PyTorch ExecuTorch

Security Assessment

June 25, 2026

Prepared for:
Open Source Technology Improvement Fund, Inc.

Prepared by: **Artur Cygan, Will Vandevanter, and Fredrik Dahlgren**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Emily Doucette, Project Manager
emily.doucette@trailofbits.com

The following engineering director was associated with this project:

Keith Hoodlet, Engineering Director, Application Security
keith.hoodlet@trailofbits.com

The following consultants were associated with this project:

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Will Vandevanter, Consultant
will.vandevanter@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 8, 2026	Pre-project kickoff call
January 20, 2026	Status update meeting #1
January 26, 2026	Status update meeting #2
February 2, 2026	Status update meeting #2
February 9, 2026	Status update meeting #4
February 17, 2026	Delivery of report draft and report readout meeting
June 25, 2026	Delivery of final comprehensive report with fix review

Executive Summary

Engagement Overview

OSTIF engaged Trail of Bits to review the security of PyTorch ExecuTorch, an on-device AI inference framework that enables deployment of PyTorch models across mobile, embedded, and edge devices. ExecuTorch provides a lightweight C++ runtime that loads and executes serialized model files in the Portable Tensor Executable (PTE) format, with support for hardware acceleration through backend delegates including XNNPACK, Vulkan, CUDA, and Cadence.

A team of three consultants conducted the review from January 12 to February 11, 2026, for a total of nine engineer-weeks of effort. Our testing efforts focused on the security of PTE file parsing and model loading, memory safety of the C++ runtime and kernel implementations, and input validation of data deserialized from model files. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. Automated testing included compilation with additional compiler warnings, static analysis with CodeQL, Cppcheck, and Semgrep, and fuzz testing of PTE file parsing and execution paths.

Observations and Impact

The ExecuTorch codebase benefits from a modular architecture with clean separation between the core runtime, backend delegates, and extensions. This makes the codebase easier to review and maintain. The project has a large test suite and fuzz testing infrastructure that we extended during the engagement to discover several of the reported findings.

Our review identified 32 findings related to weak or insufficient data validation, reflecting a pattern of insufficient input validation when processing data from PTE model files. The most pervasive issue is the use of unchecked integer arithmetic in size calculations and bounds checks. Untrusted input values from PTE files can cause integer overflows that bypass bounds validation, leading to heap buffer overflows. This pattern appears in the core tensor size computation (TOB-EXECUTORCH-3), data loader bounds checking (TOB-EXECUTORCH-22), constant segment offset validation (TOB-EXECUTORCH-17), memory allocator size calculations (TOB-EXECUTORCH-26), and across multiple backend implementations including Vulkan (TOB-EXECUTORCH-27) and CUDA (TOB-EXECUTORCH-20). In several cases, these overflows enable an attacker who supplies a malicious PTE file to achieve out-of-bounds heap writes, potentially enabling remote code execution when the file is processed by a vulnerable version of the runtime.

Beyond integer overflows, the runtime lacks consistent validation of FlatBuffer-deserialized data before use. FlatBuffer input validation can be disabled at compile time through a build flag, and the XNNPACK backend omits validation completely (TOB-EXECUTORCH-12,

TOB-EXECUTORCH-33). A flipped inequality in the constant buffer bounds check allows undersized buffers to pass validation, enabling a heap buffer overflow (TOB-EXECUTORCH-13), and the `FreeCall` instruction handler indexes the values array without bounds checking, enabling a write-zero primitive on the heap (TOB-EXECUTORCH-15). On the Python tooling side, `ETRecord` parsing uses `pickle.loads` and `torch.load` without safe deserialization options, enabling arbitrary code execution when processing untrusted `ETRecord` files (TOB-EXECUTORCH-11).

Due to the large size of the ExecuTorch library and the fact that the project was only scoped to nine engineer weeks, we did not manage to achieve comprehensive coverage of all components. Instead, we focused on providing the greatest benefit possible to the project by identifying pervasive bug classes like integer overflows and memory corruption issues, and developing tooling that the team can use to strengthen the security posture of the project going forward. Based on the number of issues identified during this review, we are convinced that additional high-severity issues remain undiscovered in the codebase.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends the following steps:

- **Remediate the findings disclosed in this report.** The eight high-severity findings enable heap buffer overflows and arbitrary code execution through malicious model files. These findings should be addressed through direct fixes or broader refactoring efforts.
- **Do not use the ExecuTorch library to load models from untrusted sources.** The ExecuTorch library has not been hardened against untrusted inputs and contains a large number of integer overflow and truncation issues, which in turn can lead to memory corruption issues and potentially code execution. Until the validation logic has been significantly improved, we recommend developers to avoid using the library to load models from untrusted sources.
- **Centralize input validation.** Make sure that all inputs are validated immediately as they are parsed from the PTE file. Have all constructors validate their inputs, ensuring that objects with invalid state cannot be created. Document function contracts, clearly describing if each function expects its input to be valid, or if it performs its own validation.
- **Adopt overflow-safe arithmetic for all bounds checks and size calculations.** The most prevalent vulnerability pattern in the codebase is unchecked integer arithmetic in expressions like `offset + size <= limit`. Use centralized utility functions like the safe arithmetic functions in the `c10` library to detect and prevent integer overflows. This would help prevent memory corruption issues due to overflowing arithmetic in bounds checks.

- **Enforce FlatBuffer verification unconditionally before accessing deserialized data.** Several findings stem from the runtime or backend delegates accessing FlatBuffer fields without first running the verifier. Verification should be mandatory regardless of build configuration, and all backend delegate implementations should be audited for missing verification.
- **Enable and address compiler warnings for implicit integer conversions.** The codebase produces hundreds of warnings related to signed/unsigned conversions and 64-to-32-bit truncations (TOB-EXECUTORCH-29). Addressing these warnings will improve code auditability and reduce the risk of latent integer-related vulnerabilities.
- **Expand the fuzz testing suite to cover backend delegate parsing and execution.** Despite the previous fuzzing efforts, we managed to uncover many severe issues with a relatively simple fuzz test (see appendix E). Integrating continuous fuzzing into the CI pipeline with sanitizers enabled would catch memory safety issues across all code paths, including backend-specific parsing.
- **Consider using a memory-safe language for the next generation of the library.** Languages like Rust and Go inherently mitigate most of the vulnerabilities identified during this review. In these languages, out-of-bounds reads and writes trigger panics rather than causing silent memory corruption. Rust additionally offers compilation flags that generate panics for integer overflows. Triggering a panic before memory corruption occurs makes debugging this type of issue easier, and it also renders exploitation effectively impossible.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	8
Medium	14
Low	19
Informational	0
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	32
Denial of Service	4
Undefined Behavior	6

Project Goals

The engagement was scoped to provide a security assessment of the PyTorch ExecuTorch. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a malicious or malformed PTE file compromise the memory safety of the ExecuTorch runtime?
- Are integer arithmetic operations on tensor dimensions, buffer sizes, and file offsets protected against overflow when processing untrusted data from PTE files?
- Does the runtime consistently validate FlatBuffer-deserialized data, including null checks on optional fields and bounds checks on array indices?
- Can untrusted input to Python-based developer tooling, such as `ETRecord` files, lead to arbitrary code execution through unsafe deserialization?
- Are external data formats, such as WAV audio files and bundled program (BPTE) files, validated before use?
- Does the codebase exhibit patterns of implicit integer conversions or other compiler-warning-level issues that can reduce readability or introduce latent vulnerabilities?

Project Targets

The engagement involved reviewing and testing the following target.

ExecuTorch

Repository	https://github.com/pytorch/executorch
Version	7793b1d6601ae793a8a7246d24c7c56c93bd0c48
Type	C++, Python
Platform	Multiple

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Static analysis.** We compiled the ExecuTorch codebase with Clang, using additional warning flags to identify implicit integer conversions, implicit signed/unsigned conversions, and 64-to-32-bit truncations. We ran Cppcheck, CodeQL, and Semgrep, with both public and Trail of Bits private query suites and rule sets, to identify common vulnerabilities, undefined behavior, and fragile code patterns. We performed variant analysis using CodeQL and Semgrep to determine whether newly discovered vulnerabilities existed elsewhere in the codebase. During the engagement, we wrote five custom Semgrep rules targeting integer overflows in bounds-check macros, unsigned integer underflow in loop termination conditions, unchecked element size arithmetic, aggregate initialization of pointer values, and unchecked arithmetic flowing into memory allocation functions. These custom rules identified 26 additional variants of already identified issues across the codebase. (For more details, we refer to the Automated Testing section.)
- **Dynamic analysis.** We developed a fuzz testing harness for PTE file parsing and model execution using libFuzzer with `AddressSanitizer` (see appendix E). This harness exercised the `Program::load` and `Method::execute` code paths with malformed PTE inputs. This approach uncovered many of the reported findings, including null pointer dereferences, out-of-bound reads, type confusion during destruction, unbounded memory allocations, and memory corruption through FlatBuffer-backed tensor mutation.
- **Manual review.** Because of the size of the ExecuTorch codebase, we focused our manual review on identifying common vulnerable code patterns across the codebase, rather than obtaining full manual coverage. We prioritized insufficient input validation of data deserialized from PTE files, integer overflows in buffer size calculations and bounds checks, missing null checks on optional FlatBuffer fields, unsafe deserialization in Python tooling, and the security of backend delegate implementations. We traced data flows from PTE file parsing through the runtime to identify locations where untrusted input values influence memory operations without adequate validation.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The Python-based export and compilation pipeline was not a primary focus of the review. The audit concentrated on the C++ runtime and model loading paths.
- Backend delegate coverage was limited to XNNPACK, Vulkan, CUDA, and Cadence. We did not review in depth other backends, including CoreML, ARM Ethos-U, Qualcomm QNN, MediaTek, Samsung, and OpenVINO.
- The kernel operator implementations were reviewed for vulnerable code patterns, but were not exhaustively audited for correctness or memory safety across all operators.
- The review did not cover the Android and iOS platform integration layers.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Clang	An open-source LLVM front end for C and C++	Appendix C.1
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix C.2
Cppcheck	An open-source tool for static analysis of C and C++ code	Appendix C.3
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C.4
libFuzzer	An in-process, coverage-guided, evolutionary fuzzing engine included with Clang/LLVM.	Appendix E

Areas of Focus

Our automated testing and verification work focused on identifying the following types of issues:

- Code quality issues and potentially fragile code patterns
- Overflow and truncation issues due to implicit integer conversions
- General undefined behavior
- Memory safety issues

Test Results

The results of this focused testing are detailed below.

Clang

We compiled the project using Clang with additional warnings enabled. This revealed a large number of issues related to implicit integer conversions and truncations that should be reviewed by the ExecuTorch team (TOB-EXECUTORCH-29).

CodeQL

We ran CodeQL with public and Trail of Bits' private query suites against the ExecuTorch codebase to identify common vulnerabilities, fragile design patterns, implicit conversions, and integer overflows. This analysis identified one issue related to faulty pointer offset arithmetic in the ETDumpgen constructor (TOB-EXECUTORCH-32).

Cppcheck

We ran Cppcheck to identify common C and C++ vulnerabilities and undefined behavior in the ExecuTorch codebase. This analysis identified a number of potential memory-safety issues in third-party dependencies, but did not result in any security-relevant findings for the ExecuTorch library itself.

Semgrep

We ran Semgrep with public and Trail of Bits' private rule sets against the ExecuTorch codebase to identify low-complexity vulnerabilities and weaknesses. These runs did not identify any issues or code quality findings. We also wrote a number of custom Semgrep rules during the engagement to identify the following types of issues:

- Integer overflows inside `ET_CHECK_OR_RETURN_ERROR` bounds checks
- Integer overflows when multiplying a count with the output from a call to `elementSize`
- Infinite loops due to vacuously true loop conditions
- Incorrectly using aggregate initialization to initialize pointers
- Integer overflows in calls to `std::accumulate` and similar functions

Running these custom Semgrep rules against the codebase identified a number of issues related to integer overflows in bounds checks using `ET_CHECK_OR_RETURN_ERROR` (TOB-EXECUTORCH-17), size calculations using `elementSize` (TOB-EXECUTORCH-19), and size calculations using `std::accumulate` and similar functions (TOB-EXECUTORCH-27).

libFuzzer

For fuzzing details, see appendix E. Our fuzzing effort identified the following findings:

- Null pointer dereference in get_output_flattening_encoding (TOB-EXECUTORCH-1)
- Null pointer dereference in get_execution_plan (TOB-EXECUTORCH-2)
- Abort on type mismatch in parseListOptionalType (TOB-EXECUTORCH-6)
- Abort on type mismatch in Method::execute_instruction (TOB-EXECUTORCH-7)
- Abort on type mismatch in parse_tensor_list (TOB-EXECUTORCH-8)
- Out-of-bounds heap read without InternalConsistency verification (TOB-EXECUTORCH-12)
- Out-of-bounds heap write in Program::get_constant_buffer_data (TOB-EXECUTORCH-13)
- Out-of-bounds read in validateTensorLayout (TOB-EXECUTORCH-16)
- Infinite loop with JF_CALL (TOB-EXECUTORCH-23)
- Unbounded memory allocation from untrusted PTE file (TOB-EXECUTORCH-28)
- Memory corruption due to unsafe mutation of FlatBuffer-backed tensor data (TOB-EXECUTORCH-30)
- Type confusion in BoxedEvalueList after MoveCall (TOB-EXECUTORCH-31)
- Incorrect pointer offset arithmetic in ETDumpGen constructor (TOB-EXECUTORCH-32)
- Out-of-bounds read due to missing FlatBuffers verification in XNNCompiler (TOB-EXECUTORCH-33)
- Out-of-bounds read due to missing XNNHeader verification (TOB-EXECUTORCH-34)
- NULL pointer dereferences in XNNCompiler::compileModel (TOB-EXECUTORCH-35)
- Crash due to unchecked map access in XNNCompiler (TOB-EXECUTORCH-36)
- Unbounded memory allocation in XNNPACK subgraph creation (TOB-EXECUTORCH-37)
- NULL pointer dereference in BackendDelegate::Init due to missing compile_specs validation (TOB-EXECUTORCH-38)

- Heap buffer overflow in XNNPACK backend due to inconsistent tensor dimension validation (TOB-EXECUTORCH-39)
- Infinite loop in XNNPACK subgraph optimization (TOB-EXECUTORCH-40)
- Out-of-bounds vector access in XNNPACK getConstantDataPtr (TOB-EXECUTORCH-41)
- Unbounded memory allocation in XNNPACK backend due to missing dimension validation (TOB-EXECUTORCH-42)

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Null pointer dereference in get_output_flattening_encoding	Data Validation	Low
2	Null pointer dereference in get_execution_plan	Data Validation	Low
3	Integer overflow in tensor nbytes size computation	Undefined Behavior	High
4	Integer overflow in Buffer DataLoader bounds check	Data Validation	Low
5	Stack buffer overflow in prepare_input_tensors	Data Validation	Low
6	Abort on type mismatch in parseListOptionalType	Data Validation	Low
7	Abort on type mismatch in Method::execute_instruction	Data Validation	Low
8	Abort on type mismatch in parseTensorList	Data Validation	Low
9	Out-of-bounds heap read in BPTE tensor deserialization	Data Validation	Low
10	Out-of-bounds heap read in WAV audio file parsing	Data Validation	Low
11	Unsafe deserialization in ETRecord parsing enables code execution	Data Validation	High
12	Out-of-bounds heap read without InternalConsistency verification	Data Validation	Medium
13	Out-of-bounds heap write in Program::get_constant_buffer_data	Data Validation	High

14	Infinite loop in XNNExecutor::resize_outputs	Denial of Service	**Medium**
15	Out-of-bounds write in FreeCall instruction handler	Data Validation	**High**
16	Out-of-bounds read in validateTensorLayout	Data Validation	**Medium**
17	Integer overflow in constant segment offset validation	Data Validation	**Medium**
18	Missing duplicate check in dim_order validation	Data Validation	**Medium**
19	Integer overflow in tensor size validation enables heap buffer overflow	Data Validation	**Medium**
20	Integer overflow in CUDA tensor copy allocation enables heap buffer overflow	Data Validation	**Medium**
21	Hardcoded element size in CUDA tensor copy enables heap buffer overflow	Data Validation	**Medium**
22	Integer overflow in data loader bounds checking	Data Validation	**High**
23	Infinite loop with JF_CALL	Denial of Service	**Low**
24	Integer overflow in program file size validation	Data Validation	**Medium**
25	Out-of-bounds write in Cadence HiFi bmm operator	Undefined Behavior	**Low**
26	Integer overflow in PlatformMemoryAllocator allocation	Undefined Behavior	**High**
27	Integer overflows in Vulkan tensor operations	Data Validation	**High**
28	Unbounded memory allocation from untrusted PTE file	Data Validation	**Low**

29	Large number of security-relevant compiler warnings	Undefined Behavior	**Undetermined**
30	Memory corruption due to unsafe mutation of FlatBuffer-backed tensor data	Data Validation	**High**
31	Type confusion in BoxedEvalueList after MoveCall	Undefined Behavior	**Medium**
32	Incorrect pointer offset arithmetic in ETDumpGen constructor	Undefined Behavior	**Low**
33	Out-of-bounds read due to missing FlatBuffers verification in XNNCompiler	Data Validation	**Medium**
34	Out-of-bounds read due to missing XNNHeader verification	Data Validation	**Medium**
35	NULL pointer dereferences in XNNCompiler::compileModel	Data Validation	**Low**
36	Crash due to unchecked map access in XNNCompiler	Data Validation	**Low**
37	Unbounded memory allocation in XNNPACK subgraph creation	Denial of Service	**Low**
38	NULL pointer dereference in BackendDelegate::Init due to missing compile_specs validation	Data Validation	**Low**
39	Heap buffer overflow in XNNPACK backend due to inconsistent tensor dimension validation	Data Validation	**Medium**
40	Infinite loop in XNNPACK subgraph optimization	Denial of Service	**Low**
41	Out-of-bounds vector access in XNNPACK getConstantDataPtr	Data Validation	**Medium**
42	Unbounded memory allocation in XNNPACK backend due to missing dimension validation	Data Validation	**Low**

Detailed Findings

1. Null pointer dereference in get_output_flattening_encoding

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-1
Target: `runtime/executor/program.cpp`	

Description

There is a null pointer dereference vulnerability in the `Program::get_output_flattening_encoding` function that can cause the program to crash when processing malformed PTE files. The `container_meta_type` is dereferenced without a prior null check, and the field is not marked as required in the `program.fbs` flatbuffer definition (figure 1.1). We include a sample minimal program and a PTE file that triggers the bug (figures 1.2 and 1.3).

```
Result<const char*> Program::get_output_flattening_encoding(
    const char* method_name) const {
  auto plan = get_execution_plan(internal_program_, method_name);
  if (!plan.ok()) {
    return plan.error();
  }
  return plan.get()->container_meta_type()->encoded_out_str()->c_str();
}
```

Figure 1.1: `container_meta_type` is dereferenced but can be null. (`runtime/executor/program.cpp`)

```
00000000  20 00 00 00 45 54 31 32  99 00 10 00 18 04 00 00  | ...ET12........|
00000010  0c 00 00 00 00 00 00 00  00 06 00 00 00 00 00 00  |................|
00000020  ea ff ff ff 04 00 00 00  02 00 00 00 0c 00 00 00  |................|
00000030  00 00 00 00 06 00 08 00  04 00 00 00 04 00 00 00  |................|
00000040  00 00 00 00 00                                    |.....|
```

Figure 1.2: A malformed PTE file without the `container_meta_type` field

```
#include <executorch/extension/data_loader/file_data_loader.h>
#include <executorch/runtime/executor/program.h>
#include <executorch/runtime/platform/runtime.h>

int main() {
  executorch::runtime::runtime_init();
```

```
  auto loader = executorch::extension::FileDataLoader::from("crash.pte");
  if (!loader.ok()) return 1;

  auto program = executorch::runtime::Program::load(&loader.get());
  if (!program.ok()) return 1;

  program->get_output_flattening_encoding("");

  return 0;
}
```

Figure 1.3: A sample program that crashes when run with `crash.pte` file from figure 1.2

Exploit Scenario

A vulnerable device is using the `get_output_flattening_encoding` method. An attacker supplies a malformed model that triggers the null pointer dereference crash and causes a denial of service.

Recommendations

Short term, add a nullptr check in `get_output_flattening_encoding` to ensure that the `container_meta_type` field is present in the `ExecutionPlan` structure.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

2. Null pointer dereference in get_execution_plan

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-2
Target: `runtime/executor/program.cpp`	

Description

There is a null pointer dereference vulnerability in the `Program::get_execution_plan` function that can cause the program to crash when processing malformed PTE files. The `name` field is dereferenced without a prior null check, and the field is not marked as required in the `program.fbs` flatbuffer definition (figure 2.1). We include a sample minimal program and a PTE file that triggers the bug (figures 2.2 and 2.3). During our tests, this null pointer dereference also manifests as a heap out-of-bounds read. We recommend reproducing the crashes with AddressSanitizer.

```
Result<executorch_flatbuffer::ExecutionPlan*> get_execution_plan(
    const executorch_flatbuffer::Program* program,
    const char* method_name) {
  auto execution_plans = program->execution_plan();
  for (size_t i = 0; i < execution_plans->size(); i++) {
    auto plan = execution_plans->GetMutableObject(i);
    if (std::strcmp(plan->name()->c_str(), method_name) == 0) {
      return plan;
    }
  }
  ET_LOG(Error, "No method named '%s' in program", method_name);
  return Error::InvalidArgument;
}
```

Figure 2.1: The name field can be null (`runtime/executor/program.cpp`)

```
00000000  20 00 00 00 45 54 31 32  00 fe ff ff 00 00 00 04  | ...ET12........|
00000010  0e 00 18 00 54 00 04 00  08 00 0c 00 10 00 14 00  |....T...........|
00000020  10 00 00 00 40 00 00 00  28 00 00 00 18 00 00 00  |....@...(.......|
00000030  20 00 00 00 04 00 00 00  5c fe fb ff fb ff ff ff  | .......\.......|
00000040  ff ff ff ff 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 4c 00 00 00  28 00 00 00 04 00 00 00  |....L...(.......|
00000060  3a fd ff ff 04 00 00 00  74 00 00 00 28 00 00 00  |:.......t...(...|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 09 00 16 00  |................|
00000080  28 00 14 00 00 00 00 00  00 00 14 00 00 00 00 00  |(...............|
00000090  20 00 24 00 16 00 00 00  00 00 04 00 f0 02 00 00  | .$.............|
000000a0  34 01 00 00 28 00 00 00  28 00 00 00 80 00 00 00  |4...(...(.......|
000000b0  28 00 00 00 20 00 00 00  04 00 00 00 02 00 00 00  |(... ...........|
000000c0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 01 72  |...............r|
```

```
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Figure 2.2: A malformed PTE file without the name field in the `ExecutionPlan`

```
#include <executorch/extension/data_loader/file_data_loader.h>
#include <executorch/runtime/executor/program.h>
#include <executorch/runtime/platform/runtime.h>

int main() {
  executorch::runtime::runtime_init();

  auto loader = executorch::extension::FileDataLoader::from("crash.pte");
  if (!loader.ok()) return 1;

  auto program = executorch::runtime::Program::load(&loader.get());
  if (!program.ok()) return 1;

  for (size_t i = 0; i < program->num_methods(); i++) {
    auto method_name_result = program->get_method_name(i);
    if (method_name_result.ok()) {
      program->method_meta(method_name_result.get());
    }
  }

  return 0;
}
```

Figure 2.3: A sample program that crashes when run with `crash.pte` file from figure 2.2

Exploit Scenario

A vulnerable device uses the `method_meta` method. An attacker supplies a malformed model that triggers the null pointer dereference crash and causes a denial of service.

Recommendations

Short term, add a nullptr check in `get_execution_plan` to ensure that the `name` field is present in the `ExecutionPlan` structure.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

3. Integer overflow in tensor nbytes size computation

Severity: **High**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-EXECUTORCH-3
Target: `runtime/core/portable_type/tensor_impl.cpp`	

Description

The `torch::executor::TensorImpl::nbytes` function in ExecuTorch's tensor implementation performs multiplication without overflow checking. When a malicious PTE model file specifies tensor dimensions that cause an integer overflow in the byte size calculation, the computed `nbytes` value wraps to zero, bypassing memory validation checks and enabling out-of-bounds heap write access.

When ExecuTorch loads a PTE model file, the runtime constructs `TensorImpl` objects for each tensor defined in the model. The application allocates memory buffers based on sizes declared in the PTE's `non_const_buffer_sizes` field, which it reads via `MethodMeta::memory_planned_buffer_size`. During memory setup, the runtime calls `nbytes` to determine how much buffer space each tensor requires, then validates that the requested memory region fits within the allocated buffers via `HierarchicalAllocator::get_offset_address`. Only after this validation succeeds do kernel operations execute. The vulnerability arises because the `nbytes` calculation overflows before the validation check, causing the check to pass for a buffer far smaller than what the kernel operations write to.

The `torch::executor::TensorImpl::nbytes` method multiplies the element count by element size without overflow detection:

```
size_t TensorImpl::nbytes() const {
  return numel_ * elementSize(type_);
}
```

Figure 3.1: The return statement multiplies `numel_` by element size; when `numel_` is 2^{62} and element size is 4, the result overflows to zero. (`runtime/core/portable_type/tensor_impl.cpp#L68–L70`)

An attacker can craft tensor dimensions such as `[2097152, 2097152, 1048576]` (equivalently $[2^{21}, 2^{21}, 2^{20}]$) that produce a `numel` value of 2^{62}. This value fits within `ssize_t` (64-bit signed integer) and is computed correctly. However, when multiplied by the element size (4 bytes for float), the result 2^{64} overflows `size_t` and wraps to zero.

During tensor parsing, the runtime calls `TensorImpl::nbytes` and passes the result to `getTensorDataPtr`, which forwards it to `HierarchicalAllocator::get_offset_address` as the `size_bytes` parameter (runtime/executor/tensor_parser_portable.cpp#L167-L173). With the overflowed zero value, the validation check passes incorrectly:

```
ET_CHECK_OR_RETURN_ERROR(
    offset_bytes + size_bytes <= buffer.size(),
    ...);
```

Figure 3.2: The bounds check passes when `size_bytes` is zero from the overflow, allowing `offset_bytes + 0 <= buffer.size()` to succeed. (runtime/core/hierarchical_allocator.h#L75–L76)

With `size_bytes` equal to zero, the bounds check passes regardless of the actual buffer size, and a valid pointer is returned. However, subsequent kernel operations iterate based on `TensorImpl::numel`, which remains at 2^{62} elements, writing far beyond the allocated buffer into adjacent heap memory.

For example, when executing an `aten::full` operation, the `torch::executor::native::full_out` kernel writes a fill value to each element:

```
for (const auto i : c10::irange(out.numel())) {
  data_out[i] = val_casted;
}
```

Figure 3.3: The kernel loop iterates `TensorImpl::numel` times (2^{62}), writing to indices beyond the allocated buffer boundary (`kernels/portable/cpu/op_full.cpp#L46–L48`)

Exploit Scenario

We created a proof-of-concept PTE model file containing a tensor with dimensions that cause overflow. The malicious tensor is defined with dimensions that produce a valid `numel` but overflow `nbytes`:

```
# Overflow dimensions: 2^21 * 2^21 * 2^20 = 2^62
# numel = 2^62 (fits in ssize_t)
# nbytes = 2^62 * 4 = 2^64 (overflows size_t to 0)

# Tensor definition in PTE (value index 1):
"sizes": [2097152, 2097152, 1048576],
"allocation_info": {
    "memory_id": 1,  # uses buffer index 1
    "memory_offset_low": 16
},

# Buffer sizes by memory ID: [id0, id1]
# Memory ID 1 (used by the overflow tensor) gets only 64 bytes:
"non_const_buffer_sizes": [0, 64]
```

```
# KernelCall invokes aten::full.out with overflow tensor as output:
"instructions": [{
    "instr_args_type": "KernelCall",
    "instr_args": {
        "op_index": 0,  # -> aten::full.out
        "args": [5, 6, 1, 1]  # sizes, fill_value, out=tensor[1], return
    }
}]
```

Figure 3.4: The PTE declares a tensor with 2^{62} elements but allocates only 64 bytes, creating a mismatch exploited by the overflow.

When loaded into `executor_runner` via `Program::load`, the runtime creates a tensor with `numel` = 2^{62} and `nbytes = 0`, then returns a valid pointer into a 64-byte heap-allocated buffer. The model's execution graph contains a `KernelCall` instruction that invokes `aten::full.out` with the overflow tensor as the output argument. When the runtime dispatches this kernel, it iterates based on the reported element count and sprays `0x41414141` across the heap, overwriting adjacent objects beyond the first 12 valid indices. The process crashes with SIGSEGV when it accesses unmapped memory, confirming the heap overflow. An attacker distributing such a model could select kernel operations and fill values to corrupt function pointers, vtables, or allocator metadata in adjacent heap objects, potentially achieving arbitrary code execution when a victim loads the malicious PTE.

Recommendations

Short term, add overflow checking to the `numel_ * elementSize(type_)` multiplication in `torch::executor::TensorImpl::nbytes`. Since `nbytes` cannot currently return an error, consider performing the overflow check in `getTensorDataPtr` before passing the value to `get_offset_address`. This prevents the issue because overflow conditions are caught before producing incorrect size values that bypass validation.

Long term, implement validation at the PTE parsing layer to reject tensor specifications with dimensions that would cause overflow when computing byte sizes. Additionally, modify the `torch::executor::HierarchicalAllocator::get_offset_address` validation to reject `size_bytes = 0` for tensors with `numel > 0`, as this combination indicates a calculation error.

4. Integer overflow in Buffer DataLoader bounds check

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-4
Target: `extension/data_loader/buffer_data_loader.h`	

Description

The `BufferDataLoader::load` method uses an addition-based bounds check that is vulnerable to integer wraparound when processing attacker-controlled offset and size values from malicious `.pte` model files.

