CyberApocalypse 2026 — GamePwn — The Salt Crown

> With the Quiet March broken, Aeron's path to the Salt Crown should be clear. Instead, Cassian waits at the Registry Altar, protected by Maelor's last covenant and the testimony of the living. Before the crown can be carried into open sky, one final claim must be answered — and the dead do not surrender their laws easily.

Category: GamePwn · **Flag:** `HTB{wh03v3r_h0lds_th3_cr0wn_s3ts_th3_rul3s_f0r_th1s_w0rld}`

Files

File	Description
`The Salt Crown.exe` (70 MB)	Godot Engine **4.7** (custom build) game, with an **encrypted** game archive appended as a PE overlay.
`challenge_core.windows.template_release.x86_64.dll` (208 KB)	A **packed / obfuscated** GDExtension (`ChallengeCore`) that holds all the real challenge logic and the flag.

The whole challenge is layered defence-in-depth: a custom encrypted PCK, Godot's own per-file encryption, a custom PE packer around the extension, runtime string obfuscation, anti-debug, and finally an AES-GCM–encrypted flag whose key is derived from the *solved* game state.

---

1. Recon

`strings` on the exe shows `Godot Engine v4.7.stable.custom_build`. A Godot game normally ships its assets in a `.pck` that is either separate or appended to the executable with a trailing `GDPC` magic. Here there is **no** `GDPC` anywhere and the file ends with a custom marker:

```
... 0c8f0d00 00000000 53435831   ->  "SCX1"
```

The PE overlay (everything after the last section, offset `0x423f000`, ~888 KB) is high-entropy and terminates in a 24-byte footer:

```
[ payload_size : u64 = 0xD8F18 ][ nonce : 12 bytes ][ magic "SCX1" : 4 bytes ]
```

So the author replaced Godot's standard `PackedSourcePCK` loader with a custom **"SCX1"** container that is decrypted at startup into a normal PCK in memory.

2. Breaking the SCX1 container (PCK layer 1)

Disassembling the exe around the `"SCX1"` comparison (`0x142a1c132`) leads to the loader. It references a 32-byte constant at `.data:0x143ecc8f0` :

```
31 f5 b2 8e c3 9b dc 74 a0 98 8c 2c 45 f4 95 42
47 3c 69 d6 c5 01 e5 a1 70 90 03 c1 78 22 07 48
```

A neighbouring routine (`fcn.140aa1130`) is a textbook **SHA-256** (initial constants `0x6a09e667 …`), used as a key-derivation step. The loader also stores that same 32-byte constant into a `FileAccessEncrypted` key buffer (`core/io/file_access_pack.cpp` strings confirm it) — i.e. it is Godot's *script encryption key*.

Trying AES with `SHA256(const)` as the key and the footer nonce as IV yields the plaintext:

```
GDPC 04 00 00 00 04 00 00 00 07 00 00 00 …   ->  Godot 4.7 PCK
```

SCX1 = AES-256-CTR, `key = SHA256(const32)`, `IV = footer_nonce(12) || 00 00 00 00`.

```
import struct, hashlib
from Crypto.Cipher import AES
from Crypto.Util import Counter
exe = open("The Salt Crown.exe","rb").read()
const = bytes.fromhex("31f5b28ec39bdc74a0988c2c45f49542473c69d6c501e5a1709003c1
key   = hashlib.sha256(const).digest()
ov    = 0x423f000                    # overlay start (end of PE sections)
payload, footer = exe[ov:ov+0xD8F18], exe[ov+0xD8F18:]
iv    = footer[8:20] + b"\x00\x00\x00\x00"
ctr   = Counter.new(128, initial_value=int.from_bytes(iv,"big"))
pck   = AES.new(key, AES.MODE_CTR, counter=ctr).decrypt(payload)   # -> valid G
```

3. The PCK (layer 2 — Godot FileAccessEncrypted)

The decrypted PCK has `pack_flags = 3` = `PACK_DIR_ENCRYPTED | PACK_REL_FILEBASE`. Layout of this custom format-4 header:

```
+0x00 "GDPC"      +0x04 fmt=4   +0x08 ver 4.7.0
+0x14 flags=3     +0x18 file_base = 112
+0x20 dir_offset = 0xD7240      (the directory sits near the end of the file)
```

At `dir_offset` there is `file_count (u32 = 81)` followed by a `FileAccessEncrypted` blob holding the directory (81 entries of `path_len, path, offset:u64, size:u64, md5:16, flags:u32`). Each individual file is *also* a `FileAccessEncrypted` blob.

