

# SHOGUN-SYNC

Total War: Shogun 2 Multiplayer Synchronization Tool

---

## Technical Specification & Architecture Reference

Internal Design, Cryptographic Verification, IPC & In-Game Hot-Reload

### Disclaimer / Legal Notice

**Notice:** Shogun-Sync is an unofficial, community-developed open-source modification and synchronization utility. This project is **not** created by, affiliated with, endorsed by, sponsored by, or connected in any way to *The Creative Assembly*, *SEGA*, or any of their parent companies, subsidiaries, or affiliates. All trademarks, registered marks, character names, and game titles belong to their respective copyright holders and are referenced here strictly for compatibility, interoperability, and identification purposes.

**Author:** KW

**Release:** v0.1.1-beta

**Core Modules:** Dear ImGui, DirectX 11/9, MinHook, CAS Storage, Zlib, Mongoose WS

## Contents

---

|          |                                                                                        |           |
|----------|----------------------------------------------------------------------------------------|-----------|
| <b>1</b> | <b>System Overview &amp; Architecture</b>                                              | <b>2</b>  |
| 1.1      | Technological Stack & Core Libraries . . . . .                                         | 3         |
| <b>2</b> | <b>Cryptographic Engine: Standalone SHA-1</b>                                          | <b>3</b>  |
| 2.1      | Message Preprocessing & Memory Padding . . . . .                                       | 3         |
| 2.2      | Message Schedule & 80-Word Expansion . . . . .                                         | 4         |
| 2.3      | Rounds & State Transformations . . . . .                                               | 4         |
| 2.4      | C API Reference: <code>kw_sha1.h</code> / <code>kw_sha1.c</code> . . . . .             | 6         |
| <b>3</b> | <b>Storage Engine: Content-Addressable Storage (CAS)</b>                               | <b>6</b>  |
| 3.1      | Object Types & Disk Hierarchy . . . . .                                                | 6         |
| 3.2      | C API Reference: <code>kw_conf.h</code> / <code>kw_conf.c</code> . . . . .             | 7         |
| <b>4</b> | <b>Compression Subsystem &amp; The Save Pre-Compression Paradox</b>                    | <b>7</b>  |
| 4.1      | Empirical Finding: The Save Pre-Compression Paradox . . . . .                          | 7         |
| 4.2      | C API Reference: <code>kw_compress.h</code> / <code>kw_compress.c</code> . . . . .     | 8         |
| <b>5</b> | <b>Filesystem Watcher &amp; Turn Synchronization</b>                                   | <b>8</b>  |
| 5.1      | Engine Reverse Engineering & Runtime Memory Analysis . . . . .                         | 8         |
| 5.1.1    | Process Topology & Execution Environment . . . . .                                     | 8         |
| 5.1.2    | Call Stack Tracing of the Serialization Pipeline . . . . .                             | 9         |
| 5.1.3    | Memory Watchpoint Analysis & Path Traversal Loop . . . . .                             | 9         |
| 5.1.4    | Architectural Decisions Derived from Memory Findings . . . . .                         | 9         |
| 5.2      | Kernel Notification Architecture . . . . .                                             | 10        |
| <b>6</b> | <b>Networking &amp; Relay Protocol Specification</b>                                   | <b>11</b> |
| 6.1      | Framing Specification . . . . .                                                        | 11        |
| 6.2      | Path Traversal Defense . . . . .                                                       | 11        |
| 6.3      | C API Reference: <code>kw_conn.h</code> / <code>kw_conn.c</code> . . . . .             | 12        |
| <b>7</b> | <b>In-Game Overlay &amp; Direct3D Hooking Subsystem</b>                                | <b>12</b> |
| 7.1      | Process Enumeration & DLL Injection . . . . .                                          | 13        |
| 7.2      | Direct3D 9 & 11 Virtual Method Table (VMT) Detours . . . . .                           | 13        |
| 7.3      | Inter-Process Communication (IPC) . . . . .                                            | 13        |
| 7.4      | In-Game Save Reload Injection Subsystem . . . . .                                      | 14        |
| 7.5      | Immediate-Mode HUD Rendering . . . . .                                                 | 14        |
| <b>8</b> | <b>Interface &amp; Operations Reference</b>                                            | <b>15</b> |
| 8.1      | Command-Line Interface (CLI) . . . . .                                                 | 15        |
| 8.2      | Graphical User Interface (GUI) . . . . .                                               | 15        |
| 8.3      | Dedicated Installer & System Deployment ( <code>ShogunSyncSetup.exe</code> ) . . . . . | 16        |
| 8.4      | Diagnostic Error Codes . . . . .                                                       | 16        |
| <b>9</b> | <b>References &amp; Further Reading</b>                                                | <b>16</b> |