```
ET_CHECK_OR_RETURN_ERROR(
    offset + size <= size_,
    InvalidArgument,
    "offset %zu + size %zu > size_ %zu",
    offset,
    size,
    size_);
return executorch::runtime::FreeableBuffer(
    data_ + offset, size, /*free_fn=*/nullptr);
```

Figure 4.1: Vulnerable bounds check `offset + size <= size_` in (`extension/data_loader/buffer_data_loader.h#L37–L45`)

The `offset` and `size` values come from the PTE file's `DataSegment` table, which an attacker controls. By choosing values where `offset + size` overflows to a small number, the bounds check passes while `data_ + offset` produces an out-of-bounds pointer.

An attacker can create a malicious `.pte` file by setting a DataSegment entry's offset field to a large value, such as `0xfffffffffffe000`. The attacker then calculates a `size` value such that `offset + size` overflows to a value less than or equal to the legitimate file size. For example, if the file is 65536 bytes, setting `size` to `0x12000` (73728) causes the addition `0xfffffffffffe000 + 0x12000` to wrap around to `0x10000` (65536), passing the bounds check. The 64-bit addition wraps around, producing a small result that passes the bounds check, even though the resulting pointer (`data_ + offset`) points to memory before the allocated buffer.

In our observation, `BufferDataLoader` is used to load `.pte` model data from a pre-allocated memory buffer rather than from disk (which looks to use `FileDataLoader`). This appears common in embedded and firmware environments where model data is compiled directly into the binary image, as shown in the Raspberry Pi Pico2 example and

ARM Ethos-U runner. We assumed that in such deployments, the model data is typically baked into firmware or retrieved from trusted storage, limiting an attacker's ability to supply a malicious `.pte` file and leading to a higher difficulty rating.

Exploit Scenario

An attacker targeting an embedded device running ExecuTorch prepares a malicious `.pte` as described above. When the victim device loads this malicious model through the standard `Program::load` API, the integer overflow vulnerability is triggered during data segment loading. The bounds check `offset + size <= size_` evaluates to true because the overflow wraps the 64-bit addition result back to a valid-looking value, but the pointer arithmetic `data_ + offset` uses the full 64-bit offset value, producing a pointer 8192 bytes before the buffer starts. The device then reads data from this out-of-bounds location, potentially exposing it through model inference outputs or side channels.

Recommendations

Short term, replace the vulnerable bounds check with an overflow-safe comparison that checks each operand individually before performing addition. Use the pattern `if (offset > size_ || size > size_ - offset)` to detect out-of-bounds access without risking integer overflow. This prevents the vulnerability because it evaluates boundary conditions in an order that cannot overflow: first verifying that the offset alone does not exceed the buffer size, then checking that the size does not exceed the remaining space after offset.

Long term, implement comprehensive input validation for all fields in `.pte` files during deserialization, before any memory operations are performed. Add range checks that reject invalid values, such as offsets near `SIZE_MAX` or sizes exceeding reasonable model data limits.

5. Stack buffer overflow in prepare_input_tensors

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-5
Target: `extension/runner_util/inputs.cpp`	

Description

The `executorch::extension::prepare_input_tensors` function prepares input data for ExecuTorch model inference. For inputs declared as scalar types (`Int`, `Double`, or `Bool`) in the model metadata, the function copies data into fixed-size stack variables using `std::memcpy` without validating that the buffer size matches the expected type size. An attacker who supplies an oversized input buffer can overwrite stack memory.

For example, when used with `executor_runner`, the --inputs flag specifies input files. The runner reads each file and passes the `(buffer, file_size)` pairs to `prepare_input_tensors`.

As shown in figure 5.1 below, the code extracts the buffer pointer and size from the `input_buffers` parameter and performs an unchecked copy. Note that `bool_input` is a 1-byte stack variable, but `buffer_size` is attacker-controlled and can exceed this size, causing the `memcpy` to write past the variable's bounds.

```
auto [buffer, buffer_size] = input_buffers.at(i);

if (tag.get() == Tag::Int) {
  int64_t int_input;
  std::memcpy(&int_input, buffer, buffer_size);
  err = method.set_input(runtime::EValue(int_input), i);
} else if (tag.get() == Tag::Double) {
  double double_input;
  std::memcpy(&double_input, buffer, buffer_size);
  err = method.set_input(runtime::EValue(double_input), i);
} else if (tag.get() == Tag::Bool) {
  bool bool_input;
  std::memcpy(&bool_input, buffer, buffer_size);
  err = method.set_input(runtime::EValue(bool_input), i);
}
```

Figure 5.1: Non-scalar inputs use unchecked `memcpy` operations such as `std::memcpy(&int_input, buffer, buffer_size);`, which copy user-controlled buffer sizes into fixed-size stack variables (`extension/runner_util/inputs.cpp#L83–L99`)

The attack surface appears limited to applications using `executor_runner` with the `--inputs` flag (or custom applications using `prepare_input_tensors`) when loading models that declare scalar inputs (`Int`, `Double`, or `Bool`). Models that take scalar configuration parameters (like sequence length or temperature) as direct inputs rather than hardcoded values are likely affected.

Exploit Scenario

An attacker identifies (or uploads) a `.pte` model that expects a scalar `Int` input and prepares a malicious input file that is 64 bytes instead of the expected 8 bytes for an `Int` input:

```
# Create oversized input file (64 bytes instead of 8)
python3 -c "print('A'*64, end='')" > oversized_int.bin

# Trigger the overflow
executor_runner --model_path model.pte --inputs oversized_int.bin
```

The runner reads the input file size and passes it as `buffer_size` to `prepare_input_tensors`. The `memcpy` operation copies all 64 bytes into an 8-byte stack variable, overwriting 56 bytes of adjacent stack memory.

Recommendations

Short term, add size validation before each scalar `memcpy` operation, mirroring the existing validation pattern used for tensor inputs. For `Tag::Int` and `Tag::Double`, have the code verify that `buffer_size == sizeof(int64_t)` and `buffer_size == sizeof(double)`, respectively. For `Tag::Bool`, verify that `buffer_size == sizeof(bool)`. Have the code return an `InvalidArgument` error if the sizes do not match. This prevents the overflow by rejecting malformed input buffers before any copy occurs.

Long term, consider using a safer abstraction for deserializing scalar values from buffers that encapsulates the size check. A templated helper function that reads exactly `sizeof(T)` bytes and fails if insufficient data is available would eliminate the possibility of introducing similar vulnerabilities in future code paths handling external data.

6. Abort on type mismatch in parseListOptionalType

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-6
Target: `runtime/executor/tensor_parser.h`	

Description

A type validation vulnerability in the `parseListOptionalType` method causes the program to abort when parsing malformed PTE files containing lists with incorrectly typed elements. The `EValue::toOptional` method requires the `EValue` to have a tag that is None or matches T, in this case, a Tensor. It is possible to include `EValues` with different tags that will force the program to crash (figures 6.1, 6.2, 6.3, 6.7).

```
new (&optional_tensor_list[output_idx])
    std::optional<T>(values[index].toOptional<T>());
evalp_list[output_idx] = &values[static_cast<size_t>(index)];
```

Figure 6.1: The `toOptional` method requires `EValue` to have a tag that matches T. (`runtime/executor/tensor_parser.h#L100–L102`)

```
template <typename T>
inline std::optional<T> toOptional() const {
  if (this->isNone()) {
    return executorch::aten::nullopt;
  }
  return this->to<T>();
}
```

Figure 6.2: `toOptional` calls `toTensor` (figure 6.3) when templated with a tensor. (`runtime/core/evalue.h`)

```
bool isTensor() const {
  return tag == Tag::Tensor;
}

executorch::aten::Tensor toTensor() && {
  ET_CHECK_MSG(isTensor(), "EValue is not a Tensor.");
  auto res = std::move(payload.as_tensor);
  clearToNone();
  return res;
}
```

Figure 6.3: ExecuTorch aborts when EValue is not a Tensor. (`runtime/core/evalue.h`)

```
==10260== ERROR: libFuzzer: deadly signal
    ...
    #7 0x0001026bcd68 in executorch::runtime::pal_abort() platform.cpp:126
    #8 0x0001026bc44c in executorch::runtime::runtime_abort() abort.cpp:20
    #9 0x0001026781a0 in executorch::runtime::EValue::toTensor() const &+0x54
(pte_parser_fuzzer:arm64+0x1000201a0)
    #10 0x0001026926a8 in
executorch::runtime::internal::evalue_to_const_ref_overload_return<executorch::runti
me::etensor::Tensor>::type
executorch::runtime::EValue::to<executorch::runtime::etensor::Tensor>() const &+0x24
(pte_parser_fuzzer:arm64+0x10003a6a8)
    #11 0x000102691e4c in std::__1::optional<executorch::runtime::etensor::Tensor>
executorch::runtime::EValue::toOptional<executorch::runtime::etensor::Tensor>()
const+0x170 (pte_parser_fuzzer:arm64+0x100039e4c)
    #12 0x00010266cfb0 in
executorch::runtime::Result<executorch::runtime::BoxedEvalueList<std::__1::optional<
executorch::runtime::etensor::Tensor>>>
executorch::runtime::deserialization::parseListOptionalType<executorch::runtime::ete
nsor::Tensor>(flatbuffers::Vector<int, unsigned int> const*,
executorch::runtime::EValue*, unsigned long,
executorch::runtime::MemoryManager*)+0x44c (pte_parser_fuzzer:arm64+0x100014fb0)
    #13 0x000102669cec in
executorch::runtime::Method::parse_values(executorch::runtime::NamedDataMap
const*)+0x2268 (pte_parser_fuzzer:arm64+0x100011cec)
    #14 0x0001026705e0 in
executorch::runtime::Method::init(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::NamedDataMap const*)+0x3e0
(pte_parser_fuzzer:arm64+0x1000185e0)
    #15 0x00010266fd04 in
executorch::runtime::Method::load(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::Program const*, executorch::runtime::MemoryManager*,
executorch::runtime::EventTracer*, executorch::runtime::NamedDataMap const*)+0x2dc
(pte_parser_fuzzer:arm64+0x100017d04)
    #16 0x00010269edbc in executorch::runtime::Program::load_method(char const*,
executorch::runtime::MemoryManager*, executorch::runtime::EventTracer*,
executorch::runtime::NamedDataMap const*) const program.cpp:321
    #17 0x00010265ad14 in LLVMFuzzerTestOneInput pte_parser_fuzzer.cpp:144
```

Figure 6.4: A stack trace from running a crashing fuzzer-generated example

Exploit Scenario

An attacker supplies a malformed model that triggers the type mismatch abort and causes a denial of service.

Recommendations

Short term, improve the EValue API to have an option to gracefully handle type mismatches, which are inevitable if the input is untrusted. For instance, EValue could implement a `Result<T> EValue::tryTo` method that would first check if the value is of the expected type.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

7. Abort on type mismatch in Method::execute_instruction

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-7
Target: `runtime/executor/method.cpp`	

Description

A type validation vulnerability in the `Method::execute_instruction` method causes the program to abort during the parsing of malformed PTE files that contain lists with incorrectly typed elements. The `EValue::toTensor` method requires the `EValue` to have a Tensor tag. It is possible to include `EValues` with different tags that will cause the program to crash (figures 7.1 and 7.2).

```
case executorch_flatbuffer::InstructionArguments::FreeCall: {
  EXECUTORCH_SCOPE_PROF("FREE_CALL");
  internal::EventTracerProfileOpScope event_tracer_op_scope =
      internal::EventTracerProfileOpScope(event_tracer_, "FREE_CALL");
  // We know that instr_args_as_FreeCall is non-null because it was checked
  // at init time.
  auto free_call = instruction->instr_args_as_FreeCall();
  auto t = values_[free_call->value_index()].toTensor();
  internal::reset_data_ptr(t);
} break;
```

Figure 7.1: The `values_` array is untrusted and can be populated with non-Tensors. (`runtime/executor/method.cpp#L1471–L1480`)

```
==13653== ERROR: libFuzzer: deadly signal
    ...
    #7 0x0001005a8d74 in executorch::runtime::pal_abort() platform.cpp:126
    #8 0x0001005a8458 in executorch::runtime::runtime_abort() abort.cpp:20
    #9 0x00010055b3ec in executorch::runtime::EValue::toTensor() &+0x54
(pte_parser_fuzzer:arm64+0x1000173ec)
    #10 0x000100568554 in executorch::runtime::Method::execute_instruction()+0x161c
(pte_parser_fuzzer:arm64+0x100024554)
    #11 0x00010056b0a4 in executorch::runtime::Method::execute()+0x86c
(pte_parser_fuzzer:arm64+0x1000270a4)
    #12 0x0001005472e4 in LLVMFuzzerTestOneInput pte_parser_fuzzer.cpp:155
```

Figure 7.2: A stack trace from running a crashing fuzzer-generated example

Exploit Scenario

An attacker supplies a malformed model that triggers the type mismatch abort and causes a denial of service.

Recommendations

Short term, improve the EValue API to have an option to gracefully handle type mismatches, which are inevitable if the input is untrusted. For instance, `EValue` could implement a `Result<T> EValue::tryTo` method that would first check if the value is of the expected type.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

8. Abort on type mismatch in parseTensorList

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-8
Target: `runtime/executor/tensor_parser_exec_aten.cpp`	

Description

A type validation vulnerability in the `parseTensorList` method causes the program to abort when parsing malformed PTE files containing lists with incorrectly typed elements. The `EValue::toTensor` method requires the `EValue` to have a Tensor tag. It is possible to include `EValues` with different tags that will cause the program to crash (figures 8.1 and 8.2).

```
new (&tensor_list[output_idx]) executorch::aten::Tensor(
    values[static_cast<size_t>(tensor_index)].toTensor());
```

Figure 7.1: The `values` array is untrusted and can be populated with non-Tensors. (`runtime/executor/tensor_parser_exec_aten.cpp`)

```
==19150== ERROR: libFuzzer: deadly signal
    ...
    #6 0x000100fe9f38 in et_pal_abort posix.cpp:98
    #7 0x000100fe8d68 in executorch::runtime::pal_abort() platform.cpp:126
    #8 0x000100fe844c in executorch::runtime::runtime_abort() abort.cpp:20
    #9 0x000100f9b2c4 in executorch::runtime::EValue::toTensor() &+0x54
(pte_parser_fuzzer:arm64+0x1000172c4)
    #10 0x000100fdef24 in
executorch::runtime::deserialization::parseTensorList(flatbuffers::Vector<int,
unsigned int> const*, executorch::runtime::EValue*, unsigned long,
executorch::runtime::MemoryManager*) tensor_parser_exec_aten.cpp:112
    #11 0x000100f958ac in
executorch::runtime::Method::parse_values(executorch::runtime::NamedDataMap
const*)+0x1eb0 (pte_parser_fuzzer:arm64+0x1000118ac)
    #12 0x000100f9c62c in
executorch::runtime::Method::init(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::NamedDataMap const*)+0x3e0
(pte_parser_fuzzer:arm64+0x10001862c)
    #13 0x000100f9bd50 in
executorch::runtime::Method::load(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::Program const*, executorch::runtime::MemoryManager*,
executorch::runtime::EventTracer*, executorch::runtime::NamedDataMap const*)+0x2dc
(pte_parser_fuzzer:arm64+0x100017d50)
    #14 0x000100fcae08 in executorch::runtime::Program::load_method(char const*,
executorch::runtime::MemoryManager*, executorch::runtime::EventTracer*,
```

```
executorch::runtime::NamedDataMap const*) const program.cpp:321
    #15 0x000100f86c8c in LLVMFuzzerTestOneInput pte_parser_fuzzer.cpp:144
```

Figure 7.2: The `values` array is untrusted and can be populated with non-Tensors. (`runtime/executor/tensor_parser_exec_aten.cpp`)

Exploit Scenario

An attacker supplies a malformed model that triggers a type-mismatch abort and causes a denial of service.

Recommendations

Short term, improve the EValue API to have an option to gracefully handle type mismatches, which are inevitable if the input is untrusted. For instance, `EValue` could implement a `Result<T> EValue::tryTo` method that would first check if the value is of the expected type.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

9. Out-of-bounds heap read in BPTE tensor deserialization

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-9
Target: `devtools/bundled_program/bundled_program.cpp`	

Description

The `executorch::bundled_program::tensor_like` function performs a `memcpy` operation using a size derived from the tensor's `sizes[]` field without validating that the source data buffer contains sufficient bytes. This allows a maliciously crafted `.bpte` file to trigger an out-of-bounds heap read, exposing adjacent heap memory contents. An application that loads untrusted `.bpte` files through these interfaces is vulnerable.

The function proceeds in three steps. First, it extracts tensor dimensions from the FlatBuffer's `sizes[]` field:

```
ET_CHECK(bundled_tensor->sizes()->size() <= kMaxDim);
int64_t ret_t_sizes[kMaxDim];
for (size_t i = 0; i < bundled_tensor->sizes()->size(); i++) {
  ret_t_sizes[i] = static_cast<int64_t>(bundled_tensor->sizes()->data()[i]);
}
```

Figure 9.1: The function reads dimensions from `sizes[]` into a local array. (/devtools/bundled_program/bundled_program.cpp#L47–L52)

Next, the function allocates a destination tensor based on those dimensions:

```
at::Tensor ret_tensor = at::zeros(
    {ret_t_sizes, bundled_tensor->sizes()->size()},
    at::dtype(static_cast<ScalarType>(bundled_tensor->scalar_type())));
```

Figure 9.2: The function then allocates a tensor with the claimed dimensions. (/devtools/bundled_program/bundled_program.cpp#L54–L56)

Finally, data from FlatBuffer's `data[]` field is copied into the new tensor:

```
memcpy(
    ret_tensor.mutable_data_ptr(),
    static_cast<const void*>(bundled_tensor->data()->Data()),
    ret_tensor.nbytes());
```

Figure 9.3: The vulnerable `memcpy` call. The copy length `ret_tensor.nbytes()` is derived from `sizes[]`, but `data[]` may contain far fewer bytes. (/devtools/bundled_program/bundled_program.cpp#L57–L60)

Exploit Scenario

An attacker crafts a malicious `.bpte` file containing a tensor whose `sizes` field claims 10,000 INT32 elements (40,000 bytes) while the `data` field provides only four bytes. When a victim application loads this file using `BundledModule` or `executor_runner`, the `tensor_like` function allocates a 40,000-byte destination buffer and copies 40,000 bytes from the four-byte source, reading 39,996 bytes of adjacent heap memory. The attacker retrieves the leaked memory contents from the output tensor.

Recommendations

Short term, validate that the data buffer size matches or exceeds the expected tensor size before copying. Add a check that compares `bundled_tensor->data()->size()` against `ret_tensor.nbytes()` and return an error if the data buffer is too small.

Long term, develop a static analysis query (e.g., using CodeQL or Semgrep) to detect patterns where a size or length value read from serialized data controls a memory operation without validation against the source buffer's actual size.

10. Out-of-bounds heap read in WAV audio file parsing

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-10
Target: `extension/llm/runner/wav_loader.h`	

Description

The `load_wav_audio_data` function in `extension/llm/runner/wav_loader.h` parses WAV file headers and uses the `Subchunk2Size` field from the file header to determine how many audio samples to copy into memory. The function does not validate that the claimed data size falls within the bounds of the actual file buffer, allowing a malicious WAV file to trigger an out-of-bounds heap read. The vulnerability affects all code paths that process user-supplied WAV files through the ExecuTorch audio loading API, including the Whisper, Parakeet, and Voxtral example applications.

The function reads `data_offset` and `data_size` directly from the WAV file header at lines 171 and 172. These values pass to the calculation of `num_samples`, which determines the range passed to `std::vector::assign`. The `assign` method copies the specified number of elements from the source buffer without any bounds checking. When `Subchunk2Size` exceeds the actual file size, the copy operation reads beyond the allocated heap buffer.

```
const char* data = buffer.data();
  size_t data_offset = header->dataOffset;
  size_t data_size = header->Subchunk2Size;
  ...
  if (bits_per_sample == 32) {
    size_t num_samples = data_size / 4;

    if (audio_format == kWavFormatIeeeFloat) {
      const float* input_buffer =
          reinterpret_cast<const float*>(data + data_offset);
      audio_data.assign(input_buffer, input_buffer + num_samples);
    }
  }
```

Figure 10.1: The attacker-controlled `Subchunk2Size` value determines num_samples, which sets the copy length for assign without validating that the data falls within the buffer bounds. (`extension/llm/runner/wav_loader.h#L170-L194`)

The `Subchunk2Size` value originates in `load_wav_header`, which reads it directly from the file without validation:

```
if (data_offset != 0) {
  header->Subchunk2Size =
      *reinterpret_cast<const int32_t*>(data + data_offset + 4);
  header->dataOffset = static_cast<uint32_t>(data_offset + 8);
}
```

Figure 10.2: The `Subchunk2Size` value is read directly from the file buffer via reinterpret_cast without validation, then propagates to `load_wav_audio_data`, where it controls the copy length. (`extension/llm/runner/wav_loader.h#L129–L133`)

We created a proof of concept (PoC) that demonstrates the vulnerability by creating a 48-byte WAV file that claims to contain 1 MB of audio data. When the `load_wav_audio_data` processes the file, it attempts to read 262,144 float samples from a buffer containing only four bytes of audio data, triggering a heap buffer overflow that AddressSanitizer detects, as shown below:

```
==1507852==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x504000000040
READ of size 1048576 at 0x504000000040 thread T0
    #0 ... in __interceptor_memmove
    #12 ... in std::vector<float>::assign<float const*, void>
    #13 ... in executorch::extension::llm::load_wav_audio_data
    #14 ... in main
0x504000000040 is located 0 bytes to the right of 48-byte region
```

Figure 10.3: AddressSanitizer output from PoC test

Exploit Scenario

An attacker crafts a malicious WAV file containing a standard 44-byte header with the `Subchunk2Size` field set to 1 MB, followed by only four bytes of actual audio data. The attacker distributes this file to a victim running a speech recognition application built on ExecuTorch. When the victim processes the file, the `load_wav_audio_data` function reads the inflated size from the header and attempts to copy 1 MB of data from a 48-byte heap allocation. This causes the function to read adjacent heap memory, potentially exposing model weights, previous audio samples, cryptographic keys, or other sensitive data allocated nearby. The attacker receives this leaked data encoded as the audio samples returned by the function.

Recommendations

Short term, add bounds validation after reading header values to ensure that the claimed data size does not exceed the buffer size. The validation should verify that `data_offset + data_size <= buffer.size()` before performing any copy operations. This prevents the out-of-bounds read by rejecting malformed files before the vulnerable code path executes.

Long term, develop a static analysis query (e.g., using CodeQL or Semgrep) to detect patterns where a size or length value read from serialized data controls a memory operation without validation against the source buffer's actual size.

11. Unsafe deserialization in ETRecord parsing enables code execution

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-11
Target: `devtools/etrecord/_etrecord.py`	

Description

The `parse_etrecord` function deserializes data from ETRecord files through multiple unsafe code paths, which can be exploited to achieve arbitrary code execution. As discussed in the documentation, ETRecords are used as debug artifacts during model development and profiling, containing serialized model graphs, weights, and reference input/output pairs to enable reproducible debugging and validation of exported models. An attacker who can supply a malicious ETRecord file can execute arbitrary code on the system that parses it. Note that the difficulty is set to high because ETRecord files are debug artifacts, likely used only in development and profiling workflows.

The first attack vector uses `pickle.loads` directly on the `reference_outputs` and `representative_inputs` entries:

```
elif entry == ETRecordReservedFileNames.REFERENCE_OUTPUTS:
    # @lint-ignore PYTHONPICKLEISBAD
    reference_outputs = pickle.loads(
        etrecord_zip.read(ETRecordReservedFileNames.REFERENCE_OUTPUTS)
    )
elif entry == ETRecordReservedFileNames.REPRESENTATIVE_INPUTS:
    # @lint-ignore PYTHONPICKLEISBAD
    representative_inputs = pickle.loads(
        etrecord_zip.read(ETRecordReservedFileNames.REPRESENTATIVE_INPUTS)
    )
```

Figure 11.1: Direct pickle deserialization of `reference_outputs` and `representative_inputs` entries extracted from the ETRecord ZIP file. (devtools/etrecord/_etrecord.py#L727–L736)

The second attack vector uses `torch.load` without `weights_only=True` when deserializing exported program artifacts. The `parse_etrecord` function calls `deserialize` on `SerializedArtifact` objects containing `state_dict`, `constants`, and `example_inputs` data read from the ZIP file:

```
serialized_artifact = SerializedArtifact(
    etrecord_zip.read(ETRecordReservedFileNames.EDGE_DIALECT_EXPORTED_PROGRAM),
```

```
    etrecord_zip.read(f"{entry}_state_dict"),
    etrecord_zip.read(f"{entry}_constants"),
    etrecord_zip.read(f"{entry}_example_inputs"),
)
edge_dialect_program = deserialize(serialized_artifact)
```

Figure 11.2: Construction of `SerializedArtifact` from ZIP entries, which is then passed to `deserialize`. (devtools/etrecord/_etrecord.py#L710–L718)

This data flows through `deserialize_torch_artifact`, which uses `torch.load` without safe deserialization:

```
def deserialize_torch_artifact(
    serialized: Union[Dict[str, Any], Tuple[Any, ...], bytes]
):
    if isinstance(serialized, (dict, tuple)):
        return serialized
    if len(serialized) == 0:
        return {}
    buffer = io.BytesIO(serialized)
    buffer.seek(0)
    artifact = torch.load(buffer)
    assert isinstance(artifact, (tuple, dict))
    return artifact
```

Figure 11.3: The `deserialize_torch_artifact` function calls `torch.load` without `weights_only=True`, enabling arbitrary code execution via `torch.load`.(exir/serde/export_serialize.py#L327–L338)

In total, a malicious `ETRecord` ZIP file can contain payloads in five different locations: `reference_outputs`, `representative_inputs`, `*_state_dict`, `*_constants`, and `*_example_inputs`. The `@lint-ignore PYTHONPICKLEISBAD` comments indicate awareness of the risk in the direct pickle usage.

Exploit Scenario

We created a proof of concept (PoC) that, when parsed by `parse_etrecord`, executes arbitrary shell commands on the host system. The PoC creates a minimal ETRecord ZIP file containing the required `ETRECORD_V0` identifier and a malicious pickle payload in the `reference_outputs` entry. The payload uses Python's `__reduce__` method to invoke `os.system` during deserialization. When a user calls `parse_etrecord` on this file, the pickle payload executes with the privileges of the user running the Python process. An attacker could distribute such a file through a model-sharing platform, include it in a compromised model repository, or send it directly to a victim, resulting in data exfiltration, installation of backdoors, or lateral movement within the victim's environment.

Recommendations

Short term, integrate a pickle scanner such as Fickling to analyze serialized data before deserialization. Fickling can statically analyze pickle bytecode to detect potentially malicious

payloads, including those using `__reduce__`, `__setstate__`, or other code execution primitives. Scan all pickle data extracted from ETRecord ZIP files before passing it to `pickle.loads` or `torch.load`, and reject any payloads that fail the safety analysis. This provides defense in depth without requiring changes to the serialization format.

Long term, consider adding `weights_only=True` to all `torch.load` calls in `deserialize_torch_artifact`, and audit all deserialization paths in the `ETRecord` and model serialization code to ensure consistent use of safe deserialization options. Establish code review guidelines that flag any use of `pickle.loads` or `torch.load` without `weights_only=True` on untrusted input as a security issue requiring explicit justification and compensating controls.

12. Out-of-bounds heap read without InternalConsistency verification

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-12
Target: `runtime/executor/program.cpp`	

Description

ExecuTorch programs can be loaded either with a `Minimal` or `InternalConsistency` verification. Without the `InternalConsistency` verification, the offset to the FlatBuffer root table is not checked and will be used to read data out of bounds. Even if requested, `InternalConsistency` may not be available due to the compilation option (`ET_ENABLE_PROGRAM_VERIFICATION`) that reduces binary size (figure 12.1). Checking that the offset is within the bounds of the FlatBuffer program is mandatory for memory safety and cannot be skipped (figures 12.2 and 12.3).

