Borderlands 4 Reverse Engineering Guide

monokrome

2025-12-28

Table of contents

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.

- :material-download:{ .lg .middle } **Download**

Get the complete guide for offline reading.

:material-file-pdf-box: PDF :material-book-open-variant: EPUB :material-kindle: MOBI

1.1. Chapters

1.1.1. Part I: Foundations

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

1.1.2. Part II: Analysis Techniques

Chapter	Title	Description
3	Memory Analysis	Process memory, dumps, pattern scanning, pointer chains
4	Save File Format	Encryption, compression, YAML structure, key derivation
5	Item Serials	Base85 encoding, bit manipulation, token parsing

1.1.3. Part III: Practical Application

Chapter	Title	Description
6	Data Extraction	Pak files, asset parsing, manifest generation
7	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.

Just want the tool reference? Go to Chapter 7: 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

Part I.

Core Concepts

2. 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.

2.1. 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.

2.2. 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.

2.3. 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—those shareable weapon codes—use big-endian for the Base85 decoding step. Always verify which you're dealing with before assuming.

2.4. 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.

2.5. 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.

2.6. 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.

2.7. 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.

2.8. 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.

2.9. 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

2.10. 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

3. 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.

3.1. 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.

3.2. 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.

3.3. 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.

3.4. 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.

3.5. 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.

3.6. 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.

3.7. 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.

3.8. 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)
```

3.9. 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.

3.10. 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.

3.11. 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)
```

3.12. 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

4. 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.

4.1. 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.

4.2. 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.

4.3. 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.

4.4. 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.

4.5. Following Pointer Chains

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

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
```

4.6. 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)
```

4.7. 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.

4.8. Practical Example: Finding Item Serials

Item serials—those shareable weapon codes—exist in memory as strings. 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.

4.9. 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.

4.10. 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
```

4.11. 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.

4.12. 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.

4.13. 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

4.14. 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.

Practical Application

5. 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.

5.1. 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.

5.2. 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.

```
.sav file
    └── AES-256-ECB encrypted
        └── zlib compressed
            └── 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.

5.3. The Encryption Layer

Open a save file in a hex editor and the first bytes tell you what you're dealing with:

```
00000000: 41 45 53 2D 32 35 36 2D 45 43 42 00 ...
          A  E  S  -  2  5  6  -  E  C  B  \0
```

"AES-256-ECB" in plain ASCII. The game literally labels its encryption scheme. Following that header (32 bytes total) comes the encrypted payload.

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.

5.4. 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] = [
    0x8E, 0x62, 0xA9, 0x4C, 0x50, 0x60, 0x1A, 0x9D,  // XOR'd with Ste
    0x1C, 0x72, 0xD2, 0xAB, 0x95, 0xFC, 0x10, 0xD0,
    0xD7, 0xA9, 0x26, 0x95, 0x70, 0x56, 0x72, 0x7D,
    0xB4, 0x24, 0x2C, 0x77, 0xAD, 0xF2, 0xB1, 0x51,
];

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.

5.5. 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.

5.6. 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.

5.6.1. Character Save Structure

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

```yaml
state:
  char_guid: EAFFA60B46492388B1ED39807437595D
  class: Char_Paladin              # Char_Paladin, Char_DarkSiren, etc
  char_name: Amon
  player_difficulty: Easy
  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...'
          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
    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
```

5.6.2. 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

5.6.3. 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

Bit	Value	Meaning
5	32	Label 2
6	64	Label 3
7	128	Label 4
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).

5.7. 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.

5.8. 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**

5.8.1. Adding to Backpack

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

```
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
```

5.8.2. 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).

5.8.3. 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.

5.9. Common Edits

Adding currency:

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

Changing character name:

```
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
1	0
2	1,100
30	821,362
50	3,430,207

The curve follows approximately `XP ≈ 202 × level^2.44`.

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.

5.10. Map Exploration Data (foddatas)

The `foddatas` section stores your map exploration progress—the areas you've uncovered as you explore each zone. This data is surprisingly large, as it tracks exploration state at a granular level for every map in the game.

5.10.1. Structure

```yaml
fodsaveversion: 2
foddatas:
  - levelname: World_P
    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 (typically 128)
`foddimensiony`	Grid height (typically 128)
`compressiontype`	Always `Zlib`
`foddata`	Base64-encoded, zlib-compressed exploration bitmap

5.10.2. 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

Level Name	In-Game Zone

5.10.3. Copying Exploration Progress

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

```
# Decrypt both saves
bl4 save decrypt source.sav source.yaml
bl4 save decrypt target.sav target.yaml

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

The foddata is substantial—a fully-explored save can have 40KB+ of exploration data compared to a fresh character's few hundred bytes.

5.11. Safehouse and World Progress

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

5.11.1. Safehouses

```
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.

5.11.2. 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
```

5.11.3. 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
      orderbounty_grasslands_2: 1
    bounties_vanguard:
      vanguardbounty_grasslands_1: 1
      vanguardbounty_mountains_1: 1
```

5.11.4. Collectibles

```
    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
```

5.12. Unlockables

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

5.12.1. Hoverdrive Skins

```
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>`.

5.12.2. Vault Hunter Rank

```
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).

5.13. 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.

5.14. 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.

5.15. Manual Decryption Walkthrough

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

Step 1: Read and parse the header

```
with open('profile.sav', 'rb') as f:
    data = f.read()

# First 12 bytes: "AES-256-ECB\0"
assert data[:11] == b'AES-256-ECB'

# Bytes 24-28: compressed size (little-endian u32)
import struct
compressed_size = struct.unpack('<I', data[24:28])[0]
print(f"Compressed size: {compressed_size}")

# Encrypted payload starts at byte 32
encrypted = data[32:]
```

Step 2: Derive the key

```
STEAM_ID = 76561198012345678  # Replace with yours

BASE_KEY = bytes([
    0x8E, 0x62, 0xA9, 0x4C, 0x50, 0x60, 0x1A, 0x9D,
    0x1C, 0x72, 0xD2, 0xAB, 0x95, 0xFC, 0x10, 0xD0,
    0xD7, 0xA9, 0x26, 0x95, 0x70, 0x56, 0x72, 0x7D,
    0xB4, 0x24, 0x2C, 0x77, 0xAD, 0xF2, 0xB1, 0x51,
])

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, prepend the header.

5.16. 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.

5.17. 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.

5.18. 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

6. 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.

6.1. What's Encoded in a Serial

A serial is self-contained. Given just the string, the game can create an identical item anywhere—your inventory, a friend's inventory, another platform entirely. This is how "gun codes" work in Borderlands communities. Copy a serial, share it, and the recipient gets the exact same weapon.

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.

6.2. The Decoding Pipeline

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

```
"@Ugr$ZCm/&tH!..."  →  Strip "@U" prefix
"gr$ZCm/&tH!..."    →  Base85 decode to bytes
[0x84, 0xA5, ...]   →  Bit-mirror each byte
[0x21, 0xA5, ...]   →  Parse as bitstream tokens
{Category: 22, Level: 50, Parts: [...]}
```

The prefix `@U` marks this as a BL4 serial. The third character indicates item type—`r` for a weapon, `e` for equipment, and so on. After stripping the prefix, everything else is Base85-encoded binary data.

6.3. 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]
```

6.4. Bit Mirroring: The Obfuscation Layer

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

Why? Probably obfuscation. It makes casual inspection harder and might relate to how the game's internal serialization works. For our purposes, it's just another step to reverse:

```rust
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
}
```

6.5. Token Parsing: The Real Structure

The mirrored bytes form a bitstream parsed MSB-first. 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.

