values: 0 or the current Wormhole chain ID. The subsequent calls to IGovernance::registerEmitterAndDomain are finally expected to revert with the invalid chain reason string, since the foreignChain must not equal zero or the current Wormhole chain ID.

– The test_CannotRegisterEmitterAndDomainInvalidEmitterAddress function first assumes that the fuzzed foreignChain parameter is non-zero and does not equal the current Wormhole chain ID, along with the fuzzed domain parameter which is also assumed to differ from the current local CCTP domain of the Wormhole CCTP integration contract. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain. The WormholeOverride::craftGovernanceVaa function is then invoked to craft an encoded Governance VAA with the GOVERNANCE_MODULE and GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN constants, along with the derived target chain ID, a Wormhole sequence of 69 and a payload which in composed of the fuzzed foreignChain and domain parameters and a bytes32(0) foreign emitter. The subsequent call to IGovernance::registerEmitterAndDomain is finally expected to revert with the emitter cannot be zero address reason string, since the foreignEmitter must not equal the zero address.

– The test_CannotRegisterEmitterAndDomainInvalidDomain function first assumes that the fuzzed foreignChain parameter is non-zero and does not equal the current Wormhole chain ID, along with the fuzzed foreignEmitter parameter which is also assumed to be non-zero. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain (which itself is assigned the local CCTP domain). The WormholeOverride::craftGovernanceVaa function is then invoked to craft an encoded Governance VAA with the GOVERNANCE_MODULE and GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN constants, along with the derived target chain ID, a Wormhole sequence of 69 and a payload which in composed of the fuzzed foreignChain, foreignEmitter and domain parameters. The subsequent call to IGovernance::registerEmitterAndDomain is finally expected to revert with the domain == localDomain() reason string, since the domain for the foreign chain must not equal the local CCTP domain.

– The test_RegisterEmitterAndDomainNoTarget function first initializes a foreignChain of 42069 along with a domain of 69420 and foreignEmitter loaded from the TESTING_FOREIGN_USDC_TOKEN_ADDRESS environment variable which corresponds to the USDC address on Avalanche Fuji. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain. The WormholeOverride::craftGovernanceVaa function is then invoked to craft an encoded Governance VAA with the GOVERNANCE_MODULE and GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN constants, along with a zero target chain ID, a Wormhole sequence of 69 and a payload which in composed of the earlier initialized foreignChain, foreignEmitter and domain variables. The subsequent call to IGovernance::registerEmitterAndDomain is expected to succeed and the getter functions on the Wormhole CCTP integration contract are then invoked to validate that the foreignChain, foreignEmitter and domain have been registered correctly, since registered emitters are relevant for all chains when the target chain is zero.

– The test_RegisterEmitterAndDomain function first assumes that the fuzzed foreignChain parameter is non-zero and does not equal the current Wormhole chain ID, along with the fuzzed foreignEmitter parameter which is assumed to be non-zero and the fuzzed domain parameter which is also assumed to differ from the current local CCTP domain of the Wormhole CCTP integration contract. This test also asserts that no emitters or domains are already registered for the chain and vice-versa for chains being registered for the domain. The WormholeOverride::craftGovernanceVaa function is then invoked to craft an encoded Governance VAA with the GOVERNANCE_MODULE and GOVERNANCE_REGISTER_EMITTER_AND_DOMAIN constants, along with the derived target chain ID, a Wormhole sequence of 69 and a payload which in composed of the fuzzed foreignChain, foreignEmitter and domain parameters. The subsequent call to IGovernance::registerEmitterAndDomain is expected to succeed and the getter functions on the Wormhole CCTP integration contract are then invoked to validate that the foreignChain, foreignEmitter and domain have been registered correctly, since registered emitters are relevant for all chains when the target chain is zero. This behavior so far is similar to that of the test above, but additionally verifies that it is not possible to register an emitter for the same chain again, this time passing a Wormhole sequence of 70 and asserting that the crafted VAA causes the call to IGovernance::registerEmitterAndDomain to revert with the chain already registered reason string.