```
 // Do extra verification if requested.
  if (verification == Verification::InternalConsistency) {
#if ET_ENABLE_PROGRAM_VERIFICATION
    EXECUTORCH_SCOPE_PROF("Program::verify_internal_consistency");
    flatbuffers::Verifier verifier(
        reinterpret_cast<const uint8_t*>(program_data->data()),
        program_data->size());
    bool ok = executorch_flatbuffer::VerifyProgramBuffer(verifier);
    ET_CHECK_OR_RETURN_ERROR(
        ok,
        InvalidProgram,
        "Verification failed; data may be truncated or corrupt");
#else
    ET_LOG(
        Info, "InternalConsistency verification requested but not available");
#endif
  }
```

Figure 12.1: The offset verification is performed only when `InternalConsistency` is requested and ExecuTorch is compiled with `ET_ENABLE_PROGRAM_VERIFICATION`. (`runtime/executor/program.cpp#L140-L156`)

```
$ ./build/executor_runner
--model-path=./fuzz/crash-13ceff2a001e1a0613203f5b943906c54db1709d
AddressSanitizer:DEADLYSIGNAL
=================================================================
==1149==ERROR: AddressSanitizer: BUS on unknown address (pc 0x000101bc0908 bp
0x00016fa88ac0 sp 0x00016fa88aa0 T0)
```

```
==1149==The signal is caused by a READ memory access.
==1149==Hint: this fault was caused by a dereference of a high value address (see
register values below).  Disassemble the provided pc to learn which register was
used.
    #0 0x000101bc0908 in int flatbuffers::ReadScalar<int>(void const*)+0x6c
(executor_runner:arm64+0x101850908)
    #1 0x000101bc07ec in flatbuffers::Table::GetVTable() const+0x18
(executor_runner:arm64+0x1018507ec)
    #2 0x000101bc06e4 in flatbuffers::Table::GetOptionalFieldOffset(unsigned short)
const+0x18 (executor_runner:arm64+0x1018506e4)
    #3 0x000101bd4988 in
flatbuffers::Vector<flatbuffers::Offset<executorch_flatbuffer::NamedData>, unsigned
int> const*
flatbuffers::Table::GetPointer<flatbuffers::Vector<flatbuffers::Offset<executorch_fl
atbuffer::NamedData>, unsigned int> const*, unsigned int>(unsigned short)+0x20
(executor_runner:arm64+0x101864988)
    #4 0x000101bd4954 in
flatbuffers::Vector<flatbuffers::Offset<executorch_flatbuffer::NamedData>, unsigned
int> const*
flatbuffers::Table::GetPointer<flatbuffers::Vector<flatbuffers::Offset<executorch_fl
atbuffer::NamedData>, unsigned int> const*, unsigned int>(unsigned short) const+0x1c
(executor_runner:arm64+0x101864954)
    #5 0x000101bd0c20 in executorch_flatbuffer::Program::named_data() const+0x18
(executor_runner:arm64+0x101860c20)
    #6 0x000101bcfcc8 in
executorch::runtime::Program::load(executorch::runtime::DataLoader*,
executorch::runtime::Program::Verification)+0xc64
(executor_runner:arm64+0x10185fcc8)
    #7 0x0001003ea574 in main+0x1760 (executor_runner:arm64+0x10007a574)
    #8 0x000186a85d50  (<unknown module>)
```

Figure 12.2: The `executor_runner` program (compiled with AddressSanitizer) uses only Minimal verification and crashes on a malicious PTE file from figure 12.3.

```
$ hexdump -C ./fuzz/crash-13ceff2a001e1a0613203f5b943906c54db1709d
00000000  0a df df df 45 54 31 32  df df df df df df df df  |....ET12........|
00000010  df df df df df df df df  df df df df df df df df  |................|
*
00000030  df df df 30 df df df df  df df df df df df df df  |...0............|
00000040
```

Figure 12.3: A malicious PTE file with an offset (highlighted in yellow) significantly bigger than its size causes the program to read data out of bounds.

Exploit Scenario

A vulnerable device loads a malicious PTE file that reads the FlatBuffer root table from out-of-bounds heap data. If the attacker can spray the heap before loading, they could execute a different model or leak sensitive information from the memory, such as PII or pointers.

Recommendations

Short term, unconditionally verify if the offset is within the FlatBuffer bounds. If the original FlatBuffer implementation is unacceptable performance-wise, reimplement the check in ExecuTorch.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

13. Out-of-bounds heap write in Program::get_constant_buffer_data

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-13
Target: `runtime/executor/program.cpp`	

Description

The `Program::get_constant_buffer_data` method has an incorrect bounds check that allows it to return a buffer that is smaller than the requested number of bytes (figure 13.1). The method has a branch that can pick a buffer from `Program.constant_buffer`, with a controllable size. The undersized buffer is then used as a destination for a copy of an input tensor that also has a controllable, larger size, which results in an out-of-bounds write (figure 13.2). Before copying, the function checks if the destination and source are of the same size; however, the `nbytes` method does not reflect the actual buffer size returned from `Program::get_constant_buffer_data`, so the check passes.

```
Result<const void*> Program::get_constant_buffer_data(
    size_t buffer_index,
    size_t nbytes) const {
  auto internal_program =
      static_cast<const executorch_flatbuffer::Program*>(internal_program_);

  // Constant data is either in a separate segment (constant_segment_data) and
  // loaded during Program::load, or stored inside the flatbuffer data
  // (constant_buffer).
  if (constant_segment_data_.data() != nullptr) {
    ...
  } else {
    // Otherwise, the constant data is stored inside Program.constant_buffer.
    const auto* constant_buffer_ptr = internal_program->constant_buffer();
    size_t num_elems =
        constant_buffer_ptr == nullptr ? 0 : constant_buffer_ptr->size();
    ET_CHECK_OR_RETURN_ERROR(
        buffer_index < num_elems,
        InvalidArgument,
        "Constant buffer index %zu invalid for program constant buffer range %zu",
        buffer_index,
        num_elems);

    const auto& constant_buffer = *constant_buffer_ptr;
    const auto* storage = constant_buffer[buffer_index]->storage();
    auto storage_size = storage == nullptr ? 0 : storage->size();
    ET_CHECK_OR_RETURN_ERROR(
        storage_size <= nbytes,
```

```
        InvalidArgument,
        "Constant buffer size %zu larger than allocated nbytes %zu",
        static_cast<size_t>(constant_buffer[buffer_index]->storage()->size()),
        nbytes);

    return storage->data();
  }
}
```

Figure 13.1: A bounds checking bug inside the `Program::get_constant_buffer_data` method; the inequality should be flipped. (`runtime/executor/program.cpp#L404–L409`)

```
Error copy_tensor_data(
    const torch::executor::Tensor& t_dst,
    const torch::executor::Tensor& t_src) {
  ...
  if (t_src.const_data_ptr() != nullptr) {
    ET_CHECK_OR_RETURN_ERROR(
        t_dst.nbytes() == t_src.nbytes(),
        InvalidArgument,
        "t_dst.nbytes() %zu != t_src.nbytes(). %zu",
        t_dst.nbytes(),
        t_src.nbytes());
    std::memcpy(
        t_dst.mutable_data_ptr(), t_src.const_data_ptr(), t_src.nbytes());
  }
  return Error::Ok;
}
```

Figure 13.2: The copy writes out of bounds because the `nbytes` method does not reflect the real buffer size. (`runtime/core/exec_aten/util/tensor_util_portable.cpp#L165–L185`)

```
$ ./build/executor_runner
--model-path=fuzz/targets/pte_parser/artifacts/crash-106a69d63adb396871fb263fc91b324
3827060a1
E 00:00:00.003304 executorch:program.cpp:266] !!DEPRECATED!! This branch is
deprecated from ExecuTorch 0.7; re-export this PTE file to ensure support on newer
runtimes.
I 00:00:00.003328 executorch:executor_runner.cpp:353] Model file
fuzz/targets/pte_parser/artifacts/crash-106a69d63adb396871fb263fc91b3243827060a1 is
loaded.
I 00:00:00.003340 executorch:executor_runner.cpp:362] Using method
I 00:00:00.003344 executorch:executor_runner.cpp:413] Setting up planned buffer 0,
size 32.
I 00:00:00.003421 executorch:executor_runner.cpp:441] Method loaded.
=================================================================
==30318==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61c0000007e8 at
pc 0x000108e301c8 bp 0x00016b65bd20 sp 0x00016b65b4d0
WRITE of size 16777216 at 0x61c0000007e8 thread T0
    #0 0x000108e301c4 in __asan_memcpy+0x520
(libclang_rt.asan_osx_dynamic.dylib:arm64+0x581c4)
    #1 0x00010617bb74 in
```

```
executorch::runtime::internal::copy_tensor_data(executorch::runtime::etensor::Tensor
const&, executorch::runtime::etensor::Tensor const&)+0x304
(executor_runner:arm64+0x1019dbb74)
    #2 0x00010618e684 in
executorch::runtime::Method::set_input(executorch::runtime::EValue const&, unsigned
long)+0xaf8 (executor_runner:arm64+0x1019ee684)
    #3 0x000104859348 in
executorch::extension::internal::fill_and_set_input(executorch::runtime::Method&,
executorch::runtime::TensorInfo&, unsigned long, void*, bool)+0x334
(executor_runner:arm64+0x1000b9348)
    #4 0x000104857d88 in
executorch::extension::prepare_input_tensors(executorch::runtime::Method&,
executorch::extension::PrepareInputTensorsOptions,
std::__1::vector<std::__1::pair<char*, unsigned long>,
std::__1::allocator<std::__1::pair<char*, unsigned long>>> const&)+0x18b8
(executor_runner:arm64+0x1000b7d88)
    #5 0x000104823574 in main+0x2b80 (executor_runner:arm64+0x100083574)
    #6 0x0001857ddd50  (<unknown module>)

0x61c0000007e8 is located 0 bytes after 1896-byte region
[0x61c000000080,0x61c0000007e8)
allocated by thread T0 here:
    #0 0x000108e34494 in __sanitizer_mz_memalign+0x78
(libclang_rt.asan_osx_dynamic.dylib:arm64+0x5c494)
    #1 0x0001859d2650 in _malloc_zone_memalign+0x13c
(libsystem_malloc.dylib:arm64+0x37650)
    #2 0x0001859b9928 in _malloc_type_aligned_alloc_outlined+0x68
(libsystem_malloc.dylib:arm64+0x1e928)
    #3 0x000185b5db68 in operator new(unsigned long, std::align_val_t)+0x4c
(libc++abi.dylib:arm64+0x16b68)
    #4 0x00010484987c in executorch::extension::(anonymous
namespace)::et_aligned_alloc(unsigned long, std::align_val_t)+0x1c
(executor_runner:arm64+0x1000a987c)
    #5 0x000104849460 in executorch::extension::FileDataLoader::load(unsigned long,
unsigned long, executorch::runtime::DataLoader::SegmentInfo const&) const+0x474
(executor_runner:arm64+0x1000a9460)
    #6 0x0001061a7f00 in
executorch::runtime::Program::load(executorch::runtime::DataLoader*,
executorch::runtime::Program::Verification)+0xbe0
(executor_runner:arm64+0x101a07f00)
    #7 0x000104822904 in main+0x1f10 (executor_runner:arm64+0x100082904)
    #8 0x0001857ddd50  (<unknown module>)
```

Figure 13.3: A malicious PTE file can cause the `copy_tensor_data` function to read out of bounds.

Exploit Scenario

A vulnerable device loads a malicious PTE file that exploits the out-of-bounds write and takes over the execution of the program.

Recommendations

Short term, fix the bounds check in `Program::get_constant_buffer_data` by flipping the inequality.

Long term, expand the fuzz testing suite and ensure that the fuzzer exercises all callable low-level API functions.

14. Infinite loop in XNNExecutor::resize_outputs

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-EXECUTORCH-14
Target: `backends/xnnpack/runtime/XNNExecutor.cpp`	

Description

The `XNNExecutor::resize_outputs` function contains an infinite loop caused by an unsigned integer underflow in the `int32_t`-to-`int64_t` conversion loop. When an XNNPACK output tensor has `ScalarType::Long` type, the function iterates backwards through the tensor elements to convert `int32_t` values to `int64_t`. However, the loop variable `j` is of type `size_t` (unsigned), and the termination condition `j >= 0` is always true for unsigned types, causing the loop to run forever.

```
if (out_tensor->scalar_type() == ScalarType::Long) {
  int64_t* data_64 = out_tensor->mutable_data_ptr<int64_t>();
  const int32_t* data_32 = out_tensor->const_data_ptr<int32_t>();
  for (size_t j = out_tensor->numel() - 1; j >= 0; --j) {
    data_64[j] = data_32[j];
  }
}
```

Figure 14.1: Infinite loop due to unsigned integer underflow (`backends/xnnpack/runtime/XNNExecutor.cpp`)

This vulnerability affects any model that uses XNNPACK delegation with output tensors of type `ScalarType::Long`, which occurs in operations like `argmax` pooling where XNNPACK internally uses int32 for indices but ExecuTorch expects int64.

We wrote a Semgrep rule to detect infinite loops due to integer underflows. This did not identify any additional issues in the codebase.

Exploit Scenario

An attacker crafts a model that delegates an argmax pooling operation to the XNNPACK backend. The output tensor for this operation has `ScalarType::Long` type. When the model executes and `resize_outputs` processes this tensor, the `int32_t`-to-`int64_t` conversion loop begins iterating backwards. After processing all valid elements, the unsigned loop counter underflows from 0 to `SIZE_MAX`. The loop then attempts to access memory at index `SIZE_MAX`, which either crashes the application immediately due to segmentation fault or, if memory access continues without crashing, causes the application

to hang indefinitely in the loop, consuming CPU resources and rendering the inference service unavailable.

Recommendations

Short term, use a signed integer type for the loop variable to avoid underflow and allow the loop to terminate correctly.

Long term, enable compiler warnings for comparisons that are always true or always false due to type limitations, such as `-Wtype-limits` in GCC/Clang. Integrate static analysis tools like Semgrep or CodeQL into the CI pipeline to detect unsigned comparison issues.

15. Out-of-bounds write in FreeCall instruction handler

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-15
Target: `runtime/executor/method.cpp`	

Description

The `FreeCall` instruction handler in `Method::execute_instruction` directly accesses the `values_` array without bounds validation. This enables an attacker to trigger out-of-bounds memory access via a malicious `.pte` file. This can lead to a "write-zero-where" primitive that allows an attacker to overwrite an arbitrary memory location, potentially bypassing security checks or corrupting data structures on the heap.

```
case executorch_flatbuffer::InstructionArguments::FreeCall: {
  EXECUTORCH_SCOPE_PROF("FREE_CALL");
  internal::EventTracerProfileOpScope event_tracer_op_scope =
      internal::EventTracerProfileOpScope(event_tracer_, "FREE_CALL");
  // We know that instr_args_as_FreeCall is non-null because it was checked
  // at init time.
  auto free_call = instruction->instr_args_as_FreeCall();
  auto t = values_[free_call->value_index()].toTensor();
  internal::reset_data_ptr(t);
} break;
```

Figure 15.2: The `FreeCall` handler directly accesses `values_` array without a bounds check. (`runtime/executor/method.cpp#L1471–L1480`)

During `Method::init`, the `FreeCall` instruction type falls through to the default case, which performs no validation of the `value_index` field.

```
case executorch_flatbuffer::InstructionArguments::JumpFalseCall: {
  // Validate the index at load time so we can trust it during
  // execution.
  auto index =
      static_cast<const executorch_flatbuffer::JumpFalseCall*>(
          instr_args)
          ->cond_value_index();
  ET_CHECK_OR_RETURN_ERROR(
      index >= 0 && static_cast<size_t>(index) < n_value_,
      InvalidProgram,
      "Index %zd negative or >= %" ET_PRIsize_t,
      static_cast<ssize_t>(index),
      n_value_);
  chain_instruction_arg_lists[instr_idx] = InstructionArgs();
```

```
} break;
default: {
  chain_instruction_arg_lists[instr_idx] = InstructionArgs();
} break;
```

Figure 15.1: `JumpFalseCall` validates its index in `Method::init`, but `FreeCall` falls through to the default case with no validation. (`runtime/executor/method.cpp#L1022–L1039`)

At execution time, the `FreeCall` handler directly indexes into the `values_` array without any bounds check. This contrasts with `MoveCall`, which uses the bounds-checked `get_value` and `mutable_value` functions:

```
const EValue& Method::get_value(size_t i) const {
  ET_CHECK_MSG(
      i < n_value_, "%" ET_PRIsize_t " >= %" ET_PRIsize_t, i, n_value_);
  return values_[i];
}

EValue& Method::mutable_value(size_t i) {
  ET_CHECK_MSG(
      i < n_value_, "%" ET_PRIsize_t " >= %" ET_PRIsize_t, i, n_value_);
  return values_[i];
}
```

Figure 15.3: The bounds-checked accessor functions are not used by the `FreeCall` handler. (`runtime/executor/method.cpp#L1648–L1658`)

When an out-of-bounds index is used, the code reads arbitrary memory and interprets it as an `EValue`. If the memory at the read location has a `tag` field value of 1 (corresponding to `Tag::Tensor`), the `toTensor` call succeeds and returns a corrupted `Tensor` object. The subsequent call to `internal::reset_data_ptr` invokes `tensor.unsafeGetTensorImpl()->set_data(nullptr)` on this corrupted tensor.

Since `TensorImpl` has no virtual functions, `unsafeGetTensorImpl` simply returns the `impl_` pointer member from the corrupted `Tensor`. The `set_data(nullptr)` call then writes zero to the `data_` field at a fixed offset from wherever `impl_` points. If an attacker can influence the write location by controlling heap layout, she is able zero out arbitrary memory locations on the heap.

Exploit Scenario

An attacker crafts a malicious `.pte` file containing a `FreeCall` instruction with a `value_index` that exceeds the bounds of the `values_` array. When an application loads and executes this model, the `FreeCall` handler reads memory beyond the `values_` array. If the attacker can influence heap layout such that the out-of-bounds read returns memory with a tag value of 1, the code interprets the adjacent bytes as a `Tensor` structure.

The `reset_data_ptr` function then calls `set_data(nullptr)` on the corrupted `TensorImpl` pointer, writing zero to an attacker-influenced memory location. While the attacker cannot control the value written, zeroing specific memory locations can bypass security checks that test for non-zero values, or trigger secondary vulnerabilities by corrupting data structures on the heap.

Recommendations

Short term, refactor the `FreeCall` handler to use the existing bounds-checked `get_value` or `mutable_value` accessor functions instead of directly indexing `values_`.

Long term, rewrite all instruction handlers to use bounds-checked accessors at execution time. Consider adding a static analysis check or unit test that verifies all array accesses through `values_` use bounds-checked methods.

16. Out-of-bounds read in validateTensorLayout

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-16
Target: `runtime/executor/tensor_parser_exec_aten.cpp`	

Description

The `validateTensorLayout` function reads from the `dim_order` array without first verifying that its size matches the `sizes` array. The function uses the length of the `sizes` array to bound its loop, then accesses both arrays using the same index. The implementation uses `Get` to read elements from the FlatBuffer vectors, which does not perform bounds checking in release builds. Thus, if a malicious `.pte` file contains a tensor where the `dim_order` array is shorter than the `sizes` array, the function will read memory beyond the `dim_order` buffer.

```
ET_NODISCARD Error validateTensorLayout(
    const executorch_flatbuffer::Tensor* s_tensor,
    const TensorLayout& expected_layout) {
  ET_CHECK_OR_RETURN_ERROR(
      static_cast<ScalarType>(s_tensor->scalar_type()) ==
          expected_layout.scalar_type(),
      InvalidExternalData,
      "Scalar type mismatch. Expected %hhd, got %hhd.",
      static_cast<int8_t>(s_tensor->scalar_type()),
      static_cast<int8_t>(expected_layout.scalar_type()));
  int dim = s_tensor->sizes()->size();
  ET_CHECK_OR_RETURN_ERROR(
      dim >= 0, InvalidExternalData, "Dim is negative: %d", dim)
  ET_CHECK_OR_RETURN_ERROR(
      static_cast<size_t>(dim) == expected_layout.sizes().size(),
      InvalidExternalData,
      "Dim mismatch. Expected %d, got %zu.",
      dim,
      expected_layout.sizes().size());
  for (int i = 0; i < dim; i++) {
    ET_CHECK_OR_RETURN_ERROR(
        s_tensor->sizes()->Get(i) == expected_layout.sizes()[i],
        InvalidExternalData,
        "Sizes mismatch. Expected %d, got %d for size at index %d.",
        s_tensor->sizes()->Get(i),
        expected_layout.sizes()[i],
        i);
    ET_CHECK_OR_RETURN_ERROR(
        s_tensor->dim_order()->Get(i) == expected_layout.dim_order()[i],
```

```
        InvalidExternalData,
        "Dim order mismatch. Expected %d, got %d for dim at index %d.",
        s_tensor->dim_order()->Get(i),
        expected_layout.dim_order()[i],
        i);
  }
  return Error::Ok;
}
```

Figure 16.1: The `validateTensorLayout` function uses the length of the `sizes` array to iterate over both the `sizes` and `dim_order` arrays (`runtime/executor/tensor_parser_exec_aten.cpp#L113–L149`)

Tensors created using the `parseTensor` API are validated to ensure that the `sizes` and `dim_order` arrays have matching lengths. However, external constant tensors created by the `Method::parse_external_constants` method bypass this check, and calls `validateTensorLayout` directly on the raw deserialized tensor without first validating that the `dim_order` array length matches the `sizes` array length.

```
const auto s_tensor = static_cast<const executorch_flatbuffer::Tensor*>(
    serialization_value->val());
// ...

Error err =
    deserialization::validateTensorLayout(s_tensor, tensor_layout.get());
```

Figure 16.2: `Method::parse_external_constants` calls `validateTensorLayout` without any array size validation (`runtime/executor/method.cpp#L352–L385`)

Exploit Scenario

An attacker crafts a malicious `.pte` file containing an external constant tensor (with `extra_tensor_info.location` set to `EXTERNAL` and no `allocation_info`). The tensor's `sizes` array has four elements but the `dim_order` array has only two elements.

When an application loads this `.pte` file with an external data map (`.ptd` file), the runtime calls `Method::parse_external_constants` to resolve external tensor data. This function directly calls `validateTensorLayout` on the raw deserialized tensor without first checking that `dim_order->size()` matches `sizes->size()`. The function then iterates from 0 to 3 (based on the `sizes` length). When `i` reaches 2, the call to `s_tensor->dim_order()->Get(2)` reads memory beyond the allocated `dim_order` buffer.

Recommendations

Short term, add a check at the beginning of the loop to verify that `s_tensor->dim_order()->size()` equals `dim` before accessing the `dim_order` array.

Long term, centralize tensor validation logic into a single function that comprehensively validates that all tensor metadata fields (sizes, dim_order, strides) have consistent lengths before any processing. Add fuzz tests that generate malformed tensor definitions with mismatched array lengths to catch similar issues in the future.

17. Integer overflow in constant segment offset validation

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-17
Target: `runtime/executor/program.cpp`	

Description

The `Program::get_constant_buffer_data` function loads constant tensor data from a pre-loaded segment using an offset read directly from the PTE file's FlatBuffer data. The implementation contains a bounds check to ensure that `offset + nbytes <= size`, but this check does not account for integer overflow.

```
Result<const void*> Program::get_constant_buffer_data(
    size_t buffer_index,
    size_t nbytes) const {
  if (constant_segment_data_.data() != nullptr) {
    // ...
    uint64_t offset = static_cast<uint64_t>(
        (*internal_program->constant_segment()->offsets())[buffer_index]);

    size_t size = constant_segment_data_.size();
    ET_CHECK_OR_RETURN_ERROR(
        offset + nbytes <= size,
        InvalidArgument,
        "Constant segment offset %" PRIu64
        " + size_bytes %zu invalid for program constant segment size %zu",
        offset,
        nbytes,
        size);

    // Offset is wrt the beginning of the constant segment.
    return static_cast<const void*>(
        static_cast<const unsigned char*>(constant_segment_data_.data()) +
        offset);
  }
  // ...
}
```

Figure 17.1: Vulnerable bounds check in `get_constant_buffer_data()` (`runtime/executor/program.cpp#L344–L413`)

The `offset` value is read from `constant_segment()->offsets()`, which is deserialized from the PTE file without prior validation. The FlatBuffer schema defines these offsets as `uint64` values.

```
table SubsegmentOffsets {
  // Index of the segment in Program.segments
  segment_index: uint;

  // Each element is an offset in bytes into the data of the segment pointed to
  // by segment_index. Offsets must be aligned to @executorch-tensor-alignment.
  offsets: [uint64];
}
```

Figure 17.2: Schema definition for subsegment offsets (schema/program.fbs#L425–L432)

The `nbytes` parameter is computed from the tensor's `sizes` and `scalar_type` fields, which also originate from the PTE file. This gives a potential attacker control over both operands of the vulnerable addition.

We wrote a Semgrep rule to identify other uses of the `ET_CHECK_OR_RETURN_ERROR` macro where the condition does not sufficiently protect against integer overflows. This identified 12 more instances across the codebase.

- `n_inputs + n_outputs == args.size()` in backends/apple/metal/runtime/metal_backend.cpp
- `n_inputs + n_outputs == args.size()` in backends/cuda/runtime/cuda_backend.cpp
- `(start_pos + num_prompt_tokens) <= (metadata_.context_len - metadata_.ar_len)` in examples/qualcomm/oss_scripts/llama/runner/prompt_processor.cpp
- `cur_pos_ + num_prompt_tokens < seq_len` in examples/qualcomm/oss_scripts/llama/runner/runner.cpp
- `offset + size <= size_` in extension/data_loader/buffer_data_loader.h
- `offset + size <= file_size_` in extension/data_loader/file_data_loader.cpp
- `offset + size <= file_size_` in extension/data_loader/file_descriptor_data_loader.cpp
- `offset + size <= file_size_` in extension/data_loader/file_descriptor_data_loader.cpp
- `offset + size <= file_size_` in extension/data_loader/mmap_data_loader.cpp

- `offset + size <= size_` in `extension/data_loader/shared_ptr_data_loader.h`
- `(segments->Get(segment_index)->offset() + segments->Get(segment_index)->size()) <= segment_end_offset` in `extension/flat_tensor/flat_tensor_data_map.cpp`
- `N + M <= size()` in `runtime/core/array_ref.h`
- `offset_bytes + size_bytes <= buffer.size()` in `runtime/core/hierarchical_allocator.h`
- `offset + nbytes <= size` in `runtime/executor/program.cpp`

Exploit Scenario

An attacker crafts a malicious PTE file containing a constant tensor with `data_buffer_idx` pointing to a large entry in `constant_segment.offsets`. When a victim application loads this PTE file, the runtime parses the tensor and calls `get_constant_buffer_data`. The addition `offset + nbytes` overflows to a small value, which passes the bounds check against the actual constant segment size. The function returns a pointer calculated as `constant_segment_data_.data() + offset`, causing a second overflow. When the tensor parser subsequently reads data through this pointer, it causes a segmentation fault resulting in denial of service, or reads data from other memory regions if the resulting address happens to be mapped.

Recommendations

Short term, replace the arithmetic bounds check with an overflow-safe comparison. Change `offset + nbytes <= size` to `offset <= size && nbytes <= size - offset`.

Long term, implement centralized bounds-checking utility functions that use compiler built-ins like `__builtin_add_overflow` to detect overflow, and mandate its use for all bounds validation involving PTE file data.

18. Missing duplicate check in dim_order validation

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-18
Target: `runtime/core/exec_aten/util/dim_order_util.h`	

Description

The `validate_dim_order` function fails to verify that the `dim_order` array contains a valid permutation of dimension indices. A valid `dim_order` must contain each index from 0 to `dims` - 1 exactly once, but the validation only checks that each value is less than the number of dimensions.

```
template <typename DimOrderType>
bool validate_dim_order(const DimOrderType* dim_order, const size_t dims) {
  for (const auto i : c10::irange(dims)) {
    if (dim_order[i] >= static_cast<DimOrderType>(dims)) {
      return false;
    }
  }
  return true;
}
```

Figure 18.1: Incomplete `dim_order` validation (runtime/core/exec_aten/util/dim_order_util.h#L26–L33)

If a PTE file contains a tensor with duplicate `dim_order` values (e.g., `[0, 0, 0]` for a 3-dimensional tensor), the validation passes because all values are less than 3. The subsequent `dim_order_to_stride_nocheck` function uses these values to index into the strides array.

```
template <typename SizesType, typename DimOrderType, typename StridesType>
inline void dim_order_to_stride_nocheck(
    const SizesType* sizes,
    const DimOrderType* dim_order,
    const size_t dims,
    StridesType* strides) {
  if (dims == 0) {
    return;
  }
  strides[dim_order[dims - 1]] = 1;
  for (int32_t i = dims - 2; i >= 0; --i) {
    if (sizes[dim_order[i + 1]] == 0) {
      strides[dim_order[i]] = strides[dim_order[i + 1]];
```

```
    } else {
      strides[dim_order[i]] =
          strides[dim_order[i + 1]] * sizes[dim_order[i + 1]];
    }
  }
}
```

Figure 18.2: Stride calculation using `dim_order` as array index. If all `dim_order` values are 0, only the first stride element is initialized. (`runtime/core/exec_aten/util/dim_order_util.h#L114–L138`)

With duplicate `dim_order` values, some stride indices are written multiple times while others are never written, leaving some stride values uninitialized. When these uninitialized stride values are later used in memory offset calculations, they can lead to out-of-bounds accesses, memory corruption, or a crash.

Exploit Scenario

A corrupted PTE file contains a 3-dimensional tensor with `dim_order` equal to `[0, 0, 0]`. When the PTE file is loaded, the runtime allocates an uninitialized strides array with three elements containing arbitrary heap values. The `dim_order_to_stride_nocheck` function writes only to `strides[0]`, leaving `strides[1]` and `strides[2]` containing uninitialized values. The tensor is then used in kernel operations that compute memory offsets using expressions like `offset + index * tensor.strides()[1]`. The garbage stride values cause the kernel to read from or write to arbitrary memory locations, resulting in memory corruption.

Recommendations

Short term, modify `validate_dim_order` to verify that the `dim_order` array is a valid permutation by checking for duplicate values. Use a bitmask to track which indices have been seen and reject the input if any index appears more than once.

Long term, consider initializing all allocated arrays to zero (or a known sentinel value) to make uninitialized memory bugs easier to detect.

19. Integer overflow in tensor size validation enables heap buffer overflow

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-19
Target: `extension/tensor/tensor_ptr.cpp`, `runtime/core/portable_type/tensor_impl.cpp`	

Description

The `make_tensor_ptr` function validates that a provided data buffer has sufficient size for the requested tensor dimensions before creating the `TensorImpl`. However, this validation relies on arithmetic operations that can overflow, allowing the check to pass even when the buffer is too small. When the tensor is subsequently accessed, this results in out-of-bounds memory access.

The vulnerable size check in `make_tensor_ptr` computes the expected buffer size by multiplying the number of elements by the element size:

```
ET_CHECK_MSG(
    data.size() ==
        executorch::aten::compute_numel(sizes.data(), sizes.size()) *
            executorch::aten::elementSize(type),
    "Data size does not match tensor size.");
```

Figure 19.1: Vulnerable size check in `make_tensor_ptr` (`extension/tensor/tensor_ptr.cpp#L150–L154`)

The `compute_numel` function also computes the total element count by multiplying all dimension sizes without overflow protection:

```
ssize_t compute_numel(const TensorImpl::SizesType* sizes, ssize_t dim) {
  ET_CHECK_MSG(
      dim == 0 || sizes != nullptr,
      "Sizes must be provided for non-scalar tensors");
  ssize_t numel = 1; // Zero-dimensional tensors (scalars) have numel == 1.
  for (const auto i : c10::irange(dim)) {
    ET_CHECK_MSG(
        sizes[i] >= 0,
        "Size must be non-negative, got %zd at dimension %zd",
        static_cast<ssize_t>(sizes[i]),
        i);
    numel *= sizes[i];
  }
  return numel;
```

```
}
```

Figure 19.2: `compute_numel` multiplies dimensions without overflow checking (`runtime/core/portable_type/tensor_impl.cpp#L30–L44`)

Since `SizesType` is `int32_t` and `ssize_t` varies by platform (32-bit on 32-bit systems, 64-bit on 64-bit systems), the multiplication can overflow which constitutes undefined behavior. The subsequent multiplication by `elementSize(type)` provides another overflow opportunity.

The same type of issue is present in the `clone_tensor_ptr` function, where the size of the new data buffer is computed without overflow checks.

