Borderlands 4 Reverse Engineering Guide

monokrome

2026-04-04

Table of contents

1. Borderlands 4 Reverse Engineering Guide **1**
- 1.1. Chapters 1
  - 1.1.1. Part I: Core Concepts 1
  - 1.1.2. Part II: Game Formats 2
  - 1.1.3. Part III: Practical Application 2
  - 1.1.4. Appendices 3
  - 1.1.5. Reference 3
- 1.2. Quick Start 4
- 1.3. Prerequisites 4
- 1.4. About This Guide 5

2. Borderlands 4 Reverse Engineering Guide **7**
- 2.1. The Journey Ahead 8
- 2.2. How This Guide Works 9
- 2.3. A Word on Philosophy 9
- 2.4. Ethics and Intent 10
- 2.5. What You'll Need 10
- 2.6. Finding Your Path 11
- 2.7. Let's Begin 12

I. Core Concepts **13**

3. Chapter 1: Binary Basics **15**
- 3.1. The Language of Computers 15
- 3.2. How Data Lives in Memory 16

3.3. The Backwards Number Problem 17
3.4. Reading a Hex Dump 18
3.5. Packing Bits Together 19
3.6. Text as Numbers 21
3.7. Why Alignment Creates Gaps 22
3.8. Putting It Together 22
3.9. Exercises 23
3.10. What's Next 24

4. Chapter 2: Unreal Engine Architecture 25
4.1. Everything Is a UObject 25
4.2. The 40-Byte Header 26
4.3. How Unreal Stores Names 27
4.4. Reflection: How Unreal Knows Itself 28
4.5. The Global Object Array 29
4.6. Pak Files: Where Assets Live 30
4.7. Usmap: The Rosetta Stone 31
4.8. Common UE5 Data Types 33
4.9. BL4's Class Structure 34
4.10. Walking Memory: A Preview 34
4.11. Exercises 35
4.12. What's Next 36

5. Chapter 3: Memory Analysis 39
5.1. Why Look at Memory? 39
5.2. Capturing a Memory Dump 40
5.3. Virtual Memory and Address Space 41
5.4. Finding the Global Structures 42
5.5. Following Pointer Chains 43
5.6. Recognizing Patterns 44
5.7. The bl4 Memory Commands 44
5.8. Practical Example: Finding Item Serials 46
5.9. Comparing Memory States 46
5.10. Validating Pointers 47

5.11. Dealing with ASLR 48
5.12. Wine/Proton Considerations 49
5.13. Exercises 49
5.14. What’s Next 50

II. Game Formats 51

6. Chapter 4: Save File Format 53
6.1. Finding Your Saves 53
6.2. The Three Layers 54
6.3. The Encryption Layer 55
6.4. Key Derivation: Your Steam ID Is the Key 55
6.5. The Compression Layer 57
6.6. The YAML Structure 57
6.6.1. Character Save Structure 57
6.6.2. Equipped Slot Mapping 61
6.6.3. State Flags 61
6.7. Working with Saves 63
6.8. Item Injection 64
6.8.1. Adding to Backpack 64
6.8.2. Equipping an Item 64
6.8.3. Live Editing Limitations 65
6.9. Common Edits 66
6.10. Map Exploration Data (foddatas) 68
6.10.1. Structure 68
6.10.2. Fog Alpha Map Format 69
6.10.3. Zone Names 71
6.10.4. Manipulating FOD Data 71
6.11. Safehouse and World Progress 72
6.11.1. Safehouses 72
6.11.2. Silos 73
6.11.3. Bounties 73
6.11.4. Collectibles 74

6.12 Unlockables . . . 74
6.12.1 Hoverdrive Skins . . . 75
6.12.2 Vault Hunter Rank . . . 75
6.13 Backups . . . 76
6.14 What Can Go Wrong . . . 76
6.15 Manual Decryption Walkthrough . . . 77
6.16 Why ECB Mode? . . . 79
6.17 Exercises . . . 79
6.18 What's Next . . . 80

7. Chapter 5: Item Serials 81
7.1. What's Encoded in a Serial . . . 81
7.2. The Decoding Pipeline . . . 82
7.3. Base85: Custom Alphabet . . . 83
7.4. Bit Mirroring: The Obfuscation Layer . . . 83
7.5. Bit Assembly: MSB Stream, LSB Data . . . 84
7.6. Token Parsing: The Real Structure . . . 85
7.7. Item Type: Determined by First Token . . . 86
7.8. Two Serial Formats . . . 86
7.8.1. Weapon Format (VarInt-first) . . . 86
7.8.2. Equipment Format (VarBit-first) . . . 87
7.9. Part Group IDs (Categories) . . . 88
7.10 Part Indices Are Context-Dependent . . . 90
7.11 Level Encoding . . . 90
7.12 Element Encoding . . . 90
7.13 Decoding a Serial Manually . . . 91
7.14 Decoding Examples . . . 92
7.14.1 Weapon Serial . . . 92
7.14.2 Equipment Serial . . . 92
7.15 Exercises . . . 93
7.16 What's Next . . . 94

8. Chapter 6: NCS Format (Nexus Config Store) 95
8.1. Overview . . . 96

8.2. Compression 97
8.2.1. Outer Header (16 bytes) 97
8.2.2. Inner Header (16+ bytes) 97
8.2.3. Format Flags 98
8.2.4. Compatibility 98
8.3. Content Format 99
8.3.1. BlobHeader (16 bytes) 99
8.3.2. Header Strings 99
8.3.3. TypeCodeTable 99
8.3.4. Three String Blocks 101
8.3.5. Binary Data Section 101
8.4. Binary Section 102
8.4.1. String Indexing 102
8.4.2. Decode Loop 102
8.4.3. Record Tags 103
8.4.4. Node Types 104
8.4.5. Remap Arrays 104
8.4.6. Hash Function 104
8.5. Inventory Parts (inv.bin) 105
8.5.1. File Structure 105
8.5.2. Parser Pipeline 106
8.5.3. Weapon Type Definitions 107
8.5.4. Serial Index Structure 108
8.5.5. Legendary Compositions 110
8.5.6. NCS vs Memory Part Names 111
8.5.7. Extracting Parts 111
8.6. Actor Definitions (gbxactor.bin) 112
8.6.1. Entry Categories 112
8.7. Entity Display Names (NameData) 113
8.7.1. Boss Names 114
8.7.2. Enemy Variants 114
8.7.3. Boss Replay and Challenge Text 114
8.7.4. Internal to Display Name Mapping 115
8.7.5. Extracting NameData 115

8.8. NCS Manifest 116
8.8.1. Manifest Header 116
8.8.2. Manifest Entry 116
8.9. DataTable Relationship 117
8.10. Known File Types 118
8.10.1. Key Files for Loot Analysis 118
8.10.2. Extracting Drop Information 119
8.11. Type Prefixes 119
8.12. Worked Example: achievement.bin 120
8.12.1. File Layout 120
8.12.2. Parsed Output 120
8.13. Compatibility Issues 121
8.13.1. Oodle SDK Requirements 121
8.13.2. String Validation 122
8.14. Future Work 122

III. Practical Application 125

9. Chapter 7: Data Extraction 127
9.1. The Game File Landscape 127
9.2. Data Source Hierarchy 128
9.3. What We Can Extract 129
9.4. What We Can't Extract 130
9.4.1. UE5 Metadata: What We Know 131
9.4.2. Where Parts Actually Live 132
9.4.3. The Key Insight: Self-Describing Parts 132
9.5. Memory Extraction: The Breakthrough 134
9.5.1. The Part Registration Structure 134
9.5.2. Verified Example 135
9.5.3. Extraction Algorithm 135
9.5.4. Why This Works 137
9.5.5. Extraction Results and Limitations 138
9.5.6. Data Pipeline: Current State 139

9.5.7. What NCS/UASSET Data Provides 140
9.5.8. NCS Serial Index Discovery 140
9.5.9. Category Derivation from NCS (Jan 2026 Discovery) 141
9.6. Empirical Validation (Fallback) 143
9.7. Extraction Tools 143
9.7.1. retoc — IoStore Extraction 143
9.7.2. uextract — Project Tool 144
9.8. The Usmap Requirement 144
9.9. Extracting Parts from Memory 145
9.9.1. Step 1: Create Memory Dump 145
9.9.2. Step 2: Extract Part Names 146
9.9.3. Step 3: Build Parts Database 146
9.10. Working with Extracted Assets 147
9.10.1. Asset Structure 147
9.10.2. Finding Specific Data 148
9.10.3. Stat Patterns 148
9.11. Oodle Compression 149
9.12. Building a Data Pipeline 150
9.13. Strategies for Complete Index Coverage 150
9.14. Summary: Data Sources 151
9.15. Exercises 152
9.16. What's Next 153

10. Chapter 8: Parts System 155
10.1. Data Sources 156
10.2. Item Types 157
10.2.1. Weapons 157
10.2.2. Shields 157
10.3. Part Naming Convention 158
10.3.1. Weapon Parts 158
10.3.2. Shield Parts 159
10.3.3. Licensed and Special Parts 159

10.4. Part Categories . . . 160
10.4.1. Weapon Parts . . . 160
10.4.2. Shield Parts . . . 160
10.4.3. Part Counts by Item Type . . . 161
10.5. Category Mappings . . . 162
10.5.1. Weapons . . . 162
10.5.2. Equipment . . . 162
10.5.3. Category Derivation . . . 163
10.6. Composition System . . . 164
10.6.1. Rarity Tiers . . . 164
10.6.2. Legendary Compositions . . . 165
10.7. Licensed Parts . . . 165
10.7.1. License Types . . . 166
10.7.2. Tediore Reload Variants . . . 166
10.7.3. Weapon-Specific Underbarrels . . . 167
10.7.4. KL (Killer License) Parts . . . 167
10.8. Serial Index Architecture . . . 168
10.8.1. GbxSerialNumberIndex Structure . . . 168
10.8.2. Part Index Resolution . . . 168
10.8.3. Registration Order . . . 169
10.9. Part Validation . . . 169
10.9.1. Using item_parts.json . . . 170
10.9.2. Using the CLI . . . 170
10.9.3. NCS vs. Memory: Source of Truth . . . 170
10.10. Level Gating . . . 171
10.10.1. Known Level-Gated Categories . . . 171
10.11. Firmware . . . 173
10.11.1. Firmware Parts . . . 173
10.11.2. Serial Encoding . . . 174
10.11.3. Class Mod Skill Interaction . . . 176
10.12. Known Gaps . . . 176
10.12.1. Rainbow Vomit Legendary Parts . . . 176
10.12.2. Non-Prefixed Parts . . . 177
10.12.3. Missing Class Mod Parts . . . 177

10.12.4. Equipment Low Resolution 178
10.13. Key Differences from BL3 178
10.14. Unresolved Questions 179

11. Chapter 9: Using bl4 Tools **181**
11.1. Building the Tools 181
11.1.1. Prerequisites 181
11.1.2. Build 182
11.2. CLI Structure 182
11.3. Save File Operations 183
11.3.1. Inspect a Save 183
11.3.2. Decrypt/Encrypt 183
11.3.3. Edit Interactively 184
11.3.4. Get/Set Values 184
11.4. Serial Operations 185
11.4.1. Decode 185
11.4.2. Compare 186
11.4.3. Modify 186
11.4.4. Batch Decode 186
11.5. Drop Rate Queries 187
11.5.1. Find Where an Item Drops 187
11.5.2. List All Drops from a Source 187
11.5.3. List All Known Items or Sources 187
11.5.4. Generate Drops Manifest 188
11.6. Items Database (idb) 188
11.6.1. Basic Operations 188
11.6.2. Import Items 189
11.6.3. Attachments 189
11.6.4. Value Attribution 189
11.6.5. Verification and Curation 190
11.6.6. Export and Merge 190
11.6.7. Community Server Sync 190
11.6.8. Database Maintenance 191

11.7. Memory Operations 191
11.7.1. With Dump File 191
11.7.2. Generate Usmap 191
11.7.3. FName Operations 192
11.7.4. Object Inspection 192
11.7.5. Parts Extraction 192
11.7.6. Object Discovery 193
11.7.7. NCS Schema Extraction 193
11.7.8. String Search 193
11.7.9. Low-Level Memory Access 193
11.7.10. Preload Library 194
11.8. NCS Operations 194
11.8.1. Decompress from Pak 194
11.8.2. Scan Decompressed Files 195
11.8.3. Show File Contents 195
11.8.4. Search 196
11.8.5. Extract Data 196
11.8.6. Debug 197
11.8.7. Statistics 198
11.9. Launch 198
11.10. uextract 199
11.10.1. Extract from IoStore 199
11.10.2. Extract from Traditional Pak Files 199
11.10.3. Dump ScriptObjects 200
11.10.4. Find Assets by Class 200
11.10.5. List Classes 200
11.11. Configuration 200
11.12. Common Workflows 201
11.12.1. Edit a Save 201
11.12.2. Analyze an Item 201
11.12.3. Import Items from Saves 202
11.12.4. Update After Game Patch 202
11.12.5. Full Asset Extraction Pipeline 202

11.13. Shell Tips 203
11.13.1. Quoting Serials 203
11.13.2. Aliases 203
11.13.3. Piping 203
11.14. Troubleshooting 204
11.14.1. "Decryption failed" 204
11.14.2. "Invalid serial" 204
11.14.3. "Memory read failed" 204
11.14.4. "Preload library not found" 205
11.14.5. "Failed to load drops manifest" 205
11.15. Quick Reference 205
11.16. What's Next? 209

IV. Appendices **211**

12. Appendix A: SDK Class Layouts **213**
12.1. Core UE5 Types 213
12.1.1. FName (8 bytes) 213
12.1.2. FVector (24 bytes) 213
12.1.3. FRotator (24 bytes) 214
12.1.4. FQuat (32 bytes) 214
12.1.5. FTransform (96 bytes) 215
12.1.6. TArray (16 bytes) 215
12.1.7. FString (16 bytes) 215
12.2. UObject Hierarchy 216
12.2.1. UObject (40 bytes) 216
12.2.2. UField (48 bytes) 216
12.2.3. UStruct (176 bytes) 217
12.2.4. UClass (512 bytes) 217
12.3. Actor Hierarchy 217
12.3.1. AActor (912 bytes) 217
12.3.2. APawn (1040 bytes) 218
12.3.3. ACharacter (1864 bytes) 218

12.4 Controller Classes 219
12.4.1 AController (1064 bytes) 219
12.4.2 APlayerController (2392+ bytes) 219
12.5 BL4/Oak Classes 220
12.5.1 AGbxPlayerController (3496 bytes) 220
12.5.2 AOakPlayerController (15424 bytes) 220
12.5.3 AGbxCharacter (15232 bytes) 221
12.5.4 AOakCharacter (38800 bytes) 221
12.6 Inventory Classes 222
12.6.1 AInventory (2224 bytes) 222
12.6.2 AWeapon (3400 bytes) 222
12.7 Currency System 223
12.7.1 FSToken (12 bytes) 223
12.7.2 FGbxCurrency (24 bytes) 223
12.7.3 UGbxCurrencyManager (64 bytes) 223
12.8 Attribute Types 224
12.8.1 FGbxAttributeFloat (12 bytes) 224
12.8.2 FGbxAttributeInteger (12 bytes) 224
12.9 Enums 225
12.9.1 ECharacterHealthCondition 225
12.9.2 EMovementMode 225
12.10 Global Pointers 226
12.11 Pattern Signatures 226
12.12 Mesh & Visibility 227
12.13 FProperty Layout 227

13. Appendix B: Weapon Parts Reference **229**
13.1 Part Naming Convention 229
13.2 Manufacturers 230
13.3 Weapon Types 230
13.3.1 Type Codes 230
13.3.2 EWeaponType Enum 231
13.4 Scope Parts by Manufacturer 231
13.4.1 BOR (Ripper) 231

13.4.2 DAD (Daedalus) 233
13.4.3 JAK (Jakobs) 234
13.4.4 MAL (Maliwan) 235
13.4.5 ORD (Order) 236
13.4.6 TED (Tediore) 237
13.4.7 TOR (Torgue) 238
13.4.8 VLA (Vladof) 240
13.5 Barrel Parts 241
13.5.1 Heavy Weapons 241
13.6 Part Index Organization 242
13.6.1 The Self-Describing Design 242
13.6.2 How Indices Are Assigned 242
13.6.3 Registration Order Pattern 243
13.6.4 Index Gaps 244
13.6.5 Implications for Modding 244
13.7 Part Compatibility Rules (Verified) 244
13.7.1 Rule 1: Prefix Determines Weapon Type . . . 245
13.7.2 Rule 2: Barrel Accessories Require Matching Barrel 245
13.7.3 Rule 3: Scope Accessories Require Matching Scope AND Lens 246
13.7.4 Rule 4: Some Magazine Modifiers Are Barrel-Specific 246
13.7.5 Rule 5: Maximum Accessory Counts 247
13.7.6 Rule 6: Legendary Barrel Restrictions 247
13.7.7 Rule 7: Barrel-Type Constraints for Magazines 247
13.7.8 Validation Algorithm 248
13.7.9 Implications for Save Editing 249
13.8 Part Selection System (Theoretical) 249
13.8.1 Class-Based Selection (from usmap) 250
13.9 Part Slot Types 250
13.10 Weapon Naming System 251
13.10.1 Primary Indices 251
13.10.2 Naming Indices (from WeaponNamingStruct) 252

13.11. Known Legendaries 252
13.11.1. By Manufacturer 252
13.12. Rarity System 253
13.12.1. Internal Format 253
13.13. Part Count by Category 254
13.13.1. Weapons 254
13.13.2. Class Mods 255
13.13.3. Firmware 255
13.13.4. Shields 256
13.13.5. Gadgets and Gear 256
13.13.6. Enhancements 256
13.13.7. Summary 257
13.14. Variant Suffixes 257
13.15. Data Files 258

14. Appendix C: Loot System Internals 261
14.1. Two Ways to Get a Legendary 261
14.1.1. Dedicated Drops 261
14.1.2. World Drops 262
14.2. How a Drop Resolves 263
14.3. Pool Sizes and Per-Item Odds 264
14.4. Rarity Estimation 265
14.5. What We Know vs. Don’t Know 266
14.6. NCS Data Sources 267
14.7. Reference: Dedicated Drop Probability Tiers 268
14.8. Reference: Rarity Weights 269
14.9. Reference: Enemy Drop Fields 269
14.10. Reference: Boss → Legendary Mappings 270
14.11. Reference: Item Pools 274
14.11.1. Boss Pool Lists 274
14.11.2. Rarity-Tiered Weapon Pools 275
14.11.3. Special Pools 275
14.12. Reference: Drop Source Types 276
14.13. Reference: Item Composition in NCS 276

14.14. Reference: Codes 277
14.14.1. Weapon Types 277
14.14.2. Manufacturers 277
14.14.3. Gear Slots 277
14.15. Reference: Drops Manifest Format 278
14.16. Reference: Loot System Classes 279
14.16.1. Pool System 279
14.16.2. Rarity Resolution 280
14.16.3. Luck System 280

15. Appendix D: Game File Structure **283**
15.1. Overview 283
15.2. File Locations 284
15.2.1. Steam (Linux) 284
15.2.2. Steam (Windows) 284
15.3. Pak Chunk Contents 284
15.4. Top-Level Content Structure 285
15.5. Player Characters (Vault Hunters) 286
15.6. Gear System 286
15.6.1. Equipment Types 286
15.6.2. Weapon System 287
15.7. GameData System 288
15.8. AI System 289
15.9. File Naming Conventions 290
15.10. Weapon Part Types 291
15.10.1. Core Weapon Parts 291
15.10.2. Attachment Parts 291
15.10.3. Manufacturer-Specific 292
15.10.4. Element & Augments 292
15.10.5. Grenade Parts 293
15.10.6. Class Mod Parts 293
15.10.7. Other 293
15.11. Key DataTables 294
15.11.1. Weapon Naming 294

15.11.2. Balance Data . . . 294
15.12. Asset Path Mapping . . . 295
15.13. Gear Types Found . . . 295
15.14. New Systems in BL4 . . . 296
15.15. Extraction Results . . . 297
15.15.1. Manufacturer Codes . . . 297
15.15.2. Balance Data Categories . . . 297
15.16. Notes . . . 298
15.17. NCS Format (Nexus Config Store) . . . 298
15.17.1. Key NCS Files . . . 298
15.17.2. Quick Reference: NCS Chunk Header . . . 299
15.17.3. Quick Reference: Format Codes . . . 299
15.17.4. Quick Reference: NCS Manifest (_NCS/) . . . 300
15.17.5. Decompression CLI . . . 300

16. Glossary **301**
16.1. A . . . 301
16.2. B . . . 302
16.3. C . . . 302
16.4. D . . . 303
16.5. E . . . 303
16.6. F . . . 303
16.7. G . . . 304
16.8. H . . . 305
16.9. I . . . 305
16.10. L . . . 305
16.11. M . . . 306
16.12. N . . . 306
16.13. O . . . 307
16.14. P . . . 307
16.15. R . . . 308
16.16. S . . . 308
16.17. T . . . 309
16.18. U . . . 309

16.19 V 310
16.20 W 310
16.21 Z 310
16.22 Symbols 311
16.23 Quick Reference: Playable Characters 311
16.24 Quick Reference: Key Offsets 311
16.25 Quick Reference: File Extensions 312

1. Borderlands 4 Reverse Engineering Guide

Zero to Hero

A comprehensive guide to understanding game internals, reverse engineering techniques, and using the bl4 tooling to analyze and modify Borderlands 4.

1.1. Chapters

1.1.1. Part I: Core Concepts

Chapter	Title	Description
1	Binary Basics	Hexadecimal, endianness, data types, and memory layout
2	Unreal Engine Architecture	UObjects, reflection system, pak files, and usmap

Chapter	Title	Description
3	Memory Analysis	Process memory, dumps, pattern scanning, pointer chains

1.1.2. Part II: Game Formats

Chapter	Title	Description
4	Save File Format	Encryption, compression, YAML structure, key derivation
5	Item Serials	Base85 encoding, bit manipulation, token parsing
6	NCS Format	Nexus Config Store: compression, content format, binary section

1.1.3. Part III: Practical Application

Chapter	Title	Description
7	Data Extraction	Pak files, NCS parsing, memory dumps, manifest generation
8	Parts System	Part categories, compositions, licensed parts, validation

Chapter	Title	Description
9	Using bl4 Tools	Complete CLI reference and practical workflows

1.1.4. Appendices

Appendix	Title	Description
A	SDK Class Layouts	Memory layouts for UObject, AOakCharacter, AWeapon, etc.
B	Weapon Parts Reference	Complete catalog of weapon parts by manufacturer
C	Loot System Internals	Drop pools, rarity weights, luck system
D	Game File Structure	Full asset tree and file organization

1.1.5. Reference

Title	Description
Glossary	Terms, definitions, and quick reference tables

1.2. Quick Start

New to reverse engineering? Start with Chapter 1: Binary Basics and work through sequentially.

Want to edit saves? Jump to Chapter 4: Save File Format.

Need to decode an item? See Chapter 5: Item Serials.

Interested in the NCS format? See Chapter 6: NCS Format for the complete format specification.

Just want the tool reference? Go to Chapter 9: Using bl4 Tools.

1.3. Prerequisites

Before starting, ensure you have:

- Basic programming knowledge (any language)
- Command line familiarity
- Rust toolchain installed (rustup.rs)
- The bl4 repository cloned and built

```
git clone https://github.com/monokrome/bl4
cd bl4
cargo build --release
```

1.4. About This Guide

This guide accompanies the **bl4** project—a Borderlands 4 save file editor and item serial decoder. It documents not just *how* to use the tools, but *why* they work, giving you the knowledge to explore further on your own.

Each chapter includes:

- **Concept explanations** with visual diagrams
- **Practical examples** you can try immediately
- **Exercises** to test your understanding
- **Tips** from real reverse engineering sessions

For the latest version of this guide, visit book.bl4.dev

2. Borderlands 4 Reverse Engineering Guide

> **Note:** The information in this guide is based on ongoing reverse engineering analysis. Some details may be incomplete or subject to revision as our understanding improves. If you find errors or have corrections, please reach out at hey@monokro.me.

Every weapon you pick up in Borderlands 4 exists as a compact binary string buried in your save file. That string—maybe 30 bytes of data, rendered as about 40 characters of Base85—encodes everything about the weapon: its manufacturer, every part attached to it, the random seed that determined its stats, even which rarity tier it rolled. The game reads that string and reconstructs the full item, parts and all.

But how does it work?

This guide exists because someone asked that question. What started as curiosity about save file formats turned into a deep dive through binary data, encryption schemes, Unreal Engine internals, and ultimately a complete understanding of how Borderlands 4 stores and encodes its item system internally.

2.1. The Journey Ahead

You're about to learn reverse engineering by doing it. We won't just explain concepts—we'll use them immediately to solve real problems. By the time you finish this guide, you'll be able to:

Decrypt and edit save files. BL4 saves are encrypted with AES-256, compressed, and structured as YAML. We'll walk through the entire process of opening them up, making changes, and putting them back together.

Decode item serials. Those cryptic strings that encode weapons use a custom Base85 alphabet, bit mirroring, and a token-based format. We'll parse them byte by byte until they make perfect sense.

Extract data from game files. Unreal Engine 5 packs everything into `.pak` containers with a custom format. We'll build tools to crack them open and pull out the good stuff—weapon definitions, part databases, drop tables.

Understand memory analysis. Sometimes the only way to find what you need is to take a snapshot of the running game. We'll learn to navigate gigabytes of process memory to locate specific structures.

None of this requires prior reverse engineering experience. If you can write basic code and use a command line, you have everything you need to start.

2.2. How This Guide Works

Each chapter builds on the previous, taking you from fundamentals to practical application. The structure follows the natural progression of a reverse engineering project:

First, we establish foundations. Binary representation, data types, memory layout—these concepts appear everywhere in game data. Understanding them makes everything else click into place.

Then, we learn Unreal Engine's architecture. BL4 runs on UE5, and knowing how Unreal organizes data (UObjects, reflection, property serialization) explains patterns we'll see repeatedly in memory dumps and pak files.

Next, we apply these concepts. We'll analyze save files, decode serials, dump process memory, and extract game assets. Each technique opens new doors.

Finally, we use the tools. The bl4 project provides command-line utilities for everything we've learned. The tools chapter serves as your practical reference for day-to-day use.

2.3. A Word on Philosophy

The best reverse engineers share a common trait: they form hypotheses and test them ruthlessly. "I think this byte controls weapon damage" leads to "let me change it and see what happens." Wrong guesses are valuable—they narrow down possibilities.

Keep notes as you explore. The documentation you're reading started as scratch files filled with hex dumps and question marks. Over time, patterns emerged, and those scratch notes became explanations. Your discoveries might become documentation too.

2.4. Ethics and Intent

This guide exists for education and personal use. It's for:

- Understanding how games work beneath the surface
- Building save editors that modify your own single-player experience
- Learning reverse engineering techniques applicable far beyond gaming
- Satisfying the curiosity that comes from wanting to know *how things work*

It's not for cheating in multiplayer, bypassing DRM, or violating terms of service. The techniques here are powerful—use them responsibly.

2.5. What You'll Need

Before we begin, make sure you have the Rust toolchain installed. We'll be building and using the bl4 tools throughout:

```
# Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# Clone and build
git clone https://github.com/monokrome/bl4
cd bl4
cargo build --release
```

A hex editor (any will do) and familiarity with your system's terminal will help. If you're on Linux, tools like `xxd` for hex dumps come pre-installed. On Windows, HxD is a solid free option.

For deeper analysis work, Ghidra (free) or IDA (commercial) let you decompile binaries. Rizin or Radare2 provide scriptable binary analysis. We'll use some of these in later chapters, but they're not strictly required to follow along.

2.6. Finding Your Path

This guide is meant to be read sequentially, but you might have a specific goal in mind:

Want to decode a weapon serial? The encoding process is fascinating, but it also requires understanding binary basics and the token format. Start at Chapter 1 and work through Chapter 5. You'll have full context.

Need to edit a save file? Chapter 4 covers the format in detail. If you're comfortable with encryption concepts and binary

data, jump there. Otherwise, Chapters 1-2 provide helpful background.

Interested in extracting game assets? Chapter 7 walks through data extraction and pak file parsing. Chapter 6 covers the NCS format that holds part and loot data. Chapter 2's coverage of Unreal Engine architecture explains why assets are structured the way they are.

Curious about the whole picture? Start from the beginning. Each chapter adds a piece to the puzzle.

2.7. Let's Begin

The first step in any reverse engineering project is understanding how computers represent data. It sounds basic, but it's foundational. Knowing that 0x41 means 'A' in ASCII, that little-endian systems store bytes backwards, and that a 4-byte integer can represent about 4 billion values—these facts become second nature, and they unlock everything else.

Turn the page to Chapter 1. We'll start with binary basics, and before you know it, you'll be reading hex dumps like they're plain English.

This guide accompanies the bl4 project: https://github.com/monokrome/bl4

Part I.

Core Concepts

3. Chapter 1: Binary Basics

Open any game file in a hex editor and you'll see a wall of numbers and letters that looks like gibberish. But it's not gibberish—it's data, organized according to rules we can learn. This chapter teaches you to read that wall of bytes like a map.

3.1. The Language of Computers

Everything in a computer reduces to numbers. Your character's health? A number. The name of that legendary weapon? A sequence of numbers representing characters. The damage calculation when you shoot an enemy? Numbers in, numbers out.

Humans count in base 10 because we have ten fingers. Computers count in base 2 because transistors have two states: on and off. This creates an immediate translation problem. The number 137 is easy for us to read, but computers see it as `10001001`—eight binary digits representing 128 + 8 + 1.

Writing out binary gets tedious fast. An 8-byte pointer becomes 64 ones and zeros. So we use hexadecimal—base 16—as a compact representation. Each hex digit represents exactly four bits,

so two hex digits represent exactly one byte. The number 137 becomes `0x89`. Much better.

Here's the relationship:

```
Decimal:     137
Binary:      1000 1001
Hexadecimal: 8    9     →  0x89
```

You don't need to do these conversions in your head. Use a calculator. What matters is recognizing that when you see `0x89` in a hex dump, you're looking at one byte with a value of 137.

3.2. How Data Lives in Memory

When a game stores your character's level, it doesn't just write "50" somewhere. It chooses a *data type*—a container of a specific size with rules about what values it can hold.

The most common types you'll encounter:

Integers store whole numbers. An `i32` (signed 32-bit integer) takes 4 bytes and can hold values from about -2 billion to +2 billion. A `u8` (unsigned 8-bit integer) takes 1 byte and holds 0-255. The 'u' means unsigned (no negative values), and the number indicates bits.

Floating point numbers store decimals. Damage multipliers, coordinates, health values—anything that needs fractional precision uses `f32` (4 bytes) or `f64` (8 bytes).

Pointers are addresses pointing to other memory locations. On 64-bit systems like modern PCs, pointers are 8 bytes (`u64`).

Type	Size	Range
`u8`	1 byte	0 to 255
`i16`	2 bytes	-32,768 to 32,767
`u32`	4 bytes	0 to ~4.3 billion
`i64`	8 bytes	±9.2 quintillion
`f32`	4 bytes	~7 digits precision

When reverse engineering, your job is figuring out which type holds which value. Is that item level a `u8` or `u16`? Only one way to find out: look at the data and test your hypothesis.

3.3. The Backwards Number Problem

Here's something that trips up every beginner. Let's say you're looking for the value 0x12345678 in a file. You search for those bytes and find… nothing. Then you search for `78 56 34 12` and there it is.

Welcome to little-endian byte order.

Intel CPUs—which means basically every PC—store multi-byte numbers with the *least significant byte first*. The "small end" comes first, hence "little-endian." It feels backwards because we read numbers from left to right, most significant digit first.

The value 0x12345678 stored on a little-endian system:

```
Memory address:  0x00  0x01  0x02  0x03
Stored bytes:    0x78  0x56  0x34  0x12
```

Big-endian systems store bytes in the order we'd expect: 0x12 first, then 0x34, then 0x56, then 0x78. Network protocols often use big-endian (it's sometimes called "network byte order").

BL4 uses both. Save files are little-endian because they're made for x86 processors. But item serials—the internal strings that encode each weapon in save data—use big-endian for the Base85 decoding step. Always verify which you're dealing with before assuming.

3.4. Reading a Hex Dump

Let's look at real data. Here's the first 64 bytes of a BL4 save file:

```
00000000: 4145 532d 3235 362d 4543 4200 0000 0000  AES-256-ECB.....
00000010: 0000 0000 0000 0000 a8de 0700 0000 0000  ................
00000020: 789c 0bc9 c82c 5600 5346 4b23 0b32 4b32  x....,.V.SFK#.2K2
00000030: 93cb 4a8b cb52 f34b 148c f513 73f3 5212  ..J..R.K....s.R.
```

Three columns: offset on the left, hex bytes in the middle, ASCII representation on the right.

The offset tells you where in the file you are. `00000020` means byte 32 (0x20 in hex).

The hex bytes are the actual data, two hex digits per byte. `4145` represents two bytes: 0x41 and 0x45.

The ASCII column shows printable characters for bytes that fall in the displayable range. Non-printable bytes show as dots. This is where strings jump out at you—look at the first line: "AES-256-ECB" is clearly visible.

Now let's decode what we're seeing:

Bytes 0x00-0x0B: The string "AES-256-ECB" followed by a null byte. This is a magic marker identifying the encryption scheme.

Bytes 0x0C-0x17: Zeros. Padding or reserved space.

Bytes 0x18-0x1B: `a8 de 07 00`. Four bytes, little-endian. That's 0x0007DEA8 = 515,752 in decimal. This is the size of the encrypted payload.

Byte 0x20: `78 9c`. These two bytes are a zlib magic number—the compressed data starts here.

Pattern recognition is the core skill. After seeing a few zlib-compressed files, you'll instantly recognize that `78 9c` signature. After enough save files, the AES-256-ECB header becomes as readable as English.

3.5. Packing Bits Together

Sometimes a single byte holds multiple values. Games do this to save space or because the values are logically related.

Consider item flags. An item might be equipped (yes/no), favorited (yes/no), marked as junk (yes/no), and new (yes/no). Four boolean values could use four bytes, but why waste three? Pack them into one:

```
Bit 7  6  5  4  3  2  1  0
    _  _  _  _  N  J  F  E

E = Equipped (bit 0)
F = Favorited (bit 1)
J = Junk (bit 2)
N = New (bit 3)
```

If you see the byte 0x0B (`00001011` in binary), that means: - Equipped: yes (bit 0 = 1) - Favorited: yes (bit 1 = 1) - Junk: no (bit 2 = 0) - New: yes (bit 3 = 1)

Extracting individual bits requires bitwise operations:

```
let flags: u8 = 0x0B;

// Check if equipped (bit 0)
let equipped = (flags & 0x01) != 0;  // true

// Check if junk (bit 2)
let junk = (flags & 0x04) != 0;  // false

// Get bits 0-1 together
let low_two_bits = flags & 0x03;  // 0x03 = 3
```

The & operator (bitwise AND) masks off the bits you don't care about. Shifting with >> moves bits to the right so you can isolate them. You'll use these operations constantly when parsing packed data formats.