VarBit starts with a 5-bit length, then that many bits of data. 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.

6.6. Item Type: Determined by First Token, Not Character

Important discovery: The third character in a serial (like `b`, `e`, `r`) is NOT an explicit type encoding. It's simply a byproduct of Base85-encoding the first token's bits. 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 "type characters" might represent the same category of item, and vice versa. Always determine type from the bitstream, not the character.

6.7. Two Serial Formats

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

6.7.1. Weapon Format (VarInt-first)

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

```
[0] VarInt: manufacturer_weapon_id   (e.g., 2 = Jakobs Pistol, 14 = Ri
[1] SoftSeparator
[2] VarInt: 0
[3] SoftSeparator
[4] VarInt: 8
[5] SoftSeparator
[6] VarInt: level_code               <- LEVEL ENCODED HERE
[7] Separator
[8] VarInt: 4
[9] SoftSeparator
[10] VarInt: seed                    <- Random seed for stats
[11] Separator
[12] Separator
[13+] Part tokens...
```

6.7.2. Equipment Format (VarBit-first)

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

```
[0] VarBit: category_identifier      <- Category * divisor
[1] Separator
[2] VarBit: level_code               <- LEVEL ENCODED HERE
[3] Separator
[4] String: (often empty)
[5] VarInt: (varies)
[6+] More data and parts...
```

For VarBit-first serials, the category is extracted using a divisor:

```
Category ≈ first_varbit / 385   (most equipment)
Category ≈ first_varbit / 8192  (some shields)
```

The divisor isn't exact—categories are approximate matches. The bl4 tools handle this automatically.

6.8. Type-Aware Category Lookups

Critical: Category numbers overlap between item types. The same category number means different things for different item types.

For example, category 20: - For weapons: Daedalus SMG - For r-type shields: Energy Shield

This means you must know the item type before interpreting the category. The bl4 tools use type-aware lookup:

```
pub fn category_name_for_type(item_type: char, category: i64) -> Optio
    match item_type {
        'r' => SHIELD_CATEGORY_NAMES.get(&category)
            .or_else(|| CATEGORY_NAMES.get(&category)),
        _ => CATEGORY_NAMES.get(&category),
    }
}
```

Known shield categories (r-type items): | Category | Type | |———-|——| | 16 | Energy Shield | | 20 | Energy Shield | | 21 | Energy Shield | | 24 | Energy Shield | | 28 | Armor Shield | | 31 | Armor Shield |

6.9. Decoding a Serial Manually

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

Step 1: Structure - Prefix: `@U` (stripped) - Type character at position 3: `r` (weapon) - 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: 16 - 16 bits of data: 180928

Part Group ID = 180928 / 8192 = 22 (Vladof SMG)

The bl4 tool handles all this:

```
bl4 serial decode '@Ugr$ZCm/&tH!t{KgK/Shxu>k'
# Output shows tokens: 180928 | 50 | {0:1} 21 {4} , 2 , , 105 102 41
```

6.10. 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.

Important: The first VarInt in a weapon serial (the "serial ID") is NOT the same as the Part Group ID. There's a mapping between them. For example, serial ID 2 = Jakobs Pistol, but Part Group ID 2 = Daedalus Pistol. The bl4 tools handle this conversion automatically via `serial_id_to_parts_category()`.

Pistols (2-7):

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

Shotguns (8-12):

ID	Manufacturer	Code
8	Daedalus	DAD_SG
9	Jakobs	JAK_SG
10	Tediore	TED_SG
11	Torgue	TOR_SG
12	Bor	BOR_SG

Assault Rifles (13-18):

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

SMGs (20-23):

ID	Manufacturer	Code
20	Daedalus	DAD_SM
21	Bor	BOR_SM
22	Vladof	VLA_SM
23	Maliwan	MAL_SM

Snipers (26-29):

ID	Manufacturer	Code
26	Jakobs	JAK_SR
27	Vladof	VLA_SR
28	Order	ORD_SR
29	Maliwan	MAL_SR

Heavy Weapons (244-247):

ID	Manufacturer	Code
244	Vladof	VLA_HW
245	Torgue	TOR_HW
246	Bor	BOR_HW
247	Maliwan	MAL_HW

Shields (279-288):

ID	Type	Code
279	Energy Shield	energy_shield
280	Bor Shield	bor_shield
281-288	Manufacturer variants	dad/jak/mal/ord/ted/tor/vla_shield

Equipment/Gadgets (300-409):

Gadget categories use a type/subtype pattern. The base type is `category / 10 * 10`, and the subtype is `category % 10`:

Category	Type	Code
300-309	Grenade Gadget	grenade_gadget
310-319	Turret Gadget	turret_gadget
320-329	Repair Kit	repkit
330-339	Terminal Gadget	terminal_gadget
400-409	Enhancement	enhancement

For example, category 321 = Repair Kit (type 32), subtype 1.

The bl4 tools handle this automatically:

```
// For gadget range (300-399), try base type
if (300..400).contains(&category) {
    let base = category / 10 * 10;
    return CATEGORY_NAMES.get(&base);
}
```

6.11. 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.

Common part indices (within each category):

Index	Typical Part Type
2	Damage modifier
3	Crit damage modifier
4	Reload speed modifier
5	Magazine size modifier
7-10	Body variants
15-18	Barrel variants

The full parts database (`share/manifest/parts_database.json`) contains 2,615 parts across 53 categories, extracted from memory analysis.

6.12. Level Encoding

Level is encoded at different positions depending on item format:

- **Weapons**: 4th VarInt (position 6 in token list) — direct encoding
- **Equipment**: VarBit immediately after the first separator (position 2) — **0-indexed storage**

6.12.1. Equipment Level Storage (0-indexed)

Equipment levels are stored as `level - 1`:

VarBit Value	Display Level	Notes
0	1	Minimum level
29	30	Mid-game
49	50	Max level
50+	Invalid	Beyond current cap

Verification: All items with `/)}}` pattern (level 50) have VarBit=49. Tested across Throwing Knives, Energy Shields, Class Mods, and Grenades.