– The test_CannotUpgradeContractInvalidImplementation function first assumes that the fuzzed garbage parameter is non-zero, along with the fuzzed newImplementation parameter which is also assumed to differ from the current implementation of the Wormhole CCTP integration contract. The WormholeOverride::craftGovernanceVaa function is then invoked to craft an encoded Governance VAA with the GOVERNANCE_MODULE and GOVERNANCE_UPGRADE_CONTRACT constants, along with the derived target chain ID, a Wormhole sequence of 69 and a payload which in composed of the fuzzed garbage and newImplementation parameters. The subsequent call to IGovernance::upgradeContract is expected to revert with the invalid address reason string, since the newImplementation must be a valid contract address left-padded with zero bytes. Another call to IGovernance::upgradeContract is then expected to revert with the invalid implementation reason string, since the newImplementation appears to be valid but does not return the necessary "circleIntegrationImplementation()" string when static-called. Using one of Wormhole's existing implementations, the call to IGovernance::upgradeContract is finally expected to revert again with the invalid implementation same reason string (however it is not clear to which chain/contract this address refers).

– The test_UpgradeContract function first deploys a new instance of the implementation contract and asserts that the proxy is not yet initialized with this new implementation. It then crafts an encoded Governance VAA with the GOVERNANCE_MODULE and GOVERNANCE_UPGRADE_CONTRACT constants, along with the derived target chain ID, a Wormhole sequence of 69 and a payload which in composed of the GOVERNANCE_MODULE constant and the address of the new implementation contract. The subsequent call to IGovernance::upgradeContract is expected to succeed and the getter functions on the Wormhole CCTP integration contract are then invoked to validate that the new implementation has been initialized correctly. Note that there is a typo in the final dev comment which appears to have been erroneously copied from above and states that the proxy should not yet be initialized with this implementation, when in fact it should be.

• InheritingWormholeCctp.t.sol is designed to test the use of the Wormhole CCTP integration contract by a contract that inherits from it. An outline of the test file is shown below:

– Setup: this test file defines two immutable variables USDC_ADDRESS and FOREIGN_USDC_ADDRESS given by the values loaded from the TESTING_USDC_TOKEN_ADDRESS and TESTING_FOREIGN_USDC_TOKEN_ADDRESS environment variables respectively. Given these correspond to the values in evm/env/testing.env, this means that the test suite is currently configured to use the USDC token on the Polygon Mumbai fork and the USDC token on the Avalanche Fuji testnet. The setupWormhole function instantiates the Wormhole contract as defined by the TESTING_WORMHOLE_ADDRESS environment variable while the setupUSDC function validates that the decimals of the USDC contract are equal to 6. The setupCircleIntegration function deploys an instance of the Setup contract alongside the implementation contract for use with the newly deployed proxy. This proxy is then cast to the ICircleIntegration interface before calling the CircleIntegrationOverride::setUpOverride function and asserting that the the state changes executed as expected. All three functions are invoked in the Forge setUp function which also deploys an instance of the InheritingWormholeCctp contract, passing the Wormhole, CCTP Token Messenger and USDC addresses as arguments to the constructor.

– The test_TransferUsdc function first bounds the transfer amount between 1 and the burn limit while also assuming that the mint receipient is not the zero address. The internal _dealAndApproveUsdc function is called and the balanceBefore variable is assigned to the USDC balance of the test contract, since these tokens were minted but are not yet transferred to the inheritedContract address. The vm.recordLogs cheatcode is used to record all transaction logs in the subsequent call to InheritingWormholeCctp::transferUsdc which is invoked with an arbitrary payload and expected to return a Wormhole sequence number of 0. The published Wormhole payloads are then extracted from the logs, using WormholeOverride::fetchWormholePublishedPayloads and asserting that the length of the array is 1, corresponding to the earlier call. This single payload is also asserted to equal the bytes returned by WormholeCctpMessages::encodeDeposit, invoked with: the fuzzed amount transferred, source CCTP domain 7 (Polygon Mumbai), target CCTP domain 1 (Avalanche Fuji), a nonce determined by the IMessageTransmitter::nextAvailableNonce function and decremented by 1, the address of the test contract as the burn source, the fuzzed `mintRecipient` and the payload. The published CCTP messages are then also extracted from the logs, using `CircleIntegrationOverride::fetchCctpMessages` and asserting that the length of the array is `1`, again corresponding to the earlier call. The `destinationCaller` of the CCTP message header is also asserted to equal that given by the inherited contract (`0xdeadbeef`) but note that no other fields in the CCTP message are checked. Finally, there is an assertion that the USDC balance of the test contract has decreased by `amount`, implying that the cross-chain transfer was successful in that the burn amount has in fact been burned but notably not verifying that the funds were indeed redeemable on the other side.