3.6. Text as Numbers

Strings are just sequences of numbers representing characters. The mapping depends on the encoding.

ASCII maps characters to single bytes. 'A' is 65 (0x41), 'a' is 97 (0x61), space is 32 (0x20). Only covers 128 characters.

UTF-8 is ASCII-compatible but extends to all Unicode characters. Multi-byte sequences for non-ASCII characters.

UTF-16 uses 2 bytes per character (or 4 for rare characters). Common in Windows APIs and some game engines.

In hex dumps, ASCII strings are easy to spot because the bytes fall in the printable range (0x20-0x7E). You'll see the actual text in the right column.

Two common string storage formats:

Null-terminated: The string ends when you hit 0x00.

```
48 65 6C 6C 6F 00  →  "Hello"
```

Length-prefixed: A length value precedes the characters.

```
05 00 00 00 48 65 6C 6C 6F  →  5 characters, then "Hello"
```

Unreal Engine uses length-prefixed strings with additional metadata. We'll cover the exact format in Chapter 2.

3.7. Why Alignment Creates Gaps

If you're looking at a hex dump of a struct and see mysterious zero bytes between fields, you've found alignment padding.

CPUs access memory most efficiently when data aligns to certain boundaries. A 4-byte integer reads fastest from an address divisible by 4. A 64-bit pointer wants an address divisible by 8.

Compilers automatically insert padding to maintain alignment:

```
struct WeaponStats {
    u8  rarity;        // Offset 0x00, 1 byte
    // 3 bytes padding  // Offset 0x01-0x03
    u32 damage;        // Offset 0x04, 4 bytes (aligned to 4)
    u8  element;       // Offset 0x08, 1 byte
    // 7 bytes padding  // Offset 0x09-0x0F
    u64 serial_ptr;    // Offset 0x10, 8 bytes (aligned to 8)
};  // Total: 24 bytes
```

The struct logically contains 14 bytes of data (1+4+1+8), but its actual size is 24 bytes because of padding. Knowing this prevents confusion when offsets don't match what you'd calculate by adding field sizes.

3.8. Putting It Together

Let's decode a small example. Say you find these bytes at the start of an item record:

```
32 00 00 00 01 00 00 00 E8 03 00 00
```

Breaking it down:

- `32 00 00 00`: Little-endian u32 = 0x00000032 = 50. Probably item level.
- `01 00 00 00`: Little-endian u32 = 1. Maybe a type ID or flag.
- `E8 03 00 00`: Little-endian u32 = 0x000003E8 = 1000. Could be damage, price, or some other stat.

Is this interpretation correct? Only way to know is test it. Change the first value to `64 00 00 00` (100 in decimal), reload the save, and see if the item is now level 100. If yes, hypothesis confirmed. If no, back to the drawing board.

This is the cycle of reverse engineering: observe, hypothesize, test, repeat.

3.9. Exercises

Exercise 1: Reading Hex

Convert these values: 1. `0xFF` to decimal 2. `1000` (decimal) to hex 3. What's the decimal value of `a8 de 07 00` as a little-endian u32?

Exercise 2: Endianness

You're looking for the value 305,419,896 (0x12345678) in a file. What byte sequence do you search for on a little-endian system?

Exercise 3: Bit Extraction

Given the byte `0xD6` (binary: `11010110`): 1. What's bit 0? 2. What's bit 7? 3. What are bits 4-6 as a 3-bit value?

Answers

Exercise 1: 1. 255 2. 0x3E8 3. 0x0007DEA8 = 515,752

Exercise 2: `78 56 34 12` (least significant byte first)

Exercise 3: 1. Bit 0 = 0 (rightmost bit) 2. Bit 7 = 1 (leftmost bit) 3. Bits 4-6 = `(0xD6 >> 4) & 0x07` = `0b101` = 5

3.10. What's Next

You now have the foundation to read binary data. But raw bytes only get you so far—you need to understand how the game engine organizes that data into meaningful structures.

Next, we'll explore Unreal Engine 5's architecture: how it tracks objects with reflection, serializes properties, and stores assets in pak files. Understanding UE5's patterns makes the difference between staring at random bytes and recognizing game data instantly.

Next: **Chapter 2: Unreal Engine Architecture**

4. Chapter 2: Unreal Engine Architecture

Borderlands 4 runs on Unreal Engine 5, and that's great news for reverse engineering. Every Unreal game—from Fortnite to Elden Ring to indie projects—shares the same fundamental architecture. Learn how Unreal organizes data once, and you've learned something applicable to hundreds of games.

This chapter explores how Unreal thinks about the world. Understanding these patterns transforms mysterious byte sequences into recognizable structures.

4.1. Everything Is a UObject

Unreal Engine has a single base class for nearly everything: `UObject`. Your character? A UObject. That legendary weapon? A UObject. The class definition that describes what a weapon *is*? Also a UObject.

This design creates a unified system where the engine can manage, serialize, and inspect anything. Need to save the game state? Iterate through UObjects. Need to find all weapons in

the world? Query UObjects by class. Need to know what fields a weapon has? Ask the UObject's class definition.

The inheritance hierarchy for a Borderlands 4 player character looks like this:

```
UObject
└── AActor (things that exist in the world)
    └── APawn (things that can be possessed/controlled)
        └── ACharacter (humanoid pawns with movement)
            └── AGbxCharacter (Gearbox's base character)
                └── AOakCharacter (BL4 player/enemy)
```

Every step adds capabilities. UObject provides basic memory management and reflection. AActor adds world position and component attachment. APawn adds controller possession. ACharacter adds a skeletal mesh and movement component. By the time you reach AOakCharacter, you have a fully-featured game entity with health, weapons, skills, and AI hooks.

4.2. The 40-Byte Header

Every UObject begins with the same 40-byte (0x28) header. Recognizing this structure in memory dumps is a core skill.

```
Offset  Size  Field          Purpose
------  ----  -----          -------
0x00    8     VTable         Pointer to virtual function table
0x08    4     ObjectFlags    Flags like RF_Transient, RF_Public
```

```
0x0C    4     InternalIndex  Position in the global object array
0x10    8     ClassPrivate   Pointer to this object's UClass
0x18    8     NamePrivate    FName (index into name pool)
0x20    8     OuterPrivate   Parent object (package, owner)
```

When you see a pointer in memory and want to know if it's a valid UObject, check if the pointer at offset 0x10 (ClassPrivate) points to something reasonable. If that pointer's object also has a sensible structure, you're probably looking at a real UObject.

The `InternalIndex` at 0x0C is particularly useful—it tells you where this object lives in Unreal's global tracking array, which we'll explore shortly.

4.3. How Unreal Stores Names

Storing the string "Damage" every time a weapon references that property would waste enormous amounts of memory. Instead, Unreal uses a global name pool called `GNames` (or `FNamePool`).

An `FName` isn't a string—it's an 8-byte value containing an index into this pool:

```
FName (8 bytes)
├── Bits 0-31:  ComparisonIndex (which name in the pool)
└── Bits 32-63: Number (instance suffix, like "Actor_5")
```

When Unreal needs the actual string, it looks up the index in the name pool. The pool itself is organized as a chunked array—multiple blocks of entries, where each entry stores the actual characters:

```
FNameEntry
├── Header (2 bytes): bit 0 = is_wide, bits 6-15 = length
└── Characters (N bytes): the actual string data
```

The first few names in any Unreal game are always the same: "None" at index 0, then "ByteProperty", "IntProperty", and so on. This predictability helps locate the name pool in memory—search for the byte sequence representing "None\0ByteProperty" and you're close to GNames.

4.4. Reflection: How Unreal Knows Itself

The reflection system is what makes Unreal games remarkably inspectable. At runtime, Unreal knows the name, type, and offset of every field in every class. This isn't magic—it's data structures describing data structures.

Every class has a `UClass` object that describes it. UClass inherits from `UStruct`, which contains the property definitions. Each property (`FProperty`) knows its name, its byte offset within the owning struct, its type, and various flags.

The key fields in UStruct:

```
class UStruct {
    UStruct* Super;             // Parent class (offset ~0x40)
    FField* ChildProperties;    // First property in linked list (offset ~0x50)
    int32 PropertiesSize;       // Total byte size of all properties
};
```

And each FProperty in the linked list:

```
class FProperty {
    FField* Next;               // Next property in chain (offset ~0x18)
    FName NamePrivate;          // Property name (offset ~0x20)
    int32 Offset_Internal;      // Byte offset in struct (offset ~0x4C)
    int32 ElementSize;          // Size of one element
    // ... type-specific data follows
};
```

To parse a weapon's damage value, you don't hardcode "damage is at offset 0x48." Instead, you find the Weapon class, walk its property chain until you find one named "Damage," read its offset, and use that. This approach survives game patches that shuffle memory layouts.

4.5. The Global Object Array

Unreal tracks all live UObjects in a global array called `GUObjectArray`. This is your index to everything in the game's memory.

The array is chunked—multiple blocks of ~65536 entries each. Each entry (`FUObjectItem`) is 24 bytes:

```
FUObjectItem (0x18 bytes)
├── Object pointer (8 bytes): the actual UObject
├── Flags (4 bytes): item-level flags
├── ClusterRootIndex (4 bytes): clustering info
└── SerialNumber (4 bytes): for weak references
```

To enumerate all objects of a specific class:

1. Get the chunk pointer from GUObjectArray
2. Read the element count
3. For each element, read the Object pointer
4. Read the object's ClassPrivate pointer
5. Resolve the class's name via FName lookup
6. If it matches your target class, you found one

In BL4, with the game running, you might find thousands of objects: hundreds of AWeapon instances, dozens of AOakCharacter instances, and tens of thousands of supporting objects like damage components and inventory slots.

4.6. Pak Files: Where Assets Live

On disk, Unreal games store assets in `.pak` archives. BL4 uses UE5's IoStore format, which splits the data across multiple files:

`.utoc` (Table of Contents): An index listing every asset, its location, size, and compression info.

`.ucas` (Container Archive): The actual compressed asset data, referenced by the utoc.

`.pak` (Legacy Format): Some assets still use the older format for compatibility.

Inside these archives, individual assets follow the Zen package format:

```
Zen Package
├── Summary (package metadata)
├── Name Map (local FNames used in this package)
├── Import Map (external assets this package references)
├── Export Map (objects defined in this package)
└── Export Data (serialized property data)
```

The export data contains the actual property values, but here's the catch: UE5 uses "unversioned" serialization. Properties are written without their names or types—just raw values in order. To parse them, you need external schema information.

4.7. Usmap: The Rosetta Stone

A `.usmap` file contains the schema needed to parse unversioned assets. It's essentially a dump of the reflection system: all class names, property names, types, and offsets.

Without usmap:

```
Raw bytes: 42 48 00 00 40 1C 00 00 ...
Meaning: ???
```

With usmap:

```
Damage (f32): 50.0
Level (u32): 7200
ElementType (enum): Fire
...
```

The usmap format is straightforward:

```
Header
├── Magic: 0x30C4
├── Version: 3 (most recent)
├── Compression: 0=None, 1=Oodle, 2=Brotli, 3=ZStd
├── CompressedSize / DecompressedSize
└── Payload
    ├── Names array (all string names)
    ├── Enums array (enum definitions)
    └── Structs array (class/struct definitions with properties)
```

The bl4 project generates usmap files from memory dumps. Our current usmap contains 64,917 names, 2,986 enums, and 16,849 struct definitions. This covers essentially every data structure BL4 uses.

4.8. Common UE5 Data Types

Certain types appear everywhere in Unreal. Recognizing them speeds up analysis.

TArray (16 bytes): Dynamic arrays.

```
├── Data pointer (8 bytes): heap allocation
├── Count (4 bytes): current elements
└── Max (4 bytes): allocated capacity
```

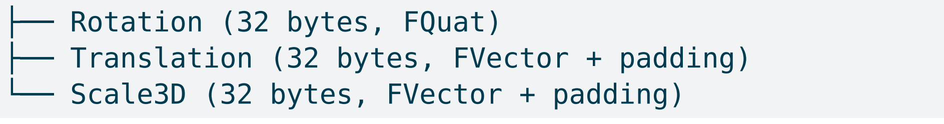
FString (16 bytes): Dynamic strings (internally a TArray). When serialized: length as i32 (negative means UTF-16), then characters, then null terminator.

FVector (24 bytes in UE5): 3D coordinates.

```
├── X (8 bytes, double)
├── Y (8 bytes, double)
└── Z (8 bytes, double)
```

Note: UE4 used 12-byte vectors with floats. UE5 switched to doubles. This is a common source of parsing errors when adapting UE4 tools.

FTransform (96 bytes): Position + rotation + scale.



```
├── Rotation (32 bytes, FQuat)
├── Translation (32 bytes, FVector + padding)
└── Scale3D (32 bytes, FVector + padding)
```

4.9. BL4's Class Structure

The Gearbox-specific classes follow predictable patterns. AOakCharacter, the player/enemy base class, is about 38KB (0x9790 bytes) and contains:

```
AOakCharacter (inherits AGbxCharacter)
├── ~0x4038: FOakDamageState (0x608 bytes)
├── ~0x4640: FOakCharacterHealthState (0x1E8 bytes)
├── ~0x5F50: FOakActiveWeaponsState (0x210 bytes)
└── ... hundreds more fields
```

AWeapon, the weapon class, runs about 3.4KB (0xD48 bytes):

```
AWeapon (inherits AInventory)
├── ~0xC40: FDamageModifierData (0x6C bytes)
├── ~0xCB8: FGbxAttributeFloat ZoomTimeScale
└── ... damage calculation fields, fire modes, etc.
```

These offsets shift between game patches. The reflection system (or a current usmap) is the authoritative source.

4.10. Walking Memory: A Preview

We'll cover memory analysis properly in Chapter 3, but here's a taste of how Unreal's architecture enables systematic exploration.

To find all weapons in memory:

1. Locate GUObjectArray (in BL4: base + 0x113878f0)
2. Read the chunk pointer and element count
3. For each object in the array:
    - Skip if the object pointer is null
    - Read ClassPrivate at object + 0x10
    - Read the class's FName at class + 0x18
    - Resolve the name through GNames
    - If it's "Weapon" or a subclass, record the address

Once you have weapon addresses, use reflection to find properties:

1. Read ChildProperties from the UClass
2. Walk the linked list of FProperty
3. For each property, read its name and offset
4. Use those offsets to read actual values from the weapon object

This two-phase approach—find objects, then decode them—works for any Unreal game.

4.11. Exercises

Exercise 1: Decode a UObject Header

Given this memory dump:

```
00000000: 50 3A 4F 14 01 00 00 00  00 00 00 02 38 04 00 00
00000010: E0 51 8B 90 01 00 00 00  38 09 00 00 00 00 00 00
00000020: 80 25 6E 91 01 00 00 00
```

What is: 1. The VTable pointer? 2. The InternalIndex? 3. The FName comparison index?

Exercise 2: Trace a Class Hierarchy

Starting from an AOakCharacter object: 1. Read ClassPrivate (offset 0x10) to get the UClass 2. Read Super (offset ~0x40) to get the parent class 3. Continue until Super is null

What classes would you encounter?

Answers

Exercise 1: 1. VTable: 0x00000001144F3A50 (bytes 0-7, little-endian) 2. InternalIndex: 0x00000438 = 1080 (bytes 0x0C-0x0F) 3. FName index: 0x00000938 = 2360 (bytes 0x18-0x1B, lower 32 bits)

Exercise 2:

```
AOakCharacter
└── AGbxCharacter
    └── ACharacter
        └── APawn
            └── AActor
                └── UObject
                    └── (Super = null, stop)
```

4.12. What's Next

You now understand how Unreal organizes its world—UObjects with reflectable properties, tracked in global arrays, stored in pak

files with usmap schemas. This knowledge transforms memory analysis from random exploration into systematic discovery.

Next, we'll put these concepts into practice by analyzing live game memory and extracting data directly from a running instance.

Next: **Chapter 3: Memory Analysis**

5. Chapter 3: Memory Analysis

There's data you can find in files, and there's data you can only find in memory. Your character's current health, the weapon in your hand, the damage numbers floating off enemies—these exist only while the game runs. To see them, we need to look inside the running process.

This chapter teaches you to capture and navigate game memory. It's where theory becomes practice, where the patterns from Chapter 2 appear as real bytes you can read and interpret.

5.1. Why Look at Memory?

Files are static. They contain assets, definitions, and saved states. But games are dynamic. At any moment, hundreds of objects exist in memory that don't correspond to any file: spawned enemies, equipped items, active effects, player stats.

Memory analysis lets you:

See decrypted data. Save files are encrypted, but the game has to decrypt them to use them. In memory, everything is plaintext.

Find runtime structures. The reflection system that tells us property offsets? It's in memory. The global arrays tracking every object? Memory. The name pool mapping indices to strings? Memory.

Watch live changes. Change your health in-game and watch the memory value update. This confirms your understanding and reveals how systems connect.

Extract type information. The usmap files that make pak parsing possible come from dumping the reflection system out of memory. There's no other source for this data.

5.2. Capturing a Memory Dump

The first step is getting the game's memory into a file you can analyze offline. The process differs by platform, but the result is the same: a multi-gigabyte file containing everything the game had loaded.

On Windows, use Process Hacker or Sysinternals procdump. Right-click the game process, create a full memory dump. Expect 15-25 GB for BL4.

On Linux with Proton, use gcore. Find the wine64-preloader process ID, then:

```
sudo gcore -o bl4_dump $(pgrep -f wine64-preloader)
```

The dump takes a minute or two. When it finishes, you have a snapshot of everything—every weapon, every enemy, every byte of game state frozen in time.

5.3. Virtual Memory and Address Space

When you see an address like `0x1513878f0`, that's a virtual address. It doesn't map directly to physical RAM—the operating system and CPU handle translation. What matters for reverse engineering is understanding the layout.

BL4 on Windows loads at a base address around `0x140000000`. From there:

```
0x140000000 - 0x14e61c000   Executable code (.text section)
0x14e61c000 - 0x15120e000   Read-only data (.rdata)
0x15120e000 - 0x15175c000   Writable data (.data, .bss)
(varying addresses)         Heap allocations
(varying addresses)         Thread stacks
```

The code section contains compiled game logic. Read-only data holds strings, vtables, and constants. Writable data contains global variables—including the crucial GNames and GUObjectArray pointers we need. Heap allocations hold dynamic objects like weapons and characters.

Knowing these ranges helps validate pointers. If you think you've found a vtable pointer but it points to 0x300000000 (not in any valid range), you know you've misinterpreted something.

5.4. Finding the Global Structures

Every Unreal game has two critical global structures we need to locate:

GNames (FNamePool): The string pool where all names live. Without it, we can't resolve FName indices to actual strings.

GUObjectArray: The master list of all UObjects. It's our index to everything in the game.

The FNamePool has a predictable signature. The first few entries are always "None", "ByteProperty", "IntProperty", and so on—Unreal's built-in types. Searching for the byte sequence "None\0ByteProperty" gets you close.

The bl4 tools automate discovery:

```
bl4 memory --dump bl4_dump.core discover gnames
# Output: Found FNamePool at 0x1512a1c80

bl4 memory --dump bl4_dump.core discover guobjectarray
# Output: Found GUObjectArray at 0x1513878f0
```

Once you have these addresses, everything else becomes accessible. Need to find all weapons? Walk GUObjectArray, resolve each object's class name through GNames, filter for "Weapon". Need to know a property's offset? Find the UClass, walk its property chain, resolve names through GNames.

5.5. Following Pointer Chains

Most interesting data requires following multiple pointers. Think of it as a treasure map where each step reveals the next.

```
flowchart LR
    A["GUObjectArray"] -->|"chunk[0]"| B["Object Item\n(24 bytes)"]
    B -->|"ptr at +0x00"| C["UObject\n(40 bytes)"]
    C -->|"ClassPrivate\n+0x10"| D["UClass"]
    C -->|"property offset"| E["Target Value\n(damage, health, etc.)"]
```

To find a weapon's damage value, the chain might be:

1. Start at GUObjectArray (known address)
2. Read the chunk pointer at offset 0x00
3. Calculate item offset: `chunk_ptr + (object_index * 24)`
4. Read the object pointer from the item
5. Read ClassPrivate at object + 0x10
6. Verify the class name is "Weapon"
7. Read the damage value at weapon + 0xC40

Each read requires careful interpretation. Is this a 4-byte integer or an 8-byte pointer? Little-endian, remember, so `78 56 34 12` is actually 0x12345678.

The bl4 tools handle this:

```
# Read 64 bytes at GUObjectArray
bl4 memory --dump bl4_dump.core read 0x1513878f0 --size 64

# Output shows the chunk pointer and element count
```

5.6. Recognizing Patterns

After enough time in hex dumps, certain patterns become instant recognition.

UObjects start with a vtable pointer (usually 0x140xxxxxx or 0x141xxxxxx), followed by flags at +0x08, an internal index at +0x0C, class pointer at +0x10, FName at +0x18, and outer at +0x20. If you see that 40-byte header pattern, you're looking at a UObject.

TArrays are 16 bytes: a data pointer (or null), then a 4-byte count, then a 4-byte capacity. The count is always less than or equal to capacity. Capacity is often a power of 2.

Floats have recognizable patterns too. The value 1.0 is always `00 00 80 3F`. The value 100.0 is `00 00 C8 42`. When you're looking at unknown data and see `00 00 80 3F`, you've probably found a float field with value 1.0.

```
Common float patterns:
0x3F800000 = 1.0    (scale, multiplier, percentage)
0x40000000 = 2.0    (damage multiplier, maybe)
0x42C80000 = 100.0  (health, percentage base)
0x43480000 = 200.0  (max values)
```

5.7. The bl4 Memory Commands

The bl4 project provides commands for common memory operations:

Reading raw memory:

```
bl4 memory --dump bl4_dump.core read 0x1513878f0 --size 64
```

Looking up FNames by string:

```
bl4 memory --dump bl4_dump.core fname-search "Damage"
```

Generating a usmap:

```
bl4 memory --dump bl4_dump.core dump-usmap
# Creates mappings.usmap with all reflection data
```

Searching for strings:

```
bl4 memory --dump bl4_dump.core scan-string "DAD_AR.part_body" -B 128 -A 128
```

Looking up an FName by index:

```
bl4 memory --dump bl4_dump.core fname 12345
```

These commands encapsulate the pointer-chasing and interpretation logic, letting you focus on what you're trying to find rather than how to read it.

5.8. Practical Example: Finding Item Serials

Item serials—the internal strings that encode each weapon in save data—exist in memory as raw character sequences. Finding them reveals where item data lives.

```
# Search for the @Ug prefix that starts all serials
grep -boa '@Ug' bl4_dump.core | head -10
```

Each hit is a potential item. Examine the surrounding bytes:

```
# Look at context around a hit
xxd -s 0x14d21a8 -l 128 bl4_dump.core
```

You'll see the serial string plus surrounding metadata—maybe a length prefix, maybe pointers to other item data. Compare multiple items at similar offsets to understand the structure.

5.9. Comparing Memory States

One of the most powerful techniques is differential analysis. Take two dumps, change one thing in-game, and compare.

Scenario: You want to find where item level is stored.

1. Take a dump with a level 50 weapon equipped
2. Use an in-game mechanic to change its level (or edit the save)
3. Take another dump with the same weapon at level 51

4. Compare the memory regions around where you found the item

```
# Extract item regions from both dumps
dd if=dump1.core of=item1.bin bs=1 skip=$((0x14d21a8)) count=$((256))
dd if=dump2.core of=item2.bin bs=1 skip=$((0x14d21a8)) count=$((256))

# Compare
xxd item1.bin > item1.txt
xxd item2.bin > item2.txt
diff item1.txt item2.txt
```

The bytes that changed between dumps are candidates for the level field. Usually only a few bytes differ, making the answer obvious.

5.10. Validating Pointers

Not every 8-byte sequence is a valid pointer. Invalid pointers lead to wrong interpretations. Always validate.

A valid pointer in BL4: - Is above 0x10000 (below that is guard pages) - Is below 0x800000000000 (user space limit) - Points to mapped memory (not arbitrary addresses)

For vtable pointers specifically: - The pointer itself should be in the data section - The first entry it points to should be in the code section

```
// Pseudocode for vtable validation
let vtable_ptr = read_u64(object_addr);
if vtable_ptr < 0x150000000 || vtable_ptr > 0x155000000 {
    // Not in .rdata where vtables live
    return false;
}

let first_entry = read_u64(vtable_ptr);
if first_entry < 0x140001000 || first_entry > 0x14e61c000 {
    // First vtable entry should point to code
    return false;
}
// Probably a valid UObject
```

5.11. Dealing with ASLR

Address Space Layout Randomization means base addresses change each time the game launches. The code section might be at 0x140000000 one session and 0x150000000 the next.

The solution: work with offsets from the PE base rather than absolute addresses. GNames isn't at 0x1512a1c80—it's at base + 0x112a1c80. The base address is easy to find (it's where the PE header is), and offsets remain constant across sessions.

When you discover a useful address, record it as an offset. When you need it next session, add the current base address.

5.12. Wine/Proton Considerations

If you're running BL4 through Proton on Linux, memory layouts differ slightly from native Windows. The dump format is ELF rather than MDMP. Address mappings may need translation.

The bl4 tools handle both formats, but be aware that tutorials and tools written for native Windows analysis may need adaptation. The concepts are identical; the mechanics differ slightly.

5.13. Exercises

Exercise 1: Find Your Steam ID

Your Steam ID is used to encrypt save files. It exists in memory while the game runs.

1. Take a memory dump while logged in
2. Search for your Steam ID as an ASCII string (it's a 17-digit number starting with 7656119)
3. Note the addresses where it appears
4. Consider: why might it appear in multiple places?

Exercise 2: Count Inventory Items

Find how many items are in your inventory:

1. Locate the player's inventory structure (hint: it's a TArray)
2. Read the Count field at offset +0x08
3. Compare with your in-game inventory count

Exercise 3: Track a Value Change

Pick something easy to change in-game (ammo count, for example):

1. Take a dump with ammo at some value
2. Fire a shot (or reload, depending on direction)
3. Take another dump
4. Search both dumps for the old and new values
5. Compare regions around the hits

5.14. What's Next

Memory analysis reveals runtime state, but most players interact with the game through save files. Those files are encrypted, compressed, and structured in a specific format.

Next, we'll crack open BL4 save files—understanding the encryption, decompression, and YAML structure that makes save editing possible.

Next: Chapter 4: Save File Format

Part II.

Game Formats

6. Chapter 4: Save File Format

Your entire Borderlands 4 experience—hundreds of hours, thousands of items, every skill point and completed mission—lives in a handful of files. These saves are encrypted, compressed, and structured in ways that seem designed to keep you out. But once you understand the layers, editing them becomes straightforward.

This chapter peels back those layers. We'll decrypt the encryption, decompress the compression, and find human-readable YAML waiting underneath.

6.1. Finding Your Saves

Save files live in predictable locations. On Linux with Proton, they're in your Steam compatdata folder (not the userdata folder):

```
~/.local/share/Steam/steamapps/compatdata/1285190/pfx/drive_c/users/steamuser/
  Documents/My Games/Borderlands 4/Saved/SaveGames/<steam_id>/Profiles/
├── profile.sav          # Your main profile (bank, golden keys, unlocks)
├── client/
│   ├── 1.sav            # Character slot 1
```

```
│   ├── 2.sav            # Character slot 2
│   ├── 3.sav            # Character slot 3
│   ├── 4.sav            # Character slot 4
│   └── 5.sav            # Character slot 5
└── ...
```

On Windows, they're typically in:

```
%USERPROFILE%\Documents\My Games\Borderlands 4\Saved\SaveGames\<steam_
```

Your Steam ID is a 17-digit number starting with 7656119. The game syncs these to Steam Cloud. When editing saves, temporarily disable cloud sync to prevent the game from overwriting your modifications or vice versa.

6.2. The Three Layers

BL4 saves are an onion. The outer layer is AES-256-ECB encryption. Peel that away, and you find zlib compression. Decompress that, and you reach YAML—the actual save data in a format you can read and edit.

```
flowchart LR
    A[".sav file"] -->|AES-256-ECB| B["Encrypted blob"]
    B -->|zlib decompress| C["Compressed data"]
    C -->|Parse| D["YAML document"]
```

To edit a save, you reverse this process: decrypt, decompress, edit the YAML, compress, encrypt. The bl4 tools handle the first four steps automatically. Let's understand each layer.

6.3. The Encryption Layer

Save files are encrypted from byte 0 — there's no plaintext header. The entire file is AES-256-ECB encrypted, then zlib compressed underneath.

AES-256-ECB means: - **AES**: Advanced Encryption Standard, the industry standard block cipher - **256**: 256-bit key (32 bytes) - **ECB**: Electronic Codebook mode, where each 16-byte block is encrypted independently

ECB mode is considered weak for security purposes—identical plaintext blocks produce identical ciphertext blocks, revealing patterns. But for save files, it doesn't matter. The goal isn't Fort Knox security; it's preventing casual tampering. And once you know the key derivation, the encryption is no obstacle at all.

6.4. Key Derivation: Your Steam ID Is the Key

The encryption key is derived from your Steam ID. Not generated randomly, not fetched from a server—just computed from a number you can easily find.

The process:

1. Start with a 32-byte base key (constant for all players)
2. Take your Steam ID as a 64-bit integer
3. Convert to 8 bytes, little-endian
4. XOR those 8 bytes with the first 8 bytes of the base key
5. Result: your personal 32-byte encryption key

```
const BASE_KEY: [u8; 32] = [
    0x35, 0xEC, 0x33, 0x77, 0xF3, 0x5D, 0xB0, 0xEA,  // XOR'd with Ste
    0xBE, 0x6B, 0x83, 0x11, 0x54, 0x03, 0xEB, 0xFB,
    0x27, 0x25, 0x64, 0x2E, 0xD5, 0x49, 0x06, 0x29,
    0x05, 0x78, 0xBD, 0x60, 0xBA, 0x4A, 0xA7, 0x87,
];

fn derive_key(steam_id: u64) -> [u8; 32] {
    let mut key = BASE_KEY;
    let steam_bytes = steam_id.to_le_bytes();
    for i in 0..8 {
        key[i] ^= steam_bytes[i];
    }
    key
}
```

Your Steam ID is a 17-digit number starting with 7656119. You can find it in your Steam profile URL, in the save file path, or in memory while the game runs. The bl4 tools require this ID to decrypt your saves.

6.5. The Compression Layer

Decrypt the payload and you'll find bytes starting with `78 9C`—the signature of zlib compression with default settings.

Zlib is straightforward. Every programming language has libraries for it. Decompress, and you get raw YAML text.

The compression is effective. A 500KB save might decompress to several megabytes of YAML. All that inventory data, skill trees, mission progress—it compresses well because YAML has lots of repeated structure.

6.6. The YAML Structure

Underneath everything, BL4 saves are YAML documents. Human-readable, text-based, editable with any text editor. This is where the interesting data lives.

6.6.1. Character Save Structure

Character saves (1.sav through 5.sav) contain all character-specific data:

```yaml
state:
  char_guid: EAFFA60B46492388B1ED398074375950
  class: Char_Paladin              # Char_Paladin, Char_DarkSiren, etc.
  char_name: Amon
  player_difficulty: Easy
```

```yaml
experience:
  - type: Character
    level: 50
    points: 3430207
  - type: Specialization
    level: 3
    points: 3084

inventory:
  items:
    backpack:
      slot_0:
        serial: '@Uge8Cmm/%Dy!gy?;m8e7QLd...'
        flags: 1
        state_flags: 513
      slot_1:
        serial: '@Ugr$)Nm/)}}!eIEIM^$QlZ...'
        flags: 1
        state_flags: 1
      # ... up to slot_21 or more depending on backpack SDUs

  equipped_inventory:
    equipped:
      slot_0:                    # Primary weapon 1
        - serial: '@Ugd77*Fg_4r=3dZfRG}KRs6...'
          flags: 1
          state_flags: 517
      slot_1:                    # Primary weapon 2
        - serial: '@UgxFw!3C0H^%<l*)jVe^47S...'
          flags: 1
          state_flags: 517
      slot_2:                    # Primary weapon 3
        - serial: '@Ugct)%Fg_4rU>wkBRG/`es7...'
```

```yaml
        flags: 1
        state_flags: 517
    slot_3:                       # Primary weapon 4
      - serial: '@Ugydj=3C0H^Ow0rtjVjck61...'
        flags: 1
        state_flags: 517
    slot_4:                       # SHIELD SLOT
      - serial: '@Uge9B?m/)}}!tjfrM>VQ_Z$...'
        flags: 1
        state_flags: 1
    slot_5:                       # Additional weapon/item
      - serial: '@Ugr$fEm/%P$!f1b>P^eCgL6...'
        flags: 1
        state_flags: 517
    slot_6:                       # Gear slot (varies)
      - serial: '@Ugr$xKm/)}}!pQufM-}RPG}...'
        flags: 1
        state_flags: 3
    slot_7:                       # Gear slot (varies)
      - serial: '@Uge8Usm/)}}!sNQ3NWCv7s8...'
        flags: 1
        state_flags: 1
    slot_8:                       # Class mod slot
      - serial: '@Ug!pHG2}TYgOpFIQhx*jtRN...'
        flags: 1
        state_flags: 3

equip_slots_unlocked:
  - 2
  - 3
  - 6
  - 7
  - 8
```

```yaml
    active_slot: 2                  # Currently selected weapon slot

  currencies:
    cash: 44971
    eridium: 210
    golden_key: shift

  ammo:
    assaultrifle: 0
    pistol: 148
    shotgun: 40
    smg: 0
    sniper: 47
    repairkit: 10

  checkpoint_name: World_P.RS_Grasslands_ClaptrapBeach
  total_playtime: 4050.224121

globals:
  time_of_day: Day
  prologue_completed: TRUE
  mainmissioncomplete: TRUE
  # ... mission flags, unlocks, etc.

stats:
  achievements:
    00_level_10: 1
    01_level_30: 1
    # ... achievement tracking
```

6.6.2. Equipped Slot Mapping

The `equipped_inventory.equipped` section uses numbered slots:

Slot	Purpose
slot_0	Primary weapon 1
slot_1	Primary weapon 2
slot_2	Primary weapon 3
slot_3	Primary weapon 4
slot_4	**Shield**
slot_5	Additional weapon slot
slot_6	Gear slot
slot_7	Gear slot
slot_8	Class mod

6.6.3. State Flags

The `state_flags` field is a bitmask indicating item status and labels:

Bit Definitions (verified in-game):

Bit	Value	Meaning
0	1	Item exists/valid (always set)
1	2	Favorite
2	4	Junk
4	16	Label 1
5	32	Label 2
6	64	Label 3
7	128	Label 4

Bit	Value	Meaning
9	512	Backpack only (NOT equipped)

Note: Favorite, Junk, and Labels 1-4 are mutually exclusive—only one can be set at a time.