6.12.2. Observed Values

Code	Binary	In-Game Level	DB Rarity Label
30	`00011110`	30	Common
50	`00110010`	50	Common
142	`10001110`	44	Common
192	`11000000`	50	Epic
200	`11001000`	50	Legendary

For codes 1-50, level equals the code directly. For codes 128+, the formula `level = 2 × (code - 120)` works for values up to 145 (which gives level 50).

Codes 192 and 200 both correspond to level 50 items but appear on items with different rarity labels in the database. The bit differences: - Bit 6 (0x40): Set in 192 and 200, clear in 142 - Bit 3 (0x08): Set in 200, clear in 192

6.12.3. Equipment First VarBit

For VarBit-first equipment, the first VarBit encodes category plus additional data:

```
category = first_varbit / divisor
remainder = first_varbit % divisor
```

Observed correlations between remainder bits and database rarity: - VarBit 107200 (remainder 64, bits 7-6 = 1) → DB shows “Epic” - VarBit 78528 (remainder 192, bits 7-6 = 3) → DB shows “Legendary”

6.13. Element Encoding

Element types are encoded as Part tokens. The observed pattern:

```
Part Index = 128 + Element ID
```

Element ID	Element	Part Token	Verified On
0	Kinetic (None)	{128}	Hellhound
5	Corrosive	{133}	Seventh Sense
8	Shock	{136}	Armored Pre-Emptive Bod
9	Radiation	{137}	(from research notes)
13	Cryo	{141}	Seventh Sense
14	Fire	{142}	Hellwalker

Multi-element weapons contain multiple element tokens:

```
Tokens: ... {136} {141} ...  → Shock + Cryo
```

6.14. The UE5 Part System

Behind serials, parts are defined as UE5 objects. The `GbxSerialNumberIndex` structure links parts to their encoding:

```
GbxSerialNumberIndex
├── Category (Int64): Part Group ID
├── scope (Byte): Root=1, Sub=2
├── status (Byte): Active, Static, etc.
└── Index (Int16): Position in category
```

Each `InventoryPartDef` contains this structure plus the part's stat modifiers, visual mesh references, and other properties.

The game's internal registration order determines indices—not alphabetical sorting. This is why we extract mappings from memory dumps rather than inferring them from file names.

6.15. Comparing Serials

When you have two similar items and want to find what differs:

```
Serial 1: @Ugd$YMq/.&{!gQaYQ1)<G9C8...
Serial 2: @Ugd$YMq/.&{!gQaYQ1)<?B8b...
```

Decode both, align the tokens, find where they diverge. The difference reveals what that section encodes. Two weapons with identical parts but different accuracy will differ only in the accuracy-related bytes.

6.16. Decoding Examples

6.16.1. Weapon Serial

Serial: `@Ugd_t@FmVuJyjIXzRG}JG7S$K^1{DjH5&-`

Decoded tokens:

```
9 ,  0 ,  8 ,  200 | 4 ,  1367 | | {172} {4} {6} {164} {167} {142} {65} {19:7}
```

- First VarInt (9): Jakobs Shotgun
- 4th VarInt (200): Level 50, additional flags set
- Seed (1367): Random seed after first separator
- Part token `{142}`: Fire element (128 + 14)

6.16.2. Equipment Serial

Serial: `@Uge8jxm/)@{!bAp5s!;381FF>eS^@w`

Decoded tokens:

```
107200 | 50 | "" 4 ,  56 | | {9} {4} {250:131} {5}
```

- First VarBit (107200): Category 279 (Energy Shield), remainder 64
- Second VarBit (50): Level 50
- Part tokens follow after separators

6.17. Exercises

Exercise 1: Identify Item Types

Given these serials, what category is each? 1. `@Uge8jxm/)@{!gQaYMipv(G&-b*Z~_` 2. `@Ugw$Yw2}TYgOvDMQhbq)?p-8<%Z7L5c7pfd;cmn_` 3. `@Ug!$ZCm/&tH!t{KgK/Shxu>k`

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

1. `e` at position 3 → Equipment (shield/enhancement)
2. `w` at position 3 → Weapon (SMG category)
3. `!` at position 3 → Class Mod

6.18. What's Next

Serials encode items completely—but where do the part definitions come from? The Part Group IDs we use come from analyzing game data. The mappings between indices and actual parts come from memory dumps and pak file extraction.

Next, we'll explore how to extract data from BL4's game files, including the investigation into whether authoritative category mappings exist anywhere we can reach them.

Next: Chapter 6: Data Extraction

7. Chapter 6: 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.

7.1. 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.

7.2. 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/
```

7.3. 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.

7.3.1. The 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.

!!! warning “False Positives” 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.

7.3.2. 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
```

7.3.3. 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

7.3.4. 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.

!!! note "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.

7.4. 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:

7.4.1. 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 index!)
```

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

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

!!! important “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.

7.4.2. 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

7.4.3. 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
```

7.4.4. 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.

!!! tip “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.

7.4.5. Extraction Results

Running the extraction on a Dec 2025 memory dump yields:

Metric	Value
Total parts extracted	1,070
FNames scanned	1,399
Categories covered	49
Match rate vs reference	84.3%
Core weapons (cat 2-29)	**100% accurate**

Distribution by type:

Type	Categories	Parts
Pistols	2-7	207
Shotguns	8-13	190
Assault Rifles	14-19	169
SMGs	20-23	135
Snipers	25-29	176
Heavy Weapons	244-247	37
Shields	279-287	44
Gadgets	300-330	146
Enhancements	400-409	23

!!! success “Core Weapon Accuracy” The extraction achieves **zero mismatches** for core weapon categories (2-29). Mismatches only occur in heavy weapons, shields, gadgets, and enhancements—likely due to different UObject layouts or reference data issues for those categories.

7.5. 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 memory-extracted mappings with empirically-verified data for comprehensive coverage.

7.6. Extraction Tools

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

7.6.2. 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 "BalanceData"

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

7.7. 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`.

7.8. Extracting Parts from Memory

Since parts only exist at runtime, memory extraction is the path forward.

7.8.1. Step 1: Create Memory Dump

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

7.8.2. 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"
  ]
}
```

7.8.3. 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"}
  }
}
```

!!! important “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.

7.9. Working with Extracted Assets

7.9.1. 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/WeaponStats/Struct
  "exports": [
    {
      "class": "ScriptStruct",
      "properties": {
        "Damage_Scale": 1.0,
        "FireRate_Scale": 1.0,
        "Accuracy_Scale": 1.0
      }
    }
  ]
}
```

7.9.2. 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
```

7.9.3. Stat Patterns

Stats follow naming conventions: `StatName_ModifierType_Index_GUID`

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

7.10. 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/ThirdP
└── 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.

