Informações de Debug do contrato `SpyErcBridge`

Esses são os argumentos que foram passados para a função analisada.

```python
def parse_args_and_execute(parser: ArgumentParser, args: Namespace) -> None:
```

```python
parser=
{'add_help': True,
 'allow_abbrev': True,
 'argument_default': 'None',
 'conflict_handler': 'error',
 'description': 'Security a...',
 'epilog': 'None',
 'exit_on_error': True,
 'fromfile_prefix_chars': 'None',
 'prefix_chars': '-',
 'prog': 'myth',
 'usage': 'None'}
```

```python
args=
{'address': 'None',
 'attacker_address': 'None',
 'beam_search': 'None',
 'bin_runtime': False,
 'call_depth_limit': 3,
 'code': 'None',
 'codefile': 'None',
 'command': 'analyze',
 'create_timeout': 30,
 'creator_address': 'None',
 'custom_modules_directory': 'None',
 'disable_coverage_strategy': False,
 'disable_dependency_pruning': False,
```

```
'disable_iprof': False,
'disable_mutation_pruner': False,
'enable_physics': False,
'enable_state_merging': False,
'enable_summaries': False,
'epic': False,
'execution_timeout': 3600,
'graph': 'None',
'infura_id': 'None',
'loop_bound': 3,
'max_depth': 128,
'modules': 'None',
'no_onchain_data': False,
'outform': 'text',
'parallel_solving': False,
'phrack': False,
'pruning_factor': 'None',
'query_signature': False,
'rpc': 'infura-mai...',
'rpctls': False,
'solc_args': 'None',
'solc_json': 'None',
'solidity_files': ['/home/mat/workspace/mythril/sContract.sol'],
'solv': 'None',
'solver_log': 'None',
'solver_timeout': 25000,
'statespace_json': 'None',
'strategy': 'bfs',
'transaction_count': 2,
'transaction_sequences': 'None',
'unconstrained_storage': False,
'v': 2}
```


1. Etapas do Pipeline de Análise do Mythril

Transformação de Código

Compilação do Solidity para Bytecode:

- O processo inicia com a compilação do código-fonte Solidity para bytecode EVM (Ethereum Virtual Machine) utilizando o compilador **solc**.
- Essa compilação pode ser realizada internamente pelo Mythril, caso o código-fonte seja fornecido. Alternativamente, o bytecode pode ser obtido diretamente se o contrato já estiver implantado na blockchain.

Representações Intermediárias (IRs):

- Após obter o bytecode, o Mythril o converte em representações intermediárias (IRs), mais adequadas para análise simbólica.

Finalidade das IRs:

- Facilitar a compreensão e manipulação das instruções de baixo nível do EVM.
- Permitir uma modelagem precisa da lógica do contrato para a análise.

As IRs incluem:

- Estruturas como o **Fluxo de Controle** (*Control Flow Graph - CFG*).
- Instruções anotadas com metadados.

Uso das IRs na Análise:

- Servem como base para construir o espaço de estados do contrato.
- Permitem identificar caminhos de execução e estados possíveis.
- Auxiliam na detecção de vulnerabilidades ao modelar o impacto de diferentes entradas no comportamento do contrato.

Desmontagem (Disassembly)

Operação do Disassembler:

- O disassembler do Mythril analisa o bytecode e extrai as instruções EVM individuais.
- Realiza o mapeamento do bytecode para instruções de alto nível, identificando funções, saltos condicionais e outras operações.

Informações Extraídas:

- **Lista de Instruções:** Sequência ordenada de instruções EVM.
- **Mapeamento de Funções:** Identificação de endereços e hashes das funções do contrato.
- **Fluxo de Controle:** Estruturas de controle como loops e condicionais.

Construção do Espaço de Estados:

- As informações extraídas são usadas para modelar todas as possíveis execuções do contrato, permitindo ao analisador explorar diferentes caminhos.

Geração de Espaço de Estados

Construção a partir do Bytecode:

- O Mythril inicia a partir do ponto de entrada do contrato e simula a execução das instruções.

- Utiliza um **Simulador de Máquina Virtual (SVM)** para rastrear estados globais que representam diferentes cenários de execução.

Informações Utilizadas:

- **Instruções e Operações:** Para compreender a lógica do contrato.
- **Fluxo de Controle:** Para determinar caminhos possíveis.
- **Variáveis e Armazenamento:** Para monitorar o estado do armazenamento.

Modelagem da Lógica do Contrato:

- O analisador simbólico mantém uma representação simbólica das variáveis, permitindo explorar combinações diferentes de valores de entrada.
- Constrói um grafo de execução que representa todas as transições de estado possíveis.

---

2. Informações de Depuração

Finalidade

- **args (Argumentos):** Contêm os parâmetros fornecidos pelo usuário ao executar o Mythril, como arquivos de entrada, estratégias de análise e opções específicas.
- **parser (Analisador de Argumentos):** Interpreta os argumentos de linha de comando e assegura a configuração correta da análise.

Contribuição para a Análise:

- Determina como a análise será conduzida e configura módulos, definindo parâmetros que afetam a profundidade e abrangência.