Godot's `FileAccessEncrypted` here is **AES-256-CFB** (128-bit segments), keyed with the **raw** `const32` (not the SHA-256). Its on-disk layout is:

```
[ md5(plaintext) : 16 ][ plaintext_len : u64 ][ IV : 16 ][ AES-256-CFB( data )
```

Note: the directory `size` field is the **plaintext** length; the on-disk encrypted blob is `40 + plaintext_len` bytes (the 40-byte FAE header). Every extracted file's MD5 verifies.

Extracting all 81 files reveals the game:

```
scripts/game/challenge_state.gdc      scenes/actors/cassian.tscn.remap
scripts/actors/cassian.gdc            scenes/levels/stormbound_game.tscn.remap
scripts/levels/stormbound_game.gdc    bin/challenge_core.gdextension
scripts/actors/aeron.gdc              .godot/exported/.../*-cassian.scn ...
```

4. Reading the GDScript (layer 3 — compiled bytecode)

`.gdc` files are compiled GDScript: `"GDSC" + version(101) + decompressed_size(u32) + zstd_frame`. Decompress with zstd, then parse the token buffer. The identifier table stores names as UTF-32 XOR-`0xB6`.

The scripts show the game is a **thin client**: `ChallengeState` (RefCounted) wraps the native `ChallengeCore` (`_core`) and only relays calls:

```
_core.reset_session()          _core.submit_event({op, arg0, arg1, …})
_core.get_public_state()        _core.render_reward_step(image)   # draws the r
```

`stormbound_game.gd` is the playable level: platformer, rooms, and a boss **Cassian** who is `covenant_protected` with `supernatural_armor`. Public-state fields expose the "legal" puzzle:

```
shared_breath_covenant { current_epoch_invalid, voluntary_quorum_lost,
                         alternative_acclamation_present, satisfied_condition_c
claim_of_maelor_severed   registry_answers_cassian { lineage, succession }
claimant_hp / max_hp       supernatural_armor        witness_count
```

```
memory_erasure_count       severance_attempt_count   death_disputed
gate_passage_unlocked      finale_reached            reward_status
```

To win you must **answer Cassian's claim in the Registry**: sever Maelor's *shared-breath covenant* (invalidate the epoch, lose the voluntary quorum, present an alternative acclamation), dispute the lineage/succession, erase the living witnesses' memories, strip the supernatural armor, and only then deplete the claimant — after which `finale_reached` becomes true and `render_reward_step` paints the flag into `reward_image`.

Crucially, the client cannot fake this: the native `ChallengeCore` keeps the authoritative, integrity-checked session; `reward_status` only becomes `REWARD_DRAWING` / `DONE` when the DLL itself is satisfied.

5. Unpacking the GDExtension DLL

The DLL has three sections named `.slt0 / .slt1 / .slt2` (the *Salt* packer). `.slt0` has `SizeOfRawData = 0` (filled at runtime); the entry point (`DllMain`, RVA `0x72220`) is an **aPLib-style decompressor** that inflates `.slt1` into `.slt0`, resolves imports via `LoadLibraryA` / `GetProcAddress`, and `VirtualProtect`s the result.

Rather than reimplement the depacker, emulate `DllMain` with Unicorn (map at the preferred image base `0x180000000` so relocations are no-ops), stub the four imports, and stop when execution first enters `.slt0`; then dump the now-unpacked section:

```
# see work/full_map.py / emu_full.py — map .slt0/.slt1/.slt2, hook the 4 IAT th
# run from EP 0x180072220 until RIP enters [0x180001000,0x180040000), dump .slt
```

The unpacked `challenge_core_library_init` is a standard godot-cpp GDExtension initialiser; at level `SCENE` it registers the `ChallengeCore` class and its methods. Strings are built at runtime (stack/obfuscated). The imports it resolves confirm the crypto:

```
bcrypt.dll: BCryptOpenAlgorithmProvider, BCryptSetProperty, BCryptGenerateSymme
            BCryptDecrypt, BCryptCreateHash/HashData/FinishHash  (+ VirtualLock
```

6. The flag: AES-256-GCM gated by the solved state

`render_reward_step` (`fcn.180007370`) drives a reward-step routine (`fcn.180006ab0`) up to 1024 times to rasterise the reward onto the `Image` . That routine:

1. allocates a `VirtualLock` -protected scratch buffer,
2. builds two 32-byte buffers and XORs them into the **AES key material** (`key = secure_buffer XOR pad`), passed through a BCrypt SHA-256,
3. AES-**256-GCM** decrypts the flag. Ciphertext, 12-byte nonce and 16-byte tag are constants near `.data:0x180041060` .