7.11. 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 "$GAME_DIR/OakGame/Content/Paks/pakchunk0-Windows_0_P.utoc" "$OUTPUT_DI

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

7.12. Summary: Data Sources

Data	Source	Extractable?
Balance/stats	Pak files	Yes
Naming strategies	Pak files	Yes
Loot pools	Pak files	Yes
Body definitions	Pak files	Yes

Data	Source	Extractable?
Part definitions	Memory dump	**Yes** (via UObject array scan)
Category mappings	Memory dump	**Yes** (embedded in part UObjects)

While parts don't exist as pak file assets, memory dumps capture the complete runtime state including all part definitions with their authoritative serial indices. The 0xFFFFFFFF sentinel pattern and UObject offset +0x20 provide reliable extraction paths.

7.13. 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?

7.14. 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 7: Using bl4 Tools

Part III.

Reference

8. Chapter 7: 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.

8.1. Building the Tools

8.1.1. 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
```

8.1.2. Build

```
cargo build --release -p bl4-cli

# Binary appears in ./target/release/bl4
```

8.2. 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)
  idb        Manage the verified items database
  launch     Launch Borderlands 4 with instrumentation
```

Aliases exist for common commands: `s` (save), `i` (inspect), `r` (serial), `m` (memory), `p` (parts).

8.3. Save File Operations

8.3.1. Inspect a Save

Quick view of save contents:

```
bl4 inspect 1.sav
bl4 inspect 1.sav --full  # Show complete YAML
```

8.3.2. 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
```

8.3.3. Edit Interactively

Opens decrypted save in your `$EDITOR`, then re-encrypts on save:

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

8.3.4. 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
```

8.4. Serial Operations

8.4.1. 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
```

Options:

```
bl4 serial decode --verbose '@Ugr...'  # Byte breakdown
bl4 serial decode --debug '@Ugr...'    # Bit-by-bit parsing
bl4 serial decode --analyze '@Ugr...'  # Token analysis
```

8.4.2. Compare

Side-by-side comparison of two serials:

```
bl4 serial compare '@Ugr$ZCm...' '@Ugr$ABC...'
```

8.4.3. Modify

Swap parts between serials:

```
bl4 serial modify '@base...' '@source...' '4,12'
```

8.4.4. Batch Decode

Decode many serials to binary for analysis:

```
bl4 serial batch-decode serials.txt serials.bin
```

8.5. Items Database (idb)

The items database tracks verified item data with source attribution.

8.5.1. 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
```

8.5.2. Import Items

```
# From save file
bl4 idb import-save 1.sav --decode --legal --source monokrome

# Decode all and populate metadata
bl4 idb decode-all
```

8.5.3. Attachments

```
bl4 idb attach '@Ugr...' screenshot.png
```

8.5.4. Value Attribution

Track values from different sources:

```
bl4 idb set-value '@Ugr...' rarity Epic --source ingame --confidence verified
bl4 idb get-values '@Ugr...' rarity
```

8.6. Memory Operations

Memory commands work with live processes or dump files.

8.6.1. With Dump File

```
bl4 memory --dump game.dmp info
bl4 memory --dump game.dmp discover gnames
bl4 memory --dump game.dmp discover guobjectarray
```

8.6.2. Generate Usmap

```
bl4 memory --dump game.dmp dump-usmap
```

8.6.3. FName Operations

```
bl4 memory --dump game.dmp fname 12345
bl4 memory --dump game.dmp fname-search "Damage"
```

8.6.4. 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
```

8.6.5. String Search

```
bl4 memory --dump game.dmp scan-string "DAD_AR.part_body" -B 128 -A 12
```

8.7. 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.

8.8. Common Workflows

8.8.1. Edit a Save

```
bl4 save edit ~/.steam/.../1.sav
# Editor opens, make changes, save & quit
# File is automatically re-encrypted
```

8.8.2. 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...'
```

8.8.3. Import Items from Saves

```
for sav in saves/*.sav; do
  bl4 idb import-save "$sav" --decode --legal
done
bl4 idb stats
```

8.8.4. 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/
```

8.9. Shell Tips

8.9.1. Quoting Serials

Serials contain `$`, `!`, `@`. Always use single quotes:

```
bl4 serial decode '@Ugr$ZCm/&tH!t{KgK/Shxu>k'
```

8.9.2. Aliases

```
alias bl4d='bl4 serial decode'
alias bl4i='bl4 inspect'
alias bl4e='bl4 save edit'
```

8.9.3. Piping

```
bl4 serial decode '@Ugr...' | grep "Category"
bl4 idb list | wc -l
```

8.10. Troubleshooting

8.10.1. “Decryption failed”

- Wrong Steam ID

- Corrupted save file
- Not a BL4 save

Verify the Steam ID matches the save file path.

8.10.2. “Invalid serial”

- Missing `@Ug` prefix
- Truncated copy
- Wrong quote type in shell

Copy the complete serial and use single quotes.

8.10.3. “Memory read failed”

- Address outside dump range
- Corrupted dump

Verify the dump covers the target address.

8.11. Quick Reference

Command	Description
`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

Command	Description
`bl4 save get <FILE> <PATH>`	Query value
`bl4 save set <FILE> <PATH> <VAL>`	Set value
`bl4 serial decode <SERIAL>`	Decode item serial
`bl4 serial compare <S1> <S2>`	Compare serials
`bl4 idb stats`	Database statistics
`bl4 idb import-save <FILE>`	Import from save
`bl4 memory --dump <F> <CMD>`	Memory analysis

8.12. 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

9. 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 hacking, cheat development, or deep reverse engineering.

9.1. Core UE5 Types

9.1.1. FName (8 bytes)

```
struct FName {
    int32_t ComparisonIndex;  // 0x00 - Index into GNames pool
    int32_t Number;           // 0x04 - Instance number (e.g., "Actor_5")
};
```

9.1.2. 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.

9.1.3. FRotator (24 bytes)

```
struct FRotator {
    double Pitch;  // 0x00
    double Yaw;    // 0x08
    double Roll;   // 0x10
};
```

9.1.4. FQuat (32 bytes)

```
struct FQuat {
    double X;  // 0x00
    double Y;  // 0x08
    double Z;  // 0x10
    double W;  // 0x18
};
```

9.1.5. 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
```

9.1.6. 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
};
```

9.1.7. FString (16 bytes)

```
struct FString {
    wchar_t* Data;    // 0x00 - Wide string pointer
    int32_t Count;    // 0x08 - String length
    int32_t Max;      // 0x0C - Buffer capacity
};
```

9.2. UObject Hierarchy

9.2.1. 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

9.2.2. UField (48 bytes)