– Internal functions: a number of helper functions are defined for the purposes of code reuse and readability. Most appear to be copied from `CircleIntegration.t.sol` and unused in the current file. These include:

  * The overloaded `_getUsdcBalance` functions return the USDC balance of either the specified `owner` address or `address(this)`. Both functions appear to be unused in the current file.
  * The `_expectRevert` function is used to assert that a given call reverts with the specified reason string. Note that to circumvent a quirk of the `vm.expectRevert` cheatcode, the encoded call is instead passed as an argument to this function, within which the actual call is executed, to ensure that the correct context is used. Also note that this function executes the call on the `circleIntegration` contract, not the `inheritedContract` address, and appears to be unused in the current file.
  * The `_dealAndApproveUsdc` function is used to mint USDC to the `inheritedContract` address by leveraging the `UsdcDeal::dealAndApprove` function. Note that whilst the inherited contract now has an approval to spend USDC from the test contract, it does not, at this stage, have any USDC in its balance.
  * The `_cctpBurnLimit` function queries a chain of contracts (`Circle Integration -> CCTP Token Messenger -> CCTP Token Minter`) for the USDC burn limit per message for the specified `domain` and returns the result having validated that it is non-zero to prevent the test from executing on a forked network where Circle has not set a burn limit.
  * The `_cctpMintLimit` function simply returns the result of calling `_cctpBurnLimit`. There is an inline comment that explains that this case of having the two limits equal is not expected to occur in practice since there is a mint allowance that is enforced by the USDC contract per registered minter; however, for the purposes of testing: "We use this out of convenience since inbound transfers can never be greater than outbound transfers (which are managed by the burn limit)". Note that this function appears to be unused in the current file.
  * The `_registerEmitterAndDomain` function registers the Avalanche emitter and domain using values hardcoded in the function body in conjunction with the `vm.store` cheatcode. The `foreignEmitter` variable is stored at the slot given by `keccak256(abi.encode(foreignChain, uint256(6)))` of the `circleIntegration` contract while the `cctpDomain` variable is stored at the slot given by `keccak256(abi.encode(foreignChain, uint256(7)))` and the `foreignChain` variable is stored at the slot given by `keccak256(abi.encode(cctpDomain, uint256(8)))`. Note that this function appears to be unused in the current file.
  * The `Error` function simply takes and immediately returns a string from/to memory but appears unused in the current file.

• `WormholeCctpMessages.t.sol` contains a single test function `test_DepositWithPayloadSerde` which is used to test the serialization and deserialization of an encoded Wormhole CCTP Deposit Message. It first assumes the source and target CCTP domains to be distinct and the CCTP payload length to be bounded between `1` and `type(uint16).max - 1`. A fake VAA is declared so that the `WormholeCctpMessages::encodeDeposit` function can be invoked with the specified `amount`, `sourceDomain`, `targetDomain`, `cctpNonce`, `burnSource`, `mintRecipient` and `payload` arguments. The resulting encoded bytes representing the Wormhole message payload are then assigned to the fake VAA and asserted to equal `147` plus the CCTP message payload, since `147` is the encoded message length including the payload ID and payload length. The payload ID is then extracted from the VAA payload and asserted to equal `1`, corresponding to a deposit payload. The `WormholeCctpMessages::decodeDeposit` function is then invoked on the fake VAA, containing the encoded Wormhole message payload, and is expected to return the same values as those passed to `WormholeCctpMessages::encodeDeposit` originally. Thus, this function tests the serialization and deserialization of a Wormhole CCTP Deposit Message but does not test the validity of the VAA itself.