```
const auto tensor_numel = static_cast<size_t>(tensor.numel());
std::vector<uint8_t> data(tensor_numel * aten::elementSize(type));
```

Figure 19.3: `clone_tensor_ptr` multiplies the number of elements with the element size without overflow checks (`extension/tensor/tensor_ptr.cpp#L207–L208`)

We wrote a Semgrep rule to identify potential integer overflows in element size arithmetic. This identified four additional potential overflow issues in the codebase.

- `data.size() * aten::elementSize(type)` in `extension/tensor/tensor_ptr.h`
- `executorch::aten::compute_numel(sizes.data(), sizes.size()) * executorch::aten::elementSize(type))` in `extension/tensor/tensor_ptr_maker.cpp`
- `numel_ * elementSize(type_)` in `runtime/core/portable_type/tensor_impl.cpp`
- `n * executorch::runtime::elementSize(scalar_type)` in `runtime/core/tensor_layout.cpp`

Exploit Scenario

An attacker provides malicious tensor dimensions to `make_tensor_ptr` when constructing a tensor from user-controlled input. The attacker selects dimension sizes such that their product overflows to a small value matching the provided buffer size. When subsequent kernel operations access the tensor data based on its declared dimensions, they read or write beyond the buffer boundaries, potentially leading to memory corruption or crashes.

Recommendations

Short term, add overflow checking to `compute_numel` and all size validation expressions. Use safe multiplication functions like `__builtin_mul_overflow` that detect overflow before it occurs, or explicit range checks that verify the multiplication result does not

exceed the maximum representable value. This prevents the check from passing when overflow would occur.

Long term, establish a consistent pattern for all arithmetic involving tensor dimensions and sizes throughout the codebase. Consider using a dedicated safe-arithmetic library or wrapper type that enforces overflow checking at compile time.

20. Integer overflow in CUDA tensor copy allocation enables heap buffer overflow

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-20
Target: `backends/cuda/runtime/shims/memory.cpp`	

Description

The `aoti_torch_copy_` function allocates temporary staging buffers when copying tensors between devices with mismatched memory strides. However, this allocation computes its size using arithmetic that can overflow, causing `malloc` to allocate a buffer far smaller than required. When the subsequent `cudaMemcpy` operation writes the full tensor data into this undersized buffer, it results in a heap buffer overflow.

The vulnerable allocation pattern appears at lines 542 and 556 in the fallback copy path:

```
size_t total_bytes = src->nbytes();
int64_t total_elements = self->numel();
// ...
src_host_data = static_cast<float*>(malloc(total_elements * sizeof(float)));
```

Figure 20.1: Allocation size computed without overflow protection (backends/cuda/runtime/shims/memory.cpp#L542)

The `total_elements` variable is an `int64_t` obtained from `self->numel()`. When multiplied by `sizeof(float)` (4 bytes), the result can overflow if `total_elements` exceeds `SIZE_MAX / 4`. When overflow occurs, `malloc` receives a value much smaller than intended, potentially allocating zero bytes.

Following the allocation, the code executes `cudaMemcpy` using `total_bytes` as the copy size:

```
ET_CUDA_CHECK_OR_RETURN_ERROR(cudaMemcpy(
    src_host_data, src->data_ptr(), total_bytes, cudaMemcpyDeviceToHost));
```

Figure 20.2: `cudaMemcpy` writes `total_bytes` into the undersized buffer (backends/cuda/runtime/shims/memory.cpp#L548)

When the allocation size overflows to a small value but `total_bytes` remains large, `cudaMemcpy` writes far beyond the allocated buffer. The same vulnerable pattern exists at

line 556 for the destination buffer allocation, with its corresponding `cudaMemcpy` at line 596.

Exploit Scenario

An attacker provides a malicious ExecuTorch model file containing tensors with dimensions designed to produce a `total_elements` value exceeding 2^{62}. When a victim loads this model and the CUDA backend performs a tensor copy operation with mismatched strides, the multiplication at line 542 overflows to zero or a small value. The `malloc` call allocates an undersized buffer, and the subsequent `cudaMemcpy` writes the full tensor data past the allocation boundary, corrupting adjacent heap metadata and enabling further exploitation.

Recommendations

Short term, add overflow checking before the multiplication by validating that `total_elements` does not exceed `SIZE_MAX / sizeof(float)`, and return an error if it does. Alternatively, use `total_bytes` directly for the allocation size since it is already correctly calculated from the tensor's actual byte size. This prevents the integer overflow by ensuring the multiplication cannot wrap around.

Long term, establish a consistent pattern for all arithmetic involving tensor dimensions and allocation sizes throughout the CUDA backend. Consider using safe multiplication functions like `__builtin_mul_overflow` that detect overflow before it occurs. This prevents similar vulnerabilities from being introduced in other code paths.

References

- CWE-190: Integer Overflow or Wraparound
- CWE-122: Heap-based Buffer Overflow

21. Hardcoded element size in CUDA tensor copy enables heap buffer overflow

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-21
Target: `backends/cuda/runtime/shims/memory.cpp`	

Description

The `aoti_torch_copy_` function allocates temporary staging buffers when copying tensors between devices with mismatched memory strides. However, this allocation hardcodes `sizeof(float)` (4 bytes) regardless of the tensor's actual data type. When the subsequent `cudaMemcpy` operation copies based on the tensor's true element size, tensors with larger data types cause a heap buffer overflow.

The vulnerable allocation pattern appears at lines 542 and 556:

```
size_t total_bytes = src->nbytes();
int64_t total_elements = self->numel();
// ...
src_host_data = static_cast<float*>(malloc(total_elements * sizeof(float)));
```

Figure 21.1: Allocation hardcodes `sizeof(float)` regardless of actual `dtype` (backends/cuda/runtime/shims/memory.cpp#L542)

The code correctly retrieves the element size at line 527 but never uses it for the allocation:

```
size_t element_size = dtype_to_element_size(self_dtype);  // Correctly computed but unused
```

Figure 21.2: Element size is computed but ignored in subsequent allocations (backends/cuda/runtime/shims/memory.cpp#L527)

Following the allocation, `cudaMemcpy` uses `total_bytes` which reflects the tensor's actual data type:

```
ET_CUDA_CHECK_OR_RETURN_ERROR(cudaMemcpy(
    src_host_data, src->data_ptr(), total_bytes, cudaMemcpyDeviceToHost));
```

Figure 21.3: `cudaMemcpy` writes `total_bytes` based on actual `dtype` (backends/cuda/runtime/shims/memory.cpp#L548)

For tensors with data types larger than 4 bytes, such as `INT64` (8 bytes per element), this creates a heap buffer overflow. The same vulnerable pattern exists at line 556 for the destination buffer allocation, with its corresponding `cudaMemcpy` at line 596.

Exploit Scenario

An attacker provides a malicious ExecuTorch model file containing INT64 tensors with controlled dimensions. When a victim loads this model and the CUDA backend performs a tensor copy operation with mismatched strides, the allocation at line 542 reserves only half the required memory. The subsequent `cudaMemcpy` writes twice as much data as allocated, overwriting adjacent heap allocations.

Recommendations

Short term, replace the hardcoded `sizeof(float)` with the actual element size already computed at line 527, or use `total_bytes` directly for allocation since it already contains the correct byte count. Apply this fix to both lines 542 and 556. This prevents the type mismatch by ensuring the allocated buffer matches the copy size.

Long term, refactor the allocation pattern to use a helper function that derives buffer sizes from tensor metadata rather than assuming a specific element size. Add assertions that verify allocation sizes match copy sizes before performing memory operations. This prevents similar type confusion vulnerabilities from occurring in future code changes.

References

- CWE-122: Heap-based Buffer Overflow
- CWE-131: Incorrect Calculation of Buffer Size
- CWE-681: Incorrect Conversion between Numeric Types

22. Integer overflow in data loader bounds checking

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-22
Target: `extension/data_loader/mmap_data_loader.cpp`, `extension/data_loader/file_data_loader.cpp`, `extension/data_loader/file_descriptor_data_loader.cpp`	

Description

The data loader implementations contain an integer overflow vulnerability in their bounds checking logic. The `MmapDataLoader`, `FileDataLoader`, and `FileDescriptorDataLoader` classes all use the expression `offset + size <= file_size_` to validate that requested data ranges fall within the bounds of the loaded file. However, this check does not account for integer overflow when adding `offset` and `size`.

```
Error MmapDataLoader::validate_input(size_t offset, size_t size) const {
  ET_CHECK_OR_RETURN_ERROR(
      // Probably had its value moved to another instance.
      fd_ >= 0,
      InvalidState,
      "Uninitialized");
  ET_CHECK_OR_RETURN_ERROR(
      offset + size <= file_size_,
      InvalidArgument,
      "File %s: offset %zu + size %zu > file_size_ %zu",
      file_name_,
      offset,
      size,
      file_size_);
  return Error::Ok;
}
```

Figure 22.1: Vulnerable bounds check in `MmapDataLoader` (`extension/data_loader/mmap_data_loader.cpp#L156-171`)

The same vulnerable pattern exists in `FileDataLoader::load` and `FileDescriptorDataLoader::load`.

The vulnerability is reachable through the `DataLoader::load` interface, which is called by `Program::load` when parsing PTE files. The `offset` and `size` values originate from the PTE file's segment metadata (`DataSegment.offset` and `DataSegment.size` fields).

```
const executorch_flatbuffer::DataSegment* data_segment =
    segments->Get(constant_segment->segment_index());
Result<FreeableBuffer> constant_segment_data = loader->load(
    segment_base_offset + data_segment->offset(),
    data_segment->size(),
    DataLoader::SegmentInfo(
        DataLoader::SegmentInfo::Type::Constant,
        constant_segment->segment_index()));
```

Figure 22.2: The `Program::load` function passes the offset and size from the data segment to the data loader (`runtime/executor/program.cpp#L228–235`)

Exploit Scenario

An attacker crafts a malicious PTE file with a `DataSegment` containing `offset = 0xFFFFFFFFFFFFFF00` and `size = 0x200`. When a victim application loads this PTE file using `Program::load`, the runtime calls the data loader's `load` method with these values. The addition `0xFFFFFFFFFFFFFF00 + 0x200` overflows to `0x100`, which passes the bounds check against the actual file size. For `MmapDataLoader`, the subsequent `mmap` call receives the original out-of-bounds offset, potentially mapping uninitialized memory or causing undefined behavior.

Recommendations

Short term, replace the arithmetic bounds check with an overflow-safe comparison, like `offset <= file_size_ && size <= file_size_ - offset`.

Long term, establish a coding standard that requires all arithmetic operations on untrusted input to use overflow-safe patterns or dedicated safe arithmetic functions. Consider adopting a safe integer library or compiler built-ins (such as `__builtin_add_overflow`) for bounds checking throughout the codebase.

23. Infinite loop with JF_CALL

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-EXECUTORCH-23
Target: `runtime/executor/method.cpp`	

Description

ExecuTorch programs can go into an infinite loop by pointing JumpFalseCall to itself (figure 23.1). There is no mechanism that prevents the creation of programs with infinite loops or a runtime limitation of executed instructions.

```
case executorch_flatbuffer::InstructionArguments::JumpFalseCall: {
  EXECUTORCH_SCOPE_PROF("JF_CALL");
  internal::EventTracerProfileOpScope event_tracer_op_scope =
      internal::EventTracerProfileOpScope(event_tracer_, "JF_CALL");
  // We know that instr_args_as_JumpFalseCall is non-null because it was
  // checked at init time.
  auto jf_call = instruction->instr_args_as_JumpFalseCall();
  // We know that index is a valid values_ index because it was checked at
  // init time.
  auto index = jf_call->cond_value_index();
  Result<bool> jf_result = parse_cond_value(values_[index]);
  if (jf_result.ok()) {
    if (!jf_result.get()) {
      next_instr_idx = jf_call->destination_instruction();
```

Figure 23.1: The next instruction can be set to the same index (runtime/executor/method.cpp#L1456)

Exploit Scenario

An attacker crafts a malicious PTE file with an infinite loop which hangs the program and consumes all the battery power.

Recommendations

Short term, add a counter for executed instructions with a configurable limit. This will prevent programs from looping infinitely.

Long term, expand the fuzz testing suite and make sure all low-level API functions that can be called are exercised by the fuzzer.

24. Integer overflow in program file size validation

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-24
Target: `runtime/executor/program.cpp`, `schema/extended_header.h`	

Description

The `Program::load` function in `program.cpp` contains an integer overflow vulnerability when calculating the expected file size from the extended header fields. The function reads `segment_base_offset` and `segment_data_size` as `uint64_t` values from the PTE file header and stores them as `size_t` local variables. It then performs an unchecked addition of these values to compute the expected file size.

```
// The header has the program size.
program_size = eh->program_size;
segment_base_offset = eh->segment_base_offset;
segment_data_size = eh->segment_data_size;

// segment_data_size was added in ET 1.0 release. For BC, only check the
// expected file size when there are no segments or when segment_data_size
// is positive (0-value may indicate no segments)
if ((segment_data_size == 0 && segment_base_offset == 0) ||
    segment_data_size > 0) {
  size_t expected = segment_base_offset == 0
      ? program_size
      : segment_base_offset + segment_data_size;
  size_t actual = loader->size().get();
  ET_CHECK_OR_RETURN_ERROR(
      expected <= actual,
      InvalidProgram,
      "File size is too small. ...");
}
```

Figure 24.1: Computing the expected file size may overflow (executorch/runtime/executor/program.cpp#L83–L103)

The `ExtendedHeader` struct defines these fields as 64-bit unsigned integers, however, they are both assigned to `size_t` local variables. On 32-bit systems, `size_t` is a 32-bit type, causing implicit truncation when assigning from `uint64_t`. Even on 64-bit systems, an attacker can craft values close to `SIZE_MAX` such that the addition `segment_base_offset + segment_data_size` wraps around to a small value. This causes the file size validation check `expected <= actual` to pass even when the actual file is smaller than required.

The `segment_base_offset` value is subsequently used in multiple arithmetic operations when loading segments:

```
Result<FreeableBuffer> constant_segment_data = loader->load(
    segment_base_offset + data_segment->offset(),
    data_segment->size(),
    // ...);
```

Figure 24.3: Use of `segment_base_offset` in segment loading (`runtime/executor/program.cpp#L230–L235`)

If the initial file size check is bypassed, these subsequent operations can produce invalid offsets, leading to out-of-bounds memory access when the data loader attempts to read from the calculated positions.

Exploit Scenario

An attacker crafts a malicious PTE file with an extended header containing large `segment_base_offset` and `segment_data_size`. On a 64-bit system, these fields sum to a small value due to 64-bit overflow. The file size validation check in `Program::load` passes because the small wrapped value is less than the actual file size. When the runtime subsequently attempts to load segments using `segment_base_offset + segment->offset()`, the resulting offset may point outside the file boundaries. Depending on the data loader implementation, this can cause the runtime to read uninitialized memory, crash, or potentially expose sensitive data from adjacent memory regions.

Recommendations

Short term, add overflow checks before performing arithmetic on values read from the extended header. Verify that `segment_base_offset + segment_data_size` does not exceed `SIZE_MAX` before computing the expected file size, and validate that segment offset calculations do not overflow before passing them to the data loader. This prevents attackers from using integer wraparound to bypass file size validation.

Long term, avoid implicit casts between different integer types, and prefer explicitly sized integer types like `uint32_t` and `uint64_t` over `unsigned int`, `unsigned long`, `unsigned long long`, and `size_t`.

25. Out-of-bounds write in Cadence HiFi bmm operator

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-EXECUTORCH-25
Target: `backends/cadence/hifi/operators/op_bmm.cpp`	

Description

The Cadence HiFi backend implementation of the batched matrix multiplication (`bmm`) operator contains an out-of-bounds write when initializing a temporary bias buffer. The `bmm_out` function allocates a temporary buffer to serve as a zero-bias input for the `xa_nn_matmul_f32xf32_f32` function, but the subsequent initialization only writes a single element past the end of the allocated memory.

```
WORD32* __restrict__ tmp =
    (WORD32* __restrict__)kernels::allocate_temp_memory(
        ctx, (batch_size * m * p) * sizeof(float));

ET_KERNEL_CHECK(ctx, tmp != nullptr, MemoryAllocationFailed, out);

tmp[batch_size * m * p] = {0};
```

Figure 25.1: Out-of-bounds write in bias buffer initialization (backends/cadence/hifi/operators/op_bmm.cpp#L79–L85)

The apparent intent of this code is to zero-initialize the entire buffer using aggregate initialization so that the matrix multiplication adds no bias. However, as it stands, the code is only assigning a single value to a single (out-of-bounds) element.

The `tmp` buffer is subsequently used as the bias parameter (`p_bias`) in a call to `xa_nn_matmul_f32xf32_f32`:

```
const FLOAT32* __restrict__ p_bias = (const FLOAT32* __restrict__)tmp;
// ...
xa_nn_matmul_f32xf32_f32(
    p_out,
    p_mat1,
    p_vec,
    p_bias,
    rows,
    cols1,
    row_stride1,
    vec_count,
    vec_offset,
```

```
    out_offset,
    out_stride);
```

Figure 25.2: Usage of the bias buffer in matrix multiplication (backends/cadence/hifi/operators/op_bmm.cpp#L97–L138)

We wrote a Semgrep rule for identifying functions attempting to use aggregate initialization to initialize pointer values. This did not identify any other instances of this issue in the codebase.

Exploit Scenario

An ML application based on the Executorch library uses the Cadence HiFi backend. The program run by the application uses the Cadence HiFi batched multiplication operator. Because the bias input to the matrix multiplication is not properly initialized, the backend silently produces wrong results, which degrades the performance of the model.

Recommendations

Short term, replace the erroneous single-element write with a proper buffer initialization using `memset` to zero the entire bias buffer.

Long-term, run tests with sanitizers like Asan and UBSan enabled to catch out-of-bounds memory accesses and undefined behavior.

26. Integer overflow in PlatformMemoryAllocator allocation

Severity: **High**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-EXECUTORCH-26
Target: `runtime/executor/platform_memory_allocator.h`	

Description

The `PlatformMemoryAllocator::allocate` method computes the total allocation size by adding three values without checking for integer overflow. When the sum of `sizeof(AllocationNode)`, the requested `size`, and the `alignment` exceeds `SIZE_MAX`, the result wraps around to a small value. The subsequent call to `pal_allocate` then allocates a buffer that is too small to hold the requested data, leading to a heap buffer overflow when the caller writes to the returned pointer.

```
void* allocate(size_t size, size_t alignment = kDefaultAlignment) override {
  if (!isPowerOf2(alignment)) {
    ET_LOG(Error, "Alignment %zu is not a power of 2", alignment);
    return nullptr;
  }

  // Allocate enough memory for the node, the data and the alignment bump.
  size_t alloc_size = sizeof(AllocationNode) + size + alignment;
  void* node_memory = runtime::pal_allocate(alloc_size);

  // ...
}
```

Figure 26.1: Vulnerable allocation size calculation (runtime/executor/platform_memory_allocator.h#L43–L51)

The base `MemoryAllocator` class uses overflow-safe arithmetic utility functions from `safe_numerics.h`. However, `PlatformMemoryAllocator` overrides the `allocate` method and does not handle overflows.

The `PlatformMemoryAllocator` is automatically instantiated as a fallback temporary allocator when no user-provided `temp_allocator` is configured in the `MemoryManager`. This occurs during `Method::load` when processing a PTE model file.

During model execution, kernel operators may request temporary memory allocations through `KernelRuntimeContext::allocate_temp`, which delegates to the temporary allocator's `allocate` method.

Exploit Scenario

During `Method::execute`, a kernel operator allocates a temporary buffer based on tensor dimensions from a corrupted model file. Since no temporary allocator is configured, the call is handled by the platform memory allocator. On 32-bit systems the allocation size computation in figure 26.1 overflows, causing the `allocate` method to return an undersized buffer. The kernel then writes data sized according to the original tensor dimensions into the undersized buffer, corrupting adjacent heap memory.

Recommendations

Short-term, add overflow checking to the allocation size calculation in `PlatformMemoryAllocator::allocate` using the existing `c10::add_overflows` function. Return `nullptr` if the addition overflows.

Long-term, run tests with sanitizers like Asan and UBSan enabled to catch out-of-bounds memory accesses and undefined behavior.

27. Integer overflows in Vulkan tensor operations

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-27
Target: `backends/vulkan/runtime/utils/VecUtils.h`, `backends/vulkan/runtime/api/containers/Tensor.cpp`	

Description

The `multiply_integers` function in the Vulkan backend is used to compute the product of tensor dimensions without checking for integer overflow. When tensor dimensions are deserialized from a PTE file, the product of these dimensions determines the number of elements in the tensor, and the size of corresponding GPU buffer allocations. If a PTE file contains a tensor with dimensions whose product exceeds `INT64_MAX`, the multiplication wraps around to a small or negative value, resulting in an undersized buffer allocation. Subsequent operations that copy data into this buffer based on the actual tensor size will write beyond the allocated memory, causing a heap buffer overflow.

The `multiply_integers` function uses `std::accumulate` with `std::multiplies<>()` to compute the product of all elements in a container as seen in figure 27.1.

```
template <
    typename C,
    std::enable_if_t<std::is_integral<typename C::value_type>::value, int> = 0>
inline int64_t multiply_integers(const C& container) {
  return std::accumulate(
      container.begin(),
      container.end(),
      static_cast<int64_t>(1),
      std::multiplies<>());
}
```

Figure 27.1: The `multiply_integers` function performs unchecked multiplication and may overflow (`backends/vulkan/runtime/utils/VecUtils.h#L472–L481`)

The `add_tensor_to_graph` function reads tensor dimensions directly from the FlatBuffer during model deserialization without validation. These are then passed to the `vTensor` constructor, which calls `multiply_integers` to compute `numel_` and `physical_numel_`:

```
vTensor::vTensor(
    vTensor& other,
    const std::vector<int64_t>& sizes,
    const std::vector<int64_t>& dim_order)
```

```
    : packed_dim_info_(other.packed_dim_info_),
      dtype_(other.dtype_),
      // Copy tensor size metadata
      sizes_(sizes.begin(), sizes.end()),
      padded_sizes_(calculate_padded_sizes(sizes_, packed_dim_info_)),
      dim_order_(dim_order.begin(), dim_order.end()),
      axis_map_(calculate_axis_map(sizes_, utils::kDefaultAxisMap)),
      strides_(calculate_strides(sizes_.size(), padded_sizes_, dim_order_)),
      numel_(utils::multiply_integers(sizes_)),
      physical_numel_(calculate_gpu_buffer_numel(dtype_, padded_sizes_)),
      // ...
      {
     // ...
}
```

Figure 27.2: The `vTensor` constructor computes numel using unchecked multiplication (backends/vulkan/runtime/api/containers/Tensor.cpp#L807–L837)

The `physical_numel_` value, derived from another call to `multiply_integers`, is used to allocate GPU buffers in the `allocate_buffer` function. This function performs a size check on the given tensor size `numel`. However, if the size computation has already overflowed, this check is ineffective since a small wrapped value will pass this check.

We wrote a Semgrep rule to identify other uses of functions like `std::reduce` and `std::accumulate` to compute buffer sizes. This identified the following ten instances in the codebase where overflows may cause memory corruption or other unexpected behavior due to overflows.

- `accumulate(io->size.begin(), io->size.end(), io->elt_size, std::multiplies<>())` in backends/arm/runtime/VGFBackend.cpp
- `accumulate(io->size.begin(), io->size.end(), io->elt_size, std::multiplies<>())` in backends/arm/runtime/VGFBackend.cpp
- `std::accumulate(dims, dims + rank, GetDataTypeSize(data_type), std::multiplies<>())` in backends/qualcomm/aot/wrappers/TensorWrapper.cpp
- `std::accumulate(container.begin(), container.end(), static_cast<int64_t>(1), std::multiplies<>())` in backends/vulkan/runtime/utils/VecUtils.h
- `std::accumulate(begin, end, static_cast<int64_t>(1), std::multiplies<>())` in backends/vulkan/runtime/utils/VecUtils.h
- `std::reduce(mCacheShape.begin(), mCacheShape.begin() + kCacheLengthDim, 1, std::multiplies<>())` in

examples/mediatek/executor_runner/llama_runner/LlamaModelChunk.cpp

- `std::reduce(mCacheShape.begin() + kCacheLengthDim + 1, mCacheShape.end(), kCacheTypeSize, std::multiplies<>())` in examples/mediatek/executor_runner/llama_runner/LlamaModelChunk.cpp
- `std::accumulate(cache_lengths_.begin(), cache_lengths_.end(), 0)` in examples/models/llama/runner/static_attention_io_manager.h
- `std::accumulate(..., executorch::runtime::elementSize(output_tensor.scalar_type()), std::multiplies<int>())` in examples/qualcomm/executor_runner/qnn_executor_runner.cpp
- `std::accumulate(x, x + size, 0.f)` in kernels/portable/cpu/vec_ops.h

Some of these instances are part of examples, which means that they are not directly exposed by the library. However, they still serve as templates for how to use the API, which may cause third-party developers to import vulnerable code patterns into their own implementations.

Exploit Scenario

An attacker crafts a malicious PTE model file containing a Vulkan delegate blob with maliciously chosen tensor dimensions. When the model is loaded, `multiply_integers` computes the product of these dimensions, which overflows `int64_t` and wraps to a small positive value. The Vulkan backend allocates a GPU buffer based on this corrupted size. When the application later executes inference with legitimately-sized input data, the `copy_into_staging` function writes the full input data into the undersized buffer, causing a heap buffer overflow.

Recommendations

Short term, add overflow checking to the `multiply_integers` function by validating that each multiplication will not exceed `INT64_MAX` before performing it.

Long term, implement validation of tensor dimensions at the deserialization boundary before dimensions are used to construct tensors and implement a single method that returns tensor sizes. Consider enforcing maximum allowable values for individual dimensions and their products based on realistic hardware constraints.

28. Unbounded memory allocation from untrusted PTE file

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-28
Target: `runtime/executor/method_meta.cpp`, `examples/portable/executor_runner/executor_runner.cpp`, `schema/program.fbs`	

Description

The ExecuTorch runtime does not validate memory buffer sizes read from PTE files before attempting allocation. Buffer sizes are defined as signed 64-bit integers in the FlatBuffer schema and are read directly from untrusted PTE files without bounds checking. These values are then cast to unsigned `size_t` and used to allocate memory, allowing malicious or malformed PTE files to request allocations of hundreds of gigabytes or even exabytes.

The vulnerability exists in the buffer size extraction method, which returns values directly from the FlatBuffer schema without validation:

```
Result<int64_t> MethodMeta::memory_planned_buffer_size(size_t index) const {
  auto num_buffers = this->num_memory_planned_buffers();
  ET_CHECK_OR_RETURN_ERROR(
      index < num_buffers,
      InvalidArgument,
      "index %zu out of range. num_buffers: %zu",
      index,
      num_buffers);
  // Index zero is reserved internally
  return s_plan_->non_const_buffer_sizes()->Get(index + 1);
}
```

Figure 28.1: Unvalidated buffer size extraction (`runtime/executor/method_meta.cpp`)

The FlatBuffer schema defines buffer sizes as signed 64-bit integers without constraints:

```
// List of buffer sizes for non_constant memory allocations
non_const_buffer_sizes: [int64];
```

Figure 28.2: FlatBuffer schema allowing unbounded int64 values (`schema/program.fbs`)

The allocation site performs an unsafe cast from `int64_t` to `size_t` without validation:

```
for (size_t id = 0; id < num_memory_planned_buffers; ++id) {
  size_t buffer_size =
      static_cast<size_t>(method_meta->memory_planned_buffer_size(id).get());
  ET_LOG(Info, "Setting up planned buffer %zu, size %zu.", id, buffer_size);
  planned_buffers.push_back(std::make_unique<uint8_t[]>(buffer_size));
  planned_spans.push_back({planned_buffers.back().get(), buffer_size});
}
```

Figure 28.3: Unchecked allocation of untrusted buffer sizes (examples/portable/executor_runner/executor_runner.cpp#L408–L415)

Overall, there are three issues. First, negative `int64_t` values in the PTE file become extremely large positive `size_t` values when cast due to two's complement representation. Second, even positive values can specify arbitrarily large allocations limited only by the `int64` range. Third, the runtime attempts allocation immediately without sanity checks or maximum size limits.

Exploit Scenario

An attacker crafts a malicious PTE file with extremely large buffer sizes in the execution plan. The attacker creates a FlatBuffer with `non_const_buffer_sizes` set to values such as 188978561026 (177 GB) or 18446741187491528708 (approximately 16 exabytes). When a victim loads this PTE file using the executor_runner or any application using the ExecuTorch runtime, the runtime reads these values from the FlatBuffer without validation. The negative or extremely large int64 values are cast to `size_t`, and the runtime attempts to allocate the requested memory. The allocation attempt causes the process to hang for extended periods attempting to allocate memory, eventually triggering an out-of-memory condition that crashes the application or exhausts system resources. The attacker achieves denial of service against any application processing PTE files, and the vulnerability prevents fuzzing tools from exploring code paths beyond the allocation point, creating a blind spot in security testing.