```
class UField : public UObject {
    UField* Next;  // 0x28 - Next field in chain
};
```

9.2.3. 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
```

9.2.4. UClass (512 bytes)

```
class UClass : public UStruct {
    char pad[96];              // 0xB0
    UObject* ClassDefaultObject;  // 0x110 - CDO pointer
    char pad[232];             // 0x118
};  // Total: 0x200
```

9.3. Actor Hierarchy

9.3.1. AActor (912 bytes)

```
class AActor : public UObject {
    char pad[416];                    // 0x28
    USceneComponent* RootComponent;   // 0x1C8
    char pad[448];                    // 0x1D0
};  // Total: 0x390
```

9.3.2. 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
```

9.3.3. ACharacter (1864 bytes)

```
class ACharacter : public APawn {
    char pad[24];                                 // 0x410
    USkeletalMeshComponent* Mesh;                 // 0x428
    UCharacterMovementComponent* Movement;        // 0x430
    char pad[784];                                // 0x438
};  // Total: 0x748
```

9.4. Controller Classes

9.4.1. 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
```

9.4.2. 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+
```

9.5. BL4/Oak Classes

9.5.1. 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
```

9.5.2. 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
```

9.5.3. AGbxCharacter (15232 bytes)

```
class AGbxCharacter : public ACharacter {
    char pad[13368];  // 0x748
};  // Total: 0x3B80
```

9.5.4. 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

9.6. Inventory Classes

9.6.1. AInventory (2224 bytes)

```
class AInventory : public AActor {
    char pad[1312];  // 0x390
};  // Total: 0x8B0
```

9.6.2. 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
```

9.7. Currency System

9.7.1. FSToken (12 bytes)

```
struct FSToken {
    int32_t Hash;  // 0x00 - Token hash
    FName Name;    // 0x04 - Token name
};
```

9.7.2. FGbxCurrency (24 bytes)

```
struct FGbxCurrency {
    FSToken Token;   // 0x00
    char pad[4];     // 0x0C
    uint64_t Amount; // 0x10 - Currency amount
};
```

9.7.3. 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

9.8. Attribute Types

9.8.1. FGbxAttributeFloat (12 bytes)

```
struct FGbxAttributeFloat {
    char pad[4];       // 0x00
    float Value;       // 0x04 - Current value
    float BaseValue;   // 0x08 - Base value before modifiers
};
```

9.8.2. FGbxAttributeInteger (12 bytes)

```
struct FGbxAttributeInteger {
    char pad[4];        // 0x00
    int32_t Value;      // 0x04
    int32_t BaseValue;  // 0x08
};
```

9.9. Enums

9.9.1. ECharacterHealthCondition

```
enum class ECharacterHealthCondition : int8_t {
    Healthy = 0,
    Injured = 1,
    Dead = 2
};
```

9.9.2. 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
};
```

9.10. 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.

9.11. 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.

9.12. 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).

9.13. 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.

10. Appendix B: Weapon Parts Reference

This appendix catalogs all known weapon parts extracted from BL4 game files, organized by manufacturer and weapon type.

10.1. 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

10.2. Manufacturers

Code	Name	Serial ID	Weapon Types
BOR	Borg	-	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.

10.3. Weapon Types

10.3.1. 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

10.3.2. 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

10.4. Scope Parts by Manufacturer

10.4.1. BOR (Borg)

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

10.4.2. 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)

10.4.3. 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

10.4.4. 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

10.4.5. 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

10.4.6. 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

10.4.7. 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

10.4.8. 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

10.5. Barrel Parts

10.5.1. 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

10.6. Part Index Organization

10.6.1. 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

10.6.2. 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.

10.6.3. 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`

10.6.4. 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

10.6.5. 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

---

10.7. 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.

10.7.1. 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.

10.7.2. 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_b`	`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`).

10.7.3. 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`.

10.7.4. 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

10.7.5. 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

10.7.6. Rule 6: Legendary Barrel Restrictions

Some legendary barrels **cannot** roll with barrel accessories:

Barrel	Accessory Restriction
JAK_SG.part_barrel_vor_slugger	Never rolls with barrel accessories
JAK_SG.part_barrel_hellwalker	Doesn't work with barrel accessories
Unique barrels (general)	Often have restricted accessory pools

10.7.7. 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

10.7.8. 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
```

10.7.9. 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

10.8. Part Selection System (Theoretical)

!!! note "Speculative" This section describes classes found in game metadata. Actual implementation may differ. The "Part Compatibility Rules" section above contains verified behavior.

10.8.1. 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

---

10.9. 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

10.10. Weapon Naming System

Parts affect weapon prefix names through the naming table system.

10.10.1. 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

10.10.2. 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…

10.11. Known Legendaries

10.11.1. By Manufacturer

Internal Name	Display Name	Type	Manufa
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	Maliwa

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

10.12. 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

10.12.1. Internal Format

```
{MFG}_{TYPE}.comp_0{N}_{rarity}_{name}
```

Examples: - `TOR_SG.comp_05_legendary_Linebacker` - `VLA_SM.comp_05_legendary_KaoSon`

10.13. Part Count by Category

Data extracted from memory dump using `bl4 memory dump-parts` and `bl4 memory build-parts-db`.

10.13.1. 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

10.13.2. 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)

10.13.3. 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.

10.13.4. 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

10.13.5. Gadgets and Gear

Category ID	Type	Parts Count
300	Grenade Gadget	82
310	Turret Gadget	52
320	Repair Kit	107
330	Terminal Gadget	61

10.13.6. 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

10.13.7. Summary

Category	Total Parts
Weapons (all types)	1,928
Class Mods	2
Shields	79
Gadgets	302
Enhancements	28
Unmapped	276
Total	**2,615**

10.14. 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

10.15. 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.

Extracted from BL4 November 2025 patch memory dump using bl4 memory analysis tools.

Last updated: December 2025 - Added MAL_SG (cat 19), bor_sr (cat 25), class mods (44, 55, 97, 140), firmware (151), shield variant (289).

11. Appendix C: Loot System Internals

This appendix documents the internal workings of BL4's loot system, based on memory analysis and game file extraction.