How Equipping Works:

When you equip an item, its serial is **copied** from `inventory.items` to `equipped_inventory.equipped`. Both copies keep bit 9 **clear** (0) to indicate the item is equipped. When unequipped, bit 9 is **set** (1) on the backpack copy and the equipped_inventory copy is removed.

Common Values:

Value	Binary	Meaning
1	`0000000001`	Equipped item
3	`0000000011`	Equipped item + favorite
513	`1000000001`	Backpack only (not equipped)
515	`1000000011`	Backpack only + Favorite
517	`1000000101`	Backpack only + Junk
529	`1000010001`	Backpack only + Label 1
545	`1000100001`	Backpack only + Label 2
577	`1001000001`	Backpack only + Label 3
641	`1010000001`	Backpack only + Label 4

Items in inventory appear as serials—those Base85-encoded strings we'll decode in Chapter 5. Each item also has `flags` (various item properties) and `state_flags` (the bitmask above).

6.7. Working with Saves

The bl4 tools make save editing straightforward.

Decrypt a save to YAML:

```
bl4 save decrypt 1.sav character.yaml
# or use stdout
bl4 save decrypt 1.sav > character.yaml
```

Edit the YAML with any text editor. Add items, change currency, modify stats.

Re-encrypt:

```
bl4 save encrypt character.yaml 1.sav
```

Or use the interactive editor (decrypts, opens in `$EDITOR`, re-encrypts on save):

```
bl4 save edit 1.sav
```

The Steam ID is configured once and stored, so you don't need to specify it each time.

6.8. Item Injection

To add items to a save, you need to:

1. **Decrypt the save**
2. **Add the item serial to the appropriate location**
3. **Re-encrypt the save**

6.8.1. Adding to Backpack

Add a new slot entry under `state.inventory.items.backpack`:

```yaml
backpack:
  slot_0:
    serial: '@Uge8Cmm/...'
    flags: 1
    state_flags: 513
  # Add new item as the next slot number
  slot_22:
    serial: '@Uge92<m/)}}!gNodNkyuCbwInLxgj=C`_2FW'
    state_flags: 513
```

6.8.2. Equipping an Item

Important: The equipped_inventory is a *reference* to a backpack item. To equip an item:

1. First, add the item to the backpack
2. Then, add a reference to the same item in equipped_inventory

```
# Step 1: Add to backpack (bit 9 clear = equipped)
state:
  inventory:
    items:
      backpack:
        slot_22:
          serial: '@Uge8jxm/)@{!bAp5s!;381FF>eS^@w'
          flags: 1
          state_flags: 1    # Equipped (bit 9 = 0)

# Step 2: Add to equipped_inventory (same serial, same flags)
    equipped_inventory:
      equipped:
        slot_4:               # Shield slot
          - serial: '@Uge8jxm/)@{!bAp5s!;381FF>eS^@w'
            flags: 1
            state_flags: 1    # Equipped
```

The same serial appears in both places—the backpack holds the actual item data, and equipped_inventory references it.

Critical: Only ONE item per slot type can have `state_flags: 1` (equipped). If you have multiple shields all marked as equipped, the game will refuse to equip any of them. Make sure all other shields in your backpack have `state_flags: 513` (backpack only, not equipped).

6.8.3. Live Editing Limitations

The game **caches character data in memory** once loaded. This means:

- Editing a save file on disk has **no effect** until the game restarts
- Switching characters doesn't reload from disk—the cache persists
- You must **fully quit and restart** the game to see save edits

Workflow for save editing: 1. Quit the game completely 2. Edit the save file 3. Restart the game 4. Load the character

Warning: Never edit a save for a character you've already loaded this session—your edits will be ignored and potentially overwritten when the game saves.

6.9. Common Edits

Adding currency:

```yaml
state:
  currencies:
    cash: 999999999
    eridium: 9999
```

Changing character name:

```yaml
state:
  char_name: NewName
```

Changing character level requires updating experience points to match:

```yaml
state:
  experience:
    - type: Character
      level: 50
      points: 3430207
```

Known character XP thresholds:

Level	XP Required	Source
1	0	exact
2	1,100	exact
30	821,362	exact
50	3,430,207	exact
55	~4,407,000	estimated
60	~5,710,000	tested

For levels 1-50, the curve follows approximately `XP ≈ 202 × level^2.44`.

For DLC levels 51-60, the XP-per-level bar grows linearly: each level requires roughly `167,900 + 13,700 × (level - 50)` more XP than the previous. Estimates were derived from in-game save editing tests and are accurate to within ~1%.

Specialization levels use separate XP tracked independently:

Level	XP Required
2	~1,265
3	~2,599
4	~4,690
5	~7,948
6	~12,718

Adding items requires valid serials. You can copy serials from other saves, the items database, or generate them (once you understand the format from Chapter 5):

```yaml
state:
  inventory:
    items:
      backpack:
        slot_22:
          serial: '@UgYOUR_ITEM_SERIAL_HERE'
          state_flags: 513
```

Invalid serials cause problems—items may not appear, or the game might crash. Always test with a backup save.

6.10. Map Exploration Data (foddatas)

The `foddatas` section stores your map exploration progress — the areas you've uncovered as you explore each zone. "FOD" stands for Fog of Discovery: the fog overlay on the in-game map that clears as you move through an area.

6.10.1. Structure

```yaml
fodsaveversion: 2
foddatas:
  - levelname: World_P
```

```yaml
    foddimensionx: 128
    foddimensiony: 128
    compressiontype: Zlib
    foddata: eJztW3tYTVkb37kMkec...  # Base64-encoded zlib data
  - levelname: Fortress_Grasslands_P
    foddimensionx: 128
    foddimensiony: 128
    compressiontype: Zlib
    foddata: eJztm3k8lO0axv...
  # ... one entry per visited zone
```

Each zone entry contains:

Field	Description
`levelname`	Internal zone identifier (e.g., `World_P`, `Fortress_Grasslands_P`)
`foddimensionx`	Grid width (always 128 in current game version)
`foddimensiony`	Grid height (always 128 in current game version)
`compressiontype`	Always `Zlib`
`foddata`	Base64-encoded, zlib-compressed fog alpha map

6.10.2. Fog Alpha Map Format

The `foddata` payload decodes to a **128x128 grayscale image** — one byte per cell, stored as a flat row-major array. Each byte is a fog alpha value:

Value	Meaning
0	Fully fogged (unexplored)
1–254	Partially revealed (gradient at fog boundary)
255	Fully revealed (explored)

To decode:

1. Base64-decode the `foddata` string
2. Zlib-decompress the result
3. Read the output as a flat array of `foddimensionx * foddimensiony` bytes
4. Index as `grid[row * foddimensionx + col]`

The decompressed size is always exactly 16,384 bytes (128 * 128). A fully-explored zone uses all 256 possible values — the intermediate values (1–254) form smooth gradients at the edges of explored regions, matching the soft feathered fog boundary visible on the in-game map.

> **ℹ Not a Bitmask**
>
> Despite the name "exploration bitmap" in early documentation, the format is not binary explored/unexplored. It's a full 8-bit alpha channel. The game renders this as a texture overlay on the zone's minimap. Cells near the player transition from 0 to 255 with a radial falloff, producing the familiar soft-edged fog reveal.

A fully-explored save has roughly 30–40% of cells at 255, with another 5–10% at intermediate values and the rest at 0 (out-of-bounds or unreachable areas). A fresh character's zones are almost entirely zeros, compressing down to a few hundred bytes.

6.10.3. Zone Names

Level Name	In-Game Zone
`Intro_P`	Tutorial area
`World_P`	Main open world hub
`Fortress_Grasslands_P`	Grasslands region
`Fortress_Shatteredlands_P`	Shattered Lands region
`Fortress_Mountains_P`	Mountains region
`ElpisElevator_P`	Elpis elevator zone
`Elpis_P`	Moon base
`UpperCity_P`	Upper city
`Vault_Grasslands_P`	Grasslands vault
`Vault_ShatteredLands_P`	Shattered Lands vault
`Vault_Mountains_P`	Mountains vault
`Raid1_P`	Raid zone

Only visited zones appear in the save. A new character has no entries; a completionist save has all 12.

6.10.4. Manipulating FOD Data

To reveal an entire zone, generate a 16,384-byte array filled with `0xFF`, zlib-compress it, and base64-encode the result. To hide everything, do the same with `0x00`.

To copy exploration data from one character to another, extract the entire `foddatas` block (including `fodsaveversion`) and replace it in the target save:

```
bl4 save decrypt -i source.sav -o source.yaml
bl4 save decrypt -i target.sav -o target.yaml

# Copy foddatas section (use text manipulation or YAML tools)
# Then re-encrypt
bl4 save encrypt -i target_modified.yaml -o target.sav
```

6.11. Safehouse and World Progress

The `openworld` section tracks your progression through the open world activities: safehouses captured, silos cleared, bounties completed, and collectibles found.

6.11.1. Safehouses

```yaml
openworld:
  activities:
    safehouses:
      safehouse_grasslands_1: 1
      safehouse_grasslands_3: 1
      safehouse_grasslands_4: 1
      safehouse_mountains_1: 1
      safehouse_mountains_2: 1
      safehouse_mountains_3: 1
      safehouse_mountains_4: 1
      safehouse_shatteredlands_2: 1
```

```
    safehouse_city_1: 1
    safehouse_city_3: 1
```

A value of `1` indicates the safehouse is captured. Missing entries or `0` means uncaptured.

6.11.2. Silos

```
  silos:
    silo_grasslands_1: 1
    silo_grasslands_2: 1
    silo_grasslands_3: 1
    silo_mountains_1: 1
    silo_mountains_2: 1
    silo_mountains_3: 1
    silo_shatteredlands_1: 1
    silo_shatteredlands_2: 1
    silo_shatteredlands_3: 1
```

6.11.3. Bounties

Three bounty types track different faction activities:

```
  bounties_augur:
    augurbounty_mountains_1: 1
    augurbounty_mountains_2: 1
    augurbounty_shatteredlands_1: 1
  bounties_order:
    orderbounty_grasslands_1: 1
```

```yaml
  orderbounty_grasslands_2: 1
bounties_vanguard:
  vanguardbounty_grasslands_1: 1
  vanguardbounty_mountains_1: 1
```

6.11.4. Collectibles

```yaml
collectibles:
  vaultsymbols:
    vaultsymbol_grasslands_4: 1
    vaultsymbol_grasslands_5: 1
  shrines:
    shrine_mountains_10: 1
  safes:
    safe_shatteredlands_10: 1
  echologs_general:
    el_g_grasslands:
      gra_gen_02: 1
      gra_gen_10: 1
      gra_mis_04: 1
```

6.12. Unlockables

The `unlockables` section tracks cosmetic items and vehicle customizations you've collected.

6.12.1. Hoverdrive Skins

```yaml
unlockables:
  unlockable_hoverdrives:
    entries:
      - unlockable_hoverdrives.jakobs_01
      - unlockable_hoverdrives.daedalus_01
      - unlockable_hoverdrives.jakobs_03
      - unlockable_hoverdrives.borg_03
      - unlockable_hoverdrives.vladof_01
      - unlockable_hoverdrives.maliwan_02
      - unlockable_hoverdrives.order_02
      - unlockable_hoverdrives.tediore_01
```

Each entry represents a vehicle skin tied to a manufacturer. The naming pattern is `unlockable_hoverdrives.<manufacturer>_<number>`.

6.12.2. Vault Hunter Rank

```yaml
highest_unlocked_vault_hunter_level: 6
```

This tracks your progression through the Vault Hunter Rank challenges. Values typically range from 1 (starting) to 6 (max rank).

6.13. Backups

Never edit saves without a backup. Before making any changes, copy your save file:

```
# Simple backup
cp 1.sav 1.sav.backup

# Or with timestamp
cp 1.sav "1.sav.$(date +%Y%m%d_%H%M%S).backup"
```

Steam Cloud will also sync your saves. If you're experimenting, disable cloud sync temporarily to prevent conflicts.

6.14. What Can Go Wrong

The game validates saves on load. Here's what happens with various issues:

Invalid YAML syntax: The game won't load the save at all. Usually a crash or error message.

Unknown fields: Generally ignored. The game skips what it doesn't recognize.

Invalid item serials: Items may not appear in inventory, or appear as corrupted/unnamed items.

Out-of-range values: Often clamped to valid ranges. Level 9999 might become level 72 (the cap).

Wrong encryption key: Decryption fails completely—you get garbage instead of zlib-compressed data.

The safest approach: make one change at a time, test immediately, keep backups.

6.15. Manual Decryption Walkthrough

If you want to understand the process without tools, here's a Python walkthrough:

Step 1: Read the encrypted file

```
with open('1.sav', 'rb') as f:
    encrypted = f.read()

# The entire file is encrypted from byte 0
print(f"Encrypted size: {len(encrypted)}")
```

Step 2: Derive the key

```
import struct
STEAM_ID = 76561198012345678  # Replace with yours

BASE_KEY = bytes([
    0x35, 0xEC, 0x33, 0x77, 0xF3, 0x5D, 0xB0, 0xEA,
    0xBE, 0x6B, 0x83, 0x11, 0x54, 0x03, 0xEB, 0xFB,
    0x27, 0x25, 0x64, 0x2E, 0xD5, 0x49, 0x06, 0x29,
    0x05, 0x78, 0xBD, 0x60, 0xBA, 0x4A, 0xA7, 0x87,
])
```

```
steam_bytes = struct.pack('<Q', STEAM_ID)
key = bytearray(BASE_KEY)
for i in range(8):
    key[i] ^= steam_bytes[i]
```

Step 3: Decrypt

```
from Crypto.Cipher import AES

# Pad to 16-byte boundary for AES
padded = encrypted + b'\x00' * (16 - len(encrypted) % 16)

cipher = AES.new(bytes(key), AES.MODE_ECB)
decrypted = cipher.decrypt(padded)[:len(encrypted)]

# Should start with 78 9C (zlib header)
print(f"First bytes: {decrypted[:4].hex()}")
```

Step 4: Decompress

```
import zlib
yaml_data = zlib.decompress(decrypted)
print(yaml_data[:500].decode('utf-8'))
```

At this point, you have the raw YAML. Edit it, then reverse the process: compress with zlib, encrypt with AES-256-ECB using the same key, and write the result directly as the `.sav` file (no header).

6.16. Why ECB Mode?

Security-conscious readers might wonder why Gearbox chose ECB mode, which cryptographers consider weak. ECB's flaw is that identical plaintext blocks produce identical ciphertext blocks, potentially revealing patterns.

For save files, this doesn't matter much. The files contain compressed data (which looks random), and the threat model is "casual tampering," not nation-state adversaries. ECB is simple to implement, requires no IV management, and works fine for this use case.

More importantly for us, ECB makes analysis easier. You can decrypt blocks independently, which simplifies debugging. It's a reasonable engineering tradeoff.

6.17. Exercises

Exercise 1: Decrypt Your Save

Use the bl4 tools to decrypt one of your saves. Examine the YAML structure. Find your character level and current cash.

Exercise 2: Make a Safe Modification

1. Back up your save
2. Decrypt to YAML
3. Add 1000 cash to your total
4. Re-encrypt
5. Load the game and verify

6. Restore the backup

Exercise 3: Find the Item List

Navigate through the decrypted YAML to `state.inventory.items`. Count your items. Notice that each item is just a serial string and some flags. The serial encodes everything—weapon type, parts, level, stats.

6.18. What's Next

You've seen that inventory items are stored as compact serial strings. But what do those strings mean? How does `@Ugr$ZCm/&tH!t{KgK/Shxu>k` encode a complete weapon with manufacturer, parts, level, and random rolls?

The next chapter decodes item serials. It's one of the most intricate pieces of the puzzle, and understanding it unlocks the ability to create, modify, or analyze any item in the game.

Next: Chapter 5: Item Serials

7. Chapter 5: Item Serials

The first time you see an item serial—something like `@Ugr$ZCm/&tH!t{KgK/Shxu>k`—it looks like line noise. Random characters that couldn't possibly mean anything. But that string contains a complete weapon: its manufacturer, every part attached to it, the level, the random seed that determined its stats. Everything needed to reconstruct the item perfectly.

This chapter decodes how serials work. By the end, you'll understand every transformation from that cryptic string to a fully-described weapon.

7.1. What's Encoded in a Serial

A serial is self-contained. Given just the string, the game can reconstruct the item completely—no external references needed. If you copy a serial from one save file into another, the recipient gets an exact duplicate of the weapon. This is an internal encoding detail, not something the game exposes to players, but understanding it is what makes save editing and item analysis possible.

Inside that string: - Item type (weapon, shield, class mod) - Manufacturer - Level - Element type (Kinetic, Corrosive, Shock,

Radiation, Cryo, Fire) - Every part (barrel, grip, scope, magazine) - Random seed for stat calculations - Additional flags (some correlate with rarity in database)

The encoding is compact. A 40-character serial describes an item that would need hundreds of bytes in a more verbose format.

7.2. The Decoding Pipeline

Serials transform through multiple stages. Understanding each stage reveals how the pieces fit together.

```
flowchart LR
    A["@Ugr$ZCm/&tH!..."] -->|Strip prefix| B["gr$ZCm/&tH!..."]
    B -->|Base85 decode| C["[0x84, 0xA5, ...]"]
    C -->|Bit-mirror| D["[0x21, 0xA5, ...]"]
    D -->|Parse bitstream| E["Category: 269\nLevel: 33\nParts: [...]"]
```

The prefix `@U` marks this as a BL4 serial. After stripping the two-character prefix, everything else is Base85-encoded binary data. The character at position 3 (the first Base85 character) varies based on the magnitude of the first encoded value—it is NOT a type discriminator, despite appearing to correlate with item types at first glance.

7.3. Base85: Custom Alphabet

BL4 doesn't use standard ASCII85. It uses a custom 85-character alphabet:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz!#$%&()*+-;<=>?@^_`{/}
```

Every 5 characters encode 4 bytes. The math: $85^5 \approx 4.4$ billion, which fits in 32 bits (4 bytes) with room to spare.

To decode, look up each character's position in the alphabet, combine them as a base-85 number, then extract 4 bytes big-endian:

```
Characters: g r $ Z C
Positions:  42 53 64 35 12

Value = 42×85⁴ + 53×85³ + 64×85² + 35×85 + 12
      = 2,225,440,262

Bytes: [0x84, 0xA5, 0x86, 0x06]
```

7.4. Bit Mirroring: The Obfuscation Layer

After Base85 decoding, each byte gets bit-reversed. 0x87 (binary `10000111`) becomes 0xE1 (binary `11100001`).

```
fn mirror_byte(b: u8) -> u8 {
    let mut result = 0;
    for i in 0..8 {
        if (b >> i) & 1 == 1 {
            result |= 1 << (7 - i);
        }
    }
    result
}
```

7.5. Bit Assembly: MSB Stream, LSB Data

After mirroring, the bytes form a bitstream that's read MSB-first (bit 7 of each byte first). But data values within the stream are encoded LSB-first — the first bit read is the least significant bit of the value.

This means framing bits (prefixes, continuation flags, magic header) work correctly as-is, but multi-bit data values must have their bits reversed after reading. Each VarInt nibble (4 bits) and each VarBit length/value field gets its bits reversed within its width.

For example, reading 4 bits that come out as `1000` (8) from the stream actually represents the value `0001` (1) after reversal. This reversal is applied per-nibble for VarInts and per-field for VarBits.

7.6. Token Parsing: The Real Structure

The first 7 bits must be `0010000` (0x10)—a magic number validating this as a proper serial.

After the magic header, the stream contains tokens identified by prefix bits:

Prefix	Token Type	Purpose
`00`	Separator	Hard boundary between sections
`01`	SoftSeparator	Softer boundary (like commas)
`100`	VarInt	Variable-length integer
`101`	Part	Part reference with optional value
`110`	VarBit	Bit-length-prefixed integer
`111`	String	Length-prefixed ASCII string

VarInt encodes integers in nibbles (4-bit chunks). Each nibble has 4 bits of value plus 1 continuation bit. Keep reading nibbles until the continuation bit is 0. Each nibble's bits are reversed after reading (MSB stream → LSB data).

VarBit starts with a 5-bit length (reversed), then that many bits of data (also reversed). More efficient for known-size values.

Part tokens reference parts by index, optionally with associated values. `{42}` means part index 42, `{42:7}` means part 42 with value 7.

7.7. Item Type: Determined by First Token

The **actual item type** is determined by parsing the first token after the magic header:

First Token	Item Type	What It Contains
VarInt (prefix `100`)	Weapon	Pistols, shotguns, rifles, SMGs, snipers
VarBit (prefix `110`)	Equipment	Shields, grenades, class mods, gadgets

This means two serials with different third characters might represent the same category of item. Always determine type from the bitstream, not the character.

7.8. Two Serial Formats

BL4 uses two distinct token structures, distinguished by the first token after the 7-bit magic header:

7.8.1. Weapon Format (VarInt-first)

Weapons start with a VarInt encoding a combined manufacturer/weapon-type ID:

```
[0] VarInt: manufacturer_weapon_id   (e.g., 4 = Jakobs Pistol, 22 = Ri
[1] SoftSeparator
[2] VarInt: 0
[3] SoftSeparator
```

```
[4] VarInt: 1
[5] SoftSeparator
[6] VarInt: level_code              <- LEVEL ENCODED HERE
[7] Separator
[8] VarInt: 2
[9] SoftSeparator
[10] VarInt: seed                   <- Random seed for stats
[11] Separator
[12] Separator
[13+] Part tokens...
```

7.8.2. Equipment Format (VarBit-first)

Equipment (shields, grenades, class mods) starts with a VarBit encoding the category:

```
[0] VarBit: category_id             <- Category ID directly
[1] SoftSeparator
[2] VarInt: 0
[3] SoftSeparator
[4] VarInt: 1
[5] SoftSeparator
[6] VarInt: level_code              <- LEVEL ENCODED HERE
[7] Separator
[8] VarInt: seed
[9] SoftSeparator
[10] VarInt: (varies)
[11+] More data and parts...
```

For VarBit-first serials, the VarBit value IS the NCS category ID directly. No divisor or formula is needed.

7.9. Part Group IDs (Categories)

The Part Group ID (also called Category ID) determines which part pool to use for decoding. Each ID corresponds to a manufacturer/weapon-type combination.

For weapons, the first VarInt (serial ID) maps directly to the NCS category. The bl4 tools handle this via `serial_id_to_parts_category()`, but for most weapons the serial ID matches the NCS category.

Pistols (2-6):

ID	Manufacturer	Code
2	Daedalus	DAD_PS
3	Jakobs	JAK_PS
4	Order	ORD_PS
5	Tediore	TED_PS
6	Torgue	TOR_PS

Shotguns (7-12):

ID	Manufacturer	Code
7	Ripper	BOR_SG
8	Daedalus	DAD_SG
9	Jakobs	JAK_SG
10	Maliwan	MAL_SG
11	Tediore	TED_SG
12	Torgue	TOR_SG

Assault Rifles (13-18, 27):

ID	Manufacturer	Code
13	Daedalus	DAD_AR
14	Tediore	TED_AR
15	Order	ORD_AR
17	Torgue	TOR_AR
18	Vladof	VLA_AR
27	Jakobs	JAK_AR

Snipers (16, 23-26):

ID	Manufacturer	Code
16	Vladof	VLA_SR
23	Ripper	BOR_SR
24	Jakobs	JAK_SR
25	Maliwan	MAL_SR
26	Order	ORD_SR

SMGs (19-22):

ID	Manufacturer	Code
19	Ripper	BOR_SM
20	Daedalus	DAD_SM
21	Maliwan	MAL_SM
22	Vladof	VLA_SM

7.10. Part Indices Are Context-Dependent

Part token `{4}` doesn't mean the same part across all weapons. The index is relative to the Part Group. Index 4 on a Vladof SMG might be a specific barrel, while index 4 on a Jakobs Pistol is something completely different.

This is why you must decode the Part Group ID first. Without knowing which pool you're indexing into, part tokens are meaningless.

7.11. Level Encoding

Level is encoded as a VarInt at position 6 in the token list for both weapon and equipment formats. The VarInt value is the level directly (e.g., 50 = level 50).

7.12. Element Encoding

Element types are encoded as Part tokens. The element part index maps to the element type through the parts database — element parts have names like `Kinetic`, `Corrosive`, `Fire`, etc.

Multi-element weapons contain multiple element part tokens:

```
Parts: ... Corrosive, Kinetic, Radiation ...  → Three elements
```

7.13. Decoding a Serial Manually

Let's walk through `@Ugr$ZCm/&tH!t{KgK/Shxu>k`:

Step 1: Structure - Prefix: `@U` (stripped) - Base85 data: `gr$ZCm/&tH!...`

Step 2: Base85 decode First 5 characters `gr$ZC`: - Positions: 42, 53, 64, 35, 12 - Value: 2,225,440,262 - Bytes: [0x84, 0xA5, 0x86, 0x06]

Continue for remaining characters.

Step 3: Bit-mirror each byte

```
Original: 84 A5 86 06 ...
Mirrored: 21 A5 61 60 ...
```

Step 4: Parse bitstream

```
Binary: 00100001 10100101 01100001 ...
        └───────┘ └────────────────...
         Magic   Tokens begin
         (0x10)
```

First token after magic: prefix `110` = VarBit - 5-bit length (reversed): 9 - 9 bits of data (reversed): 269

Category = 269 (Vladof Repair Kit)

The bl4 tool handles all this:

```
bl4 serial decode '@Ugr$ZCm/&tH!t{KgK/Shxu>k'
# Vladof Repair Kit (269) ✓
# Level: 33
```

7.14. Decoding Examples

7.14.1. Weapon Serial

Serial: `@UgxFw!2}TYgOs)+YRG}7?s3AisQ8!UBQ8Q6BQDIPXP<2qdQ2P)`

Decoded tokens:

```
22 ,  0 ,  1 ,  50 | 2 ,  3417 | | {1} {2} {5} {4} {1:12} {68} {75} {7
```

- First VarInt (22): Ripper SMG
- 4th VarInt (50): Level 50
- Seed (3417): Random seed after first separator
- Part tokens resolve to barrels, body parts, grips, elements, and legendary composition

7.14.2. Equipment Serial

Serial: `@Ugr%Scm/)}}$pj({qzigfrP>z<v^$y<L5*r(1po`

Decoded tokens:

```
321 ,  0 ,  1 ,  50 | 9 ,  1 | 2 ,  2880 | | {7} {10} {246:24} {237} ,  9 2 | {6} {2
```

- First VarBit (321): Torgue Shield
- Level: 50
- Part tokens: legendary rarity component, body armor, stat modifiers, unique legendary part

---

7.15. Exercises

Exercise 1: Identify Item Types

Given these serials, what category is each? 1. `@Uge8aum/(OZ$pj+I_5#Y(pw{;WbgA{xWRhC/` 2. `@UgxFw!2}TYgOs)+YRG}7?s3AisQ8!UBQ8Q6BQDIPXP<2qdQ2P)` 3. `@Ugr%Scm/)}}$pj({qzigfrP>z<v^$y<L5*r(1po`

Exercise 2: Decode a Manufacturer

Use `bl4 serial decode` on a weapon serial. What Part Group ID does it use? What manufacturer does that correspond to?

Exercise 3: Compare Two Items

Find two similar weapons in your inventory. Decode both serials. Which tokens differ? Can you correlate the differences to visible stats?

Exercise 1 Answers

Decode each serial with `bl4 serial decode`:

1. First token is VarBit (272) → Order Grenade Gadget
2. First token is VarInt (22) → Ripper SMG
3. First token is VarBit (321) → Torgue Shield

7.16. What's Next

Now that we understand how serials encode items, we need to understand the NCS format that stores part definitions, item pools, and loot configuration. NCS is the foundational data format that makes serial decoding meaningful.

Next: Chapter 6: NCS Format

8. Chapter 6: NCS Format (Nexus Config Store)

Try searching BL4's pak files for item pool definitions. For loot configuration tables. For achievement data. You'll find nothing — these classes don't exist as standard `.uasset` files. Gearbox moved critical game configuration into a custom binary format embedded directly in pak chunks, invisible to normal Unreal Engine extraction tools.

That format is NCS: Nexus Config Store. It holds item pools, drop rates, inventory part definitions, actor configurations, achievement data, and hundreds of DataTable serializations. If you want to understand how BL4's loot system works — what can drop, where it drops, which parts are valid for which weapons — you need to understand NCS.

This chapter is the format specification. It covers everything from Oodle-compressed pak chunks down to bit-packed tag values in the binary section. It's long, and deliberately so: NCS is a complex format with multiple encoding strategies, and the details matter when you're writing a parser.

8.1. Overview

NCS stores typed configuration records. Each NCS file has a content type (`achievement`, `inv`, `itempoollist`, `loot_config`, etc.) and contains structured entries with string-valued and binary-encoded fields. The format is space-efficient — strings are differentially encoded, values are bit-packed, and the whole thing is Oodle-compressed before being stored in a pak chunk.

The data exists at two levels:

1. **NCS Chunks** — Oodle-compressed blocks within pak files, located via an NCS manifest
2. **Decompressed Content** — The actual typed configuration data: header, strings, and binary section

Figure 6.1 shows the decompression pipeline. Figure 6.2 shows the layout of decompressed content.

```
flowchart LR
    A[PAK File] --> B[NCS Chunk<br/>16-byte header]
    B --> C{Format Flags}
    C -->|0x00000000| D[Single Block<br/>Oodle decompress]
    C -->|0x03030812| E[Multi Block<br/>256KB blocks]
    D --> F[Decompressed<br/>Content]
    E --> F
```

```
flowchart TB
    A[BlobHeader<br/>16 bytes] --> B[Header Strings<br/>null-terminate
    B --> C[TypeCodeTable<br/>type codes + bit matrix]
    C --> D[String Blocks<br/>3 blocks: values, kinds, keys]
    D --> E[Binary Data<br/>bit-packed tables/records/entries]
```

8.2. Compression

NCS data is Oodle-compressed and wrapped in a two-layer header structure. The outer header describes the NCS chunk; the inner header describes the Oodle compression format.

8.2.1. Outer Header (16 bytes)

```
Offset  Size  Type   Description
------  ----  ----   -----------
0x00    1     u8     Version (always 0x01)
0x01    3     bytes  Magic: "NCS" (0x4e 0x43 0x53)
0x04    4     u32    Compression flag (0 = raw, non-zero = Oodle)
0x08    4     u32    Decompressed size (little-endian)
0x0c    4     u32    Compressed size (little-endian)
```

The version byte has always been `0x01` in every file observed. The compression flag is zero for uncompressed data (rare) and non-zero for Oodle-compressed data (nearly all files).

8.2.2. Inner Header (16+ bytes)

```
Offset  Size  Type   Description
------  ----  ----   -----------
0x00    4     u32    Oodle magic: 0xb7756362 (big-endian)
0x04    4     bytes  Hash/checksum
0x08    4     u32    Format flags (big-endian)
0x0c    4     u32    Block count (big-endian)
```

8.2.3. Format Flags

The format flags determine whether the data is stored as a single Oodle-compressed block or split across multiple blocks:

Flags	Format	Description
`0x00000000`	Single-block	Small files, one Oodle block
`0x03030812`	Multi-block	Large files, 256KB blocks

Single-block is the common case. Multi-block files split the decompressed data into 256KB chunks, each independently Oodle-compressed. The block count field in the inner header gives the number of chunks.

8.2.4. Compatibility

Most NCS files decompress with open-source Oodle implementations. A small number (~2.4% of files) use compression parameters that require the official Oodle SDK:

File	Size	Notes
audio_event0	~18MB	Large audio mappings
coordinated_effect_filter0	~300KB	Effect filters
DialogQuietTime0	~20MB	Dialog timing
DialogStyle0	~370KB	Dialog styling

8.3. Content Format

After decompression, the data follows a structured binary format parsed by the `parse/` module pipeline: `blob.rs` → `typecodes.rs` → `decode.rs`.

8.3.1. BlobHeader (16 bytes)

Every decompressed NCS payload starts with a 16-byte header:

```
Offset  Size  Type   Description
------  ----  ----   -----------
0x00    4     u32    Entry count (little-endian)
0x04    4     u32    Flags
0x08    4     u32    String bytes (total size of header strings)
0x0c    4     u32    Reserved
```

8.3.2. Header Strings

Immediately after the BlobHeader, `string_bytes` worth of null-terminated ASCII strings. The first string is the table type name (e.g., `achievement`, `inv`, `itempoollist`). Remaining strings are dependency table names — for `inv.bin`, these include part slot categories like `inv_comp`, `barrel`, `body`, `element`, `firmware`, etc.

8.3.3. TypeCodeTable

After the header strings, the body section begins with a TypeCodeTable:

```
Offset  Size        Description
------  ----        -----------
0x00    1           type_code_count (number of type code chars)
0x01    2           type_index_count (u16 LE)
0x03    N           type code characters (e.g., 'a', 'b', 'c', 'e', 'f'
0x03+N  ceil(type_index_count * type_code_count / 8) bytes  Bit matrix
```

The type codes are single ASCII characters that describe the tag types used in the binary data section. Each character maps to a bit position in the row flags:

Tag	Bit	Description
`a`	0	Key name (pair_vec string)
`b`	1	U32 value
`c`	2	F32 value
`d`	3	Name list D
`e`	4	Name list E
`f`	5	Name list F
`h`	7	Has-self-key flag (bit 7 of row flags)
`i`	8	(format-specific)
`j`	9	(format-specific)
`l`	11	(format-specific)
`p`	15	Variant (nested node)
`z`	—	Tag section terminator (not in bit matrix)

The bit matrix has `type_index_count` rows and `type_code_count` columns. Each row's bits are combined into a `row_flags` u32 that determines how nodes of that type are decoded — bits 0-1 encode the node kind (null/leaf/array/map), bit 7 encodes whether the node carries its own key.

8.3.4. Three String Blocks

After the bit matrix, three sequential string blocks contain the decode vocabulary:

1. **Value strings** — the actual data values referenced by index
2. **Value kinds** — type annotations (e.g., `Asset`, `game_region`, `map`)
3. **Key strings** — entry and field names (e.g., `serialindex`, `mappath`, `displayname`)

Each block has a 16-byte header: `count` (u16), `declared_count` (u16), then `byte_length` bytes of null-terminated strings.

8.3.5. Binary Data Section

After the string blocks, the remaining bytes are the bit-packed binary data. This is decoded by `decode.rs` using the row flags from the TypeCodeTable and the string tables for value resolution.

The decode loop reads: - **Table IDs** referencing header strings - **Dependency lists** per table - **Remap arrays** (FixedWidthIntArray) for key and value string index remapping - **Records** with byte-aligned length prefixes - **Tags** (a through z) per record — metadata like key names, numeric values, name lists - **Entries** with key-value pairs decoded recursively as null/leaf/array/map nodes - **Dependency entries** linking records to dependent tables with serial indices

8.4. Binary Section

The binary section contains bit-packed structured data decoded by the `decode.rs` module. All NCS files use the same decode algorithm — the TypeCodeTable's row flags determine how each node type is interpreted.

8.4.1. String Indexing

The binary section references strings by index into the three string tables parsed from the TypeCodeTable:

- **Header strings** — table name and dependency names (from BlobHeader)
- **Value strings** — data values (from string block 1)
- **Value kinds** — type annotations like `Asset`, `game_region` (from string block 2)
- **Key strings** — entry and field names (from string block 3)

Bit widths for indices are computed from the declared counts: `bit_width(count)` gives the minimum bits needed to index into each table.

8.4.2. Decode Loop

The binary data is read as a bitstream. The outer loop reads table IDs (referencing header strings), dependency lists, and per-table remap arrays. Each table contains byte-aligned records:

```
[table_id] [dep_id...0] [remap_a] [remap_b] [records...]
```

Each record has: 1. **Length prefix** (32-bit, byte-aligned) — total record size in bytes 2. **Tags** — metadata bytes (`a` through `z`) until the `z` terminator 3. **Entries** — key-value pairs with 2-bit opcodes (0=end, 1=null, 2=node, 3=ref) 4. **Dependency entries** — cross-references to dependent tables

8.4.3. Record Tags

Tags provide per-record metadata. Each tag is a single ASCII byte followed by tag-specific data:

Tag	Description	Data
`a`	Key name	pair_vec string reference
`b`	U32 value	32 bits
`c`	F32 value	32 bits (interpreted as both u32 and f32)
`d`, `e`, `f`	Name lists	Sequence of pair_vec strings until “none”
`p`	Variant	Nested node value
`z`	Terminator	End of tag section

8.4.4. Node Types

Entry values are decoded recursively. The type_index from the bitstream selects a row from the TypeCodeTable's bit matrix, which determines the node kind:

Kind (bits 0-1)	Type	Decoding
0	Null	No data
1	Leaf	Value string + value kind
2	Array	Continuation-bit loop of child nodes
3	Map	Continuation-bit loop of key-value pairs

If bit 7 of the row flags is set (or kind is Map), the node reads a self-key string before the value.

8.4.5. Remap Arrays

Each table has two FixedWidthIntArrays (24-bit count + 8-bit value width): - **Remap A** — remaps key string indices - **Remap B** — remaps value string indices

These compress the index space when a table only uses a subset of the global string tables.

8.4.6. Hash Function

FNV-1a 64-bit is used for field name hashing:

```
const OFFSET_BASIS: u64 = 0xcbf29ce484222325;
const PRIME: u64 = 0x100000001b3;

fn fnv1a_64(data: &[u8]) -> u64 {
    let mut hash = OFFSET_BASIS;
    for byte in data {
        hash ^= *byte as u64;
        hash = hash.wrapping_mul(PRIME);
    }
    hash
}
```

8.5. Inventory Parts (inv.bin)

The `inv.bin` file is the **authoritative source** for valid weapon and gear parts. BL3 used explicit PartSet and PartPool assets in the pak files. BL4 moved all of that into NCS.

`inv.bin` uses the full tag-based binary format — the most complex NCS encoding. The file is large (1.4MB decompressed, 18,393 strings, 976KB of binary data) and contains the complete inventory definition for every weapon, shield, and gear item in the game.

8.5.1. File Structure

```
Offset    Content
------    -------
0x00      BlobHeader (16 bytes)
0x10      Header strings ("inv" + 39 dependency names)
0x0d      Dependencies (39 null-terminated strings)
0x1fb     Format code ("abcefhijl")
0x225     String table (18,393 null-terminated strings)
0x169e7c  END OF FILE
```

The 39 dependency strings define valid part slot categories:

```
inv_comp              primary_augment          secondary_augment
core_augment          barrel                   barrel_acc
body                  body_acc                 foregrip
grip                  magazine                 magazine_ted_thrown
magazine_acc          scope                    scope_acc
secondary_ammo        hyperion_secondary_acc payload_augment
payload               class_mod_body           passive_points
action_skill_mod      body_bolt                body_mag
element               firmware                 stat_augment
body_ele              unique                   turret_weapon
tediore_acc           tediore_secondary_acc    endgame
enemy_augment         active_augment           underbarrel
underbarrel_acc_vis   underbarrel_acc          barrel_licensed
```

8.5.2. Parser Pipeline

The NCS parser processes `inv.bin` through a three-stage pipeline in the `parse/` module:

1. **blob.rs** — Reads the decompressed header: entry count, flags, string byte counts. Extracts header strings (dependencies, type name).
2. **typecodes.rs** — Builds a `TypeCodeTable` from the format code. Reads type codes, bit matrix, and row flags. Parses 3 string blocks (value strings, value kinds, key strings) with string repair for missing null terminators.
3. **decode.rs** — Decodes the bit-packed table data into structured output: tables, records, tags, entries, and values.

The decoder produces a `Document` containing `Table`s of `Record`s. Each record has `Tag`s (type-annotated metadata), `Entry`s (key-value data), and `DepEntry`s (dependency references with serial indices).

Serial index extraction from `inv0.bin` yields **649 of 655** known indices. The 6 missing are likely due to test data from an older game version.

8.5.3. Weapon Type Definitions

Weapon types follow the pattern `{MANUFACTURER}_{WEAPONTYPE}`:

Pattern	Example	Description
`DAD_PS`	Daedalus Pistol	56 valid parts
`JAK_SG`	Jakobs Shotgun	Shotgun parts
`VLA_AR`	Vladof AR	Assault rifle parts
`TOR_HW`	Torgue Heavy Weapon	Heavy weapon parts

Manufacturers: BOR, DAD, JAK, MAL, ORD, TED, TOR, VLA

Weapon Types: PS (Pistol), SG (Shotgun), AR (Assault Rifle), SM (SMG), SR (Sniper), HW (Heavy)

Parts are listed sequentially after their weapon type definition, continuing until the next weapon type entry:

```
DAD_PS                          <- Weapon type entry
  NexusSerialized, ..., Daedalus Pistol
  Weapon_PS
  /Game/Gear/Weapons/Pistols/DAD/Body_DAD_PS.Body_DAD_PS
  ...
  DAD_PS_Barrel_01              <- Valid parts start
  DAD_PS_Barrel_01_A
  DAD_PS_Barrel_01_B
  DAD_PS_Barrel_01_C
  DAD_PS_Barrel_01_D
  DAD_PS_Body
  DAD_PS_Body_A
  DAD_PS_Body_B
  ...
  DAD_PS_Underbarrel_06         <- Last part
DAD_SG                          <- Next weapon type
```

Part names follow `{MANUFACTURER}_{WEAPONTYPE}_{SLOT}_{VARIANT}`:

- `DAD_PS_Barrel_01` — Daedalus Pistol, Barrel slot, base variant
- `DAD_PS_Barrel_01_A` — Daedalus Pistol, Barrel slot, A variant
- `JAK_SG_Grip_03` — Jakobs Shotgun, Grip slot, variant 3

8.5.4. Serial Index Structure

Each part has a `serialindex` field for serialization. The structure encodes both the index number and scope:

```
serialindex: {
  status: "Active" | "Inactive"
  index: u32                // 0-127 typically
  _category: "inv_type"    // Always "inv_type" for inventory
  _scope: "Root" | "Sub"   // "Root" for item types, "Sub" for parts
}
```

Root scope entries identify base item types (e.g., `DAD_PS`, `BOR_SG`). In serialized items, Root indices occupy the range 0–127 (bit 7 = 0).

Sub scope entries identify individual parts within an item type. Indices are unique within each type but may repeat across types. In serialized items, Sub indices use 128–255 (bit 7 = 1).

8.5.4.1. Bit 7 Flag

In actual serialized items, the Root vs Sub distinction is encoded as bit 7 of the part token index:

```
For Part indices > 142 (beyond element range):
  Bit 7 = 0 -> Root scope (core parts: body, barrel, scope)
  Bit 7 = 1 -> Sub scope (attachments: grips, foregrips, underbarrel)

Actual part index = serial_index & 0x7F  (strip bit 7)
```

Examples from Rainbow Vomit (Jakobs Shotgun):

- Serial index 4 -> `part_body_b` (Root, bit 7 = 0)
- Serial index 170 -> 42 -> `part_grip_03` (Sub, bit 7 = 1, actual = 42)

- Serial index 166 -> 38 -> `part_grip_04_hyp` (Sub, bit 7 = 1, actual = 38)

The NCS `_scope` field directly corresponds to how parts are encoded in serialized items.

8.5.5. Legendary Compositions

Legendary weapons are defined by `comp_05_legendary_*` entries with mandatory parts:

```
comp_05_legendary_Zipgun
  uni_zipper                      <- Unique naming part (red text)
  part_barrel_01_Zipgun           <- Mandatory unique barrel

comp_05_legendary_DiscJockey
  uni_discjockey
  part_barrel_02_DiscJockey

comp_05_legendary_OM              <- Oscar Mike
  part_barrel_unique_OM

comp_05_legendary_GoreMaster
  part_barrel_02_GoreMaster
```

Each composition has: an identifier (`comp_05_legendary_{name}`), a unique naming part (`uni_{name}`) that provides the display name and red text, and one or more mandatory parts (usually a unique barrel).

8.5.6. NCS vs Memory Part Names

Part names differ between NCS extraction and runtime memory dumps:

NCS Name	Memory Name
`DAD_PS_Barrel_01`	`DAD_PS.part_barrel_01`
`DAD_PS_Body_A`	`DAD_PS.part_body_a`
`DAD_PS_Grip_04`	`DAD_PS.part_grip_04_hyp`

NCS consistently has more parts (56 for `DAD_PS`) than memory extraction (34), because memory dumps only capture parts that exist as runtime UObjects during the dump. Always use NCS as the authoritative source.

8.5.7. Extracting Parts

```
# Extract all item parts (weapons + shields) to JSON
bl4 ncs extract inv4.bin -t item-parts --json -o item_parts.json

# View weapon parts for a specific pakchunk
bl4 ncs extract /path/to/ncsdata/pakchunk4-Windows_0_P -t item-parts
```

Output:

```
[
  {
    "item_id": "DAD_PS",
    "parts": ["DAD_PS_Barrel_01", "DAD_PS_Barrel_01_A", "..."],
```

```
    "legendary_compositions": ["..."]
  },
  {
    "item_id": "Armor_Shield",
    "parts": ["part_core_atl_protractor", "part_ra_armor_segment_prima
    "legendary_compositions": []
  }
]
```

8.6. Actor Definitions (gbxactor.bin)

The `gbxactor.bin` file defines game actors: characters, AI behaviors, abilities, teams, and spawn patterns. It uses the `abef` format code with approximately 1,740 entries.

8.6.1. Entry Categories

Category	Pattern	Description
`Actor_*`	`Actor_PLD_AS_Scourge_*`	Player abilities, projectiles
`Char_AI`	`Char_AI`	Base AI character definition
`Char_Enemy`	`Char_Enemy`	Base enemy character
`Char_NPC`	`Char_NPC`	Base NPC character
`Char_{Type}`	`Char_Paladin`, `Char_ExoSoldier`	Player character types

Category	Pattern	Description
`Char_{Enemy}`	`Char_ArmyBandit_*`, `Char_Psycho*`	Enemy types
`Char_Gadget_*`	`Char_Gadget_AutoTurret_Base`	Gadget/turret actors
`Team_*`	`Team_Player`, `Team_Bandit`	Team definitions
`MPart_*`	`MPartRand_Skin_Human`	Mesh part randomizers

Characters inherit from base types:

```
Char_AI
+-- Char_Enemy
|   +-- Char_ArmyBandit_SHARED
|   +-- Char_PsychoBasic
|   +-- ...
+-- Char_NPC
+-- Char_Gadget_AutoTurret_Base
```

Actor entries define behavior properties like `PatrolPauseTime`, `bCanEngagePlayers`, `Element` (NoElement, Corrosive, Cryo, Fire, Shock, Radiation), and spawn patterns. Weapon parts are *not* in `gbxactor.bin` — those are exclusively in `inv.bin`.

8.7. Entity Display Names (NameData)

NCS files contain `NameData_*` entries that map internal type names to in-game display names. Each entry follows the pattern:

```
NameData_<InternalType>, <UUID>, <DisplayName>
```

8.7.1. Boss Names

Internal Type	UUID	Display Name
NameData_Meathead	D342D6EE47173677C51C068BADA88F69	Saddleback
NameData_Meathead	B8EAFB724DAB6362B39A5592718B54B0	The Immortal Boneface
NameData_Meathead	33D8546645185A85D4575F98B67DC44B	Saddleback & Boneface
NameData_Meathead	B632671F4B8B70E97E8785A8CFE1C00B	Big Encore Saddleback

8.7.2. Enemy Variants

Enemies have elemental and rank variants, each with a unique UUID:

```
NameData_Thresher, 35D7CBFD4E844BB1624140B84DE69546, Vile Thresher
NameData_Thresher, 7505A0A34FC98F3916DABBA70974675F, Badass Thresher
NameData_Thresher, 4829F4F643423CB6F1F144B4F5A2F2CB, Burning Badass Th
NameData_Thresher, 2A0DEEE34653F2E0BD3C8BABC4D1353D, Boreal Badass Thr

NameData_Bat, 160168F945945478127AD496A3BB0673, Badass Kratch
NameData_Bat, 52A45B3F42B1A9480C36188401A6C801, Vile Kratch
NameData_Bat, 05E8994641C36AF561B109ACD197D81D, Airstrike Kratch
```

8.7.3. Boss Replay and Challenge Text

`Table_BossReplay_Costs` entries reference boss display names with location context:

```
Table_BossReplay_Costs, 2DCA8E674F8F83E700B52B959C65C2D2, Meathead Riders: Saddlebac
```

UVH challenge strings embed boss names with their locations:

```
UVH_Rankup_2_Challenges, ..., Kill Bramblesong in UVH 1 (Abandoned Auger Mine, Stone
UVH_Rankup_2_Challenges, ..., Kill Bio-Thresher Omega in UVH 1 (Fades District, Domi
UVH_Rankup_4_Challenges, ..., Kill Mimicron in UVH 3 (Order Bunker, Idolator's Noose
```

8.7.4. Internal to Display Name Mapping

Known boss translations from `itempoollist` internal names to `NameData` display names:

Internal Name (itempoollist)	Display Name (NameData)
`MeatheadRider_Jockey`	Saddleback
`Thresher_BioArmoredBig`	Bio-Thresher Omega
`Timekeeper_Guardian`	Guardian Timekeeper
`BatMatriarch`	Skyspanner Kratch
`TrashThresher`	Sludgemaw
`StrikerSplitter`	Mimicron
`Destroyer`	Bramblesong

8.7.5. Extracting NameData

```
# Get all NameData entries
strings /path/to/ncs_native/*/*.bin | grep "^NameData_" | sort -u

# Get specific boss type mappings
```

```
strings /path/to/ncs_native/*/*.bin | grep "^NameData_Meathead"

# Get challenge text with boss names
strings /path/to/ncs_native/*/*.bin | grep "UVH.*Kill"
```

8.8. NCS Manifest

Each pak file contains an NCS manifest at the `_NCS/` path. The manifest lists every NCS chunk in the pak and provides the index needed to locate each chunk.

8.8.1. Manifest Header

```
Offset  Size  Type   Description
------  ----  ----   -----------
0x00    5     bytes  Magic: "_NCS/" (0x5f 0x4e 0x43 0x53 0x2f)
0x05    1     u8     Null terminator
0x06    2     u16    Entry count (little-endian)
0x08    2     u16    Unknown (typically 0x0000)
0x0a    var   Entry  Entry records
```

8.8.2. Manifest Entry

```
length (u32) | filename (length-1 bytes) | null (u8) | index (u32)
```

Sort entries by index to get the correct order matching NCS chunk offsets in the pak file.

8.9. DataTable Relationship

NCS files contain serialized DataTable rows that reference schemas in `.uasset` files. The GUID portion of a schema name matches across both formats:

Schema file (`Struct_DedicatedDropProbability.uasset`):

```json
{
  "name": "Primary_2_A7EABE6349CCFEA454C199BC8C113D94",
  "value_type": "Double",
  "float_value": 0.0
}
```

NCS reference:

```
Table_DedicatedDropProbability
Prim2_A7EABE6349CCFEA454C199BC8C113D94
```

Numeric values (weights, probabilities) are stored as strings in NCS: `"0.200000"`, `"1.500000"`. The binary section's bit-packed indices point into the string table where these values live.

8.10. Known File Types

Type	Description	Count
`achievement`	Achievement definitions	1
`aim_assist_parameters`	Aim assist config	1
`ainodefollowsettings`	AI follow settings	1
`ainodeLeadsettings`	AI lead settings	1
`attribute`	Game attributes	~10
`audio_event`	Audio event mappings	1
`coordinated_effect_filter`	Effect filters	1
`gbx_ue_data_table`	Gearbox data tables	many
`gbxactor`	Actor definitions	1
`inv`	Inventory part definitions	1
`itempool`	Item pool definitions	many
`itempoollist`	Item pool lists (boss drops)	many
`loot_config`	Loot configuration	many
`Mission`	Mission data	many
`preferredparts`	Part preferences	1
`trait_pool`	Trait pool definitions	many
`vending_machine`	Vending inventory	many

8.10.1. Key Files for Loot Analysis

File	Purpose
`itempoollist.bin`	Boss-to-legendary mappings. `ItemPoolList_<BossName>` records with dedicated drops.
`itempool.bin`	General item pools: rarity weights, world drops, Black Market items.

File	Purpose
`loot_config.bin`	Global loot configuration parameters.
`preferredparts.bin`	Part preferences for weapon/gear generation.
`inv.bin`	Complete inventory definitions: parts, compositions, serial indices.

8.10.2. Extracting Drop Information

```
# Generate drops manifest from NCS data
bl4 drops generate "/path/to/ncs_native" -o share/manifest/drops.json --manifest-dir

# Find where an item drops
bl4 drops find hellwalker

# List drops from a specific boss
bl4 drops source Timekeeper
```

See Appendix C: Loot System Internals for detailed drop table documentation.

8.11. Type Prefixes

Some string table values carry single-letter type prefixes:

Prefix	Type	Example
`T`	Text/String	`Tnone` = string "none"
`b`	Base/Boolean	context-dependent
`F`	Float	context-dependent

8.12. Worked Example: achievement.bin

To tie the format together, here's a walkthrough of `achievement.bin` (576 bytes decompressed).

8.12.1. File Layout

```
0x000-0x00F ( 16 bytes): BlobHeader (entry_count=2, flags=0, string_by
0x010-0x01C ( 13 bytes): Header strings: "achievement"
0x01D        (  3 bytes): TypeCodeTable header: type_code_count=3, typ
0x020        (  3 bytes): Type codes: "abj"
0x023        (  2 bytes): Bit matrix (3×3 = 9 bits, padded to 2 bytes)
0x025+       (variable):  Three string blocks (value strings, value ki
...          (remaining):  Binary data (bit-packed tables, records, en
```

8.12.2. Parsed Output

The parser produces a single `achievement` table with one record containing 6 tags and 5 entries:

```
{
  "achievement": {
    "name": "achievement",
    "deps": [],
    "records": [{
      "tags": [
        {"__tag": "a", "pair": "none"},
        {"__tag": "b", "value": 1},
        {"__tag": "c", "u32_value": 1065353216, "f32_value": 1.0}
      ],
      "entries": [
        {"key": "id_achievement_10_worldevents_colosseum",
         "value": {"achievementid": "10", "achievement": "ID_Achievement_10_worldeve
        {"key": "id_achievement_11_worldevents_airship",
         "value": {"achievementid": "11", "achievement": "ID_Achievement_11_worldeve
        {"key": "id_achievement_12_worldevents_meteor",
         "value": {"achievementid": "12", "achievement": "ID_Achievement_12_worldeve
      ]
    }]
  }
}
```

8.13. Compatibility Issues

8.13.1. Oodle SDK Requirements

The four files listed in the Compression section (~2.4% of all NCS files) fail to decompress with open-source Oodle implemen-

tations. They require the official Oodle SDK. All other NCS files decompress correctly with open-source tools.

8.13.2. String Validation

When parsing the string table, valid strings should be at least 2 characters long and contain no garbage characters. Pure numeric strings (`"10"`, `"24"`) are valid. Short strings (2–3 characters) should be all lowercase or known keywords.

Watch for these invalid patterns:

- Mixed-case short strings like `"zR"` or `"D3"` — binary data misinterpreted as text
- Trailing or leading spaces
- High underscore-to-letter ratio

8.14. Future Work

Several areas of the format remain partially understood:

1. **Hash table decoding** — The u32 values after ASCII field abbreviations likely form a hash lookup table, but the exact structure isn't confirmed.
2. **Entry-to-category mapping** — How the binary section maps entries to their DLC categories.
3. **Cross-file references** — How NCS files reference each other (e.g., ItemPool entries pointing to ItemPoolList records).

4. **Runtime behavior** — How the game engine loads and indexes NCS data at runtime.
5. **Structured section encoding** — For `inv.bin`, the 107 bit-packed indices in the first entry map to 16 JSON fields, but the exact mapping algorithm isn't fully decoded.
6. **Format code semantics** — The format code letters (`abcehijl`) clearly correspond to tag types, but the full specification for each letter's encoding rules is still being worked out.
7. **Inline entry names** — The `CAaB`/`RQJ`/`IEZ` identifiers in `inv.bin`'s metadata section are parsed but their semantic meaning is unclear.

The NCS format is approximately 95% reverse-engineered. The compression layer, string tables, differential encoding, and tag-based binary section are all well understood and implemented in the `bl4-ncs` parser. The remaining unknowns are in the binary section's finer details — hash table structures, cross-file reference resolution, and the exact semantics of per-entry schema encoding. Chapter 7 covers how to extract usable game data from these parsed NCS documents.

Part III.

Practical Application

9. Chapter 7: Data Extraction

A save editor needs game data: weapon stats, part definitions, manufacturer information. You might assume this data lives neatly in game files, waiting to be extracted. The reality is more complicated—and more interesting.

This chapter explores what data we can extract, what we can't, and why. Along the way, we'll document our investigation into authoritative category mappings, including the binary analysis that revealed why some data simply doesn't exist in extractable form.

9.1. The Game File Landscape

BL4's data lives in Unreal Engine pak files, stored in IoStore format:

```
Borderlands 4/OakGame/Content/Paks/
├── pakchunk0-Windows_0_P.utoc    <- Main game assets
├── pakchunk0-Windows_0_P.ucas    <- Compressed data
├── pakchunk2-Windows_0_P.utoc    <- Audio (Wwise)
├── pakchunk3-Windows_0_P.utoc    <- Localized audio
├── global.utoc                   <- Shared engine data
└── ...
```

IoStore is UE5's container format, splitting asset indices (`.utoc`) from compressed data (`.ucas`). This differs from older PAK-only formats and requires specialized tools.

> **Note**
>
> BL4 uses IoStore (UE5's format), not legacy PAK. Tools like `repak` won't work on `.utoc/.ucas` files. You need `retoc` or similar IoStore-aware extractors.

9.2. Data Source Hierarchy

Three distinct sources feed into the parts database, each with different strengths and limitations:

```
flowchart TB
    subgraph "Static Sources"
        PAK[PAK Files<br/>Balance, naming, bodies]
        NCS[NCS Files<br/>Parts, pools, loot config]
    end
    subgraph "Runtime Sources"
        MEM[Memory Dumps<br/>Serial indices, UObjects]
    end
    PAK --> DB[(Parts Database)]
    NCS --> DB
    MEM --> DB
```

PAK files provide balance data, naming strategies, and body definitions—the structural skeleton of the item system. NCS files

contain the bulk of part data: 5,360 parts across 120 categories, complete with serial indices. Memory dumps fill in the gaps: runtime UObject data, schema information, and validation of static extractions.

The rest of this chapter walks through each source in detail.

9.3. What We Can Extract

Some game data extracts cleanly from pak files:

Balance data: Stat templates and modifiers for weapons, shields, and gear. These define base damage, fire rate, accuracy scales.

Naming strategies: How weapons get their prefix names. "Damage -> Tortuous" mappings live in extractable assets.

Body definitions: Weapon body assets that reference parts and mesh fragments.

Loot pools: Drop tables and rarity weights for different sources.

Gestalt meshes: Visual mesh fragments that parts reference.

These assets follow Unreal's content structure:

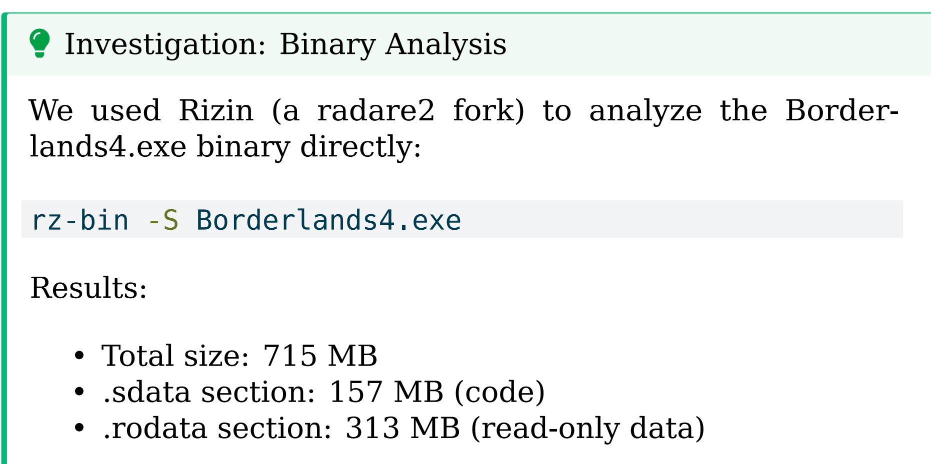
```
OakGame/Content/
├── Gear/
│   ├── Weapons/
│   │   ├── _Shared/BalanceData/
│   │   ├── Pistols/JAK/Parts/
```

```
│   │   └── ...
│   └── Shields/
├── PlayerCharacters/
│   ├── DarkSiren/
│   └── ...
└── GameData/Loot/
```

9.4. What We Can't Extract

Here's where it gets interesting. The mappings between serial tokens and actual game parts—the heart of what makes serial decoding work—don't exist as extractable pak file assets.

We wanted authoritative category mappings. Serial token {4} on a Vladof SMG should mean a specific part, and we wanted the game's own data to tell us which one. So we investigated.



Investigation: Binary Analysis

We used Rizin (a radare2 fork) to analyze the Borderlands4.exe binary directly:

```
rz-bin -S Borderlands4.exe
```

Results:

- Total size: 715 MB
- .sdata section: 157 MB (code)
- .rodata section: 313 MB (read-only data)

We searched for part prefix strings like "DAD_PS.part_" and "VLA_SM.part_barrel". Nothing. The prefixes don't exist as literal strings in the binary.
We searched for category value sequences. Serial decoding uses Part Group IDs like 2, 3, 4, 5, 6, 7 (consecutive integers stored as i64). We found one promising sequence at offset 0x02367554:

```
# Found sequence 2,3,4,5,6,7 as consecutive i64 values at 0x02367554
```

But examining the context revealed it was near crypto code—specifically "Poly1305 for x86_64, CRYPTOGAMS". Those consecutive integers were coincidental, not category definitions.
Lesson: When searching binaries for numeric patterns, verify the context. Small consecutive integers appear in many places: crypto code, lookup tables, version numbers. Always examine surrounding bytes.

9.4.1. UE5 Metadata: What We Know

From usmap analysis, we confirmed the exact structure linking parts to serials:

```
GbxSerialNumberIndex (12 bytes)
├── Category (Int64): Part Group ID
├── scope (Byte): EGbxSerialNumberIndexScope (Root=1, Sub=2)
├── status (Byte): EGbxSerialNumberIndexStatus
└── Index (Int16): Position in category
```

Every `InventoryPartDef` contains this structure. The `Category` field maps to Part Group IDs (2=Daedalus Pistol, 22=Vladof SMG,

etc.). The `Index` field determines which part token decodes to this part.

But here's the problem: we found **zero** `InventoryPartDef` assets in pak files.

```
uextract /path/to/Paks find-by-class InventoryPartDef
# Result: 0 assets found
```

9.4.2. Where Parts Actually Live

Parts aren't stored as individual pak file assets. They're:

1. **Runtime UObjects** — Created when the game initializes
2. **Code-defined** — Registrations happen in native code
3. **Self-describing** — Each part carries its own index internally

9.4.3. The Key Insight: Self-Describing Parts

Here's the crucial design pattern we discovered: **there is no separate mapping file because each part stores its own index**.

Every part UObject contains a `GbxSerialNumberIndex` structure at offset +0x28:

```
UObject + 0x28: GbxSerialNumberIndex (4 bytes)
├── Scope (1 byte)   <- EGbxSerialNumberIndexScope (Root=1, Sub=2)
├── Status (1 byte)  <- Reserved/state flags
└── Index (2 bytes)  <- THE serial index for this part
```

This is a "reverse mapping" architecture:

- **Traditional approach**: Separate lookup table maps `index -> part_name`
- **BL4's approach**: Each part stores its own index; the "mapping" IS the parts themselves

Why this design makes sense:

Benefit	Explanation
No central registry	Adding DLC parts doesn't require updating a mapping file
Self-contained	Each part is fully self-describing
Stable indices	A part's index never changes because it's intrinsic to that part
No sync issues	Impossible for mapping to drift from actual parts

Practical implication: When we extract parts from memory, we're not building a mapping from separate data—we're reading the authoritative index directly from each part. The memory dump contains the complete, correct mapping because that mapping IS the parts.

> ℹ Why Memory Dumps Are Essential
>
> Since each part carries its own index internally, and parts only exist as runtime UObjects (not pak file assets), memory dumps are the only way to capture this data. The game's binary contains the code to create parts, but the actual GbxSerialNumberIndex values are set during initialization.

9.5. Memory Extraction: The Breakthrough

Through systematic memory analysis, we discovered **authoritative part-to-index mappings can be extracted** from memory dumps. Here's the structure:

9.5.1. The Part Registration Structure

When the game loads, it creates UObjects for each part and registers them in an internal array. This array has a discoverable pattern:

```
Part Array Entry (24 bytes):
├── FName Index (4 bytes)     <- References the part name in FNamePool
├── Padding (4 bytes)         <- Always zero
├── Pointer (8 bytes)         <- Address of the part's UObject
├── Marker (4 bytes)          <- 0xFFFFFFFF sentinel value
└── Priority (4 bytes)        <- Selection priority (not the serial in
```

The serial **Index** is stored **inside the pointed UObject**, at offset +0x28:

```
UObject at Pointer (offset +0x28):
├── Scope (1 byte)            <- EGbxSerialNumberIndexScope (always 2
├── Reserved (1 byte)         <- Usually 0
└── Index (2 bytes, Int16)    <- THE SERIAL INDEX we need!
```

> **!** Category Derivation
>
> The Part Group ID (category) is **not** stored in the UObject at a fixed offset. Instead, derive it from the part name prefix

(e.g., `DAD_PS` -> category 2, `VLA_AR` -> category 17). The bl4 tool includes a complete prefix-to-category mapping.

9.5.2. Verified Example

Searching for FName `DAD_PS.part_barrel_01` (FName index 0x736a0a):

1. **Find the array entry**: FName appears in the part array with pointer 0x7ff4ca7d75d0
2. **Read offset +0x28**: At 0x7ff4ca7d75f8 we find `02 00 07 00`
3. **Parse**: Scope=2, Reserved=0, **Index=7**
4. **Derive category**: `DAD_PS` prefix -> category 2
5. **Verify**: Reference data confirms DAD_PS.part_barrel_01 has index 7

Additional verified mappings:

- `DAD_PS.part_barrel_02` -> Index 8
- `DAD_PS.part_barrel_01_Zipgun` -> Index 1
- `DAD_PS.part_barrel_02_rangefinder` -> Index 78

9.5.3. Extraction Algorithm

```
# Pseudocode for extracting all part mappings
def extract_parts(memory_dump):
    # Step 1: Build FName lookup table
    # Scan FNamePool for all names containing ".part_"
    fname_table = {}  # fname_idx -> name
```

```
    for block in fnamepool.blocks:
        for entry in block:
            if ".part_" in entry.name.lower():
                fname_table[entry.index] = entry.name

    parts = []

    # Step 2: Scan memory for 0xFFFFFFFF markers
    for marker_addr in scan(memory_dump, "ff ff ff ff"):
        # Read the 24-byte entry (marker is at offset 16)
        entry = read(marker_addr - 16, 24)

        fname_idx = entry[0:4]       # FName index
        pointer = entry[8:16]        # UObject pointer

        # Validate: known FName, padding=0, valid pointer
        if fname_idx not in fname_table:
            continue
        if entry[4:8] != 0 or not is_valid_pointer(pointer):
            continue

        # Read serial index from pointed UObject at offset +0x28
        uobject = read(pointer, 0x2C)
        scope = uobject[0x28]
        index = uobject[0x2A:0x2C]  # Int16 at bytes 2-3

        # Derive category from part name prefix
        name = fname_table[fname_idx]
        category = get_category_from_prefix(name)

        if category is not None:
            parts.append({
                'name': name,
```

```
                'category': category,
                'index': index
            })

    return parts

def get_category_from_prefix(name):
    prefix = name.split(".part_")[0].lower()
    # Pistols
    if prefix == "dad_ps": return 2
    if prefix == "jak_ps": return 3
    # ... (complete mapping in bl4 source)
    return None
```

9.5.4. Why This Works

The game registers parts at startup into internal arrays. Each entry links:

- **FName reference** -> The part's name (e.g., "VLA_SM.part_barrel_01")
- **UObject pointer** -> The full part definition, including serial index

By scanning for the 0xFFFFFFFF sentinel pattern that marks entry boundaries, we can walk these arrays and extract every part mapping the game knows about.

> **Practical Implication**
>
> Memory dumps contain **authoritative** part-to-index mappings. Extract them directly—no empirical testing required

> for known parts. Empirical validation is only needed for new parts added in patches.

9.5.5. Extraction Results and Limitations

The initial memory extraction (Dec 2025) captured roughly 1,041 parts across 47 categories—about 40% of the total. Memory extraction only captures parts that were **instantiated** at the time of the dump.

NCS extraction changed the picture entirely. Parsing the `inv*.bin` files yields the complete dataset:

Metric	Memory Extraction	NCS Extraction
Total parts	~1,041	5,360
Categories covered	47	120
Source	Runtime UObjects	Static game data
Requires game running	Yes	No

> ⚠ Memory vs NCS Coverage
>
> Memory extraction only captures parts **instantiated in memory** at the time of the dump. Significant gaps exist in index sequences—parts not spawned during the capture session are simply absent.
> NCS extraction from `inv*.bin` files provides the complete dataset without requiring the game to be running. Memory dumps remain valuable for validation and for capturing data not present in NCS files (like UObject layout details).

What we have from NCS:

- Complete part lists per category (5,360 parts across 120 categories)
- Serial indices for all extracted parts
- Category names derived from NCS keys

What still requires memory dumps:

- UObject layout verification
- Schema data (usmap generation)
- Parts from DLC/content not yet in NCS files

9.5.6. Data Pipeline: Current State

The part mapping workflow uses multiple data sources:

1. NCS Extraction (Primary)

`share/manifest/parts_database.json` contains 5,360 parts across 120 categories. Source: NCS `inv*.bin` file extraction.