```
$ ./build/executor_runner
--model-path=fuzz/targets/pte_parser/artifacts/timeout-a9c543d11a854c202898b5d33621c
de6624dd2b9
E 00:00:00.000957 executorch:program.cpp:264] !!DEPRECATED!! This branch is
deprecated from ExecuTorch 0.7; re-export this PTE file to ensure support on newer
runtimes.
I 00:00:00.000977 executorch:executor_runner.cpp:353] Model file
fuzz/targets/pte_parser/artifacts/timeout-a9c543d11a854c202898b5d33621cde6624dd2b9
is loaded.
I 00:00:00.000984 executorch:executor_runner.cpp:362] Using method
I 00:00:00.000990 executorch:executor_runner.cpp:413] Setting up planned buffer 0,
size 236.
I 00:00:00.001018 executorch:executor_runner.cpp:413] Setting up planned buffer 1,
size 0.
I 00:00:00.001025 executorch:executor_runner.cpp:413] Setting up planned buffer 2,
size 188978561026.
```

```
I 00:00:10.360782 executorch:executor_runner.cpp:413] Setting up planned buffer 3,
size 18446741187491528708.
=================================================================
==21344==ERROR: AddressSanitizer: requested allocation size 0xfffffd6000000004
(0xfffffd6000001008 after adjustments for alignment, red zones etc.) exceeds maximum
supported size of 0x10000000000 (thread T0)
    #0 0x000106a25e5c in _Znam+0x6c
(libclang_rt.asan_osx_dynamic.dylib:arm64+0x6de5c)
    #1 0x0001022d7998 in std::__1::__unique_if<unsigned char
[]>::__unique_array_unknown_bound std::__1::make_unique[abi:se190102]<unsigned char
[]>(unsigned long)+0x20 (executor_runner:arm64+0x100087998)
    #2 0x0001022d3098 in main+0x26a4 (executor_runner:arm64+0x100083098)
    #3 0x0001857ddd50  (<unknown module>)

==21344==HINT: if you don't care about these errors you may set
allocator_may_return_null=1
SUMMARY: AddressSanitizer: allocation-size-too-big
(executor_runner:arm64+0x100087998) in std::__1::__unique_if<unsigned char
[]>::__unique_array_unknown_bound std::__1::make_unique[abi:se190102]<unsigned char
[]>(unsigned long)+0x20
==21344==ABORTING
fish: Job 1, './build/executor_runner --model…' terminated by signal SIGABRT (Abort)
```

Figure 28.4: A fuzzer generated PTE file that triggers the bug

Description

Short term, add validation in the `MethodMeta::memory_planned_buffer_size` method to reject negative values and enforce a maximum buffer size limit before returning the value to callers. This validation should return an `InvalidProgram` error for out-of-range values, preventing the allocation attempt. This prevents the issue because all callers receive validated buffer sizes that are guaranteed to be non-negative and within reasonable bounds before any allocation occurs.

Long term, integrate continuous fuzzing into the development pipeline to systematically identify input validation vulnerabilities. This issue was discovered through fuzzing, demonstrating its effectiveness for file format parsing code. Establish fuzzing targets for all PTE parsing components and integrate results into continuous integration to catch regressions. Configure fuzzers with timeouts and resource limits to detect both crashes and resource exhaustion. This prevents similar issues because fuzzing automatically explores unexpected input combinations that violate implicit assumptions about data validity.

29. Large number of security-relevant compiler warnings

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-EXECUTORCH-29
Target: Multiple	

Description

Building the ExecuTorch codebase with compiler warnings enabled reveals a large number of problematic patterns that reduce code auditability and create potential vectors for undefined behavior.

Warning Type	Description	Findings
`-Wsign-conversion`	Implicit signed/unsigned conversion	681
`-Wshorten-64-to-32`	Implicit 64-bit to 32-bit truncation	201
`-Wunused-parameter`	Unused function parameter	38
`-Wimplicit-int-conversion`	Implicit integer precision loss	16
`-Wsign-compare`	Comparison of integers of different signs	5
`-Wcast-align`	Cast increases alignment requirements	2
`-Wcast-qual`	Cast discards const qualifier	1

A representative example occurs in the convolution kernel implementation, where coordinate calculations mix signed and unsigned 64-bit arithmetic before truncating to 32-bit values.

```
size_t in_y = stride_y * out_y + dilation_y * w_y - padding_y;
in_coord[2] = in_y;
// Only proceed if input y coordinate is within bounds
if (in_y >= 0 && in_y < in_H) {
  // ...
}
```

Figure 29.1: Coordinate calculation with implicit conversions (kernels/portable/cpu/op_convolution.cpp#L109–L112)

The example in figure 29.1 exhibits three problematic behaviors:

1. The subtraction `stride_y * out_y + dilation_y * w_y - padding_y` mixes signed and unsigned values. On 64-bit platforms, this causes C++ usual arithmetic conversions to promote all operands to unsigned `size_t` values. If `padding_y` exceeds the sum of the other terms, the result wraps to a large positive value rather than becoming negative.
2. On 64-bit platforms, the assignment `in_coord[2] = in_y` silently truncates a 64-bit `size_t` to a 32-bit `SizesType` (defined as `int32_t`).
3. The bounds check `in_y >= 0` is vacuously true because `in_y` is unsigned.

The `in_coord` array is used to compute an array index into the underlying input tensor data. If the value is corrupted due to implicit casts, the `conv2d_impl` may end up reading from the wrong offset within the input tensor data buffer.

While most of these instances are likely unproblematic from a security standpoint, the volume of warnings indicate a systemic issue that should be addressed, since any security-relevant problems are currently obscured by hundreds of similar but benign findings.

Exploit Scenario

A loaded PTE file contains tensor metadata which exercises an edge case in the implementation. Because of an implicit sign conversion followed by an implicit truncation in the convolution kernel, the implementation reads out-of-bounds when attempting to read the input tensor data. This severely degrades model output, causing the team to waste valuable development time attempting to triage the issue.

Recommendations

Short-term, review the findings identified by the implicit conversion warnings `-Wshorten-64-to-32` and `-Wimplicit-int-conversion`. Either refactor the problematic code path, or add a comment describing why it's safe.

Long-term, enable a targeted set of compiler warnings in the build system, and regularly address the resulting warnings. When existing warnings have been fixed, add the `-Werror`

flag to turn all compiler warnings into hard errors to ensure new issues are addressed immediately. Introduce explicit safe conversion functions (such as `narrow_cast` with runtime assertions in debug builds) to document intentional truncations and distinguish them from accidental ones.

References

- SEI CERT C Coding Standard: INT31-C

30. Memory corruption due to unsafe mutation of FlatBuffer-backed tensor data

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-30
Target: `runtime/executor/program.cpp`	

Description

The ExecuTorch runtime allows direct mutation of tensor data stored within FlatBuffer structures, violating the fundamental security principle that FlatBuffers must remain immutable after verification. When tensor data is loaded from a PTE file, the runtime obtains a pointer to data embedded in the FlatBuffer (figure 30.1), casts away the const qualifier (figure 30.2), and later uses this pointer as a destination for memory writes. This corrupts the FlatBuffer's internal structures that follow the tensor data in memory, leading to runtime instability and potential exploitation. This issue appeared after applying a fix for finding TOB-EXECUTORCH-13.

```
const auto& constant_buffer = *constant_buffer_ptr;
const auto* storage = constant_buffer[buffer_index]->storage();
auto storage_size = storage == nullptr ? 0 : storage->size();
// nbytes (requested from the program) should be less than storage_size
// (size of the constant buffer from PTE), to prevent reading out of bounds.
// in some cases storage size may be larger than nbytes because of padding;
// executorch-tensor-alignment, or 16 by default.
ET_CHECK_OR_RETURN_ERROR(
    nbytes <= storage_size,
    InvalidArgument,
    "Requested nbytes %zu exceeds constant buffer storage size %zu",
    nbytes,
    static_cast<size_t>(storage_size));

return storage->data();
```

Figure 30.1: The `Program::get_constant_buffer_data` method returns pointer directly to FlatBuffer memory (`runtime/executor/program.cpp#L402–L416`)

```
 // Constant, stored in PTE file.
} else if (data_buffer_idx > 0 && allocation_info == nullptr) {
  auto const_data =
      program->get_constant_buffer_data(data_buffer_idx, nbytes);
  if (!const_data.ok()) {
    return const_data.error();
  }
```

```
// The const_cast is 'ok' here because the program and runtime should
// guarantee that this data is never modified.
return const_cast<void*>(const_data.get());
```

Figure 30.2: The implementation of `getTensorDataPtr` (runtime/executor/tensor_parser_exec_aten.cpp#L226–L236)

The `FlatCC` security documentation explicitly warns against this practice:

It is not safe to modify buffers in-place unless the buffer is absolutely trusted. [...] Modifying a buffer is unsafe because it is possible to place one table inside another table, as an example, even if this is not valid. Such overlaps are too expensive to verify by a standard verifier. [...] Changing a field in one table would cause the other table's field to point out of bounds. This is an attack vector that anyone wanting to hack a system is likely to use.

We found an out-of-bounds read via fuzzing that exploits this issue. There are three steps:

1. A 0-sized buffer is returned from `get_constant_buffer_data`:

   `main`
   `Program::load_method`
   `Method::load`
   `Method::init`
   `Method::parse_values`
   `deserialization::parseTensor`
   `deserialization::getTensorDataPtr`
   `Program::get_constant_buffer_data` (returns address X)

2. The buffer is later used as a destination for tensor copy:

   `main`
   `extension::prepare_input_tensors`
   `extension::internal::fill_and_set_input`
   `Method::set_input`
   `internal::copy_tensor_data` (writes to X, overwrites the offset to `values`)

3. The corruption is triggered in `MethodMeta::num_inputs`:

```
==8396==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6190000086f0 at
pc 0x000102508c6c bp 0x00016f2e4470 sp 0x00016f2e4468
READ of size 4 at 0x6190000086f0 thread T0
    #0 0x000102508c68 in flatbuffers::Vector<int, unsigned int>::size() const+0x60
(executor_runner:arm64+0x1019f0c68)
    #1 0x00010251edc8 in executorch::runtime::MethodMeta::num_inputs() const+0x50
(executor_runner:arm64+0x101a06dc8)
    #2 0x00010251ee1c in executorch::runtime::MethodMeta::input_tag(unsigned long)
```

```
const+0x34 (executor_runner:arm64+0x101a06e1c)
    #3 0x000100bd19d8 in
executorch::extension::prepare_input_tensors(executorch::runtime::Method&,
executorch::extension::PrepareInputTensorsOptions,
std::__1::vector<std::__1::pair<char*, unsigned long>,
std::__1::allocator<std::__1::pair<char*, unsigned long>>> const&)+0x5b0
(executor_runner:arm64+0x1000b99d8)
    #4 0x000100b9e3d4 in main+0x2b80 (executor_runner:arm64+0x1000863d4)
    #5 0x0001857ddd50  (<unknown module>)
```

Figure 30.3: FlatBuffer structure is corrupted; the bytes describing offset of inputs field are controlled by the attacker.

Exploit Scenario

A vulnerable device loads a malicious PTE file that corrupts internal FlatBuffer structures and takes over the execution of the program.

Description

Short term, remove the possibility of writing back to the FlatBuffer structure that was loaded. The FlatBuffer verification doesn't validate if all the buffer sizes are correct and don't overlap with FlatBuffer structures. Review the code for other places where the same pattern is used.

Long term, integrate continuous fuzzing into the development pipeline to systematically identify input validation vulnerabilities. This issue was discovered through fuzzing, demonstrating its effectiveness for file format parsing code. Establish fuzzing targets for all PTE parsing components and integrate results into continuous integration to catch regressions. Configure fuzzers with timeouts and resource limits to detect both crashes and resource exhaustion. This prevents similar issues because fuzzing automatically explores unexpected input combinations that violate implicit assumptions about data validity.

31. Type confusion in BoxedEvalueList after MoveCall

Severity: **Medium**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-EXECUTORCH-31
Target: `runtime/executor/tensor_parser_exec_aten.cpp`, `runtime/core/evalue.h`, `runtime/executor/method.cpp`	

Description

The `parseTensorList` function validates that specific indices in the values array contain Tensor objects and stores pointers to those array locations. However, `MoveCall` instructions can subsequently swap the contents of the values array during execution. This creates a time-of-check-time-of-use (TOCTOU) vulnerability where the stored pointers reference memory locations whose contents have changed since validation, leading to type confusion and memory corruption during destruction.

During method initialization, `parseTensorList` validates each tensor index in the input list. The function stores pointers to the validated array positions rather than copying the values. These pointers are stored in a `BoxedEvalueList` object that persists throughout the method lifetime.

```
// For each tensor index look up the corresponding Tensor (which has been
// already allocated) and stick it in the list.
size_t output_idx = 0;
for (int32_t tensor_index : *tensor_indices) {
  ET_CHECK_OR_RETURN_ERROR(
      tensor_index >= 0 && static_cast<size_t>(tensor_index) < values_len,
      InvalidProgram,
      "Invalid value index %" PRId32 " for TensorList",
      tensor_index);

  // Placement new as the list elements are not initialized, so calling
  // copy assignment is not defined if it's non trivial.
  new (&tensor_list[output_idx]) executorch::aten::Tensor(
      values[static_cast<size_t>(tensor_index)].toTensor());
  evalp_list[output_idx] = &values[static_cast<size_t>(tensor_index)];
  output_idx++;
}
```

Figure 31.1: Type validation and pointer storage in `parseTensorList` (`runtime/executor/tensor_parser_exec_aten.cpp#L91–L107`)

During execution, `MoveCall` instructions perform assignment operations on the values array:

```
case executorch_flatbuffer::InstructionArguments::MoveCall: {
  EXECUTORCH_SCOPE_PROF("MOVE_CALL");
  internal::EventTracerProfileOpScope event_tracer_op_scope =
      internal::EventTracerProfileOpScope(event_tracer_, "MOVE_CALL");
  // We know that instr_args_as_MoveCall is non-null because it was checked
  // at init time.
  auto move_call = instruction->instr_args_as_MoveCall();
  mutable_value(move_call->move_to()) = get_value(move_call->move_from());
} break;
```

Figure 31.2: `MoveCall` instruction execution (runtime/executor/method.cpp#L1462–L1470)

If a `MoveCall` instruction targets an index that was validated by `parseTensorList`, it replaces that array position with a different value. The pointer stored in `BoxedEvalueList` still references the same memory address, but the contents now differ from what was validated.

During method destruction, the `EValue::destroy` method attempts to clean up the `BoxedEvalueList` by calling `get`:

```
} else if (
    isTensorList() &&
    payload.copyable_union.as_tensor_list_ptr != nullptr) {
  // for (auto& tensor : toTensorList()) {
  for (auto& tensor : payload.copyable_union.as_tensor_list_ptr->get()) {
    tensor.~Tensor();
  }
} else if (
```

Figure 31.3: Vulnerable destruction logic (runtime/core/evalue.h#L491–L498)

The `get` method dereferences the stored pointers and attempts to convert each `EValue` to a Tensor. When `to<Tensor>()` is invoked on an `EValue` that no longer contains a Tensor due to a `MoveCall` swap, the type check fails and the process aborts (figure 31.4).

```
executorch::aten::ArrayRef<T> BoxedEvalueList<T>::get() const {
  for (typename executorch::aten::ArrayRef<T>::size_type i = 0;
       i < wrapped_vals_.size();
       i++) {
    ET_CHECK(wrapped_vals_[i] != nullptr);
    unwrapped_vals_[i] = wrapped_vals_[i]->template to<T>();
  }
  return executorch::aten::ArrayRef<T>{unwrapped_vals_, wrapped_vals_.size()};
}
```

Figure 31.4: `BoxedEvalueList::get()` implementation calling `toTensor` (runtime/core/evalue.h#L588–L596)

The fundamental issue is a semantic mismatch: `parseTensorList` validates value semantics (the contents at a point in time) but stores pointer semantics (references to memory locations). The validation does not account for subsequent modifications to the values array, and the stored pointers bypass re-validation at time of use.

We attach `TOB-EXECUTORCH-31.pte` file generated by fuzzer. To observe this behavior in the `executor_runner`, it needs to be modified to return from `main` instead of aborting here, otherwise the `Method` destructor will not be called.

Exploit Scenario

An attacker crafts a malicious PTE file with a `TensorList` referencing indices that initially contain valid Tensors, passing validation during initialization. The attacker includes a `MoveCall` instruction that overwrites one of these indices with a non-Tensor value. During method destruction, the `BoxedEvalueList` dereferences its stored pointer and attempts to convert the swapped value to a Tensor, causing a type confusion that terminates the process. This results in a denial of service and could potentially enable more severe memory corruption if the attacker controls the timing and content of the swapped values.

Recommendations

Short term, modify the `EValue::destroy` method to avoid re-dereferencing the stored pointers in `BoxedEvalueList` when destroying `TensorList` and `ListOptionalTensor` objects. The destructor should clean up the list elements without calling methods that validate the current type of `EValues` in the values array, as these types may have changed during execution due to `MoveCall` operations. This prevents the vulnerability because the destruction logic will not attempt type conversions on values that may have been swapped after the initial validation.

Long term, integrate continuous fuzzing into the development pipeline to systematically identify input validation vulnerabilities. This issue was discovered through fuzzing, demonstrating its effectiveness for file format parsing code. Establish fuzzing targets for all PTE parsing components and integrate results into continuous integration to catch regressions. Configure fuzzers with timeouts and resource limits to detect both crashes and resource exhaustion. This prevents similar issues because fuzzing automatically explores unexpected input combinations that violate implicit assumptions about data validity.

32. Incorrect pointer offset arithmetic in ETDumpGen constructor

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-EXECUTORCH-32
Target: `devtools/etdump/etdump_flatcc.cpp`	

Description

The ETDumpGen constructor contains a pointer arithmetic error that causes incorrect calculation of buffer offsets. When computing the address past the `flatcc_builder` structure, the code performs explicit scaling on a typed pointer. Since the compiler implicitly scales offsets in arithmetic expressions containing typed pointers, this means that scaling is applied twice.

```
ETDumpGen::ETDumpGen(Span<uint8_t> buffer) {
  constexpr size_t max_alloc_buf_size = 128 * 1024;

  // Initialize the flatcc builder_ using the buffer and buffer size.

  if (buffer.data() != nullptr) {
    builder_ =
        (struct flatcc_builder*)internal::align_pointer(buffer.data(), 64);
    uintptr_t buffer_with_builder = (uintptr_t)internal::align_pointer(
        builder_ + sizeof(struct flatcc_builder), 64);
    size_t builder_size =
        (size_t)(buffer_with_builder - (uintptr_t)buffer.data());
    size_t min_buf_size = max_alloc_buf_size + builder_size;
    ET_CHECK_MSG(
        buffer.size() > min_buf_size,
        ...);
    size_t buffer_size = buffer.size() - builder_size;
    alloc_.set_buffer(
        (uint8_t*)buffer_with_builder,
        buffer_size,
        ...);
```

Figure 32.1: Incorrect pointer scaling in the ETDumpGen constructor (devtools/etdump/etdump_flatcc.cpp#L111–L134)

Because `builder_` is of type `struct flatcc_builder*`, the addition on line 120 advances by `sizeof(struct flatcc_builder)` structure elements rather than bytes. This results in an actual byte offset of `sizeof(struct flatcc_builder) * sizeof(struct flatcc_builder)`, which is most likely not intended.

The `flatcc_builder` structure is 400 bytes large on 64-bit ARM platforms. This means that the code computes an offset of 160,000 bytes instead of 400 bytes.

This miscalculation affects the `builder_size` variable, which in turn affects the minimum buffer size check and the parameters passed to `alloc_.set_buffer()`. The incorrectly computed `buffer_with_builder` address is used as the base for the allocator's buffer, pointing far beyond the intended location.

This issue was identified using the CodeQL query `cpp/suspicious-add-sizeof` from the standard query libraries provided by GitHub.

Exploit Scenario

A developer creates an `ETDumpGen` instance with a user-provided buffer for profiling model execution. Due to the double-scaled pointer arithmetic, the computed `min_buf_size` becomes artificially large, causing the size check on line 124 to fail for any reasonably-sized buffer.

Recommendations

Short term, cast the `builder_` pointer to `uint8_t*` before adding `sizeof(struct flatcc_builder)`, or use `builder_ + 1` to advance by exactly one element. This ensures the offset is computed in bytes rather than in multiples of the structure size.

Long term, enable compiler warnings for pointer arithmetic involving explicit size calculations (such as `-Wpointer-arith`), combined with static analysis tools (like CodeQL) and integrate these checks into the projects CI/CD pipeline.

33. Out-of-bounds read due to missing FlatBuffers verification in XNNCompiler

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-33
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`, `backends/xnnpack/serialization/schema.fbs`	

Description

The XNNPACK backend processes FlatBuffer data without verification, allowing malformed PTE files to trigger out-of-bounds memory reads that can lead to memory corruption. The XNNPACK backend uses its own FlatBuffer schema defined in `backends/xnnpack/serialization/schema.fbs` to serialize delegated graph data. The `compileModel` function accesses untrusted FlatBuffer data without first invoking the FlatBuffers verifier, which validates buffer structure, offset bounds, and field presence.

This vulnerability follows the same pattern as TOB-EXECUTORCH-12, where the core program loader similarly failed to verify FlatBuffer data before access, allowing out-of-bounds reads.

```
$ ./build/executor_runner --model-path=TOB-EXECUTORCH-33.pte
==80517==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x614000000200 at
pc 0x0001046d6e98 bp 0x00016b762b00 sp 0x00016b762af8
READ of size 4 at 0x614000000200 thread T0
    #0 0x0001046d6e94 in flatbuffers::IndirectHelper<unsigned int>::Read(unsigned
char const*, unsigned long)+0x70 (executor_runner:arm64+0x10003ee94)
    #1 0x0001046d4e5c in flatbuffers::VectorIterator<unsigned int, unsigned int,
unsigned char const*, unsigned int>::operator*() const+0x50
(executor_runner:arm64+0x10003ce5c)
    #2 0x0001046b4138 in std::__1::vector<unsigned long,
std::__1::allocator<unsigned long>>
executorch::backends::xnnpack::delegate::flatbufferDimsToVector<unsigned
long>(flatbuffers::Vector<unsigned int, unsigned int> const*)+0x240
(executor_runner:arm64+0x10001c138)
    #3 0x0001046b20a8 in
executorch::backends::xnnpack::delegate::defineTensor(xnn_subgraph*,
std::__1::unordered_map<unsigned int, unsigned int, std::__1::hash<unsigned int>,
std::__1::equal_to<unsigned int>, std::__1::allocator<std::__1::pair<unsigned int
const, unsigned int>>>&, fb_xnnpack::XValue const*, fb_xnnpack::XNNGraph const*,
unsigned char const*, std::__1::vector<unsigned int, std::__1::allocator<unsigned
int>>&, std::__1::vector<unsigned int, std::__1::allocator<unsigned int>>&,
executorch::backends::xnnpack::delegate::CompileAllocator&,
executorch::runtime::NamedDataMap const*,
```

```
std::__1::vector<executorch::runtime::FreeableBuffer,
std::__1::allocator<executorch::runtime::FreeableBuffer>>&,
executorch::backends::xnnpack::delegate::XNNWeightsCache*)+0x380
(executor_runner:arm64+0x10001a0a8)
    #4 0x0001046c5000 in
executorch::backends::xnnpack::delegate::XNNCompiler::compileModel(void const*,
unsigned long, executorch::backends::xnnpack::delegate::XNNExecutor*,
executorch::backends::xnnpack::delegate::XNNWeightsCache*, xnn_workspace*,
executorch::runtime::NamedDataMap const*)+0xad8 (executor_runner:arm64+0x10002d000)
    #5 0x0001046f00a0 in
executorch::backends::XnnpackBackend::init(executorch::runtime::BackendInitContext&,
executorch::runtime::FreeableBuffer*,
executorch::runtime::ArrayRef<executorch::runtime::CompileSpec>) const+0x39c
(executor_runner:arm64+0x1000580a0)
    #6 0x000106283114 in
executorch::runtime::BackendDelegate::Init(executorch_flatbuffer::BackendDelegate
const&, executorch::runtime::Program const*,
executorch::runtime::BackendInitContext&,
executorch::runtime::BackendDelegate*)+0x70c (executor_runner:arm64+0x101beb114)
    #7 0x00010628024c in
executorch::runtime::Method::init(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::NamedDataMap const*)+0xc48 (executor_runner:arm64+0x101be824c)
    #8 0x00010627f17c in
executorch::runtime::Method::load(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::Program const*, executorch::runtime::MemoryManager*,
executorch::runtime::EventTracer*, executorch::runtime::NamedDataMap const*)+0x220
(executor_runner:arm64+0x101be717c)
    #9 0x0001062a0578 in executorch::runtime::Program::load_method(char const*,
executorch::runtime::MemoryManager*, executorch::runtime::EventTracer*,
executorch::runtime::NamedDataMap const*) const+0x33c
(executor_runner:arm64+0x101c08578)
    #10 0x00010478b278 in main+0x2a10 (executor_runner:arm64+0x1000f3278)
    #11 0x0001857ddd50  (<unknown module>)
```

Figure 33.1: An AddressSanitizer crash on executing fuzzer generated example triggering the missing FlatBuffer verification in `compileModel`

Exploit Scenario

An attacker crafts a malicious PTE file containing an XNNPACK backend delegate section with out-of-bounds offsets in the FlatBuffer structure. When a service processes this file, the XNNPACK backend accesses the unverified FlatBuffer data and follows the malicious offsets to read from arbitrary memory locations. The out-of-bounds read can leak sensitive information from adjacent heap allocations, such as cryptographic keys or authentication tokens, enabling information disclosure. Alternatively, if the offset points to unmapped memory, the application crashes, causing denial of service.

Recommendations

Short term, add FlatBuffers verification before accessing the XNNPACK delegate data. In the `compileModel` function, create a `flatbuffers::Verifier` and call

`fb_xnnpack::VerifyXNNGraphBuffer()` to validate the buffer structure before processing any fields. Return an error if verification fails.

Long term, audit all backend delegate implementations for missing FlatBuffer verification and establish verification as a mandatory requirement. Document verification requirements in the backend development guide and create a shared verification utility function that all backends must call. Add regression tests that verify backends properly reject malformed FlatBuffers.

34. Out-of-bounds read due to missing XNNHeader verification

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-34
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`, `backends/xnnpack/runtime/XNNHeader.cpp`	

Description

The XNNPACK backend's header parsing function reads offset values from untrusted data and uses them for pointer arithmetic without validating that the computed pointers remain within the allocated buffer bounds.

The vulnerability exists in the `XNNHeader::Parse` function, which extracts four fields from the untrusted header data and returns them without any bounds checking (figure 34.1). These unchecked offset values are subsequently used in `XNNCompiler::compileModel` to compute pointers through arithmetic operations (figure 34.2). We show an AddressSanitizer crash from a fuzzer generated input on figure 34.3.