11.1. Loot Pool Architecture

11.1.1. Core Classes

Class	Description
ItemPoolDef	Defines a loot pool
ItemPoolEntry	Single item in a pool
ItemPoolListDef	List of multiple pools
ItemPoolSelectorDef	Selection logic
ItemPoolInstanceData	Runtime instance data
ItemPoolSelectorStateDef	Selection state

Script path: `/Script/GbxGame.ItemPoolDef`

11.1.2. Pool Types Enum

```
enum class ELootPoolTypes {
    All = 0,
    BaseLoot = 1,
    AdditionalLoot = 2,
    DedicatedDrops = 3,
    GearDrivenDrops = 4,
    MAX = 5
};
```

11.2. Rarity Tiers

11.2.1. Tier Definitions

Tier	Component ID	Material Path
Common	comp_01_common	DA_MD_BOR_Common
Uncommon	comp_02_uncommon	DA_MD_BOR_Uncommon
Rare	comp_03_rare	DA_MD_BOR_Rare
Epic	comp_04_epic	DA_MD_BOR_Epic
Legendary	comp_05_legendary	DA_MD_BOR_Legendary_01

Material path pattern: `/Game/Gear/Weapons/_Shared/Materials/BOR/DA_MD_BOR_`

11.2.2. Price Modifier Attributes

Tier	Attribute
Common	attr_calc_pricemod_rarity_common
Uncommon	attr_calc_pricemod_rarity_uncommon
Rare	attr_calc_pricemod_rarity_rare
Epic	attr_calc_pricemod_rarity_epic
Legendary	attr_calc_pricemod_rarity_legendary

11.3. Loot Weight System

11.3.1. Weight Properties

Property	Description
GrowthExponent	Level scaling exponent
BaseWeight	Base drop weight
GameStageVariance	Variance by game stage
RelativeGameStage	Relative stage modifier
GameStageTable	Stage lookup table
LootGameStages	Game stages for loot
RankLootRarityTable	Rarity by rank table

11.3.2. Weight Classes

Class	Description
RarityWeightData	Weight configuration
LocalRarityModifierData	Local rarity modifiers

11.4. Luck System

11.4.1. Core Classes

Class	Description
LootGlobalsDef	Global loot settings
LuckCategoryAttribute	Luck category attribute
LuckCategoryAttributesState	Runtime luck state
LuckCategoryDef	Luck category definition
LuckCategoryValueResolver	Resolves luck values
LuckGlobals	Global luck settings

11.4.2. Luck Categories

Category	Description
LuckCategories	Base luck modifiers
EnemyBasedLuckCategories	Per-enemy modifiers
PlayerBasedLuckCategories	Player-specific modifiers

11.5. Drop Probability Tables

11.5.1. Dedicated Drop Probability (Struct_DedicatedDropProbability)

Tier	Probability
Primary	20% (0.2)
Secondary	8% (0.08)
Tertiary	3% (0.03)
Quaternary	0% (0.0)
Shiny	1% (0.01)
TrueBoss	0% (0.0)
TrueBossShiny	0% (0.0)

11.5.2. Enemy Drops (Struct_EnemyDrops)

Fields (DoubleProperty type):

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
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

Field	Description
Enhancements_HowMany	Number of enhancements
CurrencyOrAmmo_Probability	Currency/ammo chance
CurrencyAmmo_HowMany	Amount
EXP_Multiplier	Experience multiplier
Rarity_Modifier	Rarity boost

11.6. Known Item Pools

11.6.1. Boss Pools

Pool	Description
ItemPoolList_Enemy_BaseLoot_Boss	Standard boss drop
ItemPoolList_Enemy_BaseLoot_BossRaid	Raid boss drops
ItemPoolList_ShatterlandsCommanderFortress_TrueBoss	True Boss
ItemPoolList_Timekeeper_TKBoss_TrueBoss	Timekeeper True Bo

11.6.2. Rarity Pools

Pool	Description
itempool_guns_01_common	Common weapons
itempool_guns_02_uncommon	Uncommon weapons
itempool_guns_03_rare	Rare weapons
itempool_guns_04_epic	Epic weapons

Pool	Description
itempool_guns_05_legendary	Legendary weapons

11.6.3. Special Pools

Pool	Description
ItemPool_FishCollector_Reward_Legendary	Fish collector reward
ItemPool_BlackMarket_Comp_BOR_HW_DiscJockey	Black Market
ItemPool_BlackMarket_Comp_BOR_HW_Streamer	Black Market

11.7. Lootable Objects

11.7.1. Classes

Class	Description
GbxCondition_CanOpenLootable	Open condition check
GbxCondition_ShouldLootableShowLockedPrompt	Lock prompt
LootableObjectInstanceProxy	Instance proxy
LootableObjectBehaviorMod	Behavior modifier
LootableObjectBodySettings	Body settings
LootableObjectBodyState	Runtime body state

11.8. Inventory System

11.8.1. Core Classes

Class	Description
InventoryParam	Inventory parameter
InventoryParamsDef	Parameter definitions
InventoryRarityDataTableValueResolver	Rarity resolver
InventoryRarityDef	Rarity definition
InventorySerialNumber	Item serial number
InventoryStatsContainer	Stats container
InventoryStatsPropertyResolver	Stats resolver
InventoryStatTags	Stat tags
InventoryStatAttribute	Stat attribute

11.8.2. Key Class: InventoryRarityDataTableValueResolver

This class resolves rarity values from DataTables. Potential patching target for forcing specific rarity tiers.

11.9. RNG Implementation

11.9.1. Import Table Functions

Address	Function	DLL
0x150b74e18	std::_Random_device	MSVCP140.dll
0x150b75050	BCryptGenRandom	bcrypt.dll
0x150b759d0	rand	api-ms-win-crt-utility-l1-1-0.dll
0x150b759d8	srand	api-ms-win-crt-utility-l1-1-0.dll

11.9.2. RDRAND Usage

108 `RDRAND` instructions found in the binary. These are hardware RNG entry points.

Pattern: `0F C7 F0` (rdrand eax)

11.10. Memory Addresses

11.10.1. FName Locations

Region	Description
0x05b25000-0x05f96000	Class/struct names
0x1cd44000-0x1cd55000	Property/field names

11.10.2. Specific Addresses

Data	Address	Notes
ItemPoolDef class	0x5b25150	Class definition
Pool types enum	0x65610d50	ELootPoolTypes
Rarity tiers	0x79105c0	Tier definitions
Weight properties	0x1cd44680	Weight data
Luck classes	0x5f955e0	Luck system
Legendary items	0x94e7870	Item database

11.11. Loot Chance System

11.11.1. DataTable Structure

Found `LootChanceDefinedValueRow` for configuring loot chances.