```
# Extract complete parts database from NCS files
bl4 ncs extract --extract-type manifest
```

2. Memory Dumps (Validation & Schema)

Memory dumps validate NCS data and provide schema information not available from static files.

```
# Extract indices from memory - validates NCS data
bl4 memory --dump game.dmp extract-parts -o parts_with_categories.json
```

3. Part Names (Complete)

`share/manifest/category_names.json` contains 120 category names. Source: NCS extraction alongside the parts database.

9.5.7. What NCS/UASSET Data Provides

Current NCS extraction yields 806 files with 232 unique types:

NCS Type	Contents	Part Data
`inv.bin`	13,860 inventory entries	Part attributes, stats, **serial indices**
`inv_name_part.bin`	946 part naming entries	Display names
`GbxActorPart.bin`	2,167 actor parts	Cosmetic/mesh parts with indices
`itempool.bin`	Item pool definitions	Loot tables

9.5.8. NCS Serial Index Discovery

Breakthrough: NCS `inv.bin` DOES contain serial indices for weapon parts. The indices are stored in the binary section entries.

Format: Part names in NCS use a different format than memory:

NCS Format	Memory Format	Index
`BOR_SG_Grip_01`	`BOR_SG.part_grip_01`	42
`BOR_SG_Foregrip_02`	`BOR_SG.part_foregrip_02`	81
`BOR_SG_Barrel_02_B`	`BOR_SG.part_barrel_02_b`	71

Verified matches between NCS indices and memory-extracted indices:

- `BOR_SG_Grip_01` = 42
- `BOR_SG_Grip_02` = 43
- `BOR_SG_Foregrip_01` = 50
- `BOR_SG_Foregrip_02` = 81
- `BOR_SG_Barrel_02_B` = 71
- `BOR_SG_Barrel_02_D` = 73

Important caveats:

1. Not all records with `value_0` are indices (some are attribute values)
2. Only records matching weapon part naming patterns (e.g., `XXX_YY_Part_Type`) contain indices
3. Some mismatches exist (e.g., `BOR_SG_Barrel_01` shows 4 in NCS but 7 in memory)
4. NCS may contain MORE parts than memory extraction captures

Extraction approach: Filter for records where the name matches weapon part patterns, then extract `value_0` as the serial index. Cross-reference with memory-extracted indices where available.

9.5.9. Category Derivation from NCS (Jan 2026 Discovery)

Finding: NCS files do NOT directly store category IDs. However, categories can be derived from part name prefixes:

```
// Prefix-to-category mapping
"BOR_SG" -> Category 7   (Ripper Shotgun)
"JAK_SG" -> Category 9   (Jakobs Shotgun)
"DAD_PS" -> Category 2   (Daedalus Pistol)
```

```
"VLA_AR" -> Category 18  (Vladof Assault Rifle)
// ... etc
```

Implementation: The `bl4-ncs` library includes `category_from_prefix()` function that extracts the manufacturer-weapon prefix and maps it to the corresponding category ID.

Limitations:

1. **Only works for prefixed parts**: Parts like `BOR_SG_Barrel_01_A` derive categories successfully
2. **Non-prefixed parts fail**: Generic parts (`comp_01_common`, `part_firmware_*`, `part_ra_*`) have no prefix, so category cannot be determined from NCS alone
3. **Same part name, different categories**: Parts like `comp_01_common` exist in many categories with different indices. Without category context, these cannot be uniquely identified.

Result: The NCS parser extracts **649/655 serial indices** from inv0.bin with structured table/record/entry output. Previous heuristic approaches (BinaryParserV2) used incorrect structural models and have been removed. Category limitations remain:

Conclusion: NCS provides part indices but NOT categories. For complete part database:

- Use NCS for manufacturer-specific weapon parts (BOR_SG, JAK_PS, etc.)
- Requires memory dumps or other sources for non-prefixed parts
- Categories must be derived from prefixes, not extracted from NCS data

---

9.6. Empirical Validation (Fallback)

For edge cases or when memory extraction isn't possible, empirical validation remains an option:

1. Collect serials from real game items
2. Decode the Part Group ID and part tokens
3. Record which weapon/part combinations the tokens represent
4. Validate by injecting serials into saves and checking in-game

The `parts_database.json` file combines NCS-extracted mappings with empirically-verified data for comprehensive coverage.

9.7. Extraction Tools

9.7.1. retoc — IoStore Extraction

The essential tool for BL4's pak format:

```
cargo install --git https://github.com/trumank/retoc retoc_cli

# List assets in a container
retoc list /path/to/pakchunk0-Windows_0_P.utoc

# Extract all assets
retoc unpack /path/to/pakchunk0-Windows_0_P.utoc ./output/
```

Warning

For converting to legacy format, point at the **Paks directory**, not a single file. The tool needs access to `global.utoc` for ScriptObjects:

```
retoc to-legacy /path/to/Paks/ ./output/ --no-script-objects
```

9.7.2. uextract — Project Tool

The bl4 project's custom extraction tool:

```
cargo build --release -p uextract

# List all assets
./target/release/uextract /path/to/Paks --list

# Extract with filtering
./target/release/uextract /path/to/Paks -o ./output --ifilter "Balance

# Use usmap for property resolution
./target/release/uextract /path/to/Paks -o ./output --usmap share/bord
```

9.8. The Usmap Requirement

UE5 uses "unversioned" serialization. Properties are stored without field names:

```
Versioned (old):   "Damage": 50.0, "Level": 10
Unversioned (new): 0x42480000 0x0000000A
                   └── Just values, no names
```

To parse unversioned data, you need a usmap file containing the schema—all class definitions, property names, types, and offsets.

We generate usmap from memory dumps:

```
bl4 memory --dump share/dumps/game.dmp dump-usmap

# Output: mappings.usmap
# Names: 64917, Enums: 2986, Structs: 16849, Properties: 58793
```

The project includes a pre-generated usmap at `share/manifest/mappings.usmap`.

9.9. Extracting Parts from Memory

Since parts only exist at runtime, memory extraction is the path forward for validation and schema work.

9.9.1. Step 1: Create Memory Dump

Follow Chapter 3's instructions to capture game memory while playing.

9.9.2. Step 2: Extract Part Names

```
bl4 memory --dump share/dumps/game.dmp dump-parts \
    -o share/manifest/parts_dump.json
```

This scans for strings matching `XXX_YY.part_*` patterns:

```
{
  "DAD_AR": [
    "DAD_AR.part_barrel_01",
    "DAD_AR.part_barrel_01_a",
    "DAD_AR.part_body"
  ],
  "VLA_SM": [
    "VLA_SM.part_barrel_01"
  ]
}
```

9.9.3. Step 3: Build Parts Database

```
bl4 memory --dump share/dumps/game.dmp build-parts-db \
    -i share/manifest/parts_dump.json \
    -o share/manifest/parts_database.json
```

The result maps parts to categories and indices:

```
{
  "parts": [
    {"category": 2, "index": 0, "name": "DAD_PS.part_barrel_01"},
```

```
    {"category": 22, "index": 5, "name": "VLA_SM.part_body_a"}
  ],
  "categories": {
    "2": {"count": 74, "name": "Daedalus Pistol"},
    "22": {"count": 84, "name": "Vladof SMG"}
  }
}
```


> **! Index Ordering**
>
> Part indices from memory dumps reflect the game's internal registration order—not alphabetical. Parts typically register in this order: unique variants, bodies, barrels, shields, magazines, scopes, grips, licensed parts. Alphabetical sorting produces wrong indices.

9.10. Working with Extracted Assets

9.10.1. Asset Structure

Extracted `.uasset` files follow the Zen package format:

```
Package
├── Header
├── Name Map (local FNames)
├── Import Map (external dependencies)
├── Export Map (objects defined here)
└── Export Data (serialized properties)
```

With usmap, these parse into readable JSON:

```
{
  "asset_path": "OakGame/Content/Gear/Weapons/_Shared/BalanceData/Weap
  "exports": [
    {
      "class": "ScriptStruct",
      "properties": {
        "Damage_Scale": 1.0,
        "FireRate_Scale": 1.0,
        "Accuracy_Scale": 1.0
      }
    }
  ]
}
```

9.10.2. Finding Specific Data

```
# Find legendary items
find ./bl4_assets -name "*legendary*" -type f

# Find manufacturer data
find ./bl4_assets -iname "*manufacturer*"

# Search asset contents
grep -r "Linebacker" ./bl4_assets --include="*.uasset" -l
```

9.10.3. Stat Patterns

Stats follow naming conventions: `StatName_ModifierType_Index_GUID`

Modifier	Meaning
`Scale`	Multiplier (x)
`Add`	Flat addition (+)
`Value`	Absolute override
`Percent`	Percentage bonus

9.11. Oodle Compression

BL4 uses Oodle compression (RAD Game Tools). The `retoc` tool handles decompression automatically by loading the game's DLL:

```
~/.steam/steam/steamapps/common/"Borderlands 4"/Engine/Binaries/ThirdParty/Oodle/
└── oo2core_9_win64.dll
```

Tip

If extraction fails with Oodle errors, verify the game is installed and the DLL path is accessible. On Linux, Wine must be able to load the DLL.

9.12. Building a Data Pipeline

An automated extraction script saves time when the game updates:

```
#!/bin/bash
GAME_DIR="$HOME/.steam/steam/steamapps/common/Borderlands 4"
OUTPUT_DIR="./bl4_data"
USMAP="./share/manifest/mappings.usmap"

# Extract pak files
retoc unpack "$GAME_DIR/OakGame/Content/Paks/pakchunk0-Windows_0_P.uto

# Parse with usmap
./target/release/uextract "$OUTPUT_DIR/raw" -o "$OUTPUT_DIR/parsed" --
```

9.13. Strategies for Complete Index Coverage

1. **NCS extraction (preferred)** — Extract indices from `inv.bin` `value_0` fields. This is static game data that doesn't require memory dumps.
2. **Memory dumps (validation)** — Use memory-extracted indices to validate NCS data and capture parts with mismatched formats.
3. **Multiple memory dumps** — Capture game state with different loadouts to instantiate more parts.
4. **Empirical validation** — Verify unknown indices by testing serials in-game.

Recommended workflow:

1. Extract all potential indices from NCS `inv.bin`
2. Cross-reference with memory-extracted indices
3. For matches, trust the data
4. For mismatches, investigate (NCS format may differ from memory format)
5. For NCS-only parts, treat indices as provisional until validated

---

9.14. Summary: Data Sources

Data	Source	Extractable?	Status
Balance/stats	Pak files	Yes	Complete
Naming strategies	Pak files / NCS	Yes	Complete
Loot pools	Pak files / NCS	Yes	Complete
Body definitions	Pak files	Yes	Complete
Part names	NCS (`inv.bin`)	Yes	Complete (5,360 parts)
Part serial indices	NCS + Memory	Yes	Complete (5,360 across 120 categories)

Data	Source	Extractable?	Status
Category mappings	Code analysis	Yes	Complete (120 categories)

> ❗ Current State
>
> **Part names and indices** are available from NCS data (complete list: 5,360 parts across 120 categories).
> **Memory dumps** remain valuable for validation, schema extraction (usmap), and capturing data not present in NCS files. Serial decoding works for all extracted parts. Unknown indices display as `[category:index]` placeholders until validated.

9.15. Exercises

Exercise 1: Extract and Explore

Extract the main pak file. Find balance data for a weapon type you use. What stats does the base template define?

Exercise 2: Search for Part References

Search extracted assets for references to specific parts (like "JAK_PS.part_barrel"). Where do they appear? What references them?

Exercise 3: Compare Manufacturers

Extract assets for two manufacturers (Jakobs vs Maliwan). Compare directory structures. What patterns emerge?

9.16. What's Next

We've covered the full data extraction story—what works, what doesn't, and why. The bl4 project wraps all these techniques into command-line tools.

Next, we'll tour those tools: how to decode serials, edit saves, extract data, and more, all from the command line.

Next: Chapter 8: Parts System

10. Chapter 8: Parts System

Every weapon in Borderlands 4 is an assembly. A Daedalus pistol isn't one object — it's a barrel, a grip, a magazine, a scope, and a body bolted together, each chosen from a pool of compatible parts. Swap the barrel and you change damage. Swap the grip and you change handling. The combinations run into the millions, and every one of them needs to encode into a compact serial string.

BL4's parts system lives in NCS (Nexus Config Store) files, not in Unreal Engine assets. This is a fundamental departure from Borderlands 3, where parts were defined in PartSet and PartPool uassets that tools like FModel could read directly. In BL4, you won't find a single `InventoryPartDef` in the pak files. Part definitions are native C++ objects compiled into the game binary, and the only structured record of what parts exist — and which ones go together — is the NCS inventory data.

This chapter maps the entire parts system: what parts exist, how they're named, how they compose into items, and where the remaining gaps are.

10.1. Data Sources

Two sources provide part data, and they don't agree.

Source	File	Parts	What It Provides
NCS extraction	`share/manifest/item_parts.json`	1,614	Complete item-to-parts mapping (authoritative)
Memory extraction	`share/manifest/parts_database.json`	5,368	Parts with serial indices, category IDs

NCS extraction is the authoritative source for which parts exist and which items they belong to. It's static — extracted from the game's pak files, deterministic, reproducible:

```
# Extract all item parts from NCS
bl4 ncs extract /path/to/ncsdata/pakchunk4-*/Nexus-Data-inv4.bin -t ite
```

Memory extraction captures runtime data from the game process. It provides serial indices (the numeric IDs needed to encode parts into serial strings) and category assignments, but it's incomplete — parts that aren't loaded in the current game session don't appear. A dump taken in the main menu will miss parts that only load when specific items are in your inventory.

The two sources are complementary. NCS tells you *what* parts exist. Memory tells you *how* they're indexed.

10.2. Item Types

BL4 has 30 weapon types and one shield type. Weapons are organized by manufacturer and weapon class — not every manufacturer makes every class.

10.2.1. Weapons

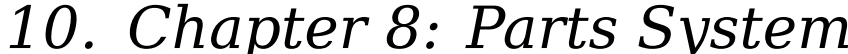
Manufacturer	PS	SG	AR	SM	SR	HW
BOR (Ripper)	-	Y	-	Y	Y	Y
DAD (Daedalus)	Y	Y	Y	Y	-	-
JAK (Jakobs)	Y	Y	Y	-	Y	-
MAL (Maliwan)	-	Y	-	Y	Y	Y
ORD (Order)	Y	-	Y	-	Y	-
TED (Tediore)	Y	Y	Y	-	-	-
TOR (Torgue)	Y	Y	Y	-	-	Y
VLA (Vladof)	-	-	Y	Y	Y	Y

Table 8.1: Weapon types by manufacturer. PS=Pistol, SG=Shotgun, AR=Assault Rifle, SM=SMG, SR=Sniper Rifle, HW=Heavy Weapon.

That's 8 manufacturers and 6 weapon classes. Each manufacturer produces 3-4 weapon classes, giving 30 distinct weapon types.

10.2.2. Shields

Shields are a single type: `Armor_Shield`, with 70 parts split between manufacturer-specific cores (44 parts) and reactive armor



augments (26 parts).

10.3. Part Naming Convention

Part names follow predictable patterns, and learning them makes the raw data readable at a glance.

10.3.1. Weapon Parts

```
{MANUFACTURER}_{WEAPONTYPE}_{SLOT}_{VARIANT}
```

The manufacturer is a three-letter code (`DAD`, `JAK`, `BOR`, etc.). The weapon type is a two-letter abbreviation (`PS`, `SG`, `AR`, `SM`, `SR`, `HW`). The slot identifies where the part attaches. The variant distinguishes between alternatives.

Examples:

- `DAD_PS_Barrel_01` — Daedalus Pistol, Barrel slot, base variant
- `DAD_PS_Barrel_01_A` — Daedalus Pistol, Barrel slot, A variant
- `JAK_SG_Grip_03` — Jakobs Shotgun, Grip slot, variant 3
- `VLA_AR_Scope_IronSight` — Vladof Assault Rifle, iron sight scope

10.3.2. Shield Parts

```
part_{type}_{name}
part_core_{manufacturer}_{effect}
part_ra_{augment}_{slot}
```

Examples:

- `part_core_dad_accelerator` — Daedalus shield core with accelerator effect
- `part_ra_armor_segment_primary` — Primary reactive armor augment

10.3.3. Licensed and Special Parts

Some parts break the standard convention:

- `LicensedPart_WeaponSpecific_UB_Tier1` — Tiered weapon-specific underbarrel
- `part_barrel_licensed_ted_mirv` — Tediore licensed barrel with MIRV reload
- `comp_05_legendary_Zipgun` — Legendary composition for the Zipgun

10.4. Part Categories

10.4.1. Weapon Parts

Each weapon type draws from the same eight part slots, though the counts vary:

Slot	Typical Count	Notes
Barrel	~10	2 base types x 4 variants, plus legendaries
Body	5	Base + A/B/C/D variants
Grip	5-8	
Magazine	4-6	
Scope	10-11	Includes iron sights
Foregrip	3	
Underbarrel	4-6	
Top Accessory	5-8	

Table 8.2: Weapon part slots and typical part counts.

10.4.2. Shield Parts

Shields use a different slot structure:

- **Core** (44 total) — manufacturer-specific, determines the shield's base behavior
- **Reactive Armor Primary** (13 types) — augments primary defense
- **Reactive Armor Secondary** (13 types) — augments secondary defense

10.4.3. Part Counts by Item Type

The full breakdown across all 31 item types totals 1,614 parts:

Item Type	Parts	Item Type	Parts
Armor_Shield	70	ORD_AR	54
BOR_HW	15	ORD_PS	59
BOR_SG	55	ORD_SR	54
BOR_SM	57	TED_AR	57
BOR_SR	55	TED_PS	60
DAD_AR	56	TED_SG	58
DAD_PS	56	TOR_AR	55
DAD_SG	56	TOR_HW	15
DAD_SM	55	TOR_PS	56
JAK_AR	56	TOR_SG	53
JAK_PS	53	VLA_AR	68
JAK_SG	53	VLA_HW	20
JAK_SR	55	VLA_SM	65
MAL_HW	15	VLA_SR	63
MAL_SG	56		
MAL_SM	58	**Total**	**1,614**
MAL_SR	56		

Table 8.3: Part counts by item type.

Heavy weapons (BOR_HW, MAL_HW, TOR_HW) have notably fewer parts at 15 each, except Vladof heavy weapons at 20. Most standard weapons cluster around 53-60 parts.

10.5. Category Mappings

The serial format assigns each item type a numeric category ID. Decoding a serial requires knowing which category you're looking at — it determines how part indices map to actual parts.

10.5.1. Weapons

Range	Type	Categories
2-6	Pistols	DAD, JAK, TED, TOR, ORD
7-12	Shotguns	BOR, DAD, JAK, TED, TOR, MAL
13-18	Assault Rifles	DAD, JAK, TED, TOR, VLA, ORD
19-22	SMGs	BOR, DAD, MAL, VLA
23-26	Snipers	BOR, JAK, VLA, ORD, MAL
244-247	Heavy Weapons	VLA, TOR, BOR, MAL

Table 8.4: Weapon category ID ranges.

10.5.2. Equipment

Category	Type	Status
44	Dark Siren Class Mod	Named, no parts in dump
55	Paladin Class Mod	Named, no parts in dump
97	Gravitar Class Mod	Named, 2 parts mapped

Category	Type	Status
140	Exo Soldier Class Mod	Named, no parts in dump
151	Firmware	Named, no parts in dump
279-288	Shields	Energy, BOR, DAD, JAK, Armor, MAL, ORD, TED, TOR, VLA
289	Shield Variant	Named, unknown subtype
300-330	Gadgets	Grenade (300), Turret (310), Repair (320), Terminal (330)
400-409	Enhancements	DAD, BOR, JAK, MAL, ORD, TED, TOR, VLA, COV, ATL

Table 8.5: Equipment category ID ranges.

10.5.3. Category Derivation

Categories can be derived from serial data depending on format:

```
VarInt-first (weapons):  category from WEAPON_INFO lookup table
VarBit-first (equipment): category = first_varbit / 384 (or / 8192 for some items)
```

The `category_from_prefix()` function in `bl4-ncs` can also derive categories from manufacturer prefixes when processing NCS data directly (e.g., `BOR_SG` maps to category 7, `JAK_SG` maps to category 9).

10.6. Composition System

Items don't just have parts — they have *compositions* that control which parts appear at each rarity tier. The composition system is how the game decides that a common Daedalus pistol gets basic parts while a legendary one gets unique, named parts.

10.6.1. Rarity Tiers

Five composition tiers correspond to the game's rarity levels:

Tier	Composition Prefix	Rarity
1	`comp_01_common`	Common (white)
2	`comp_02_uncommon`	Uncommon (green)
3	`comp_03_rare`	Rare (blue)
4	`comp_04_epic`	Epic (purple)
5	`comp_05_legendary`	Legendary (orange)

Table 8.6: Composition tiers.

Each composition can reference:

- Part slot assignments (which parts are eligible at this rarity)

- Rarity weights (`firmware_weight_XX_rarity`) controlling drop frequency
- Unique part mandates for legendaries

10.6.2. Legendary Compositions

Legendary items use `comp_05_legendary_*` compositions that mandate specific unique parts. These compositions name the legendary and pin particular parts that give it its identity:

```
comp_05_legendary_Zipgun
  uni_zipper              <- Unique naming part (red text flavor)
  part_barrel_01_Zipgun   <- Mandatory unique barrel

comp_05_legendary_GoreMaster
  part_barrel_02_GoreMaster

comp_05_legendary_OM      <- Oscar Mike
  part_barrel_unique_OM
```

The unique naming part (prefixed `uni_`) is what gives the weapon its red-text name in the game UI. The mandatory parts give the weapon its distinctive behavior — the Zipgun's barrel is what makes it a Zipgun.

10.7. Licensed Parts

Licensed parts are BL4's cross-pollination system. They let weapons gain abilities from other manufacturers — a Jakobs

shotgun with a Tediore reload, or a Vladof rifle with a Maliwan underbarrel.

10.7.1. License Types

License	Effect	Slot
Jakobs Ricochet	Critical hits ricochet to nearby targets	Barrel Acc
Hyperion Shield	Weapon shield while aiming	Barrel Acc
Hyperion Grip	Accuracy bonuses	Grip
Tediore Reload	Throw weapon on reload	Barrel Acc, Grip
Torgue Mag	Explosive magazine	Magazine
Borg Mag	Borg magazine bonus	Magazine
Forge Mag	COV repair mechanic	Magazine
Atlas UB	Atlas underbarrel	Underbarrel
Daedalus UB	Daedalus underbarrel	Underbarrel
Maliwan UB	Maliwan underbarrel	Underbarrel

Table 8.7: Licensed part types.

10.7.2. Tediore Reload Variants

The Tediore reload license has the most variation — six distinct payload types:

Part	Effect
`part_barrel_licensed_ted`	Default throw
`part_barrel_licensed_ted_combo`	Combo reload
`part_barrel_licensed_ted_mirv`	MIRV explosion
`part_barrel_licensed_ted_shooting`	Gun continues shooting mid-air
`part_barrel_licensed_ted_replicator`	Gun replicates on throw
`part_barrel_licensed_ted_replicator_multi`	Multi-replicator variant

Table 8.8: Tediore reload variants.

10.7.3. Weapon-Specific Underbarrels

Some weapons have tiered underbarrel parts that unlock progressively:

- `LicensedPart_WeaponSpecific_UB_Tier0_Vladof` — Vladof-exclusive tier 0
- `LicensedPart_WeaponSpecific_UB_Tier1` through `Tier4` — Tiered unlocks

10.7.4. KL (Killer License) Parts

Certain weapons have dedicated `_KL` suffix slots:

- `DAD_PS_KL` — Daedalus Pistol killer license
- `JAK_PS_KL` — Jakobs Pistol killer license

These appear to be dedicated slots for applying licensed part effects, separate from the standard part slots.

10.8. Serial Index Architecture

To encode a part into a serial string, the game needs a numeric index for every part. These indices are assigned by the `GbxSerialNumberProvider` system at runtime.

10.8.1. GbxSerialNumberIndex Structure

```
GbxSerialNumberIndex:
  Category  : Int64   <- Part group ID
  scope     : Byte    <- Root/Sub scope
  status    : Byte    <- Active/Static/etc.
  Index     : Int16   <- Part index within group
```

Each part is self-describing — the serial format encodes the category, scope, and index together, so a serial can be decoded without knowing the item type in advance.

10.8.2. Part Index Resolution

Part indices in serials map directly to the parts database for the item's category. The index is used as-is — no bit manipulation is needed. With the corrected bit assembly (nibble-reversed VarInt/VarBit values), part indices resolve correctly to the NCS-extracted parts database.

10.8.3. Registration Order

Part indices aren't stored in NCS for most manufacturers. Only BOR (Ripper) parts have inline serial indices embedded as null-terminated strings directly after the part name:

```
BOR_SG_Grip_01\0 42\0 part_grip_02\0 ...
BOR_SG_Grip_02\0 43\0 part_grip_03\0 ...
```

That's 36 BOR parts with extractable indices — roughly 2.2% of the total 1,614 parts. For the other 98%, indices are assigned at runtime when the game engine registers parts with `GbxSerialNumberProvider`. This means the only complete source for serial indices is a memory dump from a running game process.

> ⚠ Warning
>
> Serial index assignments may change between game versions. A parts database extracted from one patch may produce incorrect decodes on another. Always verify against the current game version.

10.9. Part Validation

With the parts data extracted, you can validate whether a given part belongs to a given item type.

10.9.1. Using item_parts.json

```python
import json

with open('share/manifest/item_parts.json') as f:
    items = json.load(f)

# Build lookup: item_id -> set of valid parts
valid_parts = {item['item_id']: set(item['parts']) for item in items}

# Check if a part is valid for an item
def is_valid_part(item_id: str, part_name: str) -> bool:
    return part_name in valid_parts.get(item_id, set())

# Example
is_valid_part('DAD_PS', 'DAD_PS_Barrel_01')  # True
is_valid_part('DAD_PS', 'JAK_PS_Barrel_01')  # False
```

10.9.2. Using the CLI

```bash
# Extract parts for a specific item type
bl4 ncs extract inv4.bin -t item-parts | grep "DAD_PS_"

# Decode a serial to see its category and parts
bl4 serial decode '@Uge8;)m/&zJ!tkr0N4>8ns8H{t!6ljj'
# Output includes: Category: Paladin Class Mod (55)
```

10.9.3. NCS vs. Memory: Source of Truth

Source	Parts for DAD_PS	Status
NCS inv.bin	56	Complete
Memory extraction	34	Incomplete

NCS is the authoritative source for valid parts. Memory extraction misses parts that aren't currently loaded in the game process. Always use NCS-extracted data for part validation. Use memory data only when you need serial indices.

10.10. Level Gating

Parts and features are gated by player level through `Att_MinGameStage_*` attributes. These control when parts can appear on dropped or vendor items — a level 5 character won't find Jakobs Ricochet licensed parts.

10.10.1. Known Level-Gated Categories

Category	Attribute Pattern
Weapon Types	`Att_MinGameStage_WeaponType_*`
Shield Types	`Att_MinGameStage_ShieldType_*`
Manufacturers	`Att_MinGameStage_Manufacturer_*`
Elements	`Att_MinGameStage_Element_*`
Licensed Parts	`Att_MinGameStage_LicensedPart_*`
Enhancement Tiers	`Att_MinGameStage_Enhancement_Stats_Tier*`
Gadgets	`Att_MinGameStage_Gadget_*`

Category	Attribute Pattern

Table 8.9: Level gating categories.

Each licensed part has its own MinGameStage attribute. The full list:

```
Att_MinGameStage_LicensedPart_JakobsRicochet
Att_MinGameStage_LicensedPart_HyperionShield
Att_MinGameStage_LicensedPart_HyperionGrip
Att_MinGameStage_LicensedPart_TedioreReload_TopACC
Att_MinGameStage_LicensedPart_TedioreReload_Grip
Att_MinGameStage_LicensedPart_TorgueMag
Att_MinGameStage_LicensedPart_ForgeMag
Att_MinGameStage_LicensedPart_BorgMag
Att_MinGameStage_LicensedPart_Atlas_UB
Att_MinGameStage_LicensedPart_Daedalus_UB
Att_MinGameStage_LicensedPart_Maliwan_UB
Att_MinGameStage_LicensedPart_WeaponSpecific_UB_Tier0_Vladof
Att_MinGameStage_LicensedPart_WeaponSpecific_UB_Tier1
Att_MinGameStage_LicensedPart_WeaponSpecific_UB_Tier2
Att_MinGameStage_LicensedPart_WeaponSpecific_UB_Tier3
Att_MinGameStage_LicensedPart_WeaponSpecific_UB_Tier4
```

> **Note**
>
> The actual level thresholds (e.g., "Jakobs Ricochet unlocks at level 15") are not stored in NCS files. They're likely defined in binary data tables or engine code and haven't been located yet.

10.11. Firmware

Firmware is an equipment modifier that can be applied to class mods, enhancements, grenades/gadgets, shields, repair kits, and other non-weapon equipment. Each item can have at most one firmware. Firmware grants passive bonuses (e.g., Deadeye improves critical hit damage, Heating Up improves fire rate).

10.11.1. Firmware Parts

Firmware parts exist in per-item-type pools, each with their own indices:

Category	Pool	Items
234	`class_mod-234.tsv`	Class mods
243	`repair_kit-243.tsv`	Repair kits
244	`heavy_weapon_gadget-244.tsv`	Heavy weapon gadgets
245	`grenade_gadget-245.tsv`	Grenade gadgets
246	`shield-246.tsv`	Shields
247	`enhancement-247.tsv`	Enhancements

The same firmware has different indices across pools. Manufacturer-specific items (e.g., 268=Jakobs Enhancement) use their base type’s pool (247=Enhancement).

10.11.2. Serial Encoding

10.11.2.1. Cross-Category Part Values

Equipment items use cross-category Part tokens to reference parts from shared pools. A `Part { index: 246, values: [23] }` on a shield means "part 23 from the shield pool (category 246)".

Part values can be encoded two ways: - **Single**: `Part { index: 246, values: [23], encoding: Single }` — one value inline - **List**: `Part { index: 246, values: [], encoding: List }` followed by `SoftSeparator` + VarInts — values stored as a list in a trailing section

The `SoftSeparator` after a `List`-encoded Part starts a value list. Each subsequent VarInt is an entry in that list, terminated by a `Separator`.

10.11.2.2. How Firmware Is Stored

Firmware is the last entry in the value list of the last cross-category Part token.

Equipment without firmware — the last cross-category Part uses `Single` encoding with its value inline:

```
Part { index: 246, values: [41], encoding: Single }   ← pinpoint perk
Separator
```

Equipment with firmware — the last cross-category Part uses `List` encoding, and firmware is the last VarInt in the list:

```
Part { index: 246, values: [], encoding: List }
SoftSeparator
Var { val: 41 }     ← pinpoint perk (moved from Single value)
Var { val: 9 }      ← firmware index (rubberband_man in shield pool)
Separator
```

When firmware is transferred to an item, the game converts the last Part from `Single(value)` to `List`, moves the original value into the trailing VarInt section, and appends the firmware index.

Items that already have a List-encoded Part (most non-legendary equipment) simply get the firmware VarInt appended to the existing list.

10.11.2.3. Class Mods

Class mods encode firmware differently: as a `Part { index: 234, values: [fw_index], encoding: Single }` token appended after the skill Parts. The firmware Part uses category 234 as both its index and its lookup pool.