# 1 System Overview & Architecture

Shogun-Sync is an unofficial synchronization utility and desynchronization recovery tool designed for *Total War: Shogun 2* multiplayer campaign sessions. It monitors the active campaign save directory in real time, computes deterministic cryptographic checksums across save files, negotiates state consistency over an encrypted WebSocket channel, and restores authoritative save state whenever simulation divergence occurs.

The system is written in C99 and C++17, compiled for the 32-bit (x86) Windows environment to match the architecture of the game process (**Shogun2.exe**). The project provides four deployment targets:

- **shogun-sync.exe**: The core console daemon managing the Win32 filesystem watcher, CAS repository, SHA-1 checksum verification, WebSocket networking, and the embedded Dear ImGui GUI when invoked with `-gui`.
- **ShogunSync.exe**: A dedicated Win32 launcher that starts the graphical interface in background mode (`CREATE_NO_WINDOW` and `SW_HIDE`), preventing unwanted command prompt windows when launched from desktop shortcuts.
- **ShogunSyncSetup.exe**: A standalone Win32 installation wizard that extracts the executables and overlay DLL from internal PE resource sections (RCDATA 101, 102, 103), creates Start Menu and Desktop shortcuts, registers the uninstaller, and configures the user PATH environment variable.
- **shogun\_overlay32.dll**: An in-process dynamic library injected into **Shogun2.exe** via remote thread execution. It detours Direct3D 9 and 11 presentation calls to display synchronization telemetry directly on screen and listens for save reload triggers.

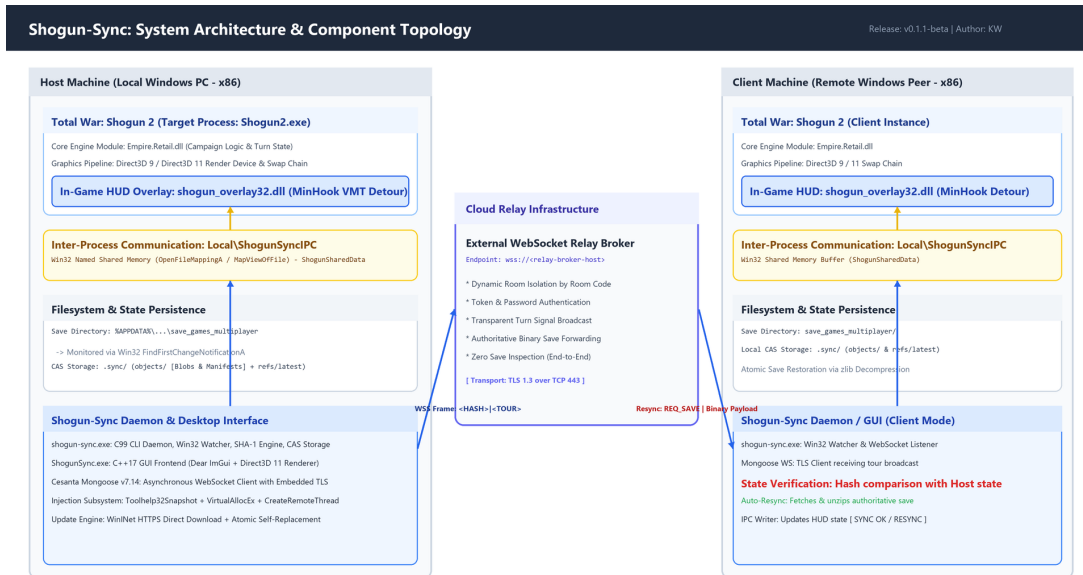


Figure 1: System topology diagram illustrating the relationship between the game processes, the Win32 filesystem watcher, the local CAS repository, the in-game overlay, and the external relay broker.