The decrypt path only runs when the DLL considers the challenge **solved** — the key material is tied to the winning game state, which is why the flag cannot simply be read out of the binary: you must satisfy the Registry's conditions (either by playing the game, or by driving the `submit_event` state machine `fcn.180013920` to the winning configuration and reproducing the key derivation).

Recovering the key and flag

Reversing the reward routine (`0x180006ab0`), the finale store (`0x180007430`) and the `submit_event` state machine (`0x180003b50`) gives the full derivation. **All constants must be read from the *decompressed* image** (`work/decompressed_full.bin` ; the on-disk `.slt1` is still aPLib-compressed) at `offset = VA − 0x180001000` .

Key material

```
K0   = XOR of six 32-byte constants @ VA 0x18004d110/130/150/170/190/1b0
     = 2c4e84f5 8c3ccf65 3bfe7606 ce11e599 0736e9f9 307da527 bc1774a4 1e374ae3
salt = 32 bytes @ 0x180041090            seed = 16 bytes @ 0x180041080 (f06e39d
```

The winning session — `submit_event` accumulates a 7-event transcript; the finale stores `session = SHA-256(transcript_preimage)` :

```
preimage = 5343545201000000 1901010001000000 2d020300000000 4303070000000000
           58040f0000000000 6e051f0000000000 81063f0000000000 b7077f0000000000
session  = SHA256(preimage) = 58c1fd69 5c617b29 9de3cf46 08fcc910 e581b033 4ff8
```

(The first 16 bytes of `session` also appear inside the GCM AAD — a self-consistency check the DLL performs; this is what ties the flag to actually solving the challenge.)

HKDF (HMAC-SHA256) — the session's first 32 bytes are folded into the IKM, so only the correct winning state yields the correct key:

```
PRK = HMAC_SHA256(salt, K0 || session[0:32])
key = HMAC_SHA256(PRK, b"stormbound/reward/v2\x00" + seed16 + b"\x01")
    = 5bc4a0b3 edffe8cd d55b47d6 f00d4455 473dd63f fd90f2dc 9b3e70ae 98ad61bf
```

Decryption — `render_reward_step` emits up to 1024 records. Record *i* is **AES-256-GCM**:

```
nonce_i      = 7563b7feefab26e0 || big-endian u32(i)
ct_i,  tag_i = 0x180041100 + i*0x30  (0x20 ciphertext + 0x10 tag)
aad          = the "SCAR" public-state descriptor @ 0x1800410b8
```

Emulating the reward routine in Unicorn (stubbing BCrypt/heap/anti-debug, injecting the winning `session`) decrypts and `decrypt_and_verify` **succeeds on all 1024 records**. Each plaintext is a 32-byte turtle-graphics record; opcode `0x39` calls the draw primitive (`0x1800070c0`), and the 532 plotted points rasterise the flag onto the reward `Image`:

```
HTB{wh03v3r_h0lds_th3_cr0wn_s3ts_th3_
rul3s_f0r_th1s_w0rld}
```

Flag

```
HTB{wh03v3r_h0lds_th3_cr0wn_s3ts_th3_rul3s_f0r_th1s_w0rld}
```

Intended solution — in-game walkthrough

The flag is only produced when the native `ChallengeCore` observes a *legitimate, correctly ordered* win. This is what a player does on a Windows machine to trigger it.

Controls: `A/D` (or stick) move · `Space/A` jump · `J/X` attack · `E` interact · `Esc` pause · `M` return to menu.