```
Part { index: 51, ... }      ← rarity
Part { index: 15, ... }      ← body
Part { index: 223, ... }     ← skill tiers...
...
Part { index: 234, values: [84], encoding: Single }  ← firmware (deadeye)
SoftSeparator
Var { val: 5 }               ← unknown
Var { val: 66 }              ← unknown
Var { val: 84 }              ← firmware index (redundant?)
Separator
```

10.11.2.4. Transfer Flag

The `String("ft")` token in the header means "firmware transferred" — the firmware was moved from another item. Items with original firmware do NOT have this flag. The flag has no effect on how firmware is read; it's metadata about provenance.

10.11.3. Class Mod Skill Interaction

On class mods, the firmware Part token sits after all skill Part tokens. When editing skills, the firmware section must be preserved in its structural position. The `apply_edits` function in `skills.rs` handles this by identifying the contiguous range of passive skill Parts and splicing replacements in-place, leaving firmware and other non-skill tokens untouched.

10.12. Known Gaps

The parts system is roughly 95% mapped, but several areas remain incomplete.

10.12.1. Rainbow Vomit Legendary Parts

The Rainbow Vomit (Jakobs Legendary Shotgun) serves as our benchmark for decode resolution. After the bit 7 discovery, 7 of 10 parts resolve correctly (70%). Three indices remain unresolved: 73, 201, and 206.

These correspond to legendary-specific parts found in pak extraction but absent from the NCS inventory files:

- `part_barrel_RainbowVomit`
- `part_mag_RainbowVomit`
- 10 elemental body variants: `part_body_ele_RainbowVomit_Cor_Fire_Shock`, etc.

Twelve legendary-specific parts total. Their serial indices need manual mapping or memory extraction.

10.12.2. Non-Prefixed Parts

Parts without manufacturer prefixes can't have categories derived from NCS:

- `comp_*` — rarity modifiers that exist in multiple categories
- `part_firmware_*` — no manufacturer prefix
- `part_ra_*` — reactive armor parts with unknown categorization

These require memory dumps or manual mapping.

10.12.3. Missing Class Mod Parts

Three of four class mod categories have no parts in any dump:

Category	Class	Parts Found
44	Dark Siren	0
55	Paladin	0
97	Gravitar	2
140	Exo Soldier	0

Only Gravitar has any parts mapped (`part_grav_asm_legendaryGravitar` and `part_grav_asm_skill_test`). The others require memory dumps taken while the relevant class mods are equipped.

10.12.4. Equipment Low Resolution

Equipment serials (class mods, firmware) decode with improving resolution as the parts database grows. Some part indices still map to nothing in the current database, particularly for equipment categories not yet fully extracted from NCS.

10.13. Key Differences from BL3

If you've worked with Borderlands 3 modding, the parts system will feel familiar in concept but different in every implementation detail.

Aspect	BL3	BL4
Part definitions	PartSet/PartPool uassets	NCS inv.bin (native C++ objects)
Part lists	GestaltPartListData for all items	Gestalt only for AI/creatures
Weapon assets	Include part pool references	Mesh components only

Aspect	BL3	BL4
Validation	Check PartSet for weapon type	Check NCS composition
Category mappings	Extractable from pak	Runtime-assigned, memory-only
Serial indices	Static in assets	Runtime-assigned by GbxSerialNumberProvider

Table 8.10: BL3 vs. BL4 parts system comparison.

The biggest practical difference: in BL3, a motivated person with FModel could extract complete part data from the pak files alone. In BL4, you need NCS parsing for part lists *and* memory extraction for serial indices. No single source gives you everything.

10.14. Unresolved Questions

A few aspects of the parts system remain unproven:

1. **Rarity filtering** — Parts have `firmware_weight_XX_rarity` values in NCS. It's unclear whether common-weighted parts can appear on legendary items.

2. **License availability** — We've confirmed that some weapons accept specific licenses (e.g., BOR SG can have Jakobs/Hyperion/Tediore licenses), but the complete mapping of which weapons accept which licenses is incomplete.
3. **Per-item restrictions** — Legendaries may impose restrictions through their compositions beyond what the base part list shows.

These gaps are tractable. More memory dumps across different characters, levels, and loadouts will fill them in. The next chapter covers the bl4 CLI tools that make this extraction practical.

11. Chapter 9: Using bl4 Tools

Everything we've learned—binary decoding, memory analysis, save file encryption, serial parsing—comes together in the bl4 command-line tools. This chapter serves as your practical reference for day-to-day use.

The tools are designed to be composable. Pipe output between commands. Chain operations together. Build your own workflows for tasks we haven't anticipated.

11.1. Building the Tools

11.1.1. Prerequisites

```
# Install Rust (if you haven't already)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env

# Clone the repository
git clone https://github.com/monokrome/bl4
cd bl4
```

11.1.2. Build

```
cargo build --release -p bl4-cli

# Binary appears in ./target/release/bl4
```

The `uextract` tool builds separately:

```
cargo build --release -p uextract

# Binary appears in ./target/release/uextract
```

11.2. CLI Structure

The bl4 CLI uses subcommands organized by function:

```
bl4 <COMMAND>

Commands:
  save       Save file operations (decrypt, encrypt, edit, get, set)
  inspect    Inspect a save file (decrypt and display info)
  configure  Configure default settings
  serial     Item serial operations (decode, encode, compare, modify)
  parts      Query parts database
  memory     Read/analyze game memory (live process or dump file)
  ncs        NCS format operations (decompress, scan, show, search, ex
  idb        Manage the verified items database
  drops      Query drop rates and locations for legendary items
  launch     Launch Borderlands 4 with instrumentation
```

Aliases exist for common commands: `s` (save), `i` (inspect), `r` (serial), `m` (memory), `n` (ncs), `d` (drops), `p` (parts), `l` (launch).

11.3. Save File Operations

11.3.1. Inspect a Save

Quick view of save contents:

```
bl4 inspect 1.sav
bl4 inspect 1.sav --full  # Show complete YAML
```

11.3.2. Decrypt/Encrypt

```
# Decrypt to stdout
bl4 save decrypt 1.sav

# Decrypt to file
bl4 save decrypt 1.sav character.yaml

# Encrypt back
bl4 save encrypt character.yaml 1.sav
```

Steam ID is auto-detected from configuration or can be specified:

```
bl4 save decrypt 1.sav --steam-id 76561198012345678
```

11.3.3. Edit Interactively

Opens decrypted save in your `$EDITOR`, then re-encrypts on save:

```
bl4 save edit 1.sav
```

11.3.4. Get/Set Values

Query specific paths:

```
bl4 save get 1.sav "state.currencies.cash"
bl4 save get 1.sav --level   # Character level
bl4 save get 1.sav --money   # Currencies
bl4 save get 1.sav --all     # Everything
```

Set values:

```
bl4 save set 1.sav "state.currencies.cash" 999999999
```

11.4. Serial Operations

11.4.1. Decode

```
bl4 serial decode '@Ugr$ZCm/&tH!t{KgK/Shxu>k'
```

Output:

```
Serial: @Ugr$ZCm/&tH!t{KgK/Shxu>k
Item type: r (Item)
Category: Vladof SMG (22)
Level: 50
Tokens: 180928 | 50 | {0:1} 21 {4} , 2 , , 105 102 41
```

> **Part Name Resolution**
>
> Part names are resolved from `share/manifest/parts_database.json`. Currently ~40% of parts have known mappings from memory extraction. Unknown parts display as `[category:index]` placeholders (e.g., `[22:5]`). See Chapter 7 for details on part data coverage.

Options:

```
bl4 serial decode --verbose '@Ugr...'  # Byte breakdown
bl4 serial decode --debug '@Ugr...'    # Bit-by-bit parsing
bl4 serial decode --analyze '@Ugr...'  # Token analysis
bl4 serial decode --rarity '@Ugr...'   # Show rarity estimate
```

The `--rarity` flag adds a rarity estimate below the decode output, showing tier probability, pool size, per-item odds, and known boss sources:

```
Rarity estimate:
  Tier: Legendary (0.000283%, ~1 in 353,490)
  Pool: Jakobs Shotgun (2 legendaries, 9 in world pool)
  Per-item: ~1 in 3,181,413
  Boss sources: 2
```

11.4.2. Compare

Side-by-side comparison of two serials:

```
bl4 serial compare '@Ugr$ZCm...' '@Ugr$ABC...'
```

11.4.3. Modify

Swap parts between serials:

```
bl4 serial modify '@base...' '@source...' '4,12'
```

11.4.4. Batch Decode

Decode many serials to binary for analysis:

```
bl4 serial batch-decode serials.txt serials.bin
```

11.5. Drop Rate Queries

The `drops` command queries a manifest of legendary item drop sources, extracted from NCS data. It tells you where items drop and what each source drops.

11.5.1. Find Where an Item Drops

```
bl4 drops find Hellwalker
bl4 drops find "Plasma Coil"
```

Output is sorted by drop rate (highest first), showing the source, source type, tier, and chance for each location.

11.5.2. List All Drops from a Source

```
bl4 drops source Timekeeper
bl4 drops source "Black Market"
bl4 drops source "Fish Collector"
```

11.5.3. List All Known Items or Sources

```
bl4 drops list              # List all known legendary items
bl4 drops list --sources    # List all known drop sources
```

11.5.4. Generate Drops Manifest

Build the drops manifest from decompressed NCS data:

```
bl4 drops generate ./ncs_output/ -o share/manifest/drops.json --manife
```

This generates both `drops.json` and `drop_pools.tsv` (pool summary with legendary counts and boss source counts per manufacturer/weapon type). The `drop_pools.tsv` is embedded at compile time for the rarity estimation API.

By default, the `find`, `source`, and `list` commands use `share/manifest/drops.json`. Override with `--manifest <PATH>`.

11.6. Items Database (idb)

The items database tracks verified item data with source attribution.

11.6.1. Basic Operations

```
bl4 idb init                        # Create database
bl4 idb stats                       # Show counts
bl4 idb list                        # List all items
bl4 idb show '@Ugr...'              # Show item details
```

The `list` command supports filtering and output format control:

```
bl4 idb list --manufacturer JAK --rarity Legendary
bl4 idb list --format json
bl4 idb list --format csv -f serial,name,rarity
```

11.6.2. Import Items

```
# From save file
bl4 idb import-save 1.sav --decode --legal --source monokrome

# From share/weapons directory structure
bl4 idb import share/weapons/

# Decode all and populate metadata
bl4 idb decode-all
```

11.6.3. Attachments

```
bl4 idb attach '@Ugr...' screenshot.png
bl4 idb attach '@Ugr...' card.png --popup    # Mark as item card view
bl4 idb attach '@Ugr...' inspect.png --detail # Mark as 3D inspect view
```

11.6.4. Value Attribution

Track values from different sources:

```
bl4 idb set-value '@Ugr...' rarity Epic --source ingame --confidence v
bl4 idb get-values '@Ugr...' rarity
```

11.6.5. Verification and Curation

```
bl4 idb verify '@Ugr...' verified --notes "Confirmed in-game screensho
bl4 idb mark-legal '@Ugr...'
bl4 idb mark-legal all                   # Mark everything as legal
bl4 idb set-source monokrome '@Ugr...'
bl4 idb set-source community --where "legal = 0"
```

11.6.6. Export and Merge

```
bl4 idb export '@Ugr...' ./item_dir/
bl4 idb merge source.db destination.db
```

11.6.7. Community Server Sync

Publish items to and pull items from the community server:

```
bl4 idb publish                              # Publish all items
bl4 idb publish --serial '@Ugr...'           # Publish one item
bl4 idb publish --attachments --dry-run      # Preview with screenshots

bl4 idb pull                                 # Pull all items
bl4 idb pull --authoritative                 # Prefer remote values
bl4 idb pull --dry-run                       # Preview what would chang
```

The default server is `https://items.bl4.dev`. Override with `--server <URL>`.

11.6.8. Database Maintenance

```
bl4 idb migrate-values              # Migrate column values to item_values table
bl4 idb migrate-values --dry-run    # Preview migration
```

11.7. Memory Operations

Memory commands work with live processes or dump files.

11.7.1. With Dump File

```
bl4 memory --dump game.dmp info
bl4 memory --dump game.dmp discover gnames
bl4 memory --dump game.dmp discover guobjectarray
```

11.7.2. Generate Usmap

```
bl4 memory --dump game.dmp dump-usmap
```

11.7.3. FName Operations

```
bl4 memory --dump game.dmp fname 12345
bl4 memory --dump game.dmp fname-search "Damage"
```

11.7.4. Object Inspection

```
bl4 memory --dump game.dmp objects --class ItemPoolDef --limit 50
bl4 memory --dump game.dmp list-objects --class-filter "Part" --limit
bl4 memory --dump game.dmp list-objects --stats
bl4 memory --dump game.dmp list-uclasses --filter "Inventory"
bl4 memory --dump game.dmp find-class-uclass
```

11.7.5. Parts Extraction

```
bl4 memory --dump game.dmp dump-parts -o parts.json
bl4 memory --dump game.dmp build-parts-db -i parts.json -o parts_db.js
```

Extract parts with authoritative Category/Index values from UObjects:

```
bl4 memory --dump game.dmp extract-parts -o parts_with_categories.json
bl4 memory --dump game.dmp extract-parts --list-fnames    # Debug: lis
bl4 memory --dump game.dmp extract-parts-raw -o parts_raw.json
```

11.7.6. Object Discovery

Find objects matching a name pattern and generate lookup maps:

```
bl4 memory --dump game.dmp find-objects-by-pattern ".part_" --limit 20
bl4 memory --dump game.dmp generate-object-map -o objects.json
```

11.7.7. NCS Schema Extraction

Extract NCS field hash-to-name mappings from process memory:

```
bl4 memory --dump game.dmp extract-ncs-schema -o share/manifests/bl4.ncsmap
```

11.7.8. String Search

```
bl4 memory --dump game.dmp scan-string "DAD_AR.part_body" -B 128 -A 128
```

11.7.9. Low-Level Memory Access

These commands work with live processes (no dump file):

```
bl4 memory read 0x7f1234567890 --size 256
bl4 memory write 0x7f1234567890 "90 90 90"
bl4 memory scan "48 8B 05 ?? ?? ?? ??"
bl4 memory patch 0x7f1234567890 --nop 5
bl4 memory patch 0x7f1234567890 --bytes "EB 05"
```

11.7.10. Preload Library

The preload library intercepts file I/O for NCS extraction and debugging:

```
bl4 memory preload info                     # Show LD_PRELOAD command
bl4 memory preload run --capture ./out -- wine64 game.exe
bl4 memory preload watch                    # Tail the preload log
bl4 memory monitor --filter "fopen"         # Monitor log with functio
```

11.8. NCS Operations

NCS (Nexus Config Store) contains item pools, loot config, and other game data not in standard PAK assets.

11.8.1. Decompress from Pak

```
# Extract all NCS chunks from a pak file
bl4 ncs decompress pakchunk0.pak -o ./ncs_output/

# Extract from specific offset
bl4 ncs decompress pakchunk0.pak --offset 0x15835

# Use native Oodle DLL for 100% compatibility (Windows only)
bl4 ncs decompress pakchunk0.pak -o ./ncs_output/ --oodle-dll /path/to
```

```
# Use external command for Oodle decompression (cross-platform)
bl4 ncs decompress pakchunk0.pak -o ./ncs_output/ --oodle-exec ./oodle_wrapper.sh
```

The `--oodle-exec` command receives `decompress <size>` arguments, compressed data via stdin, and outputs decompressed data to stdout.

11.8.2. Scan Decompressed Files

```
# List all NCS types in a directory
bl4 ncs scan ./ncs_output/

# Filter by type
bl4 ncs scan ./ncs_output/ -t itempool

# Show detailed info
bl4 ncs scan ./ncs_output/ --verbose

# Output as JSON
bl4 ncs scan ./ncs_output/ --json
```

11.8.3. Show File Contents

```
# Display parsed content
bl4 ncs show ./ncs_output/itempool0.bin

# Show all strings
bl4 ncs show ./ncs_output/itempool0.bin --all-strings
```

```
# Output as JSON or TSV
bl4 ncs show ./ncs_output/itempool0.bin --json
bl4 ncs show ./ncs_output/itempool0.bin --tsv
```

11.8.4. Search

```
# Search for pattern in entry names
bl4 ncs search ./ncs_output/ "legendary"

# Search all strings
bl4 ncs search ./ncs_output/ "damage" --all

# Limit results
bl4 ncs search ./ncs_output/ "barrel" --all -n 50
```

11.8.5. Extract Data

The `extract` command pulls structured data from NCS files. The `-t` flag selects the extraction type:

```
# Extract categorized parts manifest (produces parts_database.json + c
bl4 ncs extract ./ncs_output/ -t manifest -o share/manifest/parts_data

# Extract part serial indices from inv.bin
bl4 ncs extract ./ncs_output/ -t parts

# Extract item-to-parts mapping
bl4 ncs extract ./ncs_output/ -t item-parts --json
```

```
# Extract NexusSerialized display name mappings
bl4 ncs extract ./ncs_output/ -t names

# Extract manufacturer code-to-name mappings
bl4 ncs extract ./ncs_output/ -t manufacturers --json

# Extract raw string table
bl4 ncs extract ./ncs_output/ -t strings

# Extract string-numeric pairs
bl4 ncs extract ./ncs_output/ -t pairs

# Build serial index decoder from all inv*.bin files
bl4 ncs extract ./ncs_output/ -t decoder --json

# Extract using the binary parser (structured document output)
bl4 ncs extract ./ncs_output/ -t binary --json

# Or extract by NCS type name directly (e.g., itempool, rarity)
bl4 ncs extract ./ncs_output/ -t itempool --json
```

11.8.6. Debug

Inspect the binary structure of an NCS file:

```
bl4 ncs debug ./ncs_output/inv0.bin
bl4 ncs debug ./ncs_output/inv0.bin --hex         # Show hex dump
bl4 ncs debug ./ncs_output/inv0.bin --parse        # Parse binary with bit reader
bl4 ncs debug ./ncs_output/inv0.bin --offsets      # Show all section offsets
```

11.8.7. Statistics

```
bl4 ncs stats ./ncs_output/
bl4 ncs stats ./ncs_output/ --formats  # Show format code breakdown
```

11.9. Launch

Launch BL4 through Steam with the preload instrumentation library:

```
bl4 launch          # Shows Steam launch options, prompts for confirma
bl4 launch -y       # Skip confirmation prompt
```

This finds `libbl4_preload.so`, prints the `LD_PRELOAD` launch option string for Steam, and optionally launches the game via `steam://rungameid/1285190`. The preload library must be built first:

```
cargo build --release -p bl4-preload
```

Environment variables control preload behavior: `BL4_RNG_BIAS` (max/high/low/min), `BL4_PRELOAD_ALL` (1 to intercept all I/O), `BL4_PRELOAD_STACKS` (1 for stack traces). The log goes to `/tmp/bl4_preload.log`.

11.10. uextract

The `uextract` tool handles UE5 IoStore asset extraction—a separate binary from bl4, focused on unpacking game assets.

11.10.1. Extract from IoStore

```
# Extract all assets to JSON + uasset
uextract /path/to/Paks -o ./extracted/

# Extract with property schema and class resolution
uextract /path/to/Paks -o ./extracted/ --usmap BL4.usmap --scriptobjects scriptobject

# Filter by path and class
uextract /path/to/Paks -o ./extracted/ --filter "ItemPool" --class-filter "ItemPoolDe

# List matching files without extracting
uextract /path/to/Paks --list --filter "Weapon"

# JSON only, no raw uasset
uextract /path/to/Paks -o ./extracted/ --format json
```

11.10.2. Extract from Traditional Pak Files

```
uextract pak /path/to/file.pak -o ./extracted/
uextract pak /path/to/file.pak --extension ncs --list
```

11.10.3. Dump ScriptObjects

Generate the class hash-to-path lookup used by other commands:

```
uextract script-objects /path/to/Paks -o scriptobjects.json
```

11.10.4. Find Assets by Class

```
uextract find-by-class /path/to/Paks InventoryPartDef --scriptobjects
uextract find-by-class /path/to/Paks ItemPoolDef -o itempool_paths.txt
```

11.10.5. List Classes

```
uextract list-classes /path/to/Paks --scriptobjects scriptobjects.json
```

11.11. Configuration

Set defaults to avoid repetition:

```
bl4 configure --steam-id 76561198012345678
bl4 configure --show
```

Environment variable `BL4_ITEMS_DB` sets the default items database path.

11.12. Common Workflows

11.12.1. Edit a Save

```
bl4 save edit ~/.steam/.../1.sav
# Editor opens, make changes, save & quit
# File is automatically re-encrypted
```

11.12.2. Analyze an Item

```
# Extract serial from save
bl4 save get 1.sav "state.inventory.items[0].serial"

# Decode it
bl4 serial decode '@Ugr...'

# Look up in database
bl4 idb show '@Ugr...'

# Check where it drops
bl4 drops find "Hellwalker"
```

11.12.3. Import Items from Saves

```
for sav in saves/*.sav; do
  bl4 idb import-save "$sav" --decode --legal
done
bl4 idb stats
```

11.12.4. Update After Game Patch

```
# New memory dump
sudo gcore -o bl4_new $(pgrep -f wine64-preloader)

# Generate new usmap
bl4 memory --dump bl4_new.* dump-usmap

# Re-extract parts
bl4 memory --dump bl4_new.* extract-parts -o share/manifest/

# Rebuild parts manifest from NCS
bl4 ncs extract ./ncs_output/ -t manifest -o share/manifest/parts_data

# Regenerate drops manifest
bl4 drops generate ./ncs_output/ -o share/manifest/drops.json --manife
```

11.12.5. Full Asset Extraction Pipeline

```
# 1. Dump ScriptObjects for class resolution
uextract script-objects /path/to/Paks -o scriptobjects.json
```

```
# 2. Extract all assets with property schema
uextract /path/to/Paks -o ./extracted/ --usmap BL4.usmap --scriptobjects scriptobjec

# 3. Find specific asset types
uextract find-by-class /path/to/Paks ItemPoolDef --scriptobjects scriptobjects.json
```

11.13. Shell Tips

11.13.1. Quoting Serials

Serials contain `$`, `!`, `@`. Always use single quotes:

```
bl4 serial decode '@Ugr$ZCm/&tH!t{KgK/Shxu>k'
```

11.13.2. Aliases

```
alias bl4d='bl4 serial decode'
alias bl4i='bl4 inspect'
alias bl4e='bl4 save edit'
alias bl4f='bl4 drops find'
```

11.13.3. Piping

```
bl4 serial decode '@Ugr...' | grep "Category"
bl4 idb list | wc -l
bl4 drops list --sources | sort
```

11.14. Troubleshooting

11.14.1. "Decryption failed"

- Wrong Steam ID
- Corrupted save file
- Not a BL4 save

Verify the Steam ID matches the save file path.

11.14.2. "Invalid serial"

- Missing `@Ug` prefix
- Truncated copy
- Wrong quote type in shell

Copy the complete serial and use single quotes.

11.14.3. "Memory read failed"

- Address outside dump range
- Corrupted dump

Verify the dump covers the target address.

11.14.4. "Preload library not found"

Build the preload library before using `bl4 launch`:

```
cargo build --release -p bl4-preload
```

11.14.5. "Failed to load drops manifest"

Generate the drops manifest first, or point `--manifest` at an existing one:

```
bl4 drops generate ./ncs_output/ -o share/manifest/drops.json --manifest-dir share/ma
```

11.15. Quick Reference

Command	Description
Save	
`bl4 inspect <FILE>`	Quick save inspection
`bl4 save decrypt <IN> [OUT]`	Decrypt save to YAML
`bl4 save encrypt <IN> <OUT>`	Encrypt YAML to save
`bl4 save edit <FILE>`	Edit in $EDITOR
`bl4 save get <FILE> <PATH>`	Query value

Command	Description
bl4 save set <FILE> <PATH> <VAL>	Set value
Serial	
bl4 serial decode <SERIAL> [--rarity]	Decode item serial (with rarity estimate)
bl4 serial compare <S1> <S2>	Compare serials
bl4 serial modify <BASE> <SRC> <PARTS>	Swap parts between serials
bl4 serial batch-decode <IN> <OUT>	Batch decode to binary
Drops	
bl4 drops find <ITEM>	Find where an item drops
bl4 drops source <SOURCE>	List items from a source
bl4 drops list [--sources]	List all items or sources
bl4 drops generate <DIR> -o <OUT> [--manifest-dir <DIR>]	Generate drops manifest + pool TSV
NCS	
bl4 ncs decompress <PAK> -o <DIR>	Extract NCS from pak (--oodle-exec for full compat)
bl4 ncs scan <DIR>	List NCS types
bl4 ncs show <FILE>	Show NCS contents
bl4 ncs search <DIR> <PATTERN>	Search NCS files
bl4 ncs extract <DIR> -t <TYPE>	Extract structured data
bl4 ncs debug <FILE>	Debug binary structure
bl4 ncs stats <DIR>	Show NCS statistics

Command	Description
Items DB	
`bl4 idb init`	Create database
`bl4 idb stats`	Database statistics
`bl4 idb list`	List items (supports `--format`, `--manufacturer`, etc.)
`bl4 idb show <SERIAL>`	Show item details
`bl4 idb import-save <FILE>`	Import from save
`bl4 idb verify <SERIAL> <STATUS>`	Set verification status
`bl4 idb export <SERIAL> <DIR>`	Export item to directory
`bl4 idb merge <SRC> <DEST>`	Merge databases
`bl4 idb publish`	Publish to community server
`bl4 idb pull`	Pull from community server
`bl4 idb set-source <SRC> <IDS...>`	Set item source
`bl4 idb mark-legal <IDS...>`	Mark items as legal
`bl4 idb migrate-values`	Migrate to item_values table
Memory	
`bl4 memory --dump <F> info`	Process/dump info
`bl4 memory --dump <F> discover <TARGET>`	Discover UE5 structures
`bl4 memory --dump <F> dump-usmap`	Generate usmap file
`bl4 memory --dump <F> fname <INDEX>`	Look up FName
`bl4 memory --dump <F> extract-parts`	Extract parts from UObjects

Command	Description
`bl4 memory --dump <F> extract-parts-raw`	Extract raw part data
`bl4 memory --dump <F> find-objects-by-pattern <PAT>`	Find objects by name
`bl4 memory --dump <F> generate-object-map`	Generate object map JSON
`bl4 memory --dump <F> extract-ncs-schema`	Extract NCS field schema
`bl4 memory preload info`	Show LD_PRELOAD command
Launch	
`bl4 launch [-y]`	Launch BL4 with instrumentation
uextract	
`uextract <PAKS> -o <DIR>`	Extract IoStore assets
`uextract pak <FILE> -o <DIR>`	Extract from .pak files
`uextract script-objects <PAKS> -o <OUT>`	Dump ScriptObjects to JSON
`uextract find-by-class <PAKS> <CLASS>`	Find assets by class
`uextract list-classes <PAKS>`	List unique class hashes

11.16. What’s Next?

The appendices provide deep reference material:

- **Appendix A: SDK Class Layouts** — Memory layouts for key UE5 classes
- **Appendix B: Weapon Parts Reference** — Complete parts catalog
- **Appendix C: Loot System Internals** — Drop pools and rarity
- **Appendix D: Game File Structure** — Asset organization
- **Glossary** — Terms and quick reference

Part IV.

Appendices

12. Appendix A: SDK Class Layouts

This appendix provides detailed memory layouts for BL4's core classes, extracted from SDK analysis. These are essential for memory analysis, save editing tools, or deep reverse engineering.

12.1. Core UE5 Types

12.1.1. FName (8 bytes)

```
struct FName {
    int32_t ComparisonIndex;  // 0x00 - Index into GNames pool
    int32_t Number;           // 0x04 - Instance number (e.g., "Actor_5")
};
```

12.1.2. FVector (24 bytes)

```
struct FVector {
    double X;  // 0x00
    double Y;  // 0x08
    double Z;  // 0x10
};
```

> **Warning**
>
> UE5 uses `double` (8 bytes) for vectors, not `float` like UE4. This is a common source of offset calculation errors.

12.1.3. FRotator (24 bytes)

```
struct FRotator {
    double Pitch;  // 0x00
    double Yaw;    // 0x08
    double Roll;   // 0x10
};
```

12.1.4. FQuat (32 bytes)

```
struct FQuat {
    double X;  // 0x00
    double Y;  // 0x08
    double Z;  // 0x10
    double W;  // 0x18
};
```

12.1.5. FTransform (96 bytes)

```
struct FTransform {
    FQuat Rotation;        // 0x00 (32 bytes)
    FVector Translation;  // 0x20 (24 bytes)
    char pad[8];          // 0x38
    FVector Scale3D;      // 0x40 (24 bytes)
    char pad[8];          // 0x58
};  // Total: 0x60
```

12.1.6. TArray (16 bytes)

```
template<typename T>
struct TArray {
    T* Data;          // 0x00 - Pointer to element array
    int32_t Count;    // 0x08 - Current element count
    int32_t Max;      // 0x0C - Allocated capacity
};
```

12.1.7. FString (16 bytes)

```
struct FString {
    wchar_t* Data;    // 0x00 - Wide string pointer
    int32_t Count;    // 0x08 - String length
    int32_t Max;      // 0x0C - Buffer capacity
};
```

12.2. UObject Hierarchy

12.2.1. UObject (40 bytes)

The base class for all Unreal objects.

```
class UObject {
    void* VTable;              // 0x00 - Virtual function table
    int32_t ObjectFlags;       // 0x08 - RF_* flags
    int32_t InternalIndex;     // 0x0C - Index in GUObjectArray
    UClass* ClassPrivate;      // 0x10 - Pointer to this object's class
    FName NamePrivate;         // 0x18 - Object's name
    UObject* OuterPrivate;     // 0x20 - Parent/container object
};  // Total: 0x28
```

Offset	Size	Field	Purpose
0x00	8	VTable	Points to virtual function table
0x08	4	ObjectFlags	Object state flags
0x0C	4	InternalIndex	Position in GUObjectArray
0x10	8	ClassPrivate	Pointer to UClass
0x18	8	NamePrivate	FName (index + number)
0x20	8	OuterPrivate	Package/container pointer

12.2.2. UField (48 bytes)

```
class UField : public UObject {
    UField* Next;  // 0x28 - Next field in chain
};
```

12.2.3. UStruct (176 bytes)

```
class UStruct : public UField {
    char pad[16];               // 0x30
    UStruct* SuperStruct;       // 0x40 - Parent class/struct
    UField* Children;           // 0x48 - First child field (legacy)
    FField* ChildProperties;    // 0x50 - Property linked list (UE5)
    int32_t PropertiesSize;     // 0x58 - Total size of properties
    int16_t MinAlignment;       // 0x5C
    char pad[82];               // 0x5E
};  // Total: 0xB0
```

12.2.4. UClass (512 bytes)

```
class UClass : public UStruct {
    char pad[96];               // 0xB0
    UObject* ClassDefaultObject;  // 0x110 - CDO pointer
    char pad[232];              // 0x118
};  // Total: 0x200
```

12.3. Actor Hierarchy

12.3.1. AActor (912 bytes)

```
class AActor : public UObject {
    char pad[416];                        // 0x28
    USceneComponent* RootComponent;       // 0x1C8
    char pad[448];                        // 0x1D0
};  // Total: 0x390
```

12.3.2. APawn (1040 bytes)

```
class APawn : public AActor {
    char pad[32];                    // 0x390
    APlayerState* PlayerState;       // 0x3B0
    char pad[8];                     // 0x3B8
    AController* Controller;         // 0x3C0
    char pad[72];                    // 0x3C8
};  // Total: 0x410
```

12.3.3. ACharacter (1864 bytes)

```
class ACharacter : public APawn {
    char pad[24];                                     // 0x410
    USkeletalMeshComponent* Mesh;                     // 0x428
    UCharacterMovementComponent* Movement;            // 0x430
    char pad[784];                                    // 0x438
};  // Total: 0x748
```

12.4. Controller Classes

12.4.1. AController (1064 bytes)

```
class AController : public AActor {
    char pad[8];                      // 0x390
    APlayerState* PlayerState;    // 0x398
    char pad[48];                     // 0x3A0
    APawn* Pawn;                      // 0x3D0
    char pad[8];                      // 0x3D8
    ACharacter* Character;        // 0x3E0
    char pad[64];                     // 0x3E8
};  // Total: 0x428
```

12.4.2. APlayerController (2392+ bytes)

```
class APlayerController : public AController {
    char pad[16];                                 // 0x428
    APawn* AcknowledgedPawn;                      // 0x438
    char pad[8];                                  // 0x440
    APlayerCameraManager* CameraManager;    // 0x448
    char pad[168];                                // 0x450
    UCheatManager* CheatManager;              // 0x4F8
    UClass* CheatClass;                           // 0x500
    char pad[1104];                               // 0x508
};  // Total: 0x958+
```

12.5. BL4/Oak Classes

12.5.1. AGbxPlayerController (3496 bytes)

```
class AGbxPlayerController : public APlayerController {
    char pad[176];                          // 0x958
    ACharacter* PrimaryCharacter;           // 0xA08
    char pad[536];                          // 0xA10
    bool bUseGbxCurrencyManager;            // 0xC28
    char pad[7];                            // 0xC29
    UGbxCurrencyManager* CurrencyManager;   // 0xC30
    char pad[288];                          // 0xC38
    bool bUseRewardsManager;                // 0xD58
    char pad[7];                            // 0xD59
    UGbxRewardsManager* RewardsManager;     // 0xD60
    char pad[64];                           // 0xD68
};  // Total: 0xDA8
```

12.5.2. AOakPlayerController (15424 bytes)

```
class AOakPlayerController : public AGbxPlayerController {
    char pad[376];                      // 0xDA8
    AOakCharacter* OakCharacter;        // 0xF20
    char pad[8880];                     // 0xF28
    bool bIsCurrentlyTargeted;          // 0x31D8
    char pad[48];                       // 0x31D9
    bool bFullyAimingAtTarget;          // 0x3209
    // ... more fields
};  // Total: 0x3C40
```

12.5.3. AGbxCharacter (15232 bytes)

```
class AGbxCharacter : public ACharacter {
    char pad[13368];  // 0x748
};  // Total: 0x3B80
```

12.5.4. AOakCharacter (38800 bytes)

The main player/enemy character class. This is one of the largest classes in the game.