Class	Description
LootChanceDefinedValueRow	DataTable row for chances
GetSummary_Chance	Chance summary function

11.12. Stat Modifiers

11.12.1. From Extracted Data

Stat	Description
Damage_Scale	Base damage multiplier
Damage_Value	Flat damage value
CritDamage_Add	Critical damage bonus
FireRate_Scale	Fire rate multiplier
FireRate_Value	Fire rate value
ReloadTime_Scale	Reload speed modifier
ElementalChance_Scale	Element proc chance
Accuracy_Scale	Accuracy modifier
Spread_Scale	Spread modifier
Recoil_Scale	Recoil modifier

11.13. Injection Approaches

11.13.1. LD_PRELOAD Method (Linux/Proton)

Intercept RNG at syscall level:

```
# Bias RNG for better drops
LD_PRELOAD=/path/to/libbl4_preload.so BL4_RNG_BIAS=max ./game
```

This intercepts `getrandom()` and similar syscalls.

11.13.2. Direct Memory Patching

Target locations for modification:

Template	Target	Address
dropRate	RarityWeightData	0x5f9548e
dropRate	BaseWeight	0x6f3a44c4
dropRate	GrowthExponent	0x6f3a44b4
luck	LuckGlobals	0x5f95658
luck	LuckCategories	0x6f3a4560

11.13.3. Implementation Strategy

1. Find live `InventoryRarityDataTableValueResolver` instances
2. Locate float weight values in DataTable
3. Patch weights to favor legendary tier
4. Or patch comparison code for weight threshold

11.14. Extracted Items Database

Located at `share/manifest/items_database.json`:

Content	Count
Item pools	62
Balance items	26
Stat types	73

11.14.1. Sample Pools

- `ItemPoolList_Enemy_BaseLoot_Boss`
- `ItemPoolList_Enemy_BaseLoot_BossRaid`
- `itempool_guns_01_common`
- `itempool_guns_04_epic`
- `ItemPool_FishCollector_Reward_Legendary`

11.14.2. Generate Database

```
bl4-research items-db -m share/manifest
```

11.15. Binary Protection

The executable uses protection (likely Denuvo):

- No exports (stripped)
- References `Borderlands4.pdb` but PDB not included
- Heavy obfuscation (XOR, NOT, RCL patterns)
- Runtime analysis more effective than static

!!! warning Static binary analysis is difficult. Runtime analysis via memory attachment is recommended.

Data from live memory analysis using `bl4 memory` tool.

12. Appendix D: Game File Structure

This appendix provides a complete reference of BL4's file structure, asset organization, and content layout.

12.1. 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

12.2. File Locations

12.2.1. Steam (Linux)

```
~/.steam/steam/steamapps/common/Borderlands 4/OakGame/Content/Paks/
```

12.2.2. Steam (Windows)

```
C:\Program Files (x86)\Steam\steamapps\common\Borderlands 4\OakGame\Co
```

12.3. 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

12.4. 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
```

12.5. 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
```

12.6. Gear System

12.6.1. 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
```

12.6.2. Weapon System

```
Gear/Weapons/
├── _Manufacturer/           # Manufacturer-specific data
│   ├── BOR/                 # Borg
│   ├── 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
```

12.7. 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
```

12.8. 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
```

12.9. 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

12.10. Weapon Part Types

From `NexusConfigStoreInventory` in DefaultGame.ini:

12.10.1. 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

12.10.2. 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

12.10.3. 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

12.10.4. 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

12.10.5. 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

12.10.6. 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

12.10.7. 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

12.11. Key DataTables

12.11.1. 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

12.11.2. 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

---

12.12. 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)

12.13. 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

12.14. 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

12.15. Extraction Results

From `share/manifest/pak_summary.json`:

Field	Value
Total Assets	81,097
Stats Count	519
Naming Strategies	3
Manufacturers	9

12.15.1. Manufacturer Codes

BOR, TOR, VLA, COV, MAL, TED, DAD, JAK, ORD

12.15.2. Balance Data Categories

- firmware
- Heavy
- gadget
- repair_kit
- unknown
- grenade
- shield
- weapon

12.16. 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.

12.17. 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.

12.17.1. Why NCS Matters

Standard PAK extraction returns 0 results for key classes: - `ItemPoolDef` - Item pool definitions - `ItemPoolListDef` - Item pool lists - `loot_config` - Loot configuration

These are stored in **NCS format**, not as standard uasset files. The class definitions exist in `scriptobjects.json`, but the actual data lives in NCS.

12.17.2. File Types in NCS Format

Type	Description
`gbx_ue_data_table`	Gearbox UE data tables with item definitions
`gbxactor`	Actor definitions with loot references
`itempool`	Item pool definitions (what can drop)
`ItemPoolList`	Item pool list configurations
`loot_config`	Loot configuration with drop rates
`Mission + rewards`	Mission reward definitions
`vending_machine`	Vending machine inventory

12.17.3. gBx Header Format

NCS files use the “gBx” magic header:

```
Offset  Size  Description
------  ----  -----------
0x00    3     Magic bytes: "gBx" (0x67 0x42 0x78)
0x03    1     Variant byte: '9', '6', 'r', 0xEF, or 0xE0
0x04    ?     Oodle-compressed payload
```

Variant bytes observed: - `0x39` (‘9’) - Most common - `0x36` (‘6’) - `0x72` (‘r’) - `0xEF` - `0xE0`

Example locations in pakchunk0-Windows_0_P.pak: - Offset 18327014: `67 42 78 39...` (gBx9) - Offset 60169283: `67 42 78 ef...` - Offset 78874017: `67 42 78 72...` (gBxr)

12.17.4. Compression

NCS uses **Oodle** compression (version 9): - DLL: `oo2core_9_win64.dll` - Primary function: `OodleLZ_Decompress` - Additional: `OodleLZ_Compress`, `Oodle_GetConfigValues`

12.17.5. Decompressed Content Format

After decompression, NCS content uses a typed hierarchical format with `field|type` notation:

12.17.5.1. Type Notation

Suffix	Meaning	Example
`|map`	Nested map/object	`children|map`
`|leaf:`	String leaf value	`tags|leaf:`
`|leaf:typename`	Typed leaf	`damagesource|leaf:damagesource`
`|empty`	Boolean/empty flag	`newobjective|empty`

12.17.5.2. Known Field Names (71 total)

Top-level Structure: - `gbx_sections|map` - Gearbox sections mapping - `children|map` - Nested child objects - `dependencies|map` - Asset dependencies - `generateddependencies|map` - Generated dependencies - `sections|map` - General sections - `configs|map` - Configuration data - `attributes|map` - Attribute data

Damage System: - `damagesource|leaf:damagesource` - Damage source reference - `damagesource|map` - Damage

source mapping - `damagetags|leaf:` - Damage tags - `hitdamagesource|leaf:damagesource` - Hit damage source - `damagesourceoverride|leaf:damagesource` - Override - `reflecteddamagetags|leaf:` - Reflected damage tags - `reflectedprojectiledamagetags|leaf:` - Projectile reflection - `maxchargedamagetags|leaf:` - Max charge damage - `overheatdamagetags|leaf:` - Overheat damage - `repairkitdamagetags|leaf:` - Repair kit damage - `lessthanthresholddamagetags|leaf:` - Threshold damage - `playerdamagetags|leaf:` - Player damage

Activity/Area System: - `activityareaactortags|leaf:` - Activity area actor tags - `activityareatags|leaf:` - Activity area tags - `areatags|leaf:` - Area tags - `requiredactivityareatags|leaf:` - Required activity area - `requiredactivitytags|leaf:` - Required activity - `excludedactivityareaactortags|leaf:` - Excluded activity area actors - `excludedactivitytags|leaf:` - Excluded activity

Tag System: - `tags|leaf:` / `tags|map` - Tag data - `excludetags|leaf:` / `excludetags|map` - Exclusion tags - `excludeaddtags|leaf:` - Add to exclusion - `require_tags|leaf:` - Required tags - `reject_tags|leaf:` - Rejected tags - `requiredbasetags|leaf:` - Required base tags - `requiredtags|leaf:` - Required tags - `excludedbasetags|leaf:` - Excluded base tags - `excludedplayerdamagetags|leaf:` - Excluded player damage

Weapon/Combat: - `weaponfire|map` - Weapon fire config - `effectparameters|map` - Effect parameters - `effectparametersbysurface|map` - Surface effects - `effectoverrides|map` - Effect overrides - `ampedtag|leaf:` - Amped tag - `tedioreemptytags|leaf:` - Tediore empty - `tediorehalffullormoretags|leaf:` - Tediore half+

Mission System: - `invisiblemissiontypes|leaf:` - Invisible missions - `poamissiontypes|leaf:` - POA missions - `primarytrackedmissiontypes|leaf:` - Primary tracked - `temporarytrackedmissiontypes|leaf:` - Temporary tracked - `worldeventmissiontypes|leaf:` - World events - `missionfailsafetimerpickupty` - Failsafe timer

UI/Display: - `display_items|map` - Display items - `pips|map` - Pip display - `wheelsetups|map` - Wheel UI setups - `loaded_input_actions|map` - Input actions - `mat26_augersight|map` - Auger sight material

Other: - `stats|leaf:` - Statistics data - `apply|leaf:` - Apply action - `remove|leaf:` - Remove action - `remove_states|leaf:` - Remove states - `criteria|map` - Criteria conditions - `entry_points|map` - Entry points - `exit_points|map` - Exit points - `exclusive|leaf:` - Exclusive flag - `alias_nodes|map` - Alias nodes - `factdependencies|map` - Fact dependencies - `data|leaf:damagesource` - Data reference - `newobjective|empty` - New objective flag - `momentsuegarden|empty` - Moments UE garden - `checkleavenoescaperoom01|empty` - Escape room check

12.17.6. Hash Function

Field names are hashed using **FNV-1a 64-bit**: - Offset basis: `0xcbf29ce484222325` - Prime: `0x100000001b3`

12.17.7. SerialIndex in NCS

The SerialIndex structure maps parts to their serialization indices:

```
Offset  Size  Description
------  ----  -----------
0x00    1     Category (weapon platform)
0x01    1     Scope/Status flags
0x02    2     Index (little-endian)
```

Important: NCS indices are runtime slot positions, NOT the serial encoding indices used in item serials. The manifest's alphabetical indices are used for serial encoding.

12.17.8. NCS Parser Tool

A community parser exists (Cr4nkSt4r's NcsParser.exe): - Loads Oodle DLL dynamically - Decodes function names from XOR-encoded strings (key: 0xA7) - Outputs JSON using nlohmann JSON v3.12.0 - Requires `oo2core_9_win64.dll` from game files

12.17.9. Implementation Notes

To parse NCS files in Rust: 1. Use `oodle-safe` crate with game's Oodle DLL 2. Read gBx header to get compressed/decompressed sizes 3. Call `OodleLZ_Decompress` on payload 4. Parse decompressed content using field|type notation 5. Use FNV-1a hash for field name lookups

---

Extracted from BL4 game files using retoc and uextract tools.

13. Glossary

Quick reference for terms used throughout this guide. Page references indicate the primary location where each term is explained.

13.1. 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.

13.2. B

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.

13.3. 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.

13.4. D

DataTable UE asset type for tabular data. Contains rows of structured data. See Chapter 6.

Dedicated Drop Loot that only drops from specific enemies. See Appendix C.

13.5. E

ECB Electronic Codebook. Block cipher mode where identical plaintext blocks produce identical ciphertext. See Chapter 4.

13.6. 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 Appendix D.

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.

13.7. G

gBx Magic header for NCS files (0x67 0x42 0x78). Followed by variant byte and Oodle-compressed payload. See Appendix D.

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.

13.8. 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.

13.9. 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 6.

Item Pool Definition of possible loot drops. See Appendix C.

Item Serial Base85-encoded string representing an item's full configuration. See Chapter 5.

13.10. L

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.

13.11. 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.

13.12. 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 Appendix D.

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.

13.13. 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.

13.14. P

Pak File Legacy UE archive format (.pak). See Chapter 6.

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.

13.15. R

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 6.

RIP-Relative x64 addressing mode relative to instruction pointer. See Chapter 3.

13.16. 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.

13.17. T

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.

13.18. U

uasset Unreal asset file containing object definitions and properties. See Chapter 6.

UClass UE's class definition object. Size: 512 bytes. See Chapter 2.

ucas IoStore container archive file containing compressed asset data. See Chapter 6.

uexp Bulk data file accompanying .uasset. See Chapter 6.

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 6.

13.19. 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.

13.20. W

WASM WebAssembly. Binary format for running code in browsers. See Chapter 7.

13.21. Z

Zen Package UE5's internal package format used in IoStore containers. See Chapter 6.

zlib Compression library. Used for save file compression after encryption. See Chapter 4.

13.22. Symbols

@Ug Prefix for all BL4 item serials. See Chapter 5.

0x Prefix indicating hexadecimal number. See Chapter 1.

13.23. Quick Reference: Playable Characters

Character Name	Internal Class Name	Class Mod Label
Amon	Char_Forgeknight	Forgeknight Class Mod
Rafa	Char_ExoSoldier	Exo-Soldier Class Mod
Harlowe	Char_Gravitar	Gravitar Class Mod
Vex	Char_Siren	Siren Class Mod

13.24. Quick Reference: Key Offsets

Symbol	Offset	Description
GUObjectArray	0x113878f0	All UObjects
GNames	0x112a1c80	FName pool
GWorld	0x11532cb8	Current world
ClassPrivate	+0x10	UObject’s class
NamePrivate	+0x18	UObject’s name
OuterPrivate	+0x20	UObject’s parent
SuperStruct	+0x40	UStruct’s parent

13.25. 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.