- `CircleIntegrationComparison.t.sol` contains a series of tests that report a comparison of the gas usage when interacting with the `WormholeCctpTokenMessenger` contract via both inheritance and direct composition. The setup is very similar to that of `InheritingWormholeCctp.t.sol` except the instantiation of a composed contract in addition to the inherited contract and also a forked instance of the Circle integration contract in addition to a locally deployed instance. It is interesting to note that the `CircleIntegrationSetup::setup` function enforces no access control, so it appears possible for any caller to upgrade the implementation of the test proxy to interfere with the repos fork tests. The internal functions of this test contract are also similar to that of `InheritingWormholeCctp.t.sol` but with the addition of `_generatePayload512` which takes a `bytes32 data` parameter that is repeatedly encoded 16 times to generate a 512 byte payload. A summary of the gas comparison tests:
  - `test_Inherited__TransferUsdc` and `test_Composed__TransferUsdc` call their respective `transferUsdc` functions with the same setup and assumptions, including the `payload` parameter which is set to the result of calling `_generatePayload512`. `test_Latest__TransferTokensWithPayload` and `test_Fork__TransferTokensWithPayload` do the same but for the latest and forked Circle integration contracts respectively, using `420` for the Wormhole nonce instead of `0`. `test_Control__TransferTokensWithPayload` is used to report a control value for the gas usage of the setup steps, excluding any calls to the Circle integration contract.
  - `test_Inherited__RedeemUsdc` and `test_Composed__RedeemUsdc` call their respective `redeemUsdc` functions with the same setup and assumptions, including the `payload` parameter which is set to the result of calling `_generatePayload512`. `test_Latest__RedeemTokensWithPayload` and `test_Fork__RedeemTokensWithPayload` do the same but for the latest and forked Circle integration contracts respectively. Each of these uses the max `uint64` value as the CCTP nonce and `88` as the Wormhole sequence number. `test_Control__RedeemTokensWithPayload` is used to report a control value for the gas usage of the setup steps, excluding any calls to the Circle integration contract.
- `ComposingWithCircleIntegration.sol` contains a contract which interacts with the Circle integration contract by composing it within its own contract rather than inheriting it. It exposes the constants `_MY_BFF_DOMAIN` and `_MY_BFF_ADDR` which are set to `1` and `0xdeadbeef` respectively, along with their respective getter functions `myBffDomain` and `myBffAddr`. Within the constructor, a max approval is set on the USDC contract for the CCTP Token Messenger contract via `IUSDC::forceApprove` since it is required to pull funds from the calling contract. This differs from `Logic::transferTokensWithPayload` which only approves the tokens to spend for each individual call, seemingly simplifying the test setup. The `transferUsdc` function transfers funds from the caller (test contract) and invokes `WormholeCctpTokenMessenger::burnAndPublish` via `ICircleIntegration::transferTokensWithPayload` (defined in `Logic.sol`) with the relevant parameters. Similarly, the `redeemUsdc` function takes the encoded `RedeemParameters`, including the encoded CCTP message, CCTP attestation and encoded VAA, and invokes `WormholeCctpTokenMessenger::verifyVaaAndMint` via `ICircleIntegration::redeemTokensWithPayload` before transferring the deposit amount to the `msg.sender` address.
- `InheritingWormholeCctp.sol` contains a contract which inherits `WormholeCctpTokenMessenger` for use in the `InheritingWormholeCctpTest` contract. It exposes the constants `_MY_BFF_DOMAIN` and `_MY_BFF_ADDR` which are set to `1` and `0xdeadbeef` respectively, along with their respective getter functions `myBffDomain` and `myBffAddr`. Within the constructor, a max approval is set on the USDC contract for the CCTP Token Messenger contract since it is required to pull funds from the calling contract. This differs from `Logic::transferTokensWithPayload` which only approves the tokens to spend for each individual call, seemingly simplifying the test setup. The `transferUsdc` function transfers funds from the caller (test contract) and invokes `WormholeCctpTokenMessenger::burnAndPublish` with the relevant parameters. Similarly, the `redeemUsdc` function takes the encoded CCTP message, CCTP attestation and encoded VAA as parameters and invoked `WormholeCctpTokenMessenger::verifyVaaAndMint`. Overall, this is a light wrapper over the `WormholeCctpTokenMessenger` contract.

8.2 Script Analysis

A number of scripts that perform a variety of functions are made available in the `evm/sh`, `evm/ts/scripts` and `evm/forge/scripts` directories.

8.2.1 `evm/sh`

A tree of the `evm/sh` directory structure is shown below.