```
class AOakCharacter : public AGbxCharacter {
    char pad[1208];                              // 0x3B80
    FOakDamageState DamageState;                 // 0x4038 (size: 0x608)
    FOakCharacterHealthState HealthState;        // 0x4640 (size: 0x1E8)
    char pad[4976];                              // 0x4828
    ECharacterHealthCondition HealthCondition; // 0x5B98
    char pad[951];                               // 0x5B99
    FOakActiveWeaponsState ActiveWeapons;        // 0x5F50 (size: 0x210)
    char pad[960];                               // 0x6160
    FDownState DownState;                        // 0x6F40 (size: 0x398)
    char pad[8];                                 // 0x72D8
    AOakCharacter* ActorBeingRevived;            // 0x72E0
    // ... many more fields
    FGbxAttributeFloat AmmoRegenerate;           // 0x95E8
};  // Total: 0x9790
```

Key offsets for AOakCharacter:

Offset	Size	Field	Description
0x4038	0x608	DamageState	Damage tracking state
0x4640	0x1E8	HealthState	Health/shield values
0x5B98	1	HealthCondition	Healthy/Injured/Dead enum
0x5F50	0x210	ActiveWeapons	Equipped weapon state
0x6F40	0x398	DownState	FFYL state
0x72E0	8	ActorBeingRevived	Revive target pointer

12.6. Inventory Classes

12.6.1. AInventory (2224 bytes)

```
class AInventory : public AActor {
    char pad[1312];  // 0x390
};  // Total: 0x8B0
```

12.6.2. AWeapon (3400 bytes)

```
class AWeapon : public AInventory {
    char pad[912];                              // 0x8B0
    FDamageModifierData DamageModifierData;     // 0xC40 (size: 0x6C)
    char pad[12];                               // 0xCAC
    FGbxAttributeFloat ZoomTimeScale;           // 0xCB8
    char pad[132];                              // 0xCC4
};  // Total: 0xD48
```

12.7. Currency System

12.7.1. FSToken (12 bytes)

```
struct FSToken {
    int32_t Hash;  // 0x00 - Token hash
    FName Name;    // 0x04 - Token name
};
```

12.7.2. FGbxCurrency (24 bytes)

```
struct FGbxCurrency {
    FSToken Token;   // 0x00
    char pad[4];     // 0x0C
    uint64_t Amount; // 0x10 - Currency amount
};
```

12.7.3. UGbxCurrencyManager (64 bytes)

```
class UGbxCurrencyManager : public UObject {
    char pad[8];                      // 0x28
    TArray<FGbxCurrency> Currencies;  // 0x30
};
```

Currency indices:

Index	Currency
0	Cash
1	Eridium
2	Gold Keys
3	Unknown

12.8. Attribute Types

12.8.1. FGbxAttributeFloat (12 bytes)

```
struct FGbxAttributeFloat {
    char pad[4];       // 0x00
    float Value;       // 0x04 - Current value
    float BaseValue;   // 0x08 - Base value before modifiers
};
```

12.8.2. FGbxAttributeInteger (12 bytes)

```
struct FGbxAttributeInteger {
    char pad[4];        // 0x00
    int32_t Value;      // 0x04
    int32_t BaseValue;  // 0x08
};
```

12.9. Enums

12.9.1. ECharacterHealthCondition

```
enum class ECharacterHealthCondition : int8_t {
    Healthy = 0,
    Injured = 1,
    Dead = 2
};
```

12.9.2. EMovementMode

```
enum class EMovementMode : int8_t {
    MOVE_None = 0,
    MOVE_Walking = 1,
    MOVE_NavWalking = 2,
    MOVE_Falling = 3,
    MOVE_Swimming = 4,
    MOVE_Flying = 5,
    MOVE_Custom = 6,
    MOVE_MAX = 7
};
```

12.10. Global Pointers

These offsets are from the PE image base (0x140000000):

Global	Offset	Virtual Address	Description
GUObjectArray	0x113878f0	0x1513878f0	All UObjects
GNames	0x112a1c80	0x1512a1c80	FName string pool
GWorld	0x11532cb8	0x151532cb8	Current world
ProcessEvent	0x14f7010	0x144f7010	Event dispatcher

> **Note**
>
> These offsets are from the November 2025 patch. Earlier versions had offsets 0x1000 lower.

12.11. Pattern Signatures

For finding globals via code scanning:

```
GNames:   48 8D 0D ? ? ? ? E8 ? ? ? ? C6 05 ? ? ? ? ? 8B 05
GObjects: 48 8B 15 ? ? ? ? C1 E8 ? 48 8D 0C 49 C1 E1 ? 48 03
GWorld:   48 8B 05 ? ? ? ? 48 89 44 24 ? 48 8D 54 24 ? 4C 8D
```

The `?` bytes are wildcards. Calculate target address from RIP-relative offset in the instruction.

12.12. Mesh & Visibility

Component Offset	Field	Description
Mesh + 0x39C	LastSubmitTime	Last frame submitted
Mesh + 0x3A0	LastRenderTimeOnScreen	Last visible frame

Visibility check: If `LastSubmitTime > LastRenderTimeOnScreen`, the mesh was occluded (behind a wall).

12.13. FProperty Layout

For parsing reflection data:

```
// FField base (shared by all field types)
struct FField {
    void* VTable;           // 0x00
    UStruct* Owner;         // 0x08
    FField* Next;           // 0x10
    FName NamePrivate;      // 0x18
    uint32_t FlagsPrivate;  // 0x20
};

// FProperty extends FField
struct FProperty : FField {
    int32_t ArrayDim;       // 0x28 - Array size (1 for non-arrays)
    int32_t ElementSize;    // 0x2C - Size of one element
    uint64_t PropertyFlags; // 0x30 - CPF_* flags
```

```
    uint16_t RepIndex;      // 0x38
    char pad[2];            // 0x3A
    int32_t Offset_Internal;// 0x3C - Byte offset in struct
    // Type-specific data at 0x40+
};
```

These layouts are from SDK dumps as of November 2025. Offsets may change with game patches.

13. Appendix B: Weapon Parts Reference

This appendix catalogs all known weapon parts extracted from BL4 game files, organized by manufacturer and weapon type.

13.1. Part Naming Convention

Parts follow this pattern:

```
{MFG}_{TYPE}_{Part}_{Variant}_L{Level}_{SubVariant}
```

Component	Description	Examples
MFG	Manufacturer code	DAD, JAK, TOR
TYPE	Weapon type	AR, SG, PS, SM, SR, HW
Part	Part category	Scope, Barrel, Grip
Variant	Part variant	01, 02
Level	Part tier	L1, L2, LB (legendary)
SubVariant	Sub-variant	01, Mask, ADSMask

13.2. Manufacturers

Code	Name	Serial ID	Weapon Types
BOR	Ripper	-	SM, SG, HW, SR
COV	Children of the Vault	-	Various
DAD	Daedalus	4	AR, SM, PS, SG
DPL	Dahl	-	Turrets/Gadgets only
JAK	Jakobs	129	AR, PS, SG, SR
MAL	Maliwan	138	SM, SG, HW, SR
ORD	Order	15	AR, PS, SR
TED	Tediore	10	AR, SG, PS
TOR	Torgue	6	AR, SG, HW, PS
VLA	Vladof	134	AR, SM, HW, SR, PS

Serial ID is the first VarInt in decoded item serials.

13.3. Weapon Types

13.3.1. Type Codes

Code	Type	Description
AR	Assault Rifle	Full-auto/burst rifles
HW	Heavy Weapon	Launchers, miniguns
PS	Pistol	Handguns
SG	Shotgun	Spread weapons
SM	SMG	Submachine guns

Code	Type	Description
SR	Sniper Rifle	Precision weapons

13.3.2. EWeaponType Enum

Internal weapon type enumeration (from usmap):

Value	Type
0	None
1	Pistol
2	SMG
3	Shotgun
4	AssaultRifle
5	Sniper
6	Heavy
7	Count

13.4. Scope Parts by Manufacturer

13.4.1. BOR (Ripper)

Heavy Weapons:

Part	Variants
BOR_HW_Scope_Barrel_01	Base

Part	Variants
BOR_HW_Scope_Barrel_02	Base

Shotguns:

Part	Variants
BOR_SG_Scope_01_L1	001-009
BOR_SG_Scope_01_L2	002-008
BOR_SG_Scope_02_L1	001-008
BOR_SG_Scope_02_L2	001-005
BOR_SG_Scope_Irons	001

SMGs:

Part	Variants
BOR_SM_Scope_01_L1	009-010
BOR_SM_Scope_01_L2	001-011
BOR_SM_Scope_02_L1	beam 1-8, LED, Reticle
BOR_SM_Scope_02_L2	001-007, beam

Sniper Rifles:

Part	Variants
BOR_SR_Scope_01_L1	001-014, VFX
BOR_SR_Scope_01_L2	001-002, beam
BOR_SR_Scope_02_L1	002-009
BOR_SR_Scope_02_L2	001, beam

13.4.2. DAD (Daedalus)

Assault Rifles:

Part	Variants
DAD_AR_Scope_01_L1	01-09, BracketFix
DAD_AR_Scope_01_L2	01-06
DAD_AR_Scope_02_L1	01, 08
DAD_AR_Scope_02_L2	01-09

Pistols:

Part	Variants
DAD_PS_Scope_01_L1	01
DAD_PS_Scope_01_L2	01-08
DAD_PS_Scope_02_L1	01-04
DAD_PS_Scope_02_L2	02-03

Shotguns:

Part	Variants
DAD_SG_Scope_01_L1	01-02
DAD_SG_Scope_01_L2	02, 04
DAD_SG_Scope_02_L1	01-02
DAD_SG_Scope_02_L2	01-03

SMGs:

Part	Variants
DAD_SM_Scope_01_L1	01-03
DAD_SM_Scope_01_L2	01-08, 011-015
DAD_SM_Scope_02_L1	01
DAD_SM_Scope_02_LB	01-03 (legendary barrel)

13.4.3. JAK (Jakobs)

Assault Rifles:

Part	Variants
JAK_AR_Scope_01_L1	01-03
JAK_AR_Scope_01_L2	01-03
JAK_AR_Scope_02_L1	01-03
JAK_AR_Scope_02_L2	01-02

Pistols:

Part	Variants
JAK_PS_Scope_01_L1	01-03
JAK_PS_Scope_01_L2	01-03
JAK_PS_Scope_02_L1	01-03
JAK_PS_Scope_02_L2	01-03

Shotguns:

Part	Variants
JAK_SG_Scope_01_L1	01
JAK_SG_Scope_01_L2	02
JAK_SG_Scope_02_L1	01, 03
JAK_SG_Scope_02_L2	01

Sniper Rifles:

Part	Variants
JAK_SR_Scope_01_L1	01-03, ADSMask
JAK_SR_Scope_01_L2	01
JAK_SR_Scope_02_L1	01
JAK_SR_Scope_02_L2	01-02

13.4.4. MAL (Maliwan)

Heavy Weapons:

Part	Variants
MAL_HW_Scope_01	01-07, Proje
MAL_HW_Scope_02	01-02, Proje, Shield

Shotguns:

Part	Variants
MAL_SG_Scope_01_L1	01
MAL_SG_Scope_01_L2	01
MAL_SG_Scope_02_L1	Base
MAL_SG_Scope_02_L2	01-06, ADSMask, Projes

SMGs:

Part	Variants
MAL_SM_Scope_01_L1	01-05, Base, Proje
MAL_SM_Scope_01_L2	01-07
MAL_SM_Scope_02_L1	01-03
MAL_SM_Scope_02_L2	01-02

Sniper Rifles:

Part	Variants
MAL_SR_Scope_01_L1	01-08, ADSMask, Proje
MAL_SR_Scope_01_L2	01-07, ADSMask
MAL_SR_Scope_02_L1	02, ADSMask, Proje
MAL_SR_Scope_02_L2	01-03

13.4.5. ORD (Order)

Assault Rifles:

Part	Variants
ORD_AR_Scope_01_L1	01, 002
ORD_AR_Scope_01_L2	01
ORD_AR_Scope_02_L2	Mask

Pistols:

Part	Variants
ORD_PS_Scope_01_L1	01
ORD_PS_Scope_01_L2	01-02
ORD_PS_Scope_02_L1	01, ADSMask
ORD_PS_Scope_02_L2	01-03, ADSMask

Sniper Rifles:

Part	Variants
ORD_SR_Scope_01_L1	01
ORD_SR_Scope_01_L2	01, ADSMask
ORD_SR_Scope_02_L1	01-02, ADSMask
ORD_SR_Scope_02_L2	01-02, 002, ADSMask

13.4.6. TED (Tediore)

Assault Rifles:

Part	Variants
TED_AR_Scope_01_L1	01-03, Elements
TED_AR_Scope_01_L2	01-04
TED_AR_Scope_02_L1	ModA, Mask
TED_AR_Scope_02_L2	01, 04

Pistols:

Part	Variants
TED_PS_Scope_01_L1	01-06
TED_PS_Scope_01_L2	01-02
TED_PS_Scope_02_L1	01-04, Line
TED_PS_Scope_02_L2	01-06

Shotguns:

Part	Variants
TED_SG_Scope_01_L1	01-04, 010, 012
TED_SG_Scope_01_L2	01-02, ADSMask
TED_SG_Scope_02_L1	01, 04, ModB02
TED_SG_Scope_02_L2	01, ModB, Proje

13.4.7. TOR (Torgue)

Assault Rifles:

Part	Variants
TOR_AR_Scope_01_L1	01-06
TOR_AR_Scope_01_L2	01-05
TOR_AR_Scope_02_L1	01-04
TOR_AR_Scope_02_L2	01-04, ModB

Heavy Weapons:

Part	Variants
TOR_HW_Scope_01	01-02
TOR_HW_Scope_02	01
TOR_HW_Barrel_01	Mask
TOR_HW_Barrel_02	Mask, Splitter

Pistols:

Part	Variants
TOR_PS_Scope_01_L1	01-03, ModB
TOR_PS_Scope_01_L2	01-03
TOR_PS_Scope_02_L1	01-04, Elements_Updated
TOR_PS_Scope_02_L2	01

Shotguns:

Part	Variants
TOR_SG_Scope_01_L1	01-06
TOR_SG_Scope_01_L2	01-03, B, Compass
TOR_SG_Scope_02_L1	01

Part	Variants
TOR_SG_Scope_02_L2	01-04

13.4.8. VLA (Vladof)

Assault Rifles:

Part	Variants
VLA_AR_Scope_01_L1	01-04, BASE
VLA_AR_Scope_01_L2	01-03
VLA_AR_Scope_02_L1	01
VLA_AR_Scope_02_L2	01-02

Heavy Weapons:

Part	Variants
VLA_HW_Scope_01	01-04
VLA_HW_Scope_02	01-05, ADSMask
VLA_HW_Barrel_02	Mask

SMGs:

Part	Variants
VLA_SM_Scope_01_L1	01-08, B, BASE
VLA_SM_Scope_01_L2	01
VLA_SM_Scope_02_L1	01-02

Part	Variants
VLA_SM_Scope_02_L2	01, Mask, Triangles

Sniper Rifles:

Part	Variants
VLA_SR_Scope_01_L1	01-07
VLA_SR_Scope_01_L2	01-05, ADSMask
VLA_SR_Scope_02_L1	01, 03
VLA_SR_Scope_02_L2	01-03

13.5. Barrel Parts

13.5.1. Heavy Weapons

Manufacturer	Part	Notes
BOR	BOR_HW_Barrel_01	001-010
BOR	BOR_HW_Barrel_02	001-005, A variant
MAL	MAL_HW_Barrel_02_MIRV	MIRV launcher
TOR	TOR_HW_Barrel_01	With Mask
TOR	TOR_HW_Barrel_02	Splitter variant
VLA	VLA_HW_Barrel_02	With Mask

13.6. Part Index Organization

13.6.1. The Self-Describing Design

A critical discovery: **parts don't have external indices assigned to them—each part stores its own index internally**.

Every part UObject contains a `GbxSerialNumberIndex` at offset +0x28:

```
Part UObject + 0x28:
├── Scope (1 byte)   ← Always 2 for inventory parts
├── Status (1 byte)  ← Reserved
└── Index (2 bytes)  ← This part's serial index
```

There is no separate "part → index" mapping file. The mapping IS the parts themselves. This "reverse mapping" design means:

- Each part is self-describing and carries its own identity
- Adding new parts (e.g., DLC) doesn't require updating a central registry
- Indices are guaranteed stable because they're intrinsic to each part
- Memory extraction gives us authoritative data, not a derived mapping

13.6.2. How Indices Are Assigned

Part indices within each category are assigned based on the game's internal registration order, **not alphabetically**. Understanding this is critical for correctly decoding and encoding item serials.

13.6.3. Registration Order Pattern

Parts appear to be registered in groups by functional type:

Order	Part Type	Example
1	Unique/special variants	`part_barrel_01_zipgun`
2	Body parts	`part_body`, `part_body_a-d`
3	Base barrels	`part_barrel_01`, `part_barrel_02`
4	Shield/defensive	`part_shield_default`, `part_shield_ricochet`
5	Magazines	`part_mag_01`, `part_mag_02`
6	Scopes	`part_scope_ironsight`, `part_scope_01_*`
7	Grips	`part_grip_01`, `part_grip_02`
8	Underbarrel/foregrip	`part_underbarrel_*`, `part_foregrip_*`
9	Body magazines	`part_body_mag_smg`, `part_body_mag_ar`
10	Barrel variants	`part_barrel_01_a-d`, `part_barrel_02_a-d`
11	Licensed parts	`part_barrel_licensed_jak`, `part_barrel_licensed_ted`

13.6.4. Index Gaps

Some categories have non-contiguous indices. For example, a category might have parts at indices 1-53, skip 54-56, then continue at 57. These gaps may represent:

- Reserved slots for future DLC parts
- Parts that were removed during development
- Internal versioning or compatibility placeholders

13.6.5. Implications for Modding

When working with item serials:

1. **Never assume alphabetical order** — Part `part_barrel_01` might have index 7, not index 0
2. **Use runtime-extracted data** — Only memory dumps capture the true `GbxSerialNumberIndex` values
3. **Validate against known items** — Decode existing item serials to verify index mappings
4. **Account for gaps** — Don't assume contiguous indices when iterating

---

13.7. Part Compatibility Rules (Verified)

This section documents **verified** part compatibility rules derived from analyzing actual weapon drops and reference data. These rules are encoded in the part naming conventions and enforced by the game engine.

13.7.1. Rule 1: Prefix Determines Weapon Type

Parts are only valid for weapons matching their prefix:

Prefix	Manufacturer	Weapon Type
`DAD_PS.*`	Daedalus	Pistol
`DAD_AR.*`	Daedalus	Assault Rifle
`JAK_SG.*`	Jakobs	Shotgun
`VLA_SM.*`	Vladof	SMG

A `DAD_PS.part_barrel_01` **cannot** appear on a `DAD_AR` weapon.

13.7.2. Rule 2: Barrel Accessories Require Matching Barrel

Barrel accessories are **barrel-specific**. The naming convention encodes the dependency:

```
part_barrel_XX_Y  →  only valid with part_barrel_XX
```

Accessory Part	Valid With	Invalid With
`DAD_PS.part_barrel_01_a`	`DAD_PS.part_barrel_01`	`DAD_PS.part_barrel_02`
`DAD_PS.part_barrel_01_b`	`DAD_PS.part_barrel_01`	`DAD_PS.part_barrel_02`
`DAD_PS.part_barrel_02_a`	`DAD_PS.part_barrel_02`	`DAD_PS.part_barrel_01`
`DAD_PS.part_barrel_02_c`	`DAD_PS.part_barrel_02`	`DAD_PS.part_barrel_01`

Pattern: Extract the barrel number from accessory name (`barrel_XX_*`) and match to base barrel (`barrel_XX`).

13.7.3. Rule 3: Scope Accessories Require Matching Scope AND Lens

Scope accessories have a **two-dimensional dependency** on both scope type and lens type:

```
part_scope_acc_sXX_lYY_Z  →  only valid with part_scope_XX_lens_YY
```

Accessory Part	Valid With	Invalid With
`DAD_PS.part_scope_acc_s01_l01_a`	`part_scope_01_lens_01`	`part_scope_01_lens_02`, `part_scope_02_*`
`DAD_PS.part_scope_acc_s01_l02_b`	`part_scope_01_lens_02`	`part_scope_01_lens_01`, `part_scope_02_*`
`DAD_PS.part_scope_acc_s02_l01_a`	`part_scope_02_lens_01`	`part_scope_01_*`, `part_scope_02_lens_02`
`DAD_PS.part_scope_acc_s02_l02_b`	`part_scope_02_lens_02`	`part_scope_01_*`, `part_scope_02_lens_01`

Pattern: Parse `sXX` and `lYY` from accessory name, match to `scope_XX_lens_YY`.

13.7.4. Rule 4: Some Magazine Modifiers Are Barrel-Specific

Certain magazine stat modifiers are tied to specific barrels:

```
part_mag_*_barrel_XX  →  only valid with part_barrel_XX
```

Modifier Part	Valid With
`DAD_PS.part_mag_05_borg_barrel_01`	`part_barrel_01` + Ripper Mag
`DAD_PS.part_mag_05_borg_barrel_02`	`part_barrel_02` + Ripper Mag

13.7.5. Rule 5: Maximum Accessory Counts

Weapons have limits on how many accessories can appear:

Constraint	Limit
Total accessories per weapon	Max 3
Body accessories	Choose 1-2
Barrel accessories	Choose 2-3
Scope accessories	Not all scopes support 2

13.7.6. Rule 6: Legendary Barrel Restrictions

Some legendary barrels **cannot** roll with barrel accessories:

Barrel	Accessory Restriction
`JAK_SG.part_barrel_hot_slugger`	Never rolls with barrel accessories
`JAK_SG.part_barrel_hellwalker`	Doesn't roll with barrel accessories
Unique barrels (general)	Often have restricted accessory pools

13.7.7. Rule 7: Barrel-Type Constraints for Magazines

Some magazines are only valid for specific barrel configurations:

Magazine	Valid Barrel Type
`JAK_SG` 6x mag (single barrel)	Single-barrel weapons
`JAK_SG` 6x mag (double barrel)	Double-barrel weapons

13.7.8. Validation Algorithm

To validate a weapon's part combination:

```
def is_valid_combination(parts):
    barrel = find_barrel(parts)
    scope = find_scope(parts)

    for part in parts:
        # Rule 1: Prefix match
        if not same_prefix(part, barrel):
            return False

        # Rule 2: Barrel accessory match
        if is_barrel_accessory(part):
            if not matches_barrel(part, barrel):
                return False

        # Rule 3: Scope accessory match
        if is_scope_accessory(part):
            if not matches_scope_and_lens(part, scope):
                return False

        # Rule 4: Barrel-specific mag modifier
        if is_barrel_mag_modifier(part):
            if not matches_barrel(part, barrel):
                return False
```

```
    # Rule 5: Accessory count limits
    if count_accessories(parts) > 3:
        return False

    return True
```

13.7.9. Implications for Save Editing

When generating weapon serials for testing:

1. **Don't mix accessories** — A `barrel_01` weapon cannot have `barrel_02_a` accessory
2. **Match scope accessories** — Check both scope type AND lens type
3. **Respect legendary restrictions** — Some unique barrels have no accessories
4. **Count accessories** — Stay within the 3-accessory limit

13.8. Part Selection System (Theoretical)

> **i Speculative**
>
> This section describes classes found in game metadata. Actual implementation may differ. The "Part Compatibility Rules" section above contains verified behavior.

13.8.1. Class-Based Selection (from usmap)

The following classes exist in the game's type system, suggesting a structured selection mechanism:

Class	Likely Purpose
`PartTypeSelectionRules`	Rules for selecting parts by type
`PartTagSelectionRules`	Tag-based part filtering
`PartTagGameStageSelectionData`	Level requirements for parts
`InventorySelectionCriteria`	General selection criteria

However, **no assets implementing these classes were found in pak files**. The selection logic may be: - Hardcoded in C++ binary - Convention-based (parsed from naming) - Or a combination of both

---

13.9. Part Slot Types

From `EWeaponPartValue` enum:

Slot	Description
Grip	Weapon grip
Foregrip	Front grip
Reload	Reload mechanism
Barrel	Main barrel
Scope	Optics/scope
Melee	Melee attachment

Slot	Description
Mode	Fire mode
ModeSwitch	Mode switch mechanism
Underbarrel	Under-barrel attachment
Custom0-7	Additional custom slots

13.10. Weapon Naming System

Parts affect weapon prefix names through the naming table system.

13.10.1. Primary Indices

Index	Stat Type	Example Prefixes
2	Damage	Tortuous, Agonizing, Festering
3	CritDamage	Bleeding, Hemorrhaging, Pooling
4	ReloadSpeed	Frenetic, Manic, Rotten
5	MagSize	Bloated, Gluttonous, Hoarding
7	body_mod_a	Chosen, Promised, Tainted
8	body_mod_b	Bestowed, Cursed, Offered
9	body_mod_c	Ritualized, Summoned
10	body_mod_d	Strange
15-18	barrel_mod	Herald, Harbinger, Oracle, Prophecy

13.10.2. Naming Indices (from WeaponNamingStruct)

Field	Index	GUID
Damage	2	9DFA8E9A4AF1B3A1…
CritDamage	9	C4432C8C40CA15F0…
FireRate	10	459C49044DE26DE5…
ReloadSpeed	11	61FAACA14D48B609…
MagSize	12	C735EA434D50CD82…
Accuracy	13	5B35CC194CB71AE4…
ElementalPower	14	842A58234E5D5D79…
ADSProficiency	16	02D519604FE47BA5…
Single	18	240AB1EB411BED6B…
DamageRadius	21	EE89495D493F3450…

13.11. Known Legendaries

13.11.1. By Manufacturer

Internal Name	Display Name	Type	Manufac
DAD_AR.comp_05_legendary_OM	OM	AR	Daedalu
DAD_SG.comp_05_legendary_HeartGUn	Heart Gun	SG	Daedalu
JAK_AR.comp_05_legendary_rowan	Rowan's Call	AR	Jakobs
JAK_PS.comp_05_legendary_kingsgambit	King's Gambit	PS	Jakobs
JAK_PS.comp_05_legendary_phantom_flame	Phantom Flame	PS	Jakobs
JAK_SR.comp_05_legendary_ballista	Ballista	SR	Jakobs
MAL_HW.comp_05_legendary_GammaVoid	Gamma Void	HW	Maliwar

Internal Name	Display Name	Type	Manufacturer
MAL_SM.comp_05_legendary_OhmIGot	Ohm I Got	SM	Maliwan
TED_AR.comp_05_legendary_Chuck	Chuck	AR	Tediore
TED_PS.comp_05_legendary_Sideshow	Sideshow	PS	Tediore
TOR_HW.comp_05_legendary_ravenfire	Ravenfire	HW	Torgue
TOR_SG.comp_05_legendary_Linebacker	Linebacker	SG	Torgue
VLA_AR.comp_05_legendary_WomboCombo	Wombo Combo	AR	Vladof
VLA_HW.comp_05_legendary_AtlingGun	Atling Gun	HW	Vladof
VLA_SM.comp_05_legendary_KaoSon	Kaoson	SM	Vladof

13.12. Rarity System

Code	Tier	Color
comp_01	Common	White
comp_02	Uncommon	Green
comp_03	Rare	Blue
comp_04	Epic	Purple
comp_05	Legendary	Orange

13.12.1. Internal Format

```
{MFG}_{TYPE}.comp_0{N}_{rarity}_{name}
```

Examples: - `TOR_SG.comp_05_legendary_Linebacker` - `VLA_SM.comp_05_legendary_KaoSon`

13.13. Part Count by Category

Data extracted from memory dump using `bl4 memory dump-parts` and `bl4 memory build-parts-db`.

13.13.1. Weapons

Category ID	Manufacturer	Weapon Type	Parts Count
2	Daedalus	Pistol	74
3	Jakobs	Pistol	73
4	Tediore	Pistol	81
5	Torgue	Pistol	70
6	Order	Pistol	75
8	Daedalus	Shotgun	74
9	Jakobs	Shotgun	89
10	Tediore	Shotgun	76
11	Torgue	Shotgun	69
12	Bor	Shotgun	73
13	Daedalus	Assault Rifle	78
14	Jakobs	Assault Rifle	74
15	Tediore	Assault Rifle	79
16	Torgue	Assault Rifle	73
17	Vladof	Assault Rifle	89
18	Order	Assault Rifle	73
19	Maliwan	Shotgun	74
20	Daedalus	SMG	77
21	Bor	SMG	73
22	Vladof	SMG	84
23	Maliwan	SMG	74
25	Bor	Sniper	71
26	Jakobs	Sniper	72

Category ID	Manufacturer	Weapon Type	Parts Count
27	Vladof	Sniper	82
28	Order	Sniper	75
29	Maliwan	Sniper	76
244	Vladof	Heavy	22
245	Torgue	Heavy	32
246	Bor	Heavy	25
247	Maliwan	Heavy	19

13.13.2. Class Mods

Category ID	Player Class	Parts Count
44	Dark Siren	0 (not in dump)
55	Paladin	0 (not in dump)
97	Gravitar	2
140	Exo Soldier	0 (not in dump)

13.13.3. Firmware

Category ID	Type	Parts Count
151	Firmware	0 (parts under gadget prefixes)

Note: Firmware parts exist under `grenade_gadget.part_firmware_*`, `heavy_weapon_gadget.part_firmware_*`, and `repair_kit.part_firmware_*` prefixes.

13.13.4. Shields

Category ID	Type	Parts Count
279	Energy Shield	22
280	Bor Shield	4
281	Daedalus Shield	3
282	Jakobs Shield	3
283	Armor Shield	26
284	Maliwan Shield	9
285	Order Shield	3
286	Tediore Shield	3
287	Torgue Shield	3
288	Vladof Shield	3
289	Shield Variant	Unknown

13.13.5. Gadgets and Gear

Category ID	Type	Parts Count
300	Grenade Gadget	82
310	Turret Gadget	52
320	Repair Kit	107
330	Terminal Gadget	61

13.13.6. Enhancements

Category ID	Manufacturer	Parts Count
400	Daedalus	1

Category ID	Manufacturer	Parts Count
401	Bor	1
402	Jakobs	4
403	Maliwan	4
404	Order	4
405	Tediore	4
406	Torgue	4
407	Vladof	4
408	COV	1
409	Atlas	1

13.13.7. Summary

Category	Total Parts
Weapons (all types)	1,928
Class Mods	2
Shields	79
Gadgets	302
Enhancements	28
Unmapped	276
Total	**2,615**

13.14. Variant Suffixes

Suffix	Meaning
Mask	Texture mask asset
ADSMask	Aim-down-sights mask
Proje/Projes	Projectile-related
ModA/ModB	Alternative model
Elements	Elemental effects
VFX	Visual effects
BASE	Base variant
_a, _b, _c, _d	Stat variants
_01, _02, _03	Numbered variants

13.15. Data Files

The complete parts database is available at:

- **`share/manifest/parts_dump.json`** - Raw part names grouped by prefix
- **`share/manifest/parts_database.json`** - Full database with category/index mappings

Use `bl4 memory dump-parts` and `bl4 memory build-parts-db` to regenerate from a fresh memory dump.

For complete category mappings, composition system details, and licensed parts documentation, see Chapter 8: Parts System.

Extracted from BL4 memory dumps and NCS data using bl4 analysis tools.

Last updated: February 2026 — NCS extraction expanded to 5,360 parts across 120 categories. See `share/manifest/parts_database.json` for the current authoritative source.

14. Appendix C: Loot System Internals

This appendix explains how BL4's loot system decides what to drop, how rare each drop is, and how the bl4 toolkit estimates item rarity from serial data.

14.1. Two Ways to Get a Legendary

Every legendary in BL4 reaches the player through one of two paths: **dedicated drops** or **world drops**. They use different probability models, and understanding the difference matters for estimating how rare a given item is.

14.1.1. Dedicated Drops

Dedicated drops are tied to specific bosses. Each boss has an `ItemPoolList` in NCS that names 1–3 legendary items, each assigned to a **tier** that determines its drop chance per kill:

Tier	Chance per kill
Primary	6%
Secondary	4.5%
Tertiary	3%

The first item in a boss's pool is Primary (highest chance), the second is Secondary, and so on. When you kill a boss, the game rolls independently for each dedicated item — you can get zero, one, or (rarely) multiple legendaries from a single kill.

> **ℹ Shiny and TrueBoss Variants**
>
> Bosses have additional tier variants for special modes. **TrueBoss** (Chaos mode) raises the dedicated drop rate to 25%, making legendaries roughly 4x more common per kill. **Shiny** items are cosmetic variants at 1%, and **TrueBossShiny** at 3%.

This is the simplest path to a legendary: kill a boss, roll against a known percentage. If the Hellwalker is the Saddleback's Primary drop, you have a 6% chance per kill. Farm the boss enough and you'll get one.

14.1.2. World Drops

World drops are the other path. When any enemy dies, it can drop gear from the general loot pool. The rarity of that gear is selected by a weighted roll:

Rarity	Weight	Probability
Common	100.0	~94.18%
Uncommon	6.0	~5.65%
Rare	0.14	~0.132%
Epic	0.045	~0.0424%
Legendary	0.0003	~0.000283%

These weights come from the `rarity_balance` table in `gbx_ue_data_table0.bin`. The game sums all weights (106.1853) and picks a tier proportionally. Legendary is 0.0003 out of 106.1853 — roughly 1 in 353,000 world drops.

But that's the chance of getting *any* legendary. The chance of getting a *specific* one is much lower. If you rolled Legendary on a world drop and the game selects from the Pistol pool, there are 9 legendary pistols across all manufacturers. Your chance of the specific one you wanted is the tier probability divided by the pool size: roughly 1 in 3.2 million.

14.2. How a Drop Resolves

When an enemy dies, the game runs through several systems in sequence:

1. **Base loot roll** (`Struct_EnemyDrops`): determines what *categories* drop — guns, shields, grenades, class mods, currency. Each category has its own probability and quantity. A standard boss might drop 2 guns and 1 shield.

2. **Rarity selection**: for each item that drops, the rarity weight table determines what tier it rolls. This is where Common (94%) vs Legendary (0.0003%) is decided.

3. **Pool selection**: the game picks a specific item from the pool matching that rarity and item category. A legendary gun roll selects from `itempool_guns_05_legendary`.

4. **Dedicated drop check**: independently of the base loot, the boss's dedicated pool is checked. Each assigned legendary rolls independently at its tier percentage (Primary 6%, Secondary 4.5%, etc.).

5. **Luck modifier**: the player's luck stat modifies rarity weights at step 2. Higher luck increases the weight of rarer tiers. The exact formula uses `GrowthExponent`, `BaseWeight`, and `GameStageVariance` properties on the `RarityWeightData` class, but the specific curve is resolved at runtime and hasn't been fully extracted.

Steps 2–3 and step 4 are independent paths. An item can appear as both a dedicated drop (guaranteed from a specific boss) and a world drop (from the general pool). The dedicated path is far more likely for any given item.

14.3. Pool Sizes and Per-Item Odds

The rarity tier probability tells you how likely *any* legendary is. The pool size tells you how likely a *specific* one is. These are different questions.

Legendary items are organized into pools by manufacturer and weapon type. Jakobs Shotguns have 2 legendaries. Vladof Assault Rifles have 3. The bl4 toolkit's `drop_pools.tsv` manifest captures these counts.

For world drops, the game selects across all manufacturers within a weapon type. If it rolls "legendary pistol", it's choosing from all 9 legendary pistols (Jakobs, Daedalus, Tediore, etc. combined). The per-item probability for a specific legendary pistol from a world drop is:

```
tier_probability / world_pool_size = 0.000283% / 9 = ~0.0000314%
```

That's roughly 1 in 3.2 million world drops for a specific legendary pistol. This is why dedicated boss farming (6% per kill for a specific item) is orders of magnitude more efficient than hoping for world drops.

14.4. Rarity Estimation

The bl4 toolkit can estimate an item's rarity from its serial string. The `rarity_estimate()` method on `ItemSerial` combines several data sources:

1. **Rarity tier**: decoded from the serial's `inv_comp` part (comp_01 through comp_05)
2. **Tier probability**: looked up from the rarity weight table
3. **Manufacturer and type codes**: extracted from the serial's token stream (VarInt-first for weapons, VarBit-first for equipment)

4. **Pool data**: matched against the compile-time `drop_pools.tsv` manifest for legendary count, world pool size, and boss source count

For legendaries, the estimate divides the tier probability by the world pool size to get per-item odds. For other rarities, it reports only the tier probability (since the pool selection is less meaningful — a "common Jakobs pistol" isn't a specific item the way a legendary is).