```
Result<XNNHeader> XNNHeader::Parse(const void* data, size_t size) {
  const uint8_t* header_data = (const uint8_t*)data;

  if (size < XNNHeader::kMinSize) {
    return Error::InvalidArgument;
  }

  const uint8_t* magic_start = header_data + XNNHeader::kMagicOffset;
  if (std::memcmp(magic_start, XNNHeader::kMagic, XNNHeader::kMagicSize) != 0) {
    return Error::NotFound;
  }

  uint32_t flatbuffer_offset =
      GetUInt32LE(header_data + XNNHeader::kFlatbufferDataOffsetOffset);

  uint32_t flatbuffer_size =
      GetUInt32LE(header_data + XNNHeader::kFlatbufferDataSizeOffset);

  uint32_t constant_data_offset =
      GetUInt32LE(header_data + XNNHeader::kConstantDataOffsetOffset);

  uint64_t constant_data_size =
      GetUInt64LE(header_data + XNNHeader::kConstantDataSizeOffset);

  return XNNHeader{
      flatbuffer_offset,
```

```
      flatbuffer_size,
      constant_data_offset,
      constant_data_size};
}
```

Figure 34.1: `XNNHeader::Parse` function returns attacker-controlled offset values without validation (backends/xnnpack/runtime/XNNHeader.cpp#L43–L72)

```
ET_NODISCARD Error XNNCompiler::compileModel(
    const void* buffer_pointer,
    size_t num_bytes,
    XNNExecutor* executor,
    XNNWeightsCache* weights_cache,
    xnn_workspace_t workspace,
    const NamedDataMap* named_data_map) {
  Result<XNNHeader> header = XNNHeader::Parse(buffer_pointer, num_bytes);
  const uint8_t* flatbuffer_data = nullptr;
  const uint8_t* constant_data = nullptr;
  CompileAllocator compile_allocator;

  // Header status can only either be Error::Ok or Error::NotFound
  if (header.ok()) {
    flatbuffer_data = reinterpret_cast<const uint8_t*>(buffer_pointer) +
        header->flatbuffer_offset;
    constant_data = reinterpret_cast<const uint8_t*>(buffer_pointer) +
        header->constant_data_offset;
  } else if (header.error() == Error::NotFound) {
    flatbuffer_data = reinterpret_cast<const uint8_t*>(buffer_pointer);
  } else {
    ET_LOG(Error, "XNNHeader may be corrupt");
    return header.error();
  }

  // Temporarily support identifier XN00 and XN01
  bool is_supported_version =
      strncmp(flatbuffers::GetBufferIdentifier(flatbuffer_data), "XN00", 4) ==
          0 ||
      strncmp(flatbuffers::GetBufferIdentifier(flatbuffer_data), "XN01", 4) ==
          0;
```

Figure 34.2: `strncmp` reads out of bounds read since the `flatbuffer_data` pointer is computed from untrusted header input (backends/xnnpack/runtime/XNNCompiler.cpp#L1809–L1839)

```
$ ./build/executor_runner --model-path=TOB-EXECUTORCH-34.pte
AddressSanitizer:DEADLYSIGNAL
=================================================================
==95187==ERROR: AddressSanitizer: BUS on unknown address (pc 0x0001090a237c bp
0x00016b637630 sp 0x00016b636dc0 T0)
==95187==The signal is caused by a READ memory access.
```

```
==95187==Hint: this fault was caused by a dereference of a high value address (see
register values below).  Disassemble the provided pc to learn which register was
used.
    #0 0x0001090a237c in strncmp+0x48
(libclang_rt.asan_osx_dynamic.dylib:arm64+0x1a37c)
    #1 0x0001047f0a54 in
executorch::backends::xnnpack::delegate::XNNCompiler::compileModel(void const*,
unsigned long, executorch::backends::xnnpack::delegate::XNNExecutor*,
executorch::backends::xnnpack::delegate::XNNWeightsCache*, xnn_workspace*,
executorch::runtime::NamedDataMap const*)+0x52c (executor_runner:arm64+0x10002ca54)
    #2 0x00010481c0a0 in
executorch::backends::XnnpackBackend::init(executorch::runtime::BackendInitContext&,
executorch::runtime::FreeableBuffer*,
executorch::runtime::ArrayRef<executorch::runtime::CompileSpec>) const+0x39c
(executor_runner:arm64+0x1000580a0)
    #3 0x0001063af114 in
executorch::runtime::BackendDelegate::Init(executorch_flatbuffer::BackendDelegate
const&, executorch::runtime::Program const*,
executorch::runtime::BackendInitContext&,
executorch::runtime::BackendDelegate*)+0x70c (executor_runner:arm64+0x101beb114)
    #4 0x0001063ac24c in
executorch::runtime::Method::init(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::NamedDataMap const*)+0xc48 (executor_runner:arm64+0x101be824c)
    #5 0x0001063ab17c in
executorch::runtime::Method::load(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::Program const*, executorch::runtime::MemoryManager*,
executorch::runtime::EventTracer*, executorch::runtime::NamedDataMap const*)+0x220
(executor_runner:arm64+0x101be717c)
    #6 0x0001063cc578 in executorch::runtime::Program::load_method(char const*,
executorch::runtime::MemoryManager*, executorch::runtime::EventTracer*,
executorch::runtime::NamedDataMap const*) const+0x33c
(executor_runner:arm64+0x101c08578)
    #7 0x0001048b7278 in main+0x2a10 (executor_runner:arm64+0x1000f3278)
    #8 0x0001857ddd50  (<unknown module>)
```

Figure 34.3: AddressSanitizer crash on out-of-bounds read in strncmp from figure 34.2.

Exploit Scenario

An attacker crafts a malicious PTE file with a valid XNN header magic ("XH00") but sets `flatbuffer_offset` to a large value (e.g., 0x10000000) exceeding the buffer size. When loaded, `XNNHeader::Parse` returns the malicious offset without validation, and `XNNCompiler::compileModel` uses it for pointer arithmetic, creating an out-of-bounds pointer. Accessing the FlatBuffer identifier at this location triggers a heap buffer overflow read, potentially causing crashes, denial of service, or information disclosure.

Recommendations

Short term, add bounds validation in the `XNNHeader::Parse` function to verify that all offset and size values remain within the provided buffer bounds before returning them. Specifically, check that `flatbuffer_offset + flatbuffer_size <= size` and `constant_data_offset + constant_data_size <= size`. Make sure to check for integer overflows. Return an error if these conditions are not met. This prevents the

vulnerability by ensuring that any pointers computed from the returned offset values will remain within the valid buffer region.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

References

- CWE-823: Use of Out-of-range Pointer Offset
- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

35. NULL pointer dereferences in XNNCompiler::compileModel

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-35
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`	

Description
The `XNNCompiler::compileModel` function dereferences the `xvalues` vector pointer without validating that it is not null, causing a crash when processing malformed XNNPACK FlatBuffers. The issue also concerns a number of other fields used in this function.

Exploit Scenario
An attacker crafts a malicious ExecuTorch model file with an XNNPACK backend delegate containing a FlatBuffer where the `xvalues` vector pointer is null. When a victim application loads this model and attempts to initialize the XNNPACK backend, the `compileModel` function crashes with a null pointer dereference at the for loop iteration. This causes the application to terminate abnormally, resulting in a denial of service.

Recommendations
Short term, add a null check to verify that `flatbuffer_graph->xvalues()` is not null before dereferencing it. Review other fields for null checks as some of them are also vulnerable.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

36. Crash due to unchecked map access in XNNCompiler

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-36
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`	

Description

The XNNPACK backend's node definition functions use unchecked map access to resolve tensor identifiers, causing application termination when processing models with invalid tensor references. The `compileModel` function builds a mapping of tensor identifiers by iterating over the FlatBuffer's `xvalues` vector and populating a `remapped_ids` map. Subsequently, when processing graph nodes, the code calls `remapped_ids.at` to resolve tensor identifiers without verifying that the referenced identifiers exist in the map. When a node references a non-existent tensor identifier, the `std::unordered_map::at` method throws `std::out_of_range`, and because all node definition functions are marked `noexcept`, the exception causes `std::terminate` to be invoked, immediately crashing the application (figure 36.1).

```
Error defineGenericBinaryNode(
    xnn_subgraph_t subgraph_ptr,
    const std::unordered_map<uint32_t, uint32_t>& remapped_ids,
    const fb_xnnpack::_XNNNode2x1* graph_node,
    xnn_binary_operator op_type,
    const struct xnn_binary_params* params,
    fb_xnnpack::XNodeUnion node_type,
    uint32_t debug_handle) noexcept {
  xnn_status status = xnn_define_binary(
      subgraph_ptr,
      op_type,
      params,
      remapped_ids.at(graph_node->input1_id()),
      remapped_ids.at(graph_node->input2_id()),
      remapped_ids.at(graph_node->output_id()),
      graph_node->flags());
```

Figure 36.1: Vulnerable code using unchecked map access in `noexcept` function (backends/xnnpack/runtime/XNNCompiler.cpp#L1809–L1839)

This vulnerability affects all node definition functions in the XNNPACK backend. Analysis of the codebase revealed 74 instances of `remapped_ids.at` calls across the XNNCompiler. We include a stack trace from fuzzer in figure 36.2.

```
libc++abi: terminating due to uncaught exception of type std::out_of_range:
unordered_map::at: key not found
==47860== ERROR: libFuzzer: deadly signal
    #0 ...
    ...
    #10 0x000185b58bcc in std::terminate()+0x68 (libc++abi.dylib:arm64+0x11bcc)
    #11 0x0001028636b8 in
executorch::backends::xnnpack::delegate::defineGenericBinaryNode(xnn_subgraph*,
std::__1::unordered_map<unsigned int, unsigned int, std::__1::hash<unsigned int>,
std::__1::equal_to<unsigned int>, std::__1::allocator<std::__1::pair<unsigned int
const, unsigned int>>> const&, fb_xnnpack::_XNNNode2x1 const*, xnn_binary_operator,
xnn_binary_params const*, fb_xnnpack::XNodeUnion, unsigned int)+0xc58
(pte_parser_fuzzer:arm64+0x1000276b8)
    #12 0x00010286a0b4 in
executorch::backends::xnnpack::delegate::defineMultiplyNode(xnn_subgraph*,
std::__1::unordered_map<unsigned int, unsigned int, std::__1::hash<unsigned int>,
std::__1::equal_to<unsigned int>, std::__1::allocator<std::__1::pair<unsigned int
const, unsigned int>>> const&, fb_xnnpack::XNode const*, fb_xnnpack::XNNGraph
const*)+0x3c4 (pte_parser_fuzzer:arm64+0x10002e0b4)
    #13 0x00010286bf30 in
executorch::backends::xnnpack::delegate::XNNCompiler::compileModel(void const*,
unsigned long, executorch::backends::xnnpack::delegate::XNNExecutor*,
executorch::backends::xnnpack::delegate::XNNWeightsCache*, xnn_workspace*,
executorch::runtime::NamedDataMap const*)+0x1058
(pte_parser_fuzzer:arm64+0x10002ff30)
    #14 0x00010288fa68 in
executorch::backends::XnnpackBackend::init(executorch::runtime::BackendInitContext&,
executorch::runtime::FreeableBuffer*,
executorch::runtime::ArrayRef<executorch::runtime::CompileSpec>) const+0x568
(pte_parser_fuzzer:arm64+0x100053a68)
    #15 0x0001028af5dc in
executorch::runtime::BackendDelegate::Init(executorch_flatbuffer::BackendDelegate
const&, executorch::runtime::Program const*,
executorch::runtime::BackendInitContext&,
executorch::runtime::BackendDelegate*)+0x908 (pte_parser_fuzzer:arm64+0x1000735dc)
    #16 0x0001028ace24 in
executorch::runtime::Method::init(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::NamedDataMap const*)+0x708
(pte_parser_fuzzer:arm64+0x100070e24)
    #17 0x0001028ac278 in
executorch::runtime::Method::load(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::Program const*, executorch::runtime::MemoryManager*,
executorch::runtime::EventTracer*, executorch::runtime::NamedDataMap const*)+0x250
(pte_parser_fuzzer:arm64+0x100070278)
    #18 0x0001028c51d8 in executorch::runtime::Program::load_method(char const*,
executorch::runtime::MemoryManager*, executorch::runtime::EventTracer*,
executorch::runtime::NamedDataMap const*) const+0x50c
(pte_parser_fuzzer:arm64+0x1000891d8)
    ...
```

Figure 36.2: A strack trace from a fuzzer-generated PTE file (TOB-EXECUTORCH-36.pte) that triggers the bug

Exploit Scenario

An attacker crafts a PTE file with an XNNPACK delegate containing nodes that reference non-existent tensor identifiers. For example, a multiply node references `input1_id` value 9999, which was never defined in the tensor values section. When the service processes this file, `defineGenericBinaryNode` calls `remapped_ids.at(9999)`, throwing `std::out_of_range`. Because the function is marked `noexcept`, the runtime invokes `std::terminate`, immediately crashing the application and causing denial of service.

Recommendations

Short term, replace all `remapped_ids.at` calls with a safe accessor pattern that validates identifier existence before access. Create a helper function that checks if the identifier exists in the map using `find` or `contains`, and returns an error code if the identifier is missing. This allows the `noexcept` functions to handle missing identifiers gracefully by returning error codes rather than throwing exceptions, preventing termination and enabling proper error reporting.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

37. Unbounded memory allocation in XNNPACK subgraph creation

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-EXECUTORCH-37
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`	

Description

The XNNPACK backend fails to validate the `num_externs` field from the serialized FlatBuffer before passing it to XNNPACK's `xnn_create_subgraph` function, allowing an attacker to cause unbounded memory allocation and denial of service.

When loading an XNNPACK delegate, the `XNNCompiler::compileModel` function extracts the `num_externs` value directly from the FlatBuffer and passes it to `xnn_create_subgraph` (figure 37.1). The `num_externs` field is defined in the XNNPACK schema as a 32-bit unsigned integer without bounds. The XNNPACK library uses this value to allocate an array of `struct xnn_value` structures (figure 37.2).

```
auto flatbuffer_graph = fb_xnnpack::GetXNNGraph(flatbuffer_data);
// initialize xnnpack
xnn_status status = xnn_initialize(/*allocator =*/nullptr);
ET_CHECK_OR_RETURN_ERROR(
    xnn_status_success == status,
    Internal,
    "XNN Initialize failed with code: %s",
    xnn_status_to_string(status));

// create xnnpack subgraph
xnn_subgraph_t subgraph_ptr = nullptr;
status = xnn_create_subgraph(
    /*external_value_ids=*/flatbuffer_graph->num_externs(),
    /*flags=*/0,
    &subgraph_ptr);
```

Figure 38.1: Unvalidated num_externs passed to xnn_create_subgraph (backends/xnnpack/runtime/XNNCompiler.cpp#L1846–L1860)

```
enum xnn_status xnn_create_subgraph(uint32_t external_value_ids, uint32_t flags,
                                    xnn_subgraph_t* subgraph_out) {
  ...
  subgraph->values =
      xnn_allocate_zero_memory(external_value_ids * sizeof(struct xnn_value));
  ...
}
```

Figure 37.2: XNNPACK allocation based on `external_value_ids` parameter (`backends/xnnpack/third-party/XNNPACK/src/subgraph.c`)

Exploit Scenario

An attacker crafts a malicious PTE file targeting a system running ExecuTorch with XNNPACK backend support. The attacker modifies the XNNPACK FlatBuffer embedded in the PTE file, setting the `num_externs` field to a large value. When a victim loads this model file, ExecuTorch passes this value directly to `xnn_create_subgraph`, which attempts to allocate gigabytes of memory crashing the application.

Recommendations

Short term, add validation of the `num_externs` field before passing it to `xnn_create_subgraph`.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

38. NULL pointer dereference in BackendDelegate::Init due to missing compile_specs validation

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-38
Target: `runtime/executor/method.cpp`	

Description

The `BackendDelegate::Init` method in the ExecuTorch runtime contains a NULL pointer dereference vulnerability when processing malformed PTE files. The FlatBuffer schema defines the `compile_specs` field as an optional vector, meaning it can be NULL. However, the code unconditionally dereferences this pointer without validation, leading to a NULL pointer dereference and subsequent crash (figure 38.1).

```
CompileSpec* compile_specs;
Error err = PopulateCompileSpecs(
    delegate.compile_specs(), backend_init_context, &compile_specs);
if (err != Error::Ok) {
  ET_LOG(Error, "Failed to get compile specs for backend %s", backend_id);
  return err;
}
size_t num_compile_specs = delegate.compile_specs()->size();
```

Figure 38.1: `compile_specs` NULL pointer dereferences (`runtime/executor/method.cpp#L89–L96`)

Exploit Scenario

An attacker crafts a malicious PTE file containing a BackendDelegate entry with a NULL `compile_specs` field. When a victim application attempts to load this PTE file, a NULL pointer dereference is triggered. The application crashes with a segmentation fault, resulting in denial of service.

Recommendations

Short term, add NULL pointer validation before dereferencing `compile_specs`.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps

discover similar vulnerabilities across all backend implementations before they reach production.

39. Heap buffer overflow in XNNPACK backend due to inconsistent tensor dimension validation

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-39
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`, `backends/xnnpack/serialization/runtime_schema.fbs`	

Description

The XNNPACK backend does not validate that the `num_dims` field of an `XNNTensorValue` matches the actual size of the `dims` array when loading a serialized model. This inconsistency allows an attacker to craft a malicious FlatBuffer that causes a heap buffer overflow when the runtime attempts to read tensor dimension data.

The `XNNTensorValue` table in the FlatBuffer schema contains two independent fields that describe tensor dimensions: a scalar `num_dims` field and a vector `dims` field. The schema does not enforce any relationship between these two fields (figure 39.1). When the `defineTensor` function processes a tensor value, it converts the `dims` FlatBuffer vector into a C++ vector using `flatbufferDimsToVector`, which correctly allocates space based on the actual size of the `dims` array. However, when calling `xnn_define_tensor_value`, the code passes the unchecked `num_dims` value along with a pointer to the `dims_data` vector (figure 39.2).

The same vulnerability pattern exists in other functions that use `flatbufferDimsToVector`, including `defineStaticTransposeNode`, `defineStaticConstantPad`, `defineStaticReshape`, and `defineStaticSliceNode`.

```
table XNNTensorValue {
  // number of dimensions in the shape.
  num_dims:uint;
  // pointer to an array of @a num_dims shape dimensions. If num_dims is 0, this
pointer can be NULL.
  // XNNPACK does not keep any pointers to this array after the function returns.
  dims:[uint];
  ...
}
```

Figure 39.1: XNNTensorValue schema showing independent num_dims and dims fields (`backends/xnnpack/serialization/runtime_schema.fbs`)

```
// Get tensor dims, here we need to use a vector in order
// to properly convert the uint32_t* to size_t*
std::vector<size_t> dims_data = flatbufferDimsToVector(tensor_value->dims());

// ... later ...

status = xnn_define_tensor_value(
    /*subgraph=*/subgraph_ptr,
    /*datatype=*/getDataType(tensor_value->datatype()),
    /*num_dims=*/tensor_value->num_dims(),
    /*dims=*/dims_data.data(),
    /*data=*/buffer_ptr,
    /*external_id=*/tensor_value->external_id(),
    /*flags=*/tensor_value->flags(),
    /*id_out=*/&id);
```

Figure 39.2: Vulnerable call passing mismatched num_dims and dims pointer (`backends/xnnpack/runtime/XNNCompiler.cpp`)

Exploit Scenario

An attacker crafts a malicious ExecuTorch model file with an XNNPACK delegated subgraph where the `num_dims` field is set significantly larger than the actual size of the `dims` array. When the victim application loads this model, the XNNPACK backend reads beyond the allocated heap buffer, potentially exposing sensitive information from adjacent heap allocations or causing a crash.

Recommendations

Short term, add validation in the `defineTensor` function to verify that `tensor_value->num_dims()` matches `tensor_value->dims()->size()` before passing these values to XNNPACK. If the values do not match, return an error rather than proceeding with the inconsistent data. Apply the same validation to all other functions that use `flatbufferDimsToVector` with a separate `num_dims` parameter.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

40. Infinite loop in XNNPACK subgraph optimization

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-EXECUTORCH-40
Target: `backends/xnnpack/third-party/XNNPACK/src/subgraph.c`, `backends/xnnpack/runtime/XNNCompiler.cpp`	

Description

The XNNPACK backend contains an infinite loop vulnerability in the subgraph optimization code that allows an attacker to cause a denial of service by providing a malformed PTE file. The `xnn_subgraph_clean_up` function attempts to topologically sort computation nodes but fails to detect when no progress can be made, resulting in an infinite loop when the subgraph contains circular dependencies or unreachable nodes.

During model compilation, the XNNPACK backend deserializes delegate-specific data from the PTE file and constructs a computation subgraph. The `xnn_subgraph_clean_up` function is responsible for removing unreferenced values and sorting nodes in topological order. The function uses a while loop to iteratively schedule nodes whose input dependencies are satisfied (figure 40.1).

```
uint32_t left = 0;
uint32_t num_invalid_nodes = 0;
bool changes = false;

while (left + num_invalid_nodes < subgraph->num_nodes) {
  for (uint32_t i = left; i < subgraph->num_nodes; i++) {
    struct xnn_node* node = &subgraph->nodes[i];

    if (node->type == xnn_node_type_invalid) {
      num_invalid_nodes++;
      continue;
    }

    bool all_values_avail = true;
    for (uint32_t j = 0; all_values_avail && j < node->num_inputs; j++) {
      all_values_avail = values_ready[node->inputs[j]];
    }

    if (all_values_avail) {
      nodes_map[node->id] = left;
      node->id = left;
      for (uint32_t j = 0; j < node->num_outputs; j++) {
        values_ready[node->outputs[j]] = true;
```

```
      }
      // ... reorder nodes ...
      left++;
    }
  }
}
```

Figure 40.1: Vulnerable topological sort loop in `xnn_subgraph_clean_up` (`google/XNNPACK/src/subgraph.c#L2161–L2201`)

The loop terminates when `left + num_invalid_nodes` reaches `subgraph->num_nodes`, which assumes that each iteration either schedules a valid node (incrementing `left`) or encounters an invalid node (incrementing `num_invalid_nodes`). However, this assumption breaks when nodes exist whose input dependencies never become satisfied.

A node can only be scheduled when all its input values are marked as ready. Values are marked ready if they have no producer, are external inputs, or are produced by an already-scheduled node. If a node's inputs reference values that have no valid producer and are not external inputs, or if circular dependencies exist between nodes, the inner for loop completes without scheduling any nodes. This leaves both `left` and `num_invalid_nodes` unchanged, causing the while loop to repeat indefinitely.

The code does not validate the structure of deserialized subgraphs before attempting optimization. It assumes subgraphs are well-formed directed acyclic graphs (DAGs) as would be created by the XNNPACK API, but malformed PTE files can introduce invalid dependencies. There is no progress detection, cycle detection, or iteration limit to prevent infinite looping.

Common graph structures that trigger the infinite loop include:

- Circular dependencies where Node A depends on Node B's output and Node B depends on Node A's output
- Nodes referencing input values that have no producer and are not marked as external inputs
- Nodes referencing out-of-bounds value IDs that access invalid memory locations

Exploit Scenario

An attacker crafts a malicious PTE file with XNNPACK delegate data containing circular dependencies in the computation subgraph and distributes it as a model checkpoint. When a victim application loads the model, `XNNCompiler::compileModel` deserializes the data and invokes `xnn_subgraph_clean_up` to optimize the subgraph. The topological sort loop cannot schedule any nodes because each node's dependencies reference outputs from other unscheduled nodes, creating an unresolvable cycle. The loop continues indefinitely,

consuming CPU resources and causing the application to hang, resulting in a denial of service.

Recommendations

Short term, add an iteration counter to the while loop in `xnn_subgraph_clean_up` that aborts after `subgraph->num_nodes + 1` iterations. If the counter exceeds this limit, log an error indicating a circular dependency and return early after releasing allocated memory. This prevents the issue because it bounds the loop iterations and ensures termination even when encountering malformed subgraphs.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

41. Out-of-bounds vector access in XNNPACK getConstantDataPtr

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-41
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`	

Description

The `getConstantDataPtr` function in the XNNPACK backend accesses FlatBuffers vectors without performing bounds validation on the provided index parameter. This allows an attacker to trigger an out-of-bounds read by supplying a malformed PTE file with an invalid `buffer_idx` value, resulting in an assertion failure and denial of service.

The vulnerability exists in two code paths within the function. First, when `constant_data_ptr` is null, the function accesses the `constant_buffer` vector without validating that `buffer_idx` is within bounds (figure 41.1).

```
const uint8_t* getConstantDataPtr(
    uint32_t buffer_idx,
    GraphPtr flatbuffer_graph,
    const uint8_t* constant_data_ptr,
    const NamedDataMap* named_data_map,
    std::vector<FreeableBuffer>& freeable_buffers,
    XNNWeightsCache* weights_cache) {
  if (buffer_idx) {
    if (!constant_data_ptr) {
      const auto& constant_buffer = *flatbuffer_graph->constant_buffer();
      return constant_buffer[buffer_idx]->storage()->data();  // No bounds check
    } else {
      ConstantDataOffsetPtr constant_data_offset =
          flatbuffer_graph->constant_data()->Get(buffer_idx);  // No bounds check
      uint64_t offset = constant_data_offset->offset();
      // ...
    }
  }
}
```

Figure 40.1: Vulnerable code path accessing `constant_buffer` and `constant_data` without bounds validation (`backends/xnnpack/runtime/XNNCompiler.cpp#L176-L230`)

When `buffer_idx` is greater than or equal to the size of the respective vector, the FlatBuffers `.Get` method triggers an assertion failure in debug builds:

```
Assertion failed: (i < size()), function Get, file vector.h, line 176
```

In release builds without assertions enabled, this results in undefined behavior and potential memory corruption as the function attempts to access memory beyond the vector bounds (figure 41.2).

```
$ ./build/executor_runner --model-path=TOB-EXECUTORCH-41.pte
AddressSanitizer:DEADLYSIGNAL
=================================================================
==42560==ERROR: AddressSanitizer: BUS on unknown address (pc 0x000104c58a14 bp
0x00016b1b5010 sp 0x00016b1b4ec0 T0)
==42560==The signal is caused by a READ memory access.
==42560==Hint: this fault was caused by a dereference of a high value address (see
register values below).  Disassemble the provided pc to learn which register was
used.
    #0 0x000104c58a14 in
executorch::backends::xnnpack::delegate::getConstantDataPtr(unsigned int,
fb_xnnpack::XNNGraph const*, unsigned char const*, executorch::runtime::NamedDataMap
const*, std::__1::vector<executorch::runtime::FreeableBuffer,
std::__1::allocator<executorch::runtime::FreeableBuffer>>&,
executorch::backends::xnnpack::delegate::XNNWeightsCache*)+0x244
(executor_runner:arm64+0x100010a14)
    #1 0x000104c59a28 in
executorch::backends::xnnpack::delegate::defineTensor(xnn_subgraph*,
std::__1::unordered_map<unsigned int, unsigned int, std::__1::hash<unsigned int>,
std::__1::equal_to<unsigned int>, std::__1::allocator<std::__1::pair<unsigned int
const, unsigned int>>>&, fb_xnnpack::XValue const*, fb_xnnpack::XNNGraph const*,
unsigned char const*, std::__1::vector<unsigned int, std::__1::allocator<unsigned
int>>&, std::__1::vector<unsigned int, std::__1::allocator<unsigned int>>&,
executorch::backends::xnnpack::delegate::CompileAllocator&,
executorch::runtime::NamedDataMap const*,
std::__1::vector<executorch::runtime::FreeableBuffer,
std::__1::allocator<executorch::runtime::FreeableBuffer>>&,
executorch::backends::xnnpack::delegate::XNNWeightsCache*)+0x5b0
(executor_runner:arm64+0x100011a28)
    #2 0x000104c7bbd4 in
executorch::backends::xnnpack::delegate::XNNCompiler::compileModel(void const*,
unsigned long, executorch::backends::xnnpack::delegate::XNNExecutor*,
executorch::backends::xnnpack::delegate::XNNWeightsCache*, xnn_workspace*,
executorch::runtime::NamedDataMap const*)+0x5e0 (executor_runner:arm64+0x100033bd4)
    #3 0x000104c84200 in
executorch::backends::XnnpackBackend::init(executorch::runtime::BackendInitContext&,
executorch::runtime::FreeableBuffer*,
executorch::runtime::ArrayRef<executorch::runtime::CompileSpec>) const+0x310
(executor_runner:arm64+0x10003c200)
    #4 0x00010560f3f8 in
executorch::runtime::BackendDelegate::Init(executorch_flatbuffer::BackendDelegate
const&, executorch::runtime::Program const*,
executorch::runtime::BackendInitContext&,
executorch::runtime::BackendDelegate*)+0x800 (executor_runner:arm64+0x1009c73f8)
    #5 0x00010560d1c4 in
executorch::runtime::Method::init(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::NamedDataMap const*)+0x674 (executor_runner:arm64+0x1009c51c4)
```

```
    #6 0x00010560c6d0 in
executorch::runtime::Method::load(executorch_flatbuffer::ExecutionPlan*,
executorch::runtime::Program const*, executorch::runtime::MemoryManager*,
executorch::runtime::EventTracer*, executorch::runtime::NamedDataMap const*)+0x1e8
(executor_runner:arm64+0x1009c46d0)
    #7 0x00010561b6f8 in executorch::runtime::Program::load_method(char const*,
executorch::runtime::MemoryManager*, executorch::runtime::EventTracer*,
executorch::runtime::NamedDataMap const*) const+0x120
(executor_runner:arm64+0x1009d36f8)
    #8 0x000104cc1228 in main+0x1b58 (executor_runner:arm64+0x100079228)
    #9 0x0001857ddd50  (<unknown module>)
```

Figure 40.2: Out-of-bounds access in Release build; FlatBuffer asserts are not compiled in.

Exploit Scenario

An attacker crafts a malicious PTE file with an XNNPACK delegate containing an XValue tensor whose `constant_buffer_idx` exceeds the size of the `constant_data` vector. When a user loads this model, `XNNCompiler::compileModel` calls `getConstantDataPtr` with the malicious index during tensor definition, triggering an out-of-bounds vector access. In debug builds, this causes an assertion failure and denial of service. In release builds without assertions, the out-of-bounds read can leak sensitive memory contents or cause memory corruption that could potentially be exploited for arbitrary code execution.

Recommendations

Short term, add explicit bounds checking before accessing both the `constant_buffer` and `constant_data` FlatBuffers vectors in the `getConstantDataPtr` function. The function should validate that `buffer_idx` is less than the vector size and return a null pointer with an error log if the index is out of bounds. This prevents the out-of-bounds access by rejecting invalid indices before they reach the FlatBuffers vector accessor methods.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps discover similar vulnerabilities across all backend implementations before they reach production.

42. Unbounded memory allocation in XNNPACK backend due to missing dimension validation

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-EXECUTORCH-42
Target: `backends/xnnpack/runtime/XNNCompiler.cpp`	

Description

The XNNPACK backend does not validate tensor dimension values when deserializing models from FlatBuffer data, allowing extremely large dimension values to trigger unbounded memory allocations that cause denial of service. XNNPACK multiplies these dimensions together to calculate memory requirements, resulting in allocation requests that exceed system limits.

The vulnerability exists in the `flatbufferDimsToVector` function in `XNNCompiler.cpp`, which converts dimension values from the FlatBuffer schema to `size_t` without any bounds checking (figure 41.1). These unvalidated dimensions are then passed directly to XNNPACK functions such as `xnn_define_tensor_value` (figure 42.2).