Momento

- **parser:** Inicializado no início da execução do programa, antes de qualquer análise.
- **args:** Gerados imediatamente após o parser interpretar os argumentos fornecidos pelo usuário.

Origem dos Dados:

- Os argumentos são fornecidos manualmente pelo usuário e não extraídos do bytecode.

Uso

- **Identificação de Vulnerabilidades:** Os **args** determinam quais módulos de detecção serão usados e como os resultados serão formatados.
- **Integração com Módulos:** Configurações como `max_depth` e `strategy` influenciam a exploração do espaço de estados, enquanto parâmetros como `modules` definem quais vulnerabilidades serão procuradas.

---

3. Integração com Ferramentas Internas e Externas

SMT Solver

Uso pelo Mythril:

- O Mythril utiliza um **SMT solver** (geralmente o Z3) para resolver condições lógicas complexas.

Dados Passados ao Solver:

- **Expressões Simbólicas:** Representações matemáticas das condições do contrato.
- **Restrições (constraints):** Condições que devem ser verdadeiras para atingir um estado específico.

Retorno dos Resultados:

- Indica se as condições são satisfatíveis (**sat**) ou insatisfatíveis (**unsat**).
- Caso sejam satisfatíveis, gera exemplos concretos (inputs) para atingir uma determinada condição.

Módulos Internos

Detecção de Vulnerabilidades:

- Módulos especializados identificam padrões como reentrância ou overflow aritmético.
- O módulo de reentrância, por exemplo, monitora chamadas externas e mudanças de estado subsequentes.

Colaboração na Análise:

- O disassembler fornece instruções e mapeamentos necessários.
- A análise simbólica utiliza essas informações para explorar estados possíveis, enquanto os módulos verificam condições específicas.

---

4. Mapeamento entre Código-Fonte e Bytecode

Correlação de Vulnerabilidades

- O Mythril usa mapeamentos gerados durante a compilação para correlacionar posições no bytecode com linhas no código Solidity.

Uso dos `solc_mappings`:

- Contêm informações sobre offsets, comprimentos e índices de arquivos, permitindo mapear instruções EVM para linhas específicas no código-fonte.

Código - Observações

- Locais marcados com `# f`:
  Indicam que, durante a execução do contrato analisado, a condição **não foi satisfeita** e, portanto, **não entrou** na condicional associada.

- Locais marcados com `# v`:
  Indicam que a condição **foi satisfeita** e, durante a execução, o contrato **entrou** na condicional associada.

Essas marcações ajudam a identificar o comportamento do contrato em diferentes cenários de execução.

Python

```python
def parse_args_and_execute(parser: ArgumentParser, args: Namespace) -> None:
    """
    Parses the arguments
    :param parser: The parser
    :param args: The args
    """

    if args.epic: # f
        path = os.path.dirname(os.path.realpath(__file__))
        sys.argv.remove("--epic")
        os.system(" ".join(sys.argv) + " | python3 " + path + "/epic.py")
        sys.exit()

    if args.command not in COMMAND_LIST or args.command is None: # f
        parser.print_help()
        sys.exit()

    if args.command == VERSION_COMMAND: # f
        if args.outform == "json":
```

```
            print(json.dumps({"version_str": VERSION}))
        else:
            print("Mythril version {}".format(VERSION))
        sys.exit()

    if args.command == LIST_DETECTORS_COMMAND: # f
        modules = []
        for module in ModuleLoader().get_detection_modules():
            modules.append({"classname": type(module).__name__, "title":
module.name})
        if args.outform == "json":
            print(json.dumps(modules))
        else:
            for module_data in modules:
                print("{}: {}".format(module_data["classname"],
module_data["title"]))
        sys.exit()

    if args.command == HELP_COMMAND: # f
        parser.print_help()
        sys.exit()

    if args.command in CONCOLIC_LIST:  # f
        _ = MythrilConfig.init_mythril_dir()
        with open(args.input) as f:
            concrete_data = json.load(f)
        output_list = concolic_execution(
            concrete_data, args.branches.split(","), args.solver_timeout
        )
        json.dump(output_list, sys.stdout, indent=4)
        sys.exit()

    # Parse cmdline args
    validate_args(args)
    try:
        if args.command == FUNCTION_TO_HASH_COMMAND: # f
            contract_hash_to_address(args)
        config = set_config(args)
        solc_json = getattr(args, "solc_json", None)
        solv = getattr(args, "solv", None)
        solc_args = getattr(args, "solc_args", None)
        disassembler = MythrilDisassembler(
            eth=config.eth,
            solc_version=solv,
            solc_settings_json=solc_json,
            solc_args=solc_args,
        )
```

```
        address = load_code(disassembler, args)
        execute_command(
            disassembler=disassembler, address=address, parser=parser,
args=args
        )
    except CriticalError as ce:
        exit_with_error(getattr(args, "outform", "text"), str(ce))
    except Exception:
        exit_with_error(getattr(args, "outform", "text"),
traceback.format_exc())
```