## 1.1 Technological Stack & Core Libraries

The application is implemented without external runtime environments (such as .NET, Java, or Electron). It relies on standard Win32 APIs and compact libraries:

- **Dear ImGui (v1.91.x):** Immediate-mode graphical user interface library powering both the desktop management window and the transparent in-game HUD overlay.
- **DirectX 11 & DirectX 9 Graphics Backends:** Leverages Direct3D 11 for the desktop interface and hooks into the game's active Direct3D swap chain (supporting both DirectX 9 and 11 modes).
- **MinHook (v1.3.3):** Minimalistic x86 API detours library used for Virtual Method Table (VMT) hooking of `Present` and buffer resize routines.
- **Content-Addressable Storage (CAS):** Immutable on-disk versioning model directly inspired by Git's internal architecture, as formalized in Thibault Polge's *Write yourself a Git!* (WYAG)<sup>1</sup>.
- **Standalone Cryptographic Engine:** RFC-compliant SHA-1 implementation operating without external cryptographic provider dependencies.
- **Cesanta Mongoose (v7.14):** Asynchronous, event-driven networking library managing TLS-encrypted WebSockets (`wss://`) communicating with an external relay broker.
- **zlib (v1.3.2):** Streaming DEFLATE compression engine utilized for CAS repository object serialization and network payload framing.
- **Win32 Platform Integration:** Kernel notifications (`FindFirstChangeNotificationA`), named shared memory IPC (`Local\ShogunSyncIPC`), remote thread DLL injection, and WinINet-based atomic binary updates.

## 2 Cryptographic Engine: Standalone SHA-1

Consistency verification depends on deterministic content hashing. Shogun-Sync implements the Secure Hash Algorithm 1 (SHA-1) specified in FIPS PUB 180-1.

### 2.1 Message Preprocessing & Memory Padding

A save file is loaded into memory as a continuous byte buffer of length  $ML$  bytes. The bit-length of the raw message is:

$$L = ML \times 8 \tag{1}$$

Padding guarantees that the total message length is congruent to 448 (mod 512) bits (or 56 (mod 64) bytes), leaving exactly 64 bits (8 bytes) at the end to encode the message length  $L$  in big-endian representation.

---

<sup>1</sup><https://wyag.thb.lt/#intro>

**Padding Length Calculation Logic**

Let  $\text{diff} = (ML + 1) \bmod 64$ .

- If  $\text{diff} \leq 56$ , the required zero-padding length  $P$  is:

$$P = 56 - \text{diff} \quad (2)$$

- If  $\text{diff} > 56$ , the padding must span into the subsequent 64-byte block:

$$P = 64 - \text{diff} + 56 \quad (3)$$

The total allocated input buffer size equals:

$$M_{\text{alloc}} = ML + 1 + P + 8 \quad (4)$$

The byte immediately following the message payload ( $ML$ ) is assigned the separator value  $0x80$  ( $10000000_2$ ). Subsequent  $P$  bytes are zero-filled ( $0x00$ ), and the final 8 bytes store  $L$  in network byte order (big-endian).

## 2.2 Message Schedule & 80-Word Expansion

The padded stream is partitioned into  $N = M_{\text{alloc}}/64$  chunks of 64 bytes (512 bits). For each chunk, the first 16 words ( $W_0 \dots W_{15}$ ) are constructed from 4 consecutive big-endian bytes:

$$W_j = (B_{4j} \ll 24) \mid (B_{4j+1} \ll 16) \mid (B_{4j+2} \ll 8) \mid B_{4j+3} \quad (5)$$

For indices  $t \in [16, 79]$ , words are expanded using bitwise XOR and circular left-shifts ( $\lll$ ):

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1 \quad (6)$$

Where 32-bit left rotation by  $s$  bits is defined as:

$$\text{leftRotate}(x, s) = (x \ll s) \mid (x \gg (32 - s)) \quad (7)$$

## 2.3 Rounds & State Transformations

The compression function initializes five 32-bit state variables:

$$\begin{aligned} H_0 &= 0x67452301, & H_1 &= 0xEFCDAB89, & H_2 &= 0x98BADCFE, \\ H_3 &= 0x10325476, & H_4 &= 0xC3D2E1F0 \end{aligned}$$

Eighty rounds are executed across four distinct stages with designated non-linear functions  $f_t$  and additive constants  $K_t$ :

| Round | Interval ( $t$ )    | Nonlinear Function $f_t(B, C, D)$                  | Constant $K_t$ |
|-------|---------------------|----------------------------------------------------|----------------|
| 1     | $0 \leq t \leq 19$  | $(B \wedge C) \vee (\neg B \wedge D)$              | $0x5A827999$   |
| 2     | $20 \leq t \leq 39$ | $B \oplus C \oplus D$                              | $0x6ED9EBA1$   |
| 3     | $40 \leq t \leq 59$ | $(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ | $0x8F1BBCDC$   |
| 4     | $60 \leq t \leq 79$ | $B \oplus C \oplus D$                              | $0xCA62C1D6$   |