```
template <typename T = size_t>
std::vector<T> flatbufferDimsToVector(
    const flatbuffers::Vector<uint32_t>* fb_dims) {
  std::vector<T> dims_data;

  if (fb_dims == nullptr) {
    return dims_data;
  }

  dims_data.reserve(fb_dims->size());
  for (auto fb_dim : *fb_dims) {
    dims_data.push_back(static_cast<T>(fb_dim));  // No validation
  }
  return dims_data;
}
```

Figure 42.1: Unvalidated dimension conversion in `flatbufferDimsToVector` (`backends/xnnpack/runtime/XNNCompiler.cpp`)

```
std::vector<size_t> dims_data = flatbufferDimsToVector(tensor_value->dims());
// ...
status = xnn_define_tensor_value(
    /*subgraph=*/subgraph_ptr,
    /*datatype=*/getDataType(tensor_value->datatype()),
```

```
    /*num_dims=*/tensor_value->num_dims(),
    /*dims=*/dims_data.data(),  // Unvalidated dimensions passed to XNNPACK
```

Figure 42.2: Unvalidated dimensions passed to XNNPACK subgraph definition (backends/xnnpack/runtime/XNNCompiler.cpp)

```
$ ./build/executor_runner --model-path=TOB-EXECUTORCH-42.pte
==53074==ERROR: AddressSanitizer: requested allocation size 0x9a70d2ddc98e10
(0x9a70d2ddc99e20 after adjustments for alignment, red zones etc.) exceeds maximum
supported size of 0x10000000000 (thread T0)
    #0 0x0001045fbaf0 in posix_memalign+0x80
(libclang_rt.asan_osx_dynamic.dylib:arm64+0x5baf0)
    #1 0x000102df6698 in xnn_aligned_allocate+0x24
(executor_runner:arm64+0x1000ba698)
    #2 0x000102e540e0 in xnn_plan_memory+0x2c8 (executor_runner:arm64+0x1001180e0)
    #3 0x000102d75adc in
executorch::backends::xnnpack::delegate::XNNExecutor::prepare_args(executorch::runti
me::Span<executorch::runtime::EValue*>)+0x494 (executor_runner:arm64+0x100039adc)
    #4 0x000102d78aa0 in
executorch::backends::XnnpackBackend::execute(executorch::runtime::BackendExecutionC
ontext&, void*, executorch::runtime::Span<executorch::runtime::EValue*>) const+0x188
(executor_runner:arm64+0x10003caa0)
    #5 0x000103706be8 in executorch::runtime::Method::execute_instruction()+0x1164
(executor_runner:arm64+0x1009cabe8)
    #6 0x000103707edc in executorch::runtime::Method::execute()+0x440
(executor_runner:arm64+0x1009cbedc)
    #7 0x000102db5408 in main+0x1d38 (executor_runner:arm64+0x100079408)
```

Figure 42.3: AddressSanitizer crash showing unbounded allocation request

Exploit Scenario

An attacker distributes a malicious PTE model file with tensor dimensions set to near-maximum 32-bit values in the XNNPACK backend delegate data. When a victim loads and executes this model, XNNPACK multiplies these dimensions together to calculate memory requirements, producing allocation requests in the petabyte range. The system exhausts all available memory and crashes, causing a denial of service.

Recommendations

Short term, add validation to `flatbufferDimsToVector` to reject large dimension values. Add similar validation in `XNNExecutor::prepare_args` to check input tensor dimensions before passing them to XNNPACK. This prevents unbounded allocations by rejecting malformed model data at deserialization boundaries.

Long term, implement continuous fuzzing for PTE file parsing and execution with the XNNPACK backend enabled. Use standard fuzzing tools like libFuzzer or AFL++ to generate malformed PTE files that exercise backend delegate parsing code paths. Integrate these fuzzers into the CI/CD pipeline with AddressSanitizer and UndefinedBehaviorSanitizer enabled to automatically detect memory safety issues. This proactive approach helps

discover similar vulnerabilities across all backend implementations before they reach production.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Not Applicable	This issue is of informational severity and does not pose an immediate risk, so difficulty does not apply.
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Quality Issues

This appendix contains findings that do not have immediate or obvious security implications. However, addressing them may enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

- The schema/program.fbs definition contains a typo: "heirarchical".
- The kernel's `README.md` mentions the `executorch/kernels/fb/README.md` file, which does not exist.

C. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Though static analysis tools frequently report false positives, there are certain categories of issues that they detect with essentially perfect precision, such as undefined behavior, integer truncations and overflows, unchecked return values, and use of unsafe APIs. We recommend that the ExecuTorch team integrate these tools into their CI/CD pipeline and regularly review their findings.

C.1 Clang

We built the default build target using Clang with the following warnings enabled:

Category	Flags
General	`-Wall`, `-Wextra`, `-Wpedantic`
Type conversions	`-Wconversion`, `-Wsign-conversion`, `-Wsign-compare`, `-Wfloat-conversion`, `-Wdouble-promotion`, `-Wshorten-64-to-32`, `-Wimplicit-int-conversion`
Memory/bounds	`-Warray-bounds`, `-Wvla`, `-Wnull-dereference`, `-Wcast-align`, `-Wuninitialized`, `-Wconditional-uninitialized`
Format strings	`-Wformat=2`, `-Wformat-security`
Aliasing	`-Wstrict-aliasing=2`, `-Wcast-qual`
C++ specific	`-Wold-style-cast`, `-Wzero-as-null-pointer-constant`, `-Wnon-virtual-dtor`, `-Woverloaded-virtual`
Control flow	`-Wimplicit-fallthrough`, `-Wmisleading-indentation`, `-Wloop-analysis`
Other	`-Wshadow`, `-Wcomma`

The build included the core runtime and portable kernels (the `kernel`, `runtime`, `extension`, `examples`, and `schema` directories), but not backend-specific implementations. This produced a large number of warnings, particularly warnings related to implicit integer conversions that may lead to memory corruption or correctness issues (TOB-EXECUTORCH-29). These should be investigated and remedied by the team.

C.2. CodeQL

We used CodeQL to detect known vulnerabilities present in the codebase. This analysis did not identify any issues in the implementation. To build the CodeQL database, we ran the command `codeql database create` in the repository root directory.

```
codeql database create codeql.db \
  --language=cpp \
  --source-root=. \
  --overwrite \
  --command='sh -c "cmake -S . -B cmake-out \
    -DCMAKE_BUILD_TYPE=Debug \
    -DEXECUTORCH_ENABLE_LOGGING=ON \
    -DEXECUTORCH_ENABLE_PROGRAM_VERIFICATION=ON \
    -DEXECUTORCH_BUILD_XNNPACK=ON \
    -DEXECUTORCH_BUILD_KERNELS_OPTIMIZED=ON \
    -DEXECUTORCH_BUILD_KERNELS_QUANTIZED=ON \
    -DEXECUTORCH_BUILD_DEVTOOLS=ON \
    -DEXECUTORCH_BUILD_EXTENSION_DATA_LOADER=ON \
    -DEXECUTORCH_BUILD_EXTENSION_FLAT_TENSOR=ON \
    -DEXECUTORCH_BUILD_EXTENSION_MODULE=ON \
    -DEXECUTORCH_BUILD_EXTENSION_TENSOR=ON \
    -DEXECUTORCH_BUILD_EXTENSION_RUNNER_UTIL=ON \
    -DEXECUTORCH_BUILD_EXTENSION_EVALUE_UTIL=ON \
    -DEXECUTORCH_BUILD_EXTENSION_NAMED_DATA_MAP=ON \
    -DEXECUTORCH_BUILD_PTHREADPOOL=ON \
    -DEXECUTORCH_BUILD_CPUINFO=ON \
    && cmake --build cmake-out -j16"'
```

Figure C.1: The command used to build a CodeQL database for the ExecuTorch library

We ran the open source query suite `cpp-security-extended.qls`, as well as a number of Trail of Bits' internal query suites, on the library. This analysis identified one issue related to pointer arithmetic (TOB-EXECUTORCH-32).

To run the query suite `cpp-security-extended.qls` on an existing database, use the following command.

```
codeql database analyze
       --format=sarif-latest
       --output=cpp-security-extended.sarif
       -- codeql.db cpp-security-extended.qls
```

Figure C.2: The command used to run the query suite `cpp-security-extended.qls` on the codebase

C.3. Cppcheck

To install Cppcheck, we followed the instructions on the official website. We used the following command to run Cppcheck on the codebase. This analysis identified a number of potential memory-safety issues in third-party dependencies, but did not result in any security-relevant findings for the ExecuTorch library itself.

```
cppcheck --output-file=cppcheck.sarif --output-format=sarif .
```

Figure C.3: The command used to run Cppcheck on the ExecuTorch library

C.4. Semgrep

We ran the static analyzer Semgrep with the rule sets shown in figure C.4, as well as Trail of Bits' internal rule sets, to identify low-complexity weaknesses in the source code. These runs identified the unsafe serialization issue TOB-EXECUTORCH-11 (which we had already reported to the ExecuTorch team at this point), but did not identify any issues or code quality findings.

```
semgrep scan –pro --config auto
semgrep scan --metrics=off --config p/trailofbits
semgrep scan --metrics=off --config path/to/internal/rulesets
```

Figure C.4: These Semgrep rule sets did not identify any issues in the codebase

In addition to this, we also wrote a number of custom Semgrep rules during the engagement to identify the following types of issues:

- Integer overflows inside `ET_CHECK_OR_RETURN_ERROR` bounds checks
- Integer overflows when multiplying a count with the output from a call to `elementSize`
- Infinite loops due to vacuously true loop conditions.
- Incorrectly using aggregate initialization to initialize pointers
- Integer overflows in calls to `std::accumulate` and similar functions

Running these custom Semgrep rules against the codebase identified a number of issues related to integer overflows in bounds checks using `ET_CHECK_OR_RETURN_ERROR` (TOB-EXECUTORCH-17), size calculations using `elementSize` (TOB-EXECUTORCH-19), and size calculations using `std::accumulate` and similar functions (TOB-EXECUTORCH-27).

Use the following command to run any of the custom Semgrep rules against the codebase.

```
semgrep scan --metrics=off --config path/to/custom/rule.yaml .
```

Figure C.5: Command to run custom Semgrep rules against the library

D. CodeQL Queries

The following CodeQL query can be used to identify integer overflow vulnerabilities such as those reported in TOB-EXECUTORCH-3

IntegerOverflowInBoundsCheck.ql

The `IntegerOverflowInBoundsCheck.ql` query identifies instances of integer addition inside of a bounds check that could overflow.

```
/**
 * @name Integer overflow in bounds check
 * @description Arithmetic operations in bounds checks can overflow,
 *              causing the check to pass when it should fail.
 * @kind path-problem
 * @problem.severity error
 * @security-severity 8.5
 * @precision high
 * @id cpp/integer-overflow-bounds-check
 * @tags security
 *       external/cwe/cwe-190
 *       external/cwe/cwe-125
 *       external/cwe/cwe-787
 */

import cpp
import semmle.code.cpp.dataflow.new.DataFlow
import semmle.code.cpp.dataflow.new.TaintTracking
import semmle.code.cpp.controlflow.Guards

/**
 * Holds if the expression is a small compile-time constant unlikely to cause
overflow.
 * Constants <= 4096 are considered small (covers common buffer sizes, page sizes,
etc.)
 */
predicate isSmallConstant(Expr e) {
  exists(int val |
    val = e.getValue().toInt() and
    val >= 0 and
    val <= 4096
  )
}

/**
 * Holds if the type is a "wide" integer type (64-bit or larger),
 * making overflow much less likely in practice.
 */
predicate isWideIntegerType(Type t) {
  t.getUnspecifiedType().getSize() >= 8
}
```

```
/**
 * Holds if the expression involves a widening cast that would prevent overflow.
 * E.g., (uint64_t)a + (uint64_t)b when a and b are 32-bit.
 */
predicate hasWideningProtection(BinaryOperation op) {
  exists(Expr operand, Cast cast |
    operand = op.getAnOperand() and
    cast = operand and
    isWideIntegerType(cast.getType()) and
    not isWideIntegerType(cast.getExpr().getType().getUnspecifiedType())
  )
}

/**
 * Identifies calls to safe arithmetic built-ins that check for overflow.
 */
class SafeArithmeticCall extends FunctionCall {
  SafeArithmeticCall() {
    this.getTarget().getName() in [
      "__builtin_add_overflow",
      "__builtin_sub_overflow",
      "__builtin_mul_overflow",
      "__builtin_sadd_overflow",
      "__builtin_uadd_overflow",
      "__builtin_ssub_overflow",
      "__builtin_usub_overflow",
      "__builtin_smul_overflow",
      "__builtin_umul_overflow",
      // GCC/Clang overflow checking
      "__builtin_add_overflow_p",
      "__builtin_sub_overflow_p",
      "__builtin_mul_overflow_p"
    ]
  }
}

/**
 * Identifies arithmetic expressions (add, sub, mul) used in comparison/bounds
checks.
 */
class BoundsCheckArithmetic extends BinaryOperation {
  ComparisonOperation comparison;

  BoundsCheckArithmetic() {
    // Must be an arithmetic operation that can overflow
    (this instanceof AddExpr or this instanceof SubExpr or this instanceof MulExpr)
and
    // The arithmetic is used directly in a comparison (not deeply nested)
    exists(ComparisonOperation cmp |
      comparison = cmp and
      (
        cmp.getAnOperand() = this or
```

```
        // Allow one level of nesting (e.g., cast)
        cmp.getAnOperand().(Cast).getExpr() = this
      )
    ) and
    // Exclude operations with widening protection
    not hasWideningProtection(this) and
    // Exclude operations on already-wide types (64-bit)
    not isWideIntegerType(this.getType())
  }

  ComparisonOperation getComparison() { result = comparison }

  /** Gets the "other operand" - the one that's not tainted */
  Expr getOtherOperand(Expr taintedOperand) {
    result = this.getAnOperand() and
    taintedOperand = this.getAnOperand() and
    result != taintedOperand
  }
}

/**
 * Configuration for tracking attacker-controllable values to bounds check
arithmetic.
 */
module OverflowConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    // Source 1: Function parameters of any integer type (signed or unsigned)
    exists(Parameter p |
      source.asParameter() = p and
      p.getType().getUnspecifiedType() instanceof IntegralType
    )
    or
    // Source 2: Struct/class field accesses (common for parsed data)
    exists(FieldAccess fa |
      source.asExpr() = fa and
      fa.getType().getUnspecifiedType() instanceof IntegralType and
      // Focus on fields that are likely attacker-controlled
      not fa.getTarget().getName().matches("%count%") and
      not fa.getTarget().getName().matches("%size%") and
      not fa.getTarget().getName().matches("%len%")
    )
    or
    // Source 3: Array element accesses (reading from buffers)
    exists(ArrayExpr ae |
      source.asExpr() = ae and
      ae.getType().getUnspecifiedType() instanceof IntegralType
    )
  }

  predicate isSink(DataFlow::Node sink) {
    // Sink: operand of arithmetic in a bounds check
    exists(BoundsCheckArithmetic arith |
      sink.asExpr() = arith.getAnOperand()
```

```
    )
  }

  predicate isBarrier(DataFlow::Node node) {
    // Barrier 1: Value passes through safe arithmetic built-in
    exists(SafeArithmeticCall call |
      node.asExpr() = call.getAnArgument()
    )
    or
    // Barrier 2: Value is masked/clamped to safe range
    exists(BitwiseAndExpr mask |
      node.asExpr() = mask.getAnOperand() and
      // Mask limits the value to less than half the type's range
      exists(int maskVal |
        maskVal = mask.getAnOperand().getValue().toInt() and
        maskVal >= 0 and
        maskVal <= 65535
      )
    )
    or
    // Barrier 3: Value is bounded by std::min or similar
    exists(FunctionCall minCall |
      minCall.getTarget().getName() in ["min", "std::min", "MIN", "fmin"] and
      node.asExpr() = minCall
    )
  }
}

module OverflowFlow = TaintTracking::Global<OverflowConfig>;
import OverflowFlow::PathGraph

from
  OverflowFlow::PathNode source, OverflowFlow::PathNode sink,
  BoundsCheckArithmetic arith, Expr otherOperand
where
  OverflowFlow::flowPath(source, sink) and
  sink.getNode().asExpr() = arith.getAnOperand() and
  otherOperand = arith.getOtherOperand(sink.getNode().asExpr()) and
  // Precision: Exclude cases where the other operand is a small constant
  // (overflow unlikely when adding/multiplying by small values)
  not isSmallConstant(otherOperand)
select sink.getNode(), source, sink,
  "Potential integer overflow: $@ flows to " + arith.getOperator() +
    " operation in bounds check without overflow protection.",
  source.getNode(), source.getNode().toString()
```

Figure C.1: `IntegerOverflowInBoundsCheck.ql` for finding integer overflows

E. Fuzzing

The main fuzz harness we used to uncover a number of findings is shown in figure E.1. It is essentially a simplified version of the `executor_runner` example. We leveraged an LLM to generate the initial fuzz harness as well as the build modifications (figure E.2) and fuzzing dictionary (figure E.3). We also include commands to build and run the fuzzer (figures E.4, E.5).

The fuzz harness code is prototypical and underwent a number of tweaks and modifications during the engagement. It is not meant to be treated as an example of a solid use of ExecuTorch as a library but rather a means to find bugs. We double checked and debugged the root cause for each crash on a clean ExecuTorch build with AddressSantizer enabled to ensure that the findings are reproducible.

We ran the fuzzer mostly with AddressSanitizer and only occasionally with UndefinedBehaviorSanitizer. Thus, we advise the ExecuTorch team to extensively run sanitizers other than AddressSanitizer to uncover more issues.

```
#include <executorch/extension/data_loader/buffer_data_loader.h>
#include <executorch/extension/runner_util/inputs.h>
#include <executorch/runtime/backend/interface.h>
#include <executorch/runtime/core/hierarchical_allocator.h>
#include <executorch/runtime/executor/method.h>
#include <executorch/runtime/executor/program.h>
#include <executorch/runtime/platform/runtime.h>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <memory>
#include <vector>

using executorch::extension::BufferDataLoader;
using executorch::extension::PrepareInputTensorsOptions;
using executorch::runtime::Error;
using executorch::runtime::HierarchicalAllocator;
using executorch::runtime::MemoryAllocator;
using executorch::runtime::MemoryManager;
using executorch::runtime::Method;
using executorch::runtime::MethodMeta;
using executorch::runtime::Program;
using executorch::runtime::Result;
using executorch::runtime::Span;

__attribute__((constructor)) static void InitializeFuzzer() {
  executorch::runtime::runtime_init();
}

static uint8_t method_allocator_pool[4 * 1024U * 1024U]; // 4 MB
static uint8_t temp_allocator_pool[1024U * 1024U];
```

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  auto header_status = Program::check_header(data, size);
  (void)header_status; // Ignore result, just exercise the code path

  BufferDataLoader loader(data, size);

  Result<Program> program = Program::load(
      &loader,
      // Program::Verification::Minimal
      Program::Verification::InternalConsistency
  );

  if (program.ok()) {
    size_t num_methods = program->num_methods();
    if (num_methods > 1) return 0;

    for (size_t i = 0; i < num_methods && i < 10; i++) {
      auto method_name_result = program->get_method_name(i);

      if (method_name_result.ok()) {
        const char* method_name = method_name_result.get();

        // MethodMeta describes the memory requirements of the method.
        Result<MethodMeta> method_meta = program->method_meta(method_name);
        if (!method_meta.ok()) return 1;

        // In this example we use a statically allocated memory pool.
        MemoryAllocator method_allocator{
            MemoryAllocator(sizeof(method_allocator_pool), method_allocator_pool)};

        // Temporary memory required by kernels
        MemoryAllocator temp_allocator{
            MemoryAllocator(sizeof(temp_allocator_pool), temp_allocator_pool)};

        std::vector<std::unique_ptr<uint8_t[]>> planned_buffers; // Owns the memory
        std::vector<Span<uint8_t>> planned_spans; // Passed to the allocator

        size_t num_memory_planned_buffers =
            method_meta->num_memory_planned_buffers();
        for (size_t id = 0; id < num_memory_planned_buffers; ++id) {
          // .get() will always succeed because id < num_memory_planned_buffers.
          auto buffer_size_result = method_meta->memory_planned_buffer_size(id);
          if (!buffer_size_result.ok()) {
            return 1;
          }
          size_t buffer_size = buffer_size_result.get();
          ET_LOG(Info, "Setting up planned buffer %zu, size %zu.", id, buffer_size);
          planned_buffers.push_back(std::make_unique<uint8_t[]>(buffer_size));
          planned_spans.push_back({planned_buffers.back().get(), buffer_size});
        }
        HierarchicalAllocator planned_memory({planned_spans.data(),
            planned_spans.size()});
```

```
        // Assemble all of the allocators into the MemoryManager that the Executor
        // will use.
        MemoryManager memory_manager(
            &method_allocator, &planned_memory, &temp_allocator);

        Result<Method> method_result = program->load_method(method_name,
            &memory_manager);

        if (method_result.ok()) {
          Method& method = method_result.get();

          // Limit tensor memory allocation to 100MB to prevent excessive memory
          // usage during fuzzing of malicious PTE files
          PrepareInputTensorsOptions input_options;
          input_options.max_total_allocation_size = 100 * 1024 * 1024; // 100 MB
          input_options.max_inputs = 1024;

          auto res = executorch::extension::prepare_input_tensors(method,
              input_options);
          if (res.ok()) {

            Error exec_error = method.execute();

            // If execution succeeded, try to get outputs for coverage
            if (exec_error == Error::Ok) {
              size_t num_outputs = method.outputs_size();
              for (size_t out_idx = 0;
                  out_idx < num_outputs && out_idx < 5;
                  out_idx++) {
                // auto output_result = method.get_output(out_idx); // FIXME
                // (void)output_result;
              }
            } else { return 1; }
          }

        }
      }
    }
  }

  return 0;
}
```

Figure E.1: The main fuzzing harness exercising program execution (`fuzz/targets/pte_parser/pte_parser_fuzzer.cpp`)

```
cmake_minimum_required(VERSION 3.19)

set(EXECUTORCH_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/../../..)

# Create the PTE parser fuzzer executable
add_executable(pte_parser_fuzzer pte_parser_fuzzer.cpp)
```

```
# Link required libraries
target_link_libraries(
  pte_parser_fuzzer
  executorch_core
  extension_data_loader
  extension_runner_util
)

# Link XNNPACK backend if available (to exercise backend delegation)
if(TARGET xnnpack_backend)
  message(STATUS "Linking XNNPACK backend to pte_parser_fuzzer")
  target_link_libraries(
    pte_parser_fuzzer
    xnnpack_backend
  )
else()
  message(STATUS "XNNPACK backend not available - fuzzer will not exercise
backends")
endif()

target_compile_options(
  pte_parser_fuzzer
  PRIVATE
  -fsanitize=fuzzer,address
  -g
  -O1
)

# Fuzzer-specific link options
# Must match the sanitizer flags used in compilation
target_link_options(
  pte_parser_fuzzer
  PRIVATE
  -fsanitize=fuzzer,address
)

# The pte_parser_fuzzer needs access to FlatBuffer-generated headers
# which are created by the program_schema target
if(TARGET program_schema)
  add_dependencies(pte_parser_fuzzer program_schema)

  # Include paths for generated schema headers
  target_include_directories(
    pte_parser_fuzzer
    PRIVATE
    "${CMAKE_BINARY_DIR}/schema/include"
    "${EXECUTORCH_ROOT}/third-party/flatbuffers/include"
  )
endif()

# Set up the corpus directory
set(CORPUS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/corpus)
```

```
file(MAKE_DIRECTORY ${CORPUS_DIR})

# Install the fuzzer to a dedicated fuzz directory
install(
  TARGETS pte_parser_fuzzer
  DESTINATION bin/fuzz
  COMPONENT fuzzing
)

message(STATUS "PTE parser fuzzer target created")
message(STATUS "Corpus directory: ${CORPUS_DIR}")
message(STATUS "To run: ${CMAKE_CURRENT_BINARY_DIR}/pte_parser_fuzzer
-dict=${CMAKE_CURRENT_BINARY_DIR}/pte_parser_fuzzer.dict ${CORPUS_DIR}/")
```

Figure E.2: fuzz/targets/pte_parser/CMakeLists.txt

```
# Main file format magic (4 bytes)
# "ET" followed by 2 ASCII decimal digits indicating format version
"ET12"
"ET11"
"ET10"
"ET09"
"ET08"

# Extended header magic (4 bytes)
# "eh" followed by 2 ASCII decimal digits
"eh00"
"eh01"

# FlatBuffer file identifier (4 bytes)
# Used in some PTE file variations
"EXIR"

# ------------------------------------------------------------------------------
# Common Integer Values (Little-Endian)
# ------------------------------------------------------------------------------

# Small integers (uint32_t, little-endian)
"\x00\x00\x00\x00"
"\x01\x00\x00\x00"
"\x02\x00\x00\x00"
"\x04\x00\x00\x00"
"\x08\x00\x00\x00"
"\x10\x00\x00\x00"
"\x20\x00\x00\x00"
"\x40\x00\x00\x00"

# Common sizes (uint64_t, little-endian)
"\x00\x00\x00\x00\x00\x00\x00\x00"
"\x01\x00\x00\x00\x00\x00\x00\x00"
"\x10\x00\x00\x00\x00\x00\x00\x00"
"\x00\x10\x00\x00\x00\x00\x00\x00"
```

```
# Extended header size (56 bytes)
"\x38\x00\x00\x00"

# ------------------------------------------------------------------------------
# Alignment and Padding Patterns
# ------------------------------------------------------------------------------

# 4-byte alignment padding (common in FlatBuffers)
"\x00\x00\x00\x00"

# 8-byte alignment padding
"\x00\x00\x00\x00\x00\x00\x00\x00"

# 16-byte alignment padding
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"

# 4096-byte alignment (segment alignment in PTE files)
# Too large to include here, but 0-byte patterns help fuzzer learn about padding

# ------------------------------------------------------------------------------
# FlatBuffer Structure Markers
# ------------------------------------------------------------------------------

# FlatBuffer vtable offsets (uint16_t, little-endian)
"\x04\x00"
"\x08\x00"
"\x0c\x00"
"\x10\x00"
"\x14\x00"

# FlatBuffer vector length markers (uint32_t, little-endian)
"\x01\x00\x00\x00"
"\x02\x00\x00\x00"
"\x03\x00\x00\x00"
"\x0a\x00\x00\x00"
"\x64\x00\x00\x00"

# FlatBuffer string terminators
"\x00"

# FlatBuffer table offsets (int32_t, little-endian)
"\x00\x00\x00\x00"
"\x04\x00\x00\x00"
"\x08\x00\x00\x00"
"\x0c\x00\x00\x00"

# ------------------------------------------------------------------------------
# Common Operator and Method Names
# ------------------------------------------------------------------------------

# These strings commonly appear in PTE files as operator or method names
# libFuzzer will try inserting these at various positions
```

```
# Operator namespace prefixes
"aten::"
"native::"
"quantized::"
"custom::"

# Common operators
"add"
"mul"
"sub"
"div"
"relu"
"conv"
"linear"
"matmul"

# Common method names
"forward"
"inference"
"execute"
"main"

# Tensor dtype strings
"float32"
"float16"
"int32"
"int64"
"uint8"
"int8"

# ------------------------------------------------------------------------------
# Segment Type Identifiers
# ------------------------------------------------------------------------------

# These may appear in segment metadata
"Program"
"Constant"
"Backend"
"Mutable"
"External"

# ------------------------------------------------------------------------------
# Common Data Type Identifiers
# ------------------------------------------------------------------------------

# ScalarType enum values (int8_t)
# Byte/UInt8
"\x00"
# Char/Int8
"\x01"
# Short/Int16
"\x02"
# Int/Int32
```

```
"\x03"
# Long/Int64
"\x04"
# Half/Float16
"\x05"
# Float/Float32
"\x06"
# Double/Float64
"\x07"
# Bool
"\x0b"

# ----------------------------------------------------------------------------
# Version Numbers
# ----------------------------------------------------------------------------

# Common version markers
"\x00\x00"
"\x01\x00"
"\x02\x00"

# ----------------------------------------------------------------------------
# Backend Delegation (XNNPACK)
# ----------------------------------------------------------------------------

# XNNPACK backend identifier (used in PTE delegate metadata)
"XnnpackBackend"

# XNNPACK FlatBuffer file identifier (4 bytes)
# This magic appears at the start of XNNPACK delegate blobs
"XN01"

# XNNPACK namespace (appears in flatbuffer schema)
"fb_xnnpack"

# Common XNNPACK operator names
"xnn_datatype_fp32"
"xnn_datatype_fp16"
"xnn_datatype_qint8"
"xnn_datatype_quint8"

# XNNPACK node types (these may appear in delegate blobs)
"XNNGraph"
"XNNNode"
"XValue"
"XNNTensor"

# Delegate-related keywords
"delegate"
"backend"
"preprocess"
"compile_spec"
```

Figure E.3: The LLM-generated fuzzing dictionary (`fuzz/targets/pte_parser/pte_parser_fuzzer.dict`)

```
cmake -B fuzz/build -GNinja -DCMAKE_BUILD_TYPE=Debug -DEXECUTORCH_BUILD_FUZZING=ON
-DEXECUTORCH_BUILD_XNNPACK=ON -DEXECUTORCH_ENABLE_LOGGING=OFF
-DEXECUTORCH_BUILD_EXTENSION_DATA_LOADER=ON .
```

Figure E.4: The build command

```
build/fuzz/targets/pte_parser/pte_parser_fuzzer targets/pte_parser/corpus
-dict=./targets/pte_parser/pte_parser_fuzzer.dict -timeout=1
```

Figure E.5: Command for running the fuzzer

F. Semgrep Rules

During this review, we wrote a number of custom Semgrep rules to identify issues like integer overflows in bounds checks, integer overflows in size calculations based on the elementSize function, integer underflows leading to infinite loops, aggregate initialization of pointers, and integer overflows in calls to `std::accumulate` and similar functions. These rules are provided below.