```
Rarity estimate:
  Tier: Legendary (0.000283%, ~1 in 353,490)
  Pool: Jakobs Shotgun (2 legendaries, 9 in world pool)
  Per-item: ~1 in 3,181,413
  Boss sources: 2
```

> **Warning**
>
> These are estimates, not exact values. The rarity weight table is extracted from NCS data tables and matches community testing results, but luck modifiers, game stage scaling, and category selection probabilities are resolved at runtime and aren't captured here.

14.5. What We Know vs. Don't Know

Extracted from NCS data:

- Dedicated drop probabilities per tier (from `Table_DedicatedDropProbabilit`

- Rarity weights for the base tier selection (from `rarity_balance`)
- Boss-to-legendary assignments and tier ordering (from `itempoollist.bin`)
- Pool sizes per manufacturer/weapon type (from drops manifest)
- Enemy drop category probabilities and quantities (from `Table_EnemyDrops`)

Not yet extracted:

- Luck system curve (how much luck shifts rarity weights, resolved at runtime)
- Category selection probabilities within a rarity tier (shield vs weapon vs grenade)
- Game stage scaling effects on weights (the `GrowthExponent` / `GameStageVariance` interaction)
- Tier assignments for most dedicated drops (only 1 of 133 boss drops has explicit tier context in NCS — the rest are inferred by pool ordering)

14.6. NCS Data Sources

Drop information is stored across several NCS files within pak archives:

File	Contents
`itempoollist.bin`	Boss → legendary item mappings with tier assignments
`itempool.bin`	Item pool definitions, rarity weights, world drop membership

File	Contents
`gbx_ue_data_table0.bin`	`Table_DedicatedDropProbability`, `rarity_balance`, `Table_EnemyDrops`, `Table_LootableBalance`
`loot_config.bin`	Global loot configuration parameters
`preferredparts.bin`	Part preferences for item generation

Numeric values in these tables are stored as strings in NCS (`"0.060000"`, `"1.500000"`). The binary section's bit-packed indices point into the string table where these values live.

14.7. Reference: Dedicated Drop Probability Tiers

From `Table_DedicatedDropProbability` in `gbx_ue_data_table0.bin`. Schema defined in `Struct_DedicatedDropProbability.uasset` with a single `DoubleProperty` field.

Tier	Row Name	Index	Probability
Primary	`Primary_2_<GUID>`	2	6%
Secondary	`Secondary_4_<GUID>`	4	4.5%
Tertiary	`Tertiary_6_<GUID>`	6	3%
Shiny	`Shiny_9_<GUID>`	9	1%
TrueBoss	`TrueBoss_12_<GUID>`	12	25%
TrueBossShiny	`TrueBossShiny_14_<GUID>`	14	3%
Quaternary	`Quaternary_16_<GUID>`	16	0%

14.8. Reference: Rarity Weights

From `rarity_balance` in `gbx_ue_data_table0.bin`. Total weight: 106.1853.

Tier	Component ID	Weight	Probability
Common	`comp_01_common`	100.0	94.18%
Uncommon	`comp_02_uncommon`	6.0	5.65%
Rare	`comp_03_rare`	0.14	0.132%
Epic	`comp_04_epic`	0.045	0.0424%
Legendary	`comp_05_legendary`	0.0003	0.000283%

14.9. Reference: Enemy Drop Fields

From `Struct_EnemyDrops`. Each enemy tier has rows in `Table_EnemyDrops` controlling what categories drop and in what quantities.

Field	Description
Guns_Probability	Chance to drop guns
Guns_HowMany	Number of guns
Shields_Probability	Shield drop chance
Shields_HowMany	Number of shields
GrenadesOrGadgets_Probability	Grenade/gadget chance

Field	Description
GrenadesGadgets_HowMany	Number to drop
ClassMods_Probability	Class mod chance
ClassMods_HowMany	Number of class mods
RepKits_Probability	Repair kit chance
RepKits_HowMany	Number of repair kits
Enhancements_Probability	Enhancement chance
Enhancements_HowMany	Number of enhancements
CurrencyOrAmmo_Probability	Currency/ammo chance
CurrencyAmmo_HowMany	Amount
EXP_Multiplier	Experience multiplier
Rarity_Modifier	Rarity boost

14.10. Reference: Boss → Legendary Mappings

Extracted from `itempoollist.bin`. Each boss has 1–3 dedicated legendary drops. The first item listed is Primary tier (6%), second is Secondary (4.5%), third is Tertiary (3%).

Boss	Primary Drop	Secondary Drop	Third Drop
Arjay	ORD_SR Fisheye	DAD_SG HeartGun	-
Backhive	VLA_SR StopGap	-	-
Bango	JAK_PS Phantom_Flame	BOR_SM Prince	-

Boss	Primary Drop	Secondary Drop	Third Drop
BatMatriarch	TOR_SG Linebacker	BOR_SM hellfire	-
BattleWagon	VLA_SR Finnty	TOR_AR Bugbear	-
BlasterBrute	JAK_SG Slugger	MAL_SG Kaleidosplode	-
Bloomreaper	MAL_SG Mantra	-	-
CityCat	DAD_SG Bod	-	-
CloningLeader	TED_PS Sideshow	-	-
Destroyer	JAK_SR Boomslang	BOR_SG Convergence	-
Donk	JAK_AR Rowdy	TED_PS Inscriber	-
Drillerhole	ORD_AR GMR	MAL_SR Katagawa	-
DroneKeeper	ORD_PS Bully	TED_AR DividedFocus	-
FirstCorrupt	JAK_PS KingsGambit	DAD_PS Rangefinder	-
GlidePackPsycho	TOR_SG LeadBalloon	-	-
Grasslands_Commander	BOR_SG GoldenGod	BOR_SG GoreMaster	VLA_SM Onslaught
Grasslands_Guardian	TED_SG HeavyTurret	-	-

Boss	Primary Drop	Secondary Drop	Third Drop
Hovercart	VLA_AR WomboCombo	TOR_AR PotatoThrower	-
KOTO-Motherbase-Brute	TED_PS ATLien	-	-
KotoLieutenant	ORD_PS RocketReload	VLA_AR DualDamage	-
LeaderHologram	TED_AR Chuck	-	-
Meat-Plant-Gunship	MAL_SR Asher	-	-
Meat-head-Rider	VLA_AR Lucian	-	-
Meat-head-Rider_Jockey	JAK_SG Hellwalker	-	-
Mountain-Commander	TED_PS RubysGrasp	-	-
Mountain-Guardian	VLA_SM KaoSon	-	-
Pango	BOR_SR Stray	-	-

Boss	Primary Drop	Secondary Drop	Third Drop
Redguard	VLA_AR WF	JAK_AR Rowan	JAK_AR BonnieClyde
RockAndRoll	JAK_PS QuickDraw	JAK_SG TKsWave	-
ShatterlandsCommanderElpis	MAL_SM OhmIGot	-	-
ShatterlandsCommanderFortress	TOR_PS QueensRest	-	-
ShatterlandsGuardian	TED_SG Anarchy	-	-
SideCity_Psycho	ORD_AR Goalkeeper	JAK_PS SeventhSense	-
SkullOrchid	TOR_PS Roach	-	-
SoldierAncient	DAD_AR Om	-	-
SpiderJumbo	ORD_PS NoisyCricket	-	-

Boss	Primary Drop	Secondary Drop	Third Drop
StealthPredator	BOR_SR Vamoose	-	-
StrikerSplitter	DAD_SM Luty	-	-
SurpriseAttack	DAD_SM Bloodstarved	-	-
Thresher_ArmoredBig	JAK_SR Truck	-	-
Timekeeper_Guardian	MAL_SM PlasmaCoil	DAD_AR StarHelix	-
Timekeeper_TKBoss	ORD_SR Symmetry	JAK_SR Ballista	-
TrashThresher	MAL_SG Kickballer	VLA_SM BeeGun	-
UpgradedElectiMole	DAD_PS Zipgun	JAK_SG RainbowVomit	-

14.11. Reference: Item Pools

14.11.1. Boss Pool Lists

Pool	Description
ItemPoolList_Enemy_BaseLoot_Boss	Standard boss drops
ItemPoolList_Enemy_BaseLoot_BossRaid	Raid boss drops
ItemPoolList_Enemy_BaseLoot_BossMini	Mini-boss drops
ItemPoolList_Enemy_BaseLoot_BossVault	Vault boss drops
ItemPoolList_*_TrueBoss	True Boss (Chaos mode) variants

14.11.2. Rarity-Tiered Weapon Pools

Pool	Rarity
`itempool_guns_01_common`	Common
`itempool_guns_02_uncommon`	Uncommon
`itempool_guns_03_rare`	Rare
`itempool_guns_04_epic`	Epic
`itempool_guns_05_legendary`	Legendary

14.11.3. Special Pools

Pool	Description
ItemPool_FishCollector_Reward_Legendary	Fish collector reward
ItemPool_BlackMarket_Comp_BOR_HW_DiscJockey	Black Market exclusive
ItemPool_BlackMarket_Comp_BOR_HW_Streamer	Black Market exclusive

14.12. Reference: Drop Source Types

Type	Description
Boss	Dedicated boss drop with per-kill tier probability
Mission	Side/main mission reward (guaranteed on completion)
BlackMarket	Black Market vendor exclusive
Special	Fish Collector, challenges, event rewards
WorldDrop	General legendary pool (rarity-weighted)

14.13. Reference: Item Composition in NCS

Legendary items follow a consistent naming pattern:

```
MANUFACTURER_TYPE.comp_05_legendary_NAME
```

Examples: `JAK_SG.comp_05_legendary_Hellwalker`, `MAL_SM.comp_05_legendary_` `DAD_SHIELD.comp_05_legendary_angel`.

Component	Rarity
`comp_01_common`	Common
`comp_02_uncommon`	Uncommon
`comp_03_rare`	Rare
`comp_04_epic`	Epic
`comp_05_legendary`	Legendary

14.14. Reference: Codes

14.14.1. Weapon Types

Code	Type
AR	Assault Rifle
PS	Pistol
SG	Shotgun
SM	SMG
SR	Sniper Rifle
HW	Heavy Weapon

14.14.2. Manufacturers

Code	Manufacturer	Specialty
BOR	Ripper	-
DAD	Daedalus	Shields
JAK	Jakobs	High damage, slow fire
MAL	Maliwan	Elemental
ORD	Order	-
TED	Tediore	Throw-to-reload
TOR	Torgue	Explosives
VLA	Vladof	High fire rate

14.14.3. Gear Slots

Slot	Type Code
Weapon 1–4	AR, PS, SG, SM, SR, HW
Shield	SHIELD
Grenade	GRENADE
Class Mod	CM
Artifact	ARTIFACT
Repair Kit	RK, REPAIR_KIT

14.15. Reference: Drops Manifest Format

The `drops.json` manifest generated by `bl4 drops generate` contains:

```
{
  "version": 1,
  "probabilities": {
    "Primary": 0.06,
    "Secondary": 0.045,
    "Tertiary": 0.03,
    "Shiny": 0.01,
    "TrueBoss": 0.25,
    "TrueBossShiny": 0.03
  },
  "drops": [
    {
      "source": "MeatheadRider_Jockey",
      "source_display": "Saddleback",
      "source_type": "Boss",
```

```
      "manufacturer": "JAK",
      "gear_type": "SG",
      "item_name": "Hellwalker",
      "item_id": "JAK_SG.comp_05_legendary_Hellwalker",
      "pool": "itempool_jak_sg_05_legendary_Hellwalker_shiny",
      "drop_tier": "Primary",
      "drop_chance": 0.06
    }
  ]
}
```

The companion `drop_pools.tsv` summarizes legendary counts and boss source counts per manufacturer/weapon type pool. It is embedded at compile time for the rarity estimation API.

14.16. Reference: Loot System Classes

Classes relevant to the loot pipeline, discovered via memory analysis:

14.16.1. Pool System

Class	Role
ItemPoolDef	Defines a loot pool (`/Script/GbxGame.ItemPoolDef`)
ItemPoolEntry	Single item in a pool
ItemPoolListDef	Ordered list of pools (boss drop lists)

Class	Role
ItemPoolSelectorDef	Selection logic between pools

```
enum class ELootPoolTypes {
    All = 0,
    BaseLoot = 1,
    AdditionalLoot = 2,
    DedicatedDrops = 3,
    GearDrivenDrops = 4,
    MAX = 5
};
```

14.16.2. Rarity Resolution

Class	Role
RarityWeightData	Weight configuration (BaseWeight, GrowthExponent, GameStageVariance)
LocalRarityModifierData	Local rarity modifiers (area/event bonuses)
InventoryRarityDataTableValueResolver	Resolves rarity values from DataTables at runtime
InventoryRarityDef	Rarity tier definition
LootChanceDefinedValueRow	DataTable row for loot chance configuration

14.16.3. Luck System

Class	Role
LuckGlobals	Global luck settings
LuckCategoryDef	Luck category definition
LuckCategoryAttribute	Per-category luck attribute
LuckCategoryValueResolver	Resolves luck values at runtime

Three luck category groups: `LuckCategories` (base), `EnemyBasedLuckCategories` (per-enemy), `PlayerBasedLuckCategories` (player-specific).

Data from NCS file extraction and live memory analysis.

15. Appendix D: Game File Structure

This appendix provides a complete reference of BL4's file structure, asset organization, and content layout.

15.1. Overview

Property	Value
Engine	Unreal Engine 5.5
Asset Format	IoStore (.utoc/.ucas) with Zen packages
Total Assets	~119,299 files in pak archives
Extracted to Manifest	81,097 assets
Internal Codename	Oak2
Max Players	4

15.2. File Locations

15.2.1. Steam (Linux)

```
~/.steam/steam/steamapps/common/Borderlands 4/OakGame/Content/Paks/
```

15.2.2. Steam (Windows)

```
C:\Program Files (x86)\Steam\steamapps\common\Borderlands 4\OakGame\Co
```

15.3. Pak Chunk Contents

Chunk	Contents
pakchunk0-Windows_0_P	Core game assets, weapons, gear
pakchunk2-Windows_0_P	Audio (Wwise .bnk)
pakchunk3-Windows_0_P	Localized audio
pakchunk10-Windows_0_P	Large assets

15.4. Top-Level Content Structure

```
OakGame/Content/
├── AI/                     # Enemy AI, NPCs, bosses
├── Atlases/                # Texture atlases
├── Cinematics/             # Cutscene assets
├── Common/                 # Shared materials/resources
├── Dialog/                 # Dialogue assets
├── Editor/                 # Editor-only assets
├── Fonts/                  # Font assets
├── GameData/               # Core game configuration
├── Gear/                   # All equipment
├── GeometryCollections/    # Physics destruction meshes
├── Gore/                   # Gore effects
├── Grapple/                # Grappling hook system
├── InteractiveObjects/     # World interactables
├── LevelArt/               # Level-specific art
├── LevelLighting/          # Lighting setups
├── Maps/                   # Game maps/levels
├── Missions/               # Mission data
├── Pickups/                # Item pickup visuals
├── PlayerCharacters/       # Vault Hunters
├── UI/                     # UI assets
├── uiresources/            # UI resource files
├── WeatherOcclusionBakedData/
└── World/                  # World building assets
```

15.5. Player Characters (Vault Hunters)

```
PlayerCharacters/
├── Customizations/          # Player cosmetics
├── DarkSiren/               # Character: Dark Siren
├── ExoSoldier/              # Character: Exo Soldier
├── Gravitar/                # Character: Gravitar
├── Paladin/                 # Character: Paladin
├── _Shared/                 # Shared character resources
└── Temporary/               # Development/testing
```

15.6. Gear System

15.6.1. Equipment Types

```
Gear/
├── ArmorShard/              # Armor shard items
├── Enhancements/            # Enhancement items
├── Firmware/                # Firmware upgrades
├── Gadgets/                 # Deployable gadgets
│   ├── HeavyWeapons/        # Heavy weapon gadgets
│   ├── Terminals/           # Terminal gadgets
│   └── Turrets/             # Turret gadgets
├── GrenadeGadgets/          # Grenades
│   ├── Manufacturer/
│   │   └── VLA/             # Vladof grenades
```

```
│   └── _Shared/
├── RepairKits/                # Repair kit items
├── ShieldBooster/             # Shield boosters
├── shields/                   # Shields
│   ├── BalanceData/
│   ├── Manufacturer/
│   │   └── VLA/               # Vladof shields
│   └── _Shared/
├── Vehicles/                  # Vehicle equipment
│   └── HoverDrives/           # Hover drive upgrades
├── Weapons/                   # Guns
└── _Shared/                   # Shared gear resources
    ├── BalanceData/
    │   ├── Anoints/           # Anointment system
    │   ├── Economy/           # Currency/cost data
    │   └── Rarity/            # Rarity definitions
```

15.6.2. Weapon System

```
Gear/Weapons/
├── _Manufacturer/             # Manufacturer-specific data
│   ├── BOR/                   # Ripper
│   ├── JAK/                   # Jakobs
│   └── TED/                   # Tediore
├── Pistols/                   # Pistol weapons
├── Shotguns/                  # Shotgun weapons
├── SMG/                       # SMG weapons
├── Sniper/                    # Sniper weapons
└── _Shared/
    └── BalanceData/
        ├── BarrelData/            # Barrel parts
```

```
        ├── _BaseWeaponData/    # Base weapon stats
        ├── BorgChargeData/     # Borg charge mechanics
        ├── COV/                # COV overheat/repair
        ├── Elemental/          # Elemental damage types
        ├── MagazineData/       # Magazine parts
        ├── Order/              # Order faction weapons
        ├── Rarity/             # Weapon rarity modifiers
        ├── ScopeData/          # Scope parts
        ├── TED/                # Tediore-specific data
        ├── UnderbarrelData/    # Underbarrel attachments
        ├── UniqueData/         # Legendary/unique data
        └── WeaponStats/        # Base stat definitions
```

15.7. GameData System

```
GameData/
├── Activities/             # Activity/event system
├── Animation/              # Animation configs
├── Audio/                  # Audio settings
├── Balance/                # Game balance tables
│   └── Structs/
│       ├── Struct_ChallengeReward_ECHOTokens
│       └── Struct_BossReplay_Costs
├── Cinematics/             # Cinematic triggers
├── Damage/                 # Damage system
│   └── StatusEffects/      # Status effect definitions
├── DataTables/             # Generic data tables
│   └── Structs/
```

```
│       ├── Struct_FloatColumn
│       └── Struct_BaseDamage
├── Discovery/              # Discovery/exploration system
├── Globals/                # Global game settings
├── Impacts/                # Impact effects
├── Input/                  # Input bindings
├── Loot/                   # Loot system
│   ├── Balance/
│   │   └── DataTables/
│   │       ├── Struct_EnemyDrops
│   │       └── Struct_DedicatedDropProbability
│   └── LootSchedule/
├── Lootables/              # Lootable containers
├── Map/                    # Map data
├── Missions/               # Mission definitions
├── StatusEffects/          # Status effect data
├── WaypointPath/           # Navigation paths
└── WorldPainter/           # World generation
```

15.8. AI System

```
AI/
├── ArmyBandit/             # Bandit enemy faction
├── ArmyOrder/              # Order enemy faction
├── Bosses/                 # Boss enemies
│   ├── GrassBoss/
│   ├── Guardian/
│   ├── MountBoss/
```

```
│   ├── ShatterBoss/
│   ├── TKBoss/
│   └── _Shared/
├── Creatures/              # Non-humanoid enemies
├── Critters/               # Small creatures
├── NPC/                    # Non-enemy NPCs
└── _Shared/                # Shared AI resources
```

15.9. File Naming Conventions

Prefix	Type	Example
Struct_*	Structure definitions	Struct_EnemyDrops
DT_*	Data Tables	DT_WeaponStats
Body_*	Body/mesh definitions	Body_Pistol_01
DST_*	Destruction definitions	DST_Barrel
M_*	Materials	M_Metal_Base
MI_*	Material Instances	MI_Weapon_Red
MF_*	Material Functions	MF_Damage_Flash
AS_*	Animation Sequences	AS_Reload
Script_*	Blueprint scripts	Script_WeaponFire
StatusEffect_*	Status effects	StatusEffect_Burn

15.10. Weapon Part Types

From `NexusConfigStoreInventory` in DefaultGame.ini:

15.10.1. Core Weapon Parts

Part	Description
body	Main weapon body
body_acc	Body accessories
body_mag	Body magazine attachment
body_ele	Body elemental
body_bolt	Bolt mechanism
barrel	Weapon barrel
barrel_acc	Barrel accessories
barrel_licensed	Licensed barrel variants
magazine	Magazine
magazine_acc	Magazine accessories
magazine_borg	Borg magazine type
magazine_ted_thrown	Tediore thrown magazine

15.10.2. Attachment Parts

Part	Description
scope	Optical scopes
scope_acc	Scope accessories
rail	Rail attachments
bottom	Bottom attachments
grip	Weapon grip

Part	Description
foregrip	Forward grip
underbarrel	Underbarrel attachment
underbarrel_acc	Underbarrel accessories
underbarrel_acc_vis	Visible underbarrel accessories

15.10.3. Manufacturer-Specific

Part	Description
tediore_acc	Tediore accessories
tediore_secondary_acc	Tediore secondary accessories
hyperion_secondary_acc	Hyperion secondary accessories
turret_weapon	Turret weapon parts

15.10.4. Element & Augments

Part	Description
element	Primary element
secondary_ele	Secondary element
secondary_ammo	Secondary ammo type
primary_augment	Primary augment slot
secondary_augment	Secondary augment slot
enemy_augment	Enemy-dropped augment
active_augment	Active skill augment
endgame	Endgame modifications
unique	Unique/legendary parts

15.10.5. Grenade Parts

Part	Description
payload	Grenade payload
payload_augment	Payload augment
stat_augment	Stat augment
curative	Healing effect
augment	General augment
utility_behavior	Utility behavior

15.10.6. Class Mod Parts

Part	Description
class_mod_body	Class mod body
action_skill_mod	Action skill modifier
core_augment	Core augment
core_plus_augment	Core plus augment
passive_points	Passive skill points
special_passive	Special passive abilities
stat_group1/2/3	Stat groups

15.10.7. Other

Part	Description
firmware	Firmware upgrades
augment_element	Elemental augments
augment_element_resist	Elemental resistance
augment_element_nova	Nova effect

Part	Description
augment_element_splat	Splat effect
augment_element_immunity	Elemental immunity
detail	Detail parts
skin	Weapon skins
vile	Vile rarity parts
pearl_elem	Pearl elemental
pearl_stat	Pearl stat bonus

15.11. Key DataTables

15.11.1. Weapon Naming

Asset Path	Contents
/Game/Gear/Weapons/_Shared/NamingStrategies/WeaponNamingStruct	Weapon prefix naming
/Game/Gear/Weapons/_Shared/NamingStrategies/DAD_LicensedPart_Table_Struct	Daedalus licensed parts
/Game/Gear/Weapons/_Shared/NamingStrategies/TED_PayloadPrefix_Table_Struct	Tediore payload prefixes

15.11.2. Balance Data

Asset Path	Contents
/Game/Gear/Weapons/_Shared/BalanceData/BarrelData/*	Barrel stat modifiers
/Game/Gear/Weapons/_Shared/BalanceData/MagazineData/*	Magazine stat modifiers
/Game/Gear/Weapons/_Shared/BalanceData/Rarity/*	Rarity tier modifiers
/Game/Gear/Weapons/_Shared/BalanceData/Elemental/*	Elemental damage data
/Game/GameData/DataTables/Structs/Struct_BaseDamage	Class mod stat tables

---

15.12. Asset Path Mapping

Game paths use `/Game/` prefix:

Game Path	Extracted Path
/Game/Gear/Weapons/…	OakGame/Content/Gear/Weapons/…
/Script/OakGame.ClassName	Engine script class (not extractable)

15.13. Gear Types Found

Type	Description
ClassMod	Character class modifications
Enhancement	Enhancement items
Firmware	Firmware upgrades
Gadget	Deployable gadgets
Grenade	Grenade items
RepairKit	Repair kits
Shield	Shield equipment

15.14. New Systems in BL4

BL4 introduces several systems not present in BL3:

System	Description
Repair Kits	Healing/repair items
Enhancements	Enhancement slot items
Firmware	Firmware upgrade system
Gadgets	Turrets, Terminals, Heavy Weapons
Armor Shards	Armor shard items
Shield Boosters	Shield booster pickups
Hover Drives	Vehicle hover drive upgrades
Borg	New manufacturer with charge mechanics
Pearl	Pearl rarity tier
Vile	Vile rarity tier

15.15. Extraction Results

From `share/manifest/pak_summary.json`:

Field	Value
Total Assets	81,097
Stats Count	519
Naming Strategies	3
Manufacturers	9

15.15.1. Manufacturer Codes

BOR, TOR, VLA, COV, MAL, TED, DAD, JAK, ORD

15.15.2. Balance Data Categories

- firmware
- Heavy
- gadget
- repair_kit
- unknown
- grenade
- shield
- weapon

15.16. Notes

1. **Structure vs Data**: Files named `Struct_*` are type definitions (schemas), not actual data tables.
2. **IoStore Format**: BL4 uses UE5's IoStore container format with Zen packages.
3. **Missing Data**: Per-item balance data is derived from parts at runtime rather than stored in data files.
4. **Compression**: BL4 uses Oodle compression for IoStore containers.

15.17. NCS Format (Nexus Config Store)

NCS is Gearbox's format for storing item pool definitions, part data, and other game configuration that isn't in standard PAK assets. For the complete NCS format specification, see Chapter 6: NCS Format.

15.17.1. Key NCS Files

File	Purpose
`inv.bin`	Inventory part definitions, serial indices
`itempoollist.bin`	Boss → legendary item mappings
`itempool.bin`	Item pool definitions, rarity weights
`loot_config.bin`	Global loot configuration

File	Purpose
`gbxactor.bin`	Actor/enemy definitions
`achievement.bin`	Achievement definitions
`attribute.bin`	Game attribute values

15.17.2. Quick Reference: NCS Chunk Header

```
Offset  Size  Type   Description
------  ----  ----   -----------
0x00    1     u8     Version (always 0x01)
0x01    3     bytes  Magic: "NCS" (0x4e 0x43 0x53)
0x04    4     u32    Compression flag (0 = raw, non-zero = Oodle)
0x08    4     u32    Decompressed size (little-endian)
0x0c    4     u32    Compressed size (little-endian)
```

15.17.3. Quick Reference: Format Codes

Code	Description	Examples
`abjx`	Extended entries with dependents	achievement, preferredparts
`abij`	Indexed entries	itempoollist, aim_assist
`abjl`	Labeled entries	inv_name_part
`abhj`	Hash-indexed entries	inv_params
`abpe`	Property-based entries	audio_event

15.17.4. Quick Reference: NCS Manifest (`_NCS/`)

Each pak file contains a manifest at `_NCS/` listing all NCS data stores. Entries contain filename + index; sorting by index gives correct chunk order.

15.17.5. Decompression CLI

```
# Default (oozextract backend, ~97% compatibility)
bl4 ncs decompress pakchunk0.pak -o output/

# Native Oodle DLL (100% compatibility)
bl4 ncs decompress pakchunk0.pak -o output/ --oodle-dll /path/to/oo2co

# External command (cross-platform)
bl4 ncs decompress pakchunk0.pak -o output/ --oodle-exec ./oodle_wrapp
```

Extracted from BL4 game files using retoc and uextract tools.

16. Glossary

Quick reference for terms used throughout this guide. Page references indicate the primary location where each term is explained.

16.1. A

AActor Base class for all objects that can be placed in a level. Size: 912 bytes. See Appendix A.

AES-256-ECB Advanced Encryption Standard with 256-bit key in Electronic Codebook mode. Used by BL4 for save file encryption. See Chapter 4.

ASLR Address Space Layout Randomization. Security feature that randomizes memory addresses on each launch. See Chapter 3.

AOakCharacter BL4's main player/enemy character class. Size: 38,800 bytes. Contains health, damage, and weapon state. See Appendix A.

16.2. B

bl4-community Axum-based REST API server for sharing verified item data between users. Part of the bl4 monorepo. See Chapter 9.

Base85 Number encoding using 85 printable ASCII characters. BL4 uses a custom alphabet for item serials. See Chapter 5.

Big-Endian Byte order where most significant byte comes first. Used in Base85 decoding. See Chapter 1.

Bit Mirroring Reversing the bit order within each byte (e.g., 0b10000111 → 0b11100001). Part of serial decoding. See Chapter 5.

Bitstream Sequence of bits read without byte alignment. Used in item serial encoding. See Chapter 5.

16.3. C

CDO Class Default Object. UE's template object containing default property values. See Chapter 2.

ClassPrivate UObject field at offset 0x10 pointing to the object's UClass. See Chapter 2.

Comparison Index The 32-bit index portion of an FName, used to look up strings in GNames. See Chapter 2.

16.4. D

Differential Encoding NCS string compression where subsequent strings store only their difference from the first. The prefix `1airship` means "replace first character and append 'airship'". See Chapter 6.

DataTable UE asset type for tabular data. Contains rows of structured data. See Chapter 7.

Dedicated Drop Loot that only drops from specific enemies. See Appendix C.

16.5. E

ECB Electronic Codebook. Block cipher mode where identical plaintext blocks produce identical ciphertext. See Chapter 4.

16.6. F

FField UE5's property descriptor base class. Replaces UProperty from UE4. See Appendix A.

FName Unreal's string identifier. 8 bytes containing index and instance number. See Chapter 2.

FNV-1a Fowler-Noll-Vo hash function (variant 1a). Used by NCS format for field name lookups. 64-bit version with offset basis 0xcbf29ce484222325 and prime 0x100000001b3. See Chapter 6.

FOD (Fog of Discovery) The map fog overlay that clears as you explore. Stored in save files as a 128x128 grayscale alpha map per zone (0=fogged, 255=revealed, intermediate values for soft edges). See Chapter 4.

FNamePool Global string pool containing all FName strings. Also called GNames. See Chapter 2.

FProperty UE5 property descriptor. Contains offset, size, and type information. See Appendix A.

FString Unreal's dynamic string type. 16 bytes containing pointer, count, and capacity. See Appendix A.

FTransform 96-byte structure containing rotation (FQuat), translation (FVector), and scale. See Appendix A.

FVector 24-byte 3D vector using doubles (not floats like UE4). See Appendix A.

16.7. G

gBx Magic header for NCS files (0x67 0x42 0x78). Followed by variant byte and Oodle-compressed payload. See Chapter 6.

GNames Global FName string pool. Located at offset 0x112a1c80 from PE base. See Chapter 2.

GUObjectArray Global array containing all UObjects. Located at offset 0x113878f0. See Chapter 2.

GWorld Pointer to current UWorld. Located at offset 0x11532cb8. See Appendix A.

16.8. H

Heap Memory region for dynamic allocations. Valid range: 0x10000-0x800000000000. See Chapter 3.

Hexadecimal Base-16 number system using 0-9 and A-F. See Chapter 1.

16.9. I

InternalIndex UObject field at offset 0x0C containing position in GUObjectArray. See Chapter 2.

IoStore UE5's container format using .utoc (table of contents) and .ucas (data) files. See Chapter 7.

Item Pool Definition of possible loot drops. See Appendix C.

Item Serial Base85-encoded string representing an item's full configuration. See Chapter 5.

16.10. L

Licensed Parts BL4's cross-manufacturer part system. A weapon can gain abilities from other manufacturers (e.g., Jakobs Ricochet on a Vladof rifle). Level-gated

via `Att_MinGameStage_LicensedPart_*` attributes. See Chapter 8.

Little-Endian Byte order where least significant byte comes first. Used by x86/x64 and save files. See Chapter 1.

Luck Game system that modifies loot rarity chances. See Appendix C.

16.11. M

Magic Header Fixed bit pattern at start of data. Item serials use 0010000 (7 bits). See Chapter 5.

MDMP Windows minidump format. Used for memory dump files. See Chapter 3.

16.12. N

NCS (Nexus Config Store) Gearbox's format for storing item pools, part data, and game configuration. Uses gBx header with Oodle compression. Contains data not found in standard PAK assets. See Chapter 6.

NamePrivate UObject field at offset 0x18 containing the object's FName. See Chapter 2.

Nibble Half a byte (4 bits). Used in VarInt encoding. See Chapter 5.

16.13. O

ObjectFlags UObject field at offset 0x08 containing RF_* state flags. See Chapter 2.

Oodle Compression algorithm used in UE5 IoStore containers and NCS files. BL4 uses version 9 (oo2core_9_win64.dll). See Chapter 6 and Appendix D.

OuterPrivate UObject field at offset 0x20 pointing to parent/container object. See Chapter 2.

16.14. P

Parts System BL4's item assembly system where weapons are composed of individual parts (barrel, grip, scope, etc.) drawn from per-manufacturer pools. Defined in NCS `inv.bin`, not PAK assets. See Chapter 8.

Pak File Legacy UE archive format (.pak). See Chapter 7.

Part Token type in item serials representing weapon components. See Chapter 5.

PE Base Base address of Windows executable (0x140000000 for BL4). See Chapter 3.

PKCS7 Padding scheme for block ciphers. See Chapter 4.

Pointer Chain Sequence of pointer dereferences to reach target data. See Chapter 3.

16.15. R

Root/Sub Scope In BL4's part system, the `GbxSerialNumberIndex.scope` field distinguishes Root (core item structure) from Sub (modular attachment) parts. In serials, part indices map directly to the parts database — no bit manipulation needed. See Chapter 8.

Rarity Item quality tier (Common, Uncommon, Rare, Epic, Legendary). See Appendix B.

Reflection UE's runtime type information system. See Chapter 2.

retoc Tool for extracting IoStore containers. See Chapter 7.

RIP-Relative x64 addressing mode relative to instruction pointer. See Chapter 3.

16.16. S

Separator Token type in item serials marking section boundaries. See Chapter 5.

Serial See Item Serial.

Steam ID Unique identifier for Steam accounts. Used to derive encryption key. See Chapter 4.

SuperStruct UStruct field at offset 0x40 pointing to parent class. See Appendix A.

16.17. T

Tag-Based Encoding The most complex NCS binary format, used by `inv.bin` and `gbxactor.bin`. Each byte acts as a type tag (0x61=pair, 0x62=u32, 0x63=u32f32, 0x64-0x66=list, 0x70=variant, 0x7a=end) determining how to interpret following data. See Chapter 6.

TArray Unreal's dynamic array template. 16 bytes containing pointer, count, max. See Appendix A.

Token Discrete data element in item serial bitstream. Types: VarInt, VarBit, Part, String, Separator. See Chapter 5.

16.18. U

uasset Unreal asset file containing object definitions and properties. See Chapter 7.

UClass UE's class definition object. Size: 512 bytes. See Chapter 2.

ucas IoStore container archive file containing compressed asset data. See Chapter 7.

uexp Bulk data file accompanying .uasset. See Chapter 7.

UObject Base class for all Unreal objects. Size: 40 bytes. See Chapter 2.

Usmap Mapping file containing UE reflection data for parsing unversioned assets. See Chapter 2.

UStruct UE's structure/class layout descriptor. Size: 176 bytes. See Chapter 2.

utoc IoStore table of contents file. See Chapter 7.

16.19. V

VarBit Token type with 5-bit length prefix followed by N data bits. See Chapter 5.

VarInt Variable-length integer using 4-bit nibbles with continuation bits. See Chapter 5.

Virtual Address (VA) Memory address in process's virtual address space. See Chapter 3.

VTable Virtual function table pointer at offset 0x00 of UObjects. See Chapter 2.

16.20. W

WASM WebAssembly. Binary format for running code in browsers. See Chapter 9.

16.21. Z

Zen Package UE5's internal package format used in IoStore containers. See Chapter 7.

zlib Compression library. Used for save file compression after encryption. See Chapter 4.

16.22. Symbols

@Ug Prefix for all BL4 item serials. See Chapter 5.
0x Prefix indicating hexadecimal number. See Chapter 1.

16.23. Quick Reference: Playable Characters

Character Name	Internal Class Name	Class Mod Label
Amon	Char_Paladin	Paladin Class Mod
Rafa	Char_ExoSoldier	Exo Soldier Class Mod
Harlowe	Char_Gravitar	Gravitar Class Mod
Vex	Char_DarkSiren	Dark Siren Class Mod

16.24. Quick Reference: Key Offsets

Symbol	Offset	Description
GUObjectArray	0x113878f0	All UObjects
GNames	0x112a1c80	FName pool
GWorld	0x11532cb8	Current world

Symbol	Offset	Description
ClassPrivate	+0x10	UObject’s class
NamePrivate	+0x18	UObject’s name
OuterPrivate	+0x20	UObject’s parent
SuperStruct	+0x40	UStruct’s parent

16.25. Quick Reference: File Extensions

Extension	Description
.sav	Encrypted save file
.pak	Legacy archive
.utoc	IoStore index
.ucas	IoStore data
.uasset	Asset file
.uexp	Bulk data
.usmap	Mapping file

This glossary covers terms from all chapters and appendices.