- `deploy_contracts.sh` invokes the `deploy_contracts.sol` script in the `evm/forge/scripts` directory, passing the `RPC` and `PRIVATE_KEY` environment variables as options along with the `--broadcast --slow` flags which broadcasts the transactions and encures a transaction is sent only after its previous one has been confirmed and succeeded.
- `deploy_implementation_only.sh` invokes the `deploy_implementation_only.sol` script in the `evm/forge/scripts` directory with verbosity level 2, passing the `RPC` and `PRIVATE_KEY` environment variables as options along with the `--broadcast --slow` flags. The command `set -euo pipefail` is used to modify the behavior of the script to make it more robust and secure. `-e` causes the script to exit immediately if any command within it exits with a non-zero status (an error). This helps catch errors and prevents the script from continuing with potentially undefined or unexpected state. `-u` causes Bash to treat undefined variables as an error when they are referenced, and immediately exits. This can prevent bugs that occur due to typos in variable names or assumptions about the environment in which the script is running. `-o pipefail` affects pipelines (chains of commands connected with |). Normally, the exit status of a pipeline is the exit status of the last command. With pipefail enabled, the pipeline's status is the exit status of the last command to exit with a non-zero status, or zero if all commands exit successfully. This makes it easier to detect failures in any part of a pipeline.
- `make_ethers_types.sh` invokes the `typechain` command to generate the `ethers` types for the artifacts in the `evm/out` directory, passing the `--target ethers-v5` flag to ensure the types are compatible with the `ethers` library. The resulting typechain files are written to the `evm/ts/src` directory.
- `submit_testnet_registration.sh` invokes the `generate_registration_vaa.sol` script in the `evm/forge/scripts` directory with verbosity level 2, passing the `RPC` and `PRIVATE_KEY` environment variables as options along with the `--broadcast --slow` flags. It also exports a series of environment variables from the positional arguments passed to the script, including `TARGET_CHAIN`, `FOREIGN_CHAIN`, `FOREIGN_EMITTER`, `FOREIGN_DOMAIN` and `SIGNER_KEY`. The `set -euo pipefail` command is also used here.
- `upgrade_proxy.sh` invokes the `ts-node` command to run the `upgrade_proxy.ts` script in the `evm/ts` directory, passing all the positional arguments with `$@`. The `set -euo pipefail` command is also used here.
- `verify_implementation.sh` runs the `forge verify-contract` command to verify the implementation contract on Etherscan, passing the hardcoded `v0.8.19` solc compiler version and `--watch` option which waits for the verification result after submission. `RELEASE_EVM_CHAIN_ID` and `CIRCLE_INTEGRATION_IMPLEMENTATION` are referenced as an environment variables, and the Etherscan API key is passed as an argument.
- `verify_proxy.sh` runs the `forge verify-contract` command to verify the proxy contract on Etherscan, passing the hardcoded `v0.8.19` solc compiler version and `--watch` option which waits for the verification result after submission. `RELEASE_EVM_CHAIN_ID` and `CIRCLE_INTEGRATION_PROXY` are referenced as an environment variables, and the Etherscan API key is passed as an argument. Abi-encoded calldata representing constructor arguments and setup data is also crafted using a combination of environment variables, hardcoded variables and the `cast calldata` and `cast abi-encode` commands.
- `verify_setup.sh` runs the `forge verify-contract` command to verify the setup contract on Etherscan, passing the hardcoded `v0.8.19` solc compiler version and `--watch` option which waits for the verification result after submission. `RELEASE_EVM_CHAIN_ID` and `CIRCLE_INTEGRATION_SETUP` are referenced as an environment variables, and the Etherscan API key is passed as an argument.

8.2.2 `evm/ts/scripts`

A tree of the `evm/ts/scripts` directory structure is shown below.

- `contract_governance.ts` instantiates the relevent Wormhole and Circle integration contracts for use with a mock Circle Governance Emitter which produces Governance VAAs for the `CircleAttestation` contract. This file tests whether an emitter and its domain can be successfully registered but does not appear to be in use anywhere.
- `sample.env` contains a number of environment variables used solely in the `contract_governance.ts` script.
- `upgrade_proxy.ts` provides a simple command-line interface for upgrading the proxy contract. It is invoked by the `upgrade_proxy.sh` script in the `evm/sh` directory and is passed all the positional arguments with `$@`. If the positional arguments are not provided, the script prompts the user to enter them manually within its `setUp` function, the output of which is then used in its `main` function to send the upgrade transaction.

8.2.3 `evm/forge/scripts`

A tree of the `evm/forge/scripts` directory structure is shown below.

- `deploy_contracts.sol` first loads the Wormhole and Circle contracts from environment variables using `vm.envAddress` in the `setUp` function. The `Setup` contract is then used to upgrade and initialize the proxy contract with the deployed implementation contract within `deployCircleIntegration` which is invoked between broadcast commands in the `run` function.
- `deploy_implementation_only.sol` simply deploys the Circle integration implementation contract and logs the address of the deployed contract without upgrading the proxy contract.
- `deploy_mock_contracts.sol` deploys a mock Circle integration contract that appears to no longer exist.
- `read_governance_variables.sol` instantiates the Circle integration proxy contract using the `vm.envAddress` cheatcode in the `setUp` function. This contract is then used to read the registered emitter, domain and initialized state variables from the Circle integration contract and log them to the console, using the `vm.envUint` and `vm.envAddress` cheatcodes to provide the relevant arguments.
- `submit_testnet_registration.sol` provides a number of helper functions to make and sign a Governance observation which is then used to craft and submit a Governance VAA to register an emitter and domain.