F.1. Rules Identifying Integer Overflows in Bounds Checks

```
rules:
  # ------------------------------------------------------------------------
  # Example: offset + nbytes <= size
  # ------------------------------------------------------------------------
  - id: integer-overflow-addition-lhs-le
    message: |
      Potential integer overflow in bounds check. The expression `$A + $B <= $C`
      can overflow before the comparison, causing the check to incorrectly pass.

      VULNERABLE:
        ET_CHECK_OR_RETURN_ERROR($A + $B <= $C, ...)

      SAFE ALTERNATIVES:
        ET_CHECK_OR_RETURN_ERROR($A <= $C && $B <= $C - $A, ...)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK_OR_RETURN_ERROR($A + $B <= $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($A + $B < $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR(($A + $B) <= $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR(($A + $B) < $C, ...)
    metadata:
      category: security
      technology: [cpp]
      confidence: HIGH
      likelihood: MEDIUM
      impact: HIGH
```

Figure F.1: Rule identifying overflows inside ET_CHECK_OR_RETURN_ERROR

```
rules:
  # ------------------------------------------------------------------------
  # Example: size >= offset + nbytes
  # ------------------------------------------------------------------------
  - id: integer-overflow-addition-rhs-ge
    message: |
      Potential integer overflow in bounds check. The expression `$C >= $A + $B`
      can overflow before the comparison, causing the check to incorrectly pass.
```

```
      VULNERABLE:
        ET_CHECK_OR_RETURN_ERROR($C >= $A + $B, ...)

      SAFE ALTERNATIVES:
        ET_CHECK_OR_RETURN_ERROR($A <= $C && $B <= $C - $A, ...)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK_OR_RETURN_ERROR($C >= $A + $B, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C > $A + $B, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C >= ($A + $B), ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C > ($A + $B), ...)
    metadata:
      category: security
      confidence: HIGH
```

Figure F.2: Rule identifying overflows inside ET_CHECK_OR_RETURN_ERROR

```
rules:
  # ------------------------------------------------------------------------
  # Example: count * elem_size <= buffer_size
  # ------------------------------------------------------------------------
  - id: integer-overflow-multiplication-lhs-le
    message: |
      Potential integer overflow in bounds check. The expression `$A * $B <= $C`
      can overflow before the comparison, causing the check to incorrectly pass.

      VULNERABLE:
        ET_CHECK_OR_RETURN_ERROR($A * $B <= $C, ...)

      SAFE ALTERNATIVES:
        ET_CHECK_OR_RETURN_ERROR($B == 0 || $A <= $C / $B, ...)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK_OR_RETURN_ERROR($A * $B <= $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($A * $B < $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR(($A * $B) <= $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR(($A * $B) < $C, ...)
    metadata:
      category: security
      confidence: HIGH
```

Figure F.3: Rule identifying overflows inside ET_CHECK_OR_RETURN_ERROR

```
rules:
  # ------------------------------------------------------------------------
  # Example: buffer_size >= count * elem_size
  # ------------------------------------------------------------------------
  - id: integer-overflow-multiplication-rhs-ge
    message: |
```

```
      Potential integer overflow in bounds check. The expression `$C >= $A * $B`
      can overflow before the comparison, causing the check to incorrectly pass.

      VULNERABLE:
        ET_CHECK_OR_RETURN_ERROR($C >= $A * $B, ...)

      SAFE ALTERNATIVES:
        ET_CHECK_OR_RETURN_ERROR($B == 0 || $C / $B >= $A, ...)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK_OR_RETURN_ERROR($C >= $A * $B, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C > $A * $B, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C >= ($A * $B), ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C > ($A * $B), ...)
    metadata:
      category: security
      confidence: HIGH
```

Figure F.4: Rule identifying overflows inside ET_CHECK_OR_RETURN_ERROR

```
rules:
  # ------------------------------------------------------------------------
  # Example: buffer_size == offset + nbytes
  # ------------------------------------------------------------------------
  - id: integer-overflow-addition-equality
    message: |
      Potential integer overflow in equality check. The expression involving
      `$A + $B` can overflow before the comparison, leading to unexpected behavior.

      VULNERABLE:
        ET_CHECK_OR_RETURN_ERROR($A + $B == $C, ...)

      SAFE ALTERNATIVES:
        ET_CHECK_OR_RETURN_ERROR($A <= $C && $C - $A == $B, ...)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK_OR_RETURN_ERROR($A + $B == $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($A + $B != $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C == $A + $B, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C != $A + $B, ...)
    metadata:
      category: security
      confidence: MEDIUM
```

Figure F.5: Rule identifying overflows inside ET_CHECK_OR_RETURN_ERROR

```
rules:
  # ------------------------------------------------------------------------
  # Example: count * elem_size == buffer_size
```

```
  # ------------------------------------------------------------------------------
  - id: integer-overflow-multiplication-equality
    message: |
      Potential integer overflow in equality check. The expression involving
      `$A * $B` can overflow before the comparison.

      SAFE ALTERNATIVES:
        ET_CHECK_OR_RETURN_ERROR(!__builtin_mul_overflow($A, $B, &product) &&
product == $C, ...)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK_OR_RETURN_ERROR($A * $B == $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($A * $B != $C, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C == $A * $B, ...)
          - pattern: ET_CHECK_OR_RETURN_ERROR($C != $A * $B, ...)
    metadata:
      category: security
      confidence: MEDIUM
```

Figure F.6: Rule identifying overflows inside ET_CHECK_OR_RETURN_ERROR

```
rules:
  # ------------------------------------------------------------------------------
  # Overflows in ET_CHECK and ET_CHECK_MSG
  # ------------------------------------------------------------------------------
  - id: integer-overflow-generic-check-macro-addition
    message: |
      Potential integer overflow in check macro. Arithmetic operations in
      bounds checks can overflow before the comparison is evaluated.

      VULNERABLE:
        ET_CHECK($A + $B <= $C)

      SAFE ALTERNATIVES:
        ET_CHECK($A <= $C && $B <= $C - $A)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK($A + $B <= $C)
          - pattern: ET_CHECK($A + $B < $C)
          - pattern: ET_CHECK($C >= $A + $B)
          - pattern: ET_CHECK($C > $A + $B)
          - pattern: ET_CHECK_MSG($A + $B <= $C, ...)
          - pattern: ET_CHECK_MSG($A + $B < $C, ...)
          - pattern: ET_CHECK_MSG($C >= $A + $B, ...)
          - pattern: ET_CHECK_MSG($C > $A + $B, ...)
    metadata:
      category: security
      confidence: HIGH
```

```yaml
  - id: integer-overflow-generic-check-macro-multiplication
    message: |
      Potential integer overflow in check macro. Multiplication in bounds
      checks can overflow before the comparison is evaluated.

      VULNERABLE:
        ET_CHECK($A * $B <= $C)

      SAFE ALTERNATIVES:
        ET_CHECK($B == 0 || $A <= $C / $B)
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: ET_CHECK($A * $B <= $C)
          - pattern: ET_CHECK($A * $B < $C)
          - pattern: ET_CHECK($C >= $A * $B)
          - pattern: ET_CHECK($C > $A * $B)
          - pattern: ET_CHECK_MSG($A * $B <= $C, ...)
          - pattern: ET_CHECK_MSG($A * $B < $C, ...)
          - pattern: ET_CHECK_MSG($C >= $A * $B, ...)
          - pattern: ET_CHECK_MSG($C > $A * $B, ...)
    metadata:
      category: security
      confidence: HIGH
```

Figure F.7: Rule identifying overflows inside ET_CHECK and ET_CHECK_MSG

```yaml
rules:
  # ------------------------------------------------------------------------
  # Overflows in assert and TORCH_CHECK
  # ------------------------------------------------------------------------
  - id: integer-overflow-assert-addition
    message: |
      Potential integer overflow in assertion. The arithmetic in `$A + $B`
      can overflow before the comparison, making the assertion ineffective.
    severity: WARNING
    languages: [cpp, c]
    patterns:
      - pattern-either:
          - pattern: assert($A + $B <= $C)
          - pattern: assert($A + $B < $C)
          - pattern: assert($C >= $A + $B)
          - pattern: assert($C > $A + $B)
          - pattern: TORCH_CHECK($A + $B <= $C, ...)
          - pattern: TORCH_CHECK($A + $B < $C, ...)
          - pattern: TORCH_CHECK($C >= $A + $B, ...)
          - pattern: TORCH_CHECK($C > $A + $B, ...)
    metadata:
      category: security
      confidence: MEDIUM

  - id: integer-overflow-assert-multiplication
```

```
  message: |
    Potential integer overflow in assertion. The arithmetic in `$A * $B`
    can overflow before the comparison, making the assertion ineffective.
  severity: WARNING
  languages: [cpp, c]
  patterns:
    - pattern-either:
        - pattern: assert($A * $B <= $C)
        - pattern: assert($A * $B < $C)
        - pattern: assert($C >= $A * $B)
        - pattern: assert($C > $A * $B)
        - pattern: TORCH_CHECK($A * $B <= $C, ...)
        - pattern: TORCH_CHECK($A * $B < $C, ...)
        - pattern: TORCH_CHECK($C >= $A * $B, ...)
        - pattern: TORCH_CHECK($C > $A * $B, ...)
  metadata:
    category: security
    confidence: MEDIUM
```

Figure F.8: Rule identifying overflows inside `assert` and TORCH_CHECK

F.2. Rule Identifying Overflows in `elementSize`-Based Size Calculations

```yaml
rules:
  - id: overflow-in-element-size-arithmetic
    languages: [cpp, c]
    severity: ERROR
    message: >-
      Multiplying a count by elementSize() without overflow checking. If the count
      is large, the multiplication wraps to a small value, causing undersized
      allocation and buffer overflow.
    metadata:
      confidence: HIGH
      impact: HIGH
      category: security
    patterns:
      - pattern-either:
          - pattern: $COUNT * $ELEMENT_SIZE($ELEMENT_TYPE)
          - pattern: $ELEMENT_SIZE($ELEMENT_TYPE) * $COUNT
      - metavariable-regex:
          metavariable: $ELEMENT_SIZE
          regex: .*elementSize
```

Figure F.9: Rule identifying overflows in size calculations using `elementSize`

F.3. Rule Identifying Infinite Loops due to Loop Variable Underflow

```
rules:
  - id: unsigned-decrement-loop-gte-zero
    languages: [cpp, c]
    severity: ERROR
    message: >-
      '$VAR' is an unsigned type ($TYPE), so the condition '$VAR >= 0' is always
      true. When '$VAR' is 0 and decremented, it wraps to the maximum value
      instead of becoming negative. Consider using a signed type instead.
    metadata:
      confidence: HIGH
      impact: HIGH
      category: security
    patterns:
      - pattern-either:
          - pattern: |
              for ($TYPE $VAR = $INIT; $VAR >= 0; --$VAR) { ... }
          - pattern: |
              for ($TYPE $VAR = $INIT; $VAR >= 0; $VAR--) { ... }
          - pattern: |
              for ($TYPE $VAR = $INIT; 0 <= $VAR; --$VAR) { ... }
          - pattern: |
              for ($TYPE $VAR = $INIT; 0 <= $VAR; $VAR--) { ... }
      - metavariable-regex:
          metavariable: $TYPE
          regex:
^(size_t|std::size_t|unsigned|unsigned\s+int|unsigned\s+long|unsigned\s+long\s+long|
unsigned\s+short|unsigned\s+char|uint8_t|uint16_t|uint32_t|uint64_t|uintptr_t|uintma
x_t|std::uint8_t|std::uint16_t|std::uint32_t|std::uint64_t)$
```

Figure F.10: Rule identifying infinite loops due to loop variable underflow

F.4. Rule Identifying Aggregate Initialization of Pointers

```yaml
rules:
  - id: pointer-aggregate-initialization
    languages: [c, cpp]
    severity: WARNING
    message: >-
      Suspicious brace/list initializer assigned to a pointer index. This may
      indicate unintended aggregate initialization. Use scalar assignment or
      memset for buffers.
    metadata:
      category: correctness
      confidence: MEDIUM
      impact: MEDIUM
    pattern-either:
      - pattern: $P[$I] = {...};
```

Figure F.11: Rule identifying aggregate initialization of pointers

F.5. Rules Identifying Potential Integer Overflows in Calls to Functions like std::accumulate

```
rules:
  - id: integer-overflow-in-std-accumulate
    languages: [cpp]
    severity: WARNING
    message: >-
      Use of std::accumulate with $OP to aggregate integers without overflow
      Checking. When multiplying or adding many integers, the result can overflow
      the accumulator type ($INIT), leading to undefined behavior (signed) or
      wraparound (unsigned).
    metadata:
      confidence: MEDIUM
      impact: HIGH
    patterns:
      - pattern-either:
          - pattern: std::accumulate($BEGIN, $END, $INIT, std::multiplies<$T>())
          - pattern: std::accumulate($BEGIN, $END, $INIT, std::multiplies<>())
          - pattern: std::accumulate($BEGIN, $END, $INIT, std::plus<$T>())
          - pattern: std::accumulate($BEGIN, $END, $INIT, std::plus<>())
          - pattern: |
              std::accumulate($BEGIN, $END, $INIT, []($A, $B) { return $A * $B; })
          - pattern: |
              std::accumulate($BEGIN, $END, $INIT, []($A, $B) -> $RET { return $A *
$B; })
          - pattern: |
              std::accumulate($BEGIN, $END, $INIT, []($A, $B) { return $A + $B; })
          - pattern: |
              std::accumulate($BEGIN, $END, $INIT, []($A, $B) -> $RET { return $A +
$B; })
          - pattern: std::accumulate($BEGIN, $END, $INIT)
```

Figure F.12: Rule identifying potential integer overflows in calls to `std::accumulate`

```
rules:
  - id: integer-overflow-in-transform-reduce
    languages: [cpp]
    severity: WARNING
    message: >-
      Use of std::transform_reduce with multiplication or addition to aggregate
      integers without overflow checking. The result can overflow the accumulator
      type, leading to undefined behavior (signed) or wraparound (unsigned).
    metadata:
      confidence: MEDIUM
      impact: HIGH
    patterns:
      - pattern-either:
          - pattern: std::transform_reduce($BEGIN, $END, $INIT,
std::multiplies<$T>(), $TRANSFORM)
          - pattern: std::transform_reduce($BEGIN, $END, $INIT, std::multiplies<>(),
$TRANSFORM)
```

```
          - pattern: std::transform_reduce($BEGIN, $END, $INIT, std::plus<$T>(),
$TRANSFORM)
          - pattern: std::transform_reduce($BEGIN, $END, $INIT, std::plus<>(),
$TRANSFORM)
```

Figure F.13: Rule identifying potential integer overflows in calls to `std::transform_reduce`

```
rules:
  - id: integer-overflow-in-std-reduce
    languages: [cpp]
    severity: WARNING
    message: >-
      Use of std::reduce with $OP to aggregate integers without overflow checking.
      When multiplying or adding many integers, the result can overflow the
      Accumulator type, leading to undefined behavior (signed) or wraparound
      (unsigned).
    metadata:
      confidence: MEDIUM
      impact: HIGH
    patterns:
      - pattern-either:
          - pattern: std::reduce($BEGIN, $END, $INIT, std::multiplies<$T>())
          - pattern: std::reduce($BEGIN, $END, $INIT, std::multiplies<>())
          - pattern: std::reduce($BEGIN, $END, $INIT, std::plus<$T>())
          - pattern: std::reduce($BEGIN, $END, $INIT, std::plus<>())
          - pattern: std::reduce($BEGIN, $END, $INIT)
          - pattern: std::reduce($BEGIN, $END)
```

Figure F.14: Rule identifying potential integer overflows in calls to `std::reduce`

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From April 27 to April 29, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the ExecuTorch team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Apart from fixes to the individual issues in the report, the team has also worked towards strengthening the overall security posture of the library. This has involved fixing compiler warnings, expanding fuzz testing to more targets, and developing a new program validation layer to catch invalid or malicious PTE files early.

In summary, of the 42 issues described in this report, the ExecuTorch team has resolved 37 issues, has partially resolved three issues, and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Null pointer dereference in get_output_flattening_encoding	Low	Resolved
2	Null pointer dereference in get_execution_plan	Low	Resolved
3	Integer overflow in tensor nbytes size computation	High	Resolved
4	Integer overflow in Buffer DataLoader bounds check	Low	Resolved
5	Stack buffer overflow in prepare_input_tensors	Low	Resolved
6	Abort on type mismatch in parseListOptionalType	Low	Resolved
7	Abort on type mismatch in Method::execute_instruction	Low	Resolved
8	Abort on type mismatch in parseTensorList	Low	Resolved

9	Out-of-bounds heap read in BPTE tensor deserialization	Low	Resolved
10	Out-of-bounds heap read in WAV audio file parsing	Low	Resolved
11	Unsafe deserialization in ETRecord parsing enables code execution	High	Resolved
12	Out-of-bounds heap read without InternalConsistency verification	Medium	Resolved
13	Out-of-bounds heap write in Program::get_constant_buffer_data	High	Resolved
14	Infinite loop in XNNExecutor::resize_outputs	Medium	Resolved
15	Out-of-bounds write in FreeCall instruction handler	High	Resolved
16	Out-of-bounds read in validateTensorLayout	Medium	Resolved
17	Integer overflow in constant segment offset validation	Medium	Resolved
18	Missing duplicate check in dim_order validation	Medium	Resolved
19	Integer overflow in tensor size validation enables heap buffer overflow	Medium	Resolved
20	Integer overflow in CUDA tensor copy allocation enables heap buffer overflow	Medium	Resolved
21	Hardcoded element size in CUDA tensor copy enables heap buffer overflow	Medium	Resolved
22	Integer overflow in data loader bounds checking	High	Resolved
23	Infinite loop with JF_CALL	Low	Resolved

24	Integer overflow in program file size validation	Medium	Resolved
25	Out-of-bounds write in Cadence HiFi bmm operator	Low	Resolved
26	Integer overflow in PlatformMemoryAllocator allocation	High	Resolved
27	Integer overflows in Vulkan tensor operations	High	Resolved
28	Unbounded memory allocation from untrusted PTE file	Low	Partially Resolved
29	Large number of security-relevant compiler warnings	Undetermined	Partially Resolved
30	Memory corruption due to unsafe mutation of FlatBuffer-backed tensor data	High	Partially Resolved
31	Type confusion in BoxedEvalueList after MoveCall	Medium	Resolved
32	Incorrect pointer offset arithmetic in ETDumpGen constructor	Low	Resolved
33	Out-of-bounds read due to missing FlatBuffers verification in XNNCompiler	Medium	Resolved
34	Out-of-bounds read due to missing XNNHeader verification	Medium	Resolved
35	NULL pointer dereferences in XNNCompiler::compileModel	Low	Resolved
36	Crash due to unchecked map access in XNNCompiler	Low	Resolved
37	Unbounded memory allocation in XNNPACK subgraph creation	Low	Resolved

38	NULL pointer dereference in BackendDelegate::Init due to missing compile_specs validation	Low	Resolved
39	Heap buffer overflow in XNNPACK backend due to inconsistent tensor dimension validation	Medium	Resolved
40	Infinite loop in XNNPACK subgraph optimization	Low	Unresolved
41	Out-of-bounds vector access in XNNPACK getConstantDataPtr	Medium	Resolved
42	Unbounded memory allocation in XNNPACK backend due to missing dimension validation	Low	Unresolved

Detailed Fix Review Results

TOB-EXECUTORCH-1: Null pointer dereference in get_output_flattening_encoding
Resolved in PR #16704. The implementation now checks that both `container_meta_type` and `encoded_out_str` are non-null and return descriptive errors if the checks fail.

TOB-EXECUTORCH-2: Null pointer dereference in get_execution_plan
Resolved in PR #16795. The `get_execution_plan` function now checks that both the execution plan and the name are non-null before dereferencing them.

TOB-EXECUTORCH-3: Integer overflow in tensor nbytes size computation
Resolved in PR #19121. The implementation now calls `validate_program` when a program is loaded. This function validates different parts of the loaded program. In particular, it ensures in `validate_tensor` that the tensor size computation in `TensorImpl::nbytes` will not overflow.

TOB-EXECUTORCH-4: Integer overflow in Buffer DataLoader bounds check
Resolved in PR #16707. The `BufferDataLoader` constructor now uses `c10::add_overflows` to detect overflows, and checks that no overflow occurs.

TOB-EXECUTORCH-5: Stack buffer overflow in prepare_input_tensors
Resolved in PR #16797. The `prepare_input_tensors` function now validates the provided buffer size against the type before copying the input, which prevents buffer overflows.

TOB-EXECUTORCH-6: Abort on type mismatch in parseListOptionalType
Resolved in PR #19040. The `EValue` type now has fallible `tryTo` accessors that return `Result` instances. The `parseListOptionalType` function is updated to use `tryToOptional` and returns an error if the conversion fails.

TOB-EXECUTORCH-7: Abort on type mismatch in Method::execute_instruction
Resolved in PR #19040. The `FreeCall` handler now uses the fallible `tryToTensor` API and sets the `err` variable if conversion fails.

TOB-EXECUTORCH-8: Abort on type mismatch in parseTensorList
Resolved in PR #19040. The `parseTensorList` function has been updated to use the `tryToTensor` API, and returns an error if conversion fails.

TOB-EXECUTORCH-9: Out of bounds heap read in BPTE tensor deserialization
Resolved in PR #17163. The `tensor_like` function now checks the source data size against the size returned from a call to `nbytes` on the destination tensor, and copies at most `ret_tensor.nbytes` of data.

TOB-EXECUTORCH-10: Out of bounds heap read in WAV audio file parsing
Resolved in PR #16915. The `load_wav_audio_data` function now validates the `Subchunk2Size` field from the header to ensure that the corresponding audio data is wholly contained within the data read from the file.

TOB-EXECUTORCH-11: Unsafe deserialization in ETRecord parsing enables code execution
Resolved in PR #18133. The `ETRecord` parser and the `deserialize_torch_artifact` function now both use `torch.load` with `weights_only` set to `True` to prevent code execution during load.

TOB-EXECUTORCH-12: Out-of-bounds heap read without InternalConsistency verification
Resolved in PR #18597. The `Program::load` method now ensures that the root offset is inside the program buffer, regardless of whether `InternalConsistency` verification is requested or not. Additionally, `EXECUTORCH_ENABLE_PROGRAM_VERIFICATION` is enabled by default.

TOB-EXECUTORCH-13: Out-of-bounds heap write in Program::get_constant_buffer_data
Resolved in PR #16887. The inequality in `ET_CHECK_OR_RETURN_ERROR` is now reversed, which fixes the validation issue.

TOB-EXECUTORCH-14: Infinite loop in XNNExecutor::resize_outputs
Resolved in PR #17181. The loop variable is now a `ssize_t`, which fixes the infinite loop.

TOB-EXECUTORCH-15: Out-of-bounds write in FreeCall instruction handler
Resolved in PR #18176. The `FreeCall` index is not checked in `Method::init`, mirroring `JumpCall`. PR #18658 adds a similar check to `MoveCall`, and also adds a validation macro to make the test easier to read.

TOB-EXECUTORCH-16: Out-of-bounds read in validateTensorLayout
Resolved in PR #17131. The `validateTensorLayout` function now contains an explicit check to ensure that `dim` is equal to the size of the `dim_order` vector.

TOB-EXECUTORCH-17: Integer overflow in constant segment offset validation
Resolved in PR #18662. The main issue described in the finding is resolved in PR #18662 by reordering the bounds check. We also verified that the additional issues listed in the finding have been resolved correctly.

TOB-EXECUTORCH-18: Missing duplicate check in dim_order validation
Resolved in PR #17314. The `validate_dim_order` function now uses a bitmask to track previously seen values and correctly rejects repeat values.

TOB-EXECUTORCH-19: Integer overflow in tensor size validation enables heap buffer overflow
Resolved in PR #19074. The `make_tensor_ptr` function now uses the new function `safe_numel` to compute the number of elements in the tensor. The `safe_numel` function performs overflow checks using `c10::mul_overflows`.

The remaining issues listed in the finding except for `TensorImpl::nbytes` are fixed in the PRs #19075 and #19055 in the same way, using `c10::mul_overflows`. The overflow in `TensorImpl::nbytes` is mitigated by the fix for TOB-EXECUTORCH-3 (see above).

TOB-EXECUTORCH-20: Integer overflow in CUDA tensor copy allocation enables heap buffer overflow
Resolved. The `aoti_torch_copy_` function now calls `SlimTensor::copy_`, which fixes the issue.

TOB-EXECUTORCH-21: Hardcoded element size in CUDA tensor copy enables heap buffer overflow
Resolved. The `aoti_torch_copy_` function now calls `SlimTensor::copy_`, which fixes the issue.

TOB-EXECUTORCH-22: Integer overflow in data loader bounds checking
Resolved in PR #18676. The sum `offset + size` is now computed using overflow-aware arithmetic, and the size validation also checks for overflows in each of the affected data loaders.

TOB-EXECUTORCH-23: Infinite loop with JF_CALL
Resolved in PR #18679. The library now tracks the number of executed instructions in the `Method::execute` method, and returns an error if the counter reaches a pre-defined upper bound.

TOB-EXECUTORCH-24: Integer overflow in program file size validation
Resolved in PR #18662. The `Program::load` method now includes an explicit overflow check to ensure that the computation of the expected file size does not overflow.

TOB-EXECUTORCH-25: Out-of-bounds write in Cadence HiFi bmm operator
Resolved in PR #17453. The `bmm_out` function now uses `memset` to correctly initialize the temporary buffer.

TOB-EXECUTORCH-26: Integer overflow in PlatformMemoryAllocator allocation
Resolved in PR #18680. The allocator now uses overflow-aware arithmetic to compute the allocation size.

TOB-EXECUTORCH-27: Integer overflows in Vulkan tensor operations
Resolved in PR #18792. The main issue reported in the finding is resolved by having the `multiply_integers` function check for overflow before updating the result in each iteration. The remaining issues mentioned in the finding have also been addressed.

TOB-EXECUTORCH-28: Unbounded memory allocation from untrusted PTE file
Partially resolved in PR #18724. The `MethodMeta::memory_planned_buffer_size` method now ensures that the returned size is non-negative. However, it does not enforce an upper bound, allowing malicious files to cause the library to crash due to excessively large allocations.

TOB-EXECUTORCH-29: Large number of security-relevant compiler warnings
Partially resolved. The Executorch team has made some progress to reduce the number of compiler warnings.

TOB-EXECUTORCH-30: Memory corruption due to unsafe mutation of FlatBuffer-backed tensor data
Partially resolved in PR #19522. A new `ET_ENABLE_DEPRECATED_CONSTANT_BUFFER` flag has been added that can be used to disable loading PTE files with constant buffers. This flag is set to 0 in open-source builds to disable PTE files with constant buffers for third-party developers. The flag is currently enabled for internal builds, but the Executorch team has indicated that they are planning to migrate away from using this feature internally as well.

TOB-EXECUTORCH-31: Type confusion in BoxedEvalueList after MoveCall
Resolved in PR #18962. The `destroy_elements` function now avoids dereferencing the wrapped values in the `BoxedEvalueList`, which prevents type confusion during destruction.

TOB-EXECUTORCH-32: Incorrect pointer offset arithmetic in ETDumpGen constructor
Resolved in PR #18782. The implementation now advances the `builder_` pointer by one, which fixes the double scaling issue in the `EtDumpGen` constructor.

TOB-EXECUTORCH-33: Out of bounds read due to missing FlatBuffers verification in XNNCompiler
Resolved in PR #18784. The `XNNHeader::Parse` method now creates a FlatBuffer verifier and uses it to validate the FlatBuffer data.

TOB-EXECUTORCH-34: Out of bounds read due to missing XNNHeader verification
Resolved in PR #18784. The `XNNHeader::Parse` method now includes overflow-safe checks to verify that the FlatBuffer data and constant data regions are contained within the input buffer.

TOB-EXECUTORCH-35: NULL pointer dereferences in XNNCompiler::compileModel
Resolved in PR #18799. The `XNNCompiler::compileModel` method now checks that the FlatBuffer graph, and the `xvalues` and `xnodes` fields, are not null before dereferencing these pointers, which prevents NULL dereferences.

TOB-EXECUTORCH-36: Crash due to unchecked map access in XNNCompiler
Resolved in PR #19008. The XNNPACK backend now uses the new `REMAP_ID` macro to safely access key-value pairs in the `remapped_ids` map. This macro returns an error if the key is not present, which avoids exceptions.

TOB-EXECUTORCH-37: Unbounded memory allocation in XNNPACK subgraph creation
Resolved in PR #18909. The maximum number of external values is now bounded below 4096 in `XNNCompiler::compileModel`.

TOB-EXECUTORCH-38: NULL pointer dereference in BackendDelegate::Init due to missing compile_specs validation
Resolved in PR #18805. The implementation of the `BackendDelegate::Init` method now checks that the pointer returned by the `compile_specs` method is not NULL before dereferencing it.

TOB-EXECUTORCH-39: Heap buffer overflow in XNNPACK backend due to inconsistent tensor dimension validation
Resolved in PR #19013. The `defineTensor` function now reads the size from the `dims_data` vector directly, which prevents the issue.

TOB-EXECUTORCH-40: Infinite loop in XNNPACK subgraph optimization
Unresolved. The ExecuTorch team has opted not to address the issue, since the root cause is in the third-party implementation of XNNPACK.

TOB-EXECUTORCH-41: Out-of-bounds vector access in XNNPACK getConstantDataPtr
Resolved in PR #18820. The `getConstantDataPtr` function has been updated to return a Result, and returns an error on both code paths if the buffer index is too large.

TOB-EXECUTORCH-42: Unbounded memory allocation in XNNPACK backend due to missing dimension validation
Unresolved. The ExecuTorch team is considering enforcing an upper bound for the maximum number of dimensions in `flatbufferDimsToVector`. However, it is unclear how to best determine a reasonable upper bound, and so they have not agreed on one yet.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up with our latest news and announcements, please follow @trailofbits on X or LinkedIn and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.