The core rule — why you can't just fight the boss. At the **Registry Altar**, Cassian is protected by two things, and the Registry *rejects every attack* until both are removed:

1. **Maelor's shared-breath covenant** → his divine armour (`supernatural_armor`).
2. **The testimony of the living** → `witness_count` — *"THE WITNESSED CANNOT DIE."*

Swinging early gives *"THE ORDEAL REJECTS THIS ATTEMPT — RETURNING TO THE LAST ENTRY"*; rushing the gate gives *"PASSAGE WITHOUT SEVERANCE. ACCESS NOT CERTIFIED."* (that direct gate is a decoy — `direct_gate_decoy`). `ChallengeCore` records a strict ordered transcript, so any shortcut is rolled back to the last valid entry.

Rooms (in order): The Salt-Flat Shore → The Broken Causeway → Crownspire Lower Wall → The Claimant's Forecourt → The Registry Altar → The Constraint Record.

Steps

1. **Fight to the Registry.** Clear the Vaultrune **warrant-men** (clerk / spear / sword / shield enemies) blocking each room — *"Clear his warrant-men."* / *"VAULTRUNE GUARDS STILL HOLD THE ROAD."*

2. **Open the three "old channels"** (`E` to interact) to satisfy the covenant-severing conditions — you need **COVENANT CONDITIONS: 3/3**:

Channel	Transition	Condition satisfied
`storm_circuit`	`bound_to_gate → free`	**EPOCH INVALID**
`tide_sluice`	`closed → open`	**QUORUM LOST**

Channel	Transition	Condition satisfied
`brine`	`sealed → flooded`	**ALTERNATIVE ACCLAMATION**

"Open the old channels when the gate begins to lie… the altar severs only when the current epoch fails, voluntary quorum is lost, and another acclamation stands present."

1. **Erase the living witnesses.** Consume the witness memories (the "Uncrowned One" feeds on them) to drive `witness_count → 0` — *"MEMORY CONSUMED: …"*. The memories are Cassian's identity: **MOTHERS_FACE** ("the face that first named you"), **THE_BROTHEL_ROOM**, **FIRST_KINDNESS**, **BIRTH_NAME**, **REASON_FOR_POWER**.

2. **Sever the claim.** With 3/3 conditions met *and* the witnesses gone, trigger `SEVER(CASSIAN, CLAIM_OF_MAELOR)` → *"DIVINE ARMOUR FRACTURED"* → *"CLAIM SEVERED"* → *"UNCROWNED COVENANT: SEVERED."* `supernatural_armor` is now off and damage is accepted.

3. **Defeat Cassian.** Attack to deplete his HP (**THE BLACK HEIR %04d/%04d**), then obtain the **DEATH DISPUTED** recognition — *"A death without recognition is merely an accusation."*

4. **Leave through the now-unlocked gate** (`gate_passage_unlocked`, `finale_reached`). The game renders **THE CONSTRAINT RECORD** — the reward image containing the flag.

Why it yields the flag. Performing the above legitimately and in order makes `ChallengeCore` build the correct 7-step transcript, hash it into the session, HKDF-derive the AES-256-GCM key, and decrypt+paint the reward. Tamper or skip a step and the derived key is wrong, giving *"THE CONSTRAINT RECORD FAILED AUTHENTICATION"* (`REWARD_AUTH_FAILED`) instead of the flag. This is the same result reached offline in §6 — a mutual cross-check.

Solution summary

1. Carve the PE overlay; recognise the custom **SCX1** footer.

2. Decrypt it: **AES-256-CTR**, key `SHA256(const32@0x143ecc8f0)`, IV `nonce||0000` → real PCK.
3. Parse the format-4 PCK; decrypt the directory + each file with Godot **FileAccessEncrypted** (**AES-256-CFB**, raw `const32`).
4. Decompress the `.gdc` (zstd) → understand the thin-client architecture and the win condition.
5. Emulate the **aPLib** `DllMain` to unpack the `ChallengeCore` GDExtension.
6. Locate the **AES-256-GCM** flag decryptor and recover the key from the solved state.

Artifacts (`./work/`)

- `decrypted.pck` — the recovered Godot PCK
- `pck_extracted/` — all 81 decrypted game files (`*.gdc.dec` = decompressed bytecode)
- `challenge_core_unpacked.dll`, `slt0.bin` — the unpacked GDExtension
- `emu_full.py`, `full_map.py`, `name2global.json` — Unicorn harnesses