Inside each transformation round:

$$\begin{aligned} \text{temp} &= (A \lll 5) + f_t(B, C, D) + E + K_t + W_t \pmod{2^{32}} \\ E &= D, \quad D = C, \quad C = B \lll 30, \quad B = A, \quad A = \text{temp} \end{aligned}$$

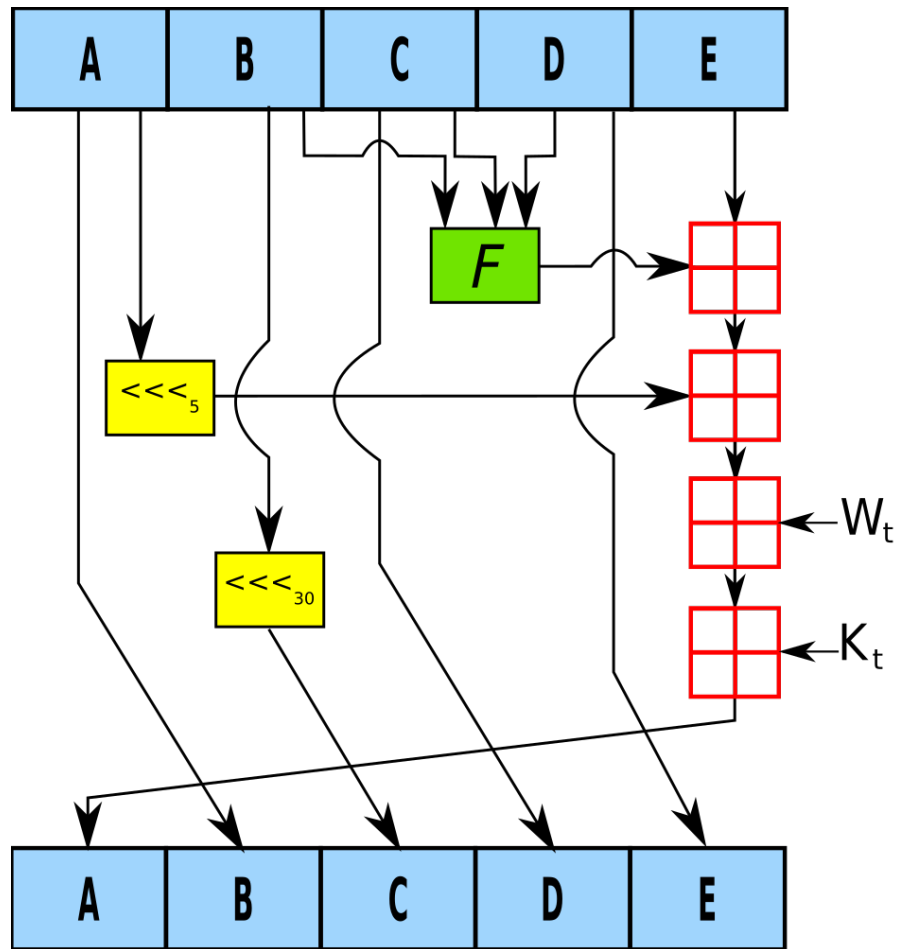


Figure 2: SHA-1 compression cycle per round showing the interaction between registers  $A, B, C, D, E$ , nonlinear function  $f_t$ , constant  $K_t$ , and expanded word  $W_t$ . Source: Wikipedia (<https://en.wikipedia.org/wiki/SHA-1>).

## 2.4 C API Reference: kw\_sha1.h / kw\_sha1.c

- `void sha1(const uint8_t *message, size_t ML, uint32_t result[5])`  
Computes the 160-bit SHA-1 digest over an arbitrary buffer of ML bytes. Populates `result` with five 32-bit state words. Emits `Error 017` if heap allocation fails.
- `void resultToChar(uint32_t result[], char* resultChar, size_t ML)`  
Formats the five 32-bit state words into a 40-character hexadecimal null-terminated string.
- `void printS(uint32_t result[])`  
Prints the formatted 40-character hexadecimal digest to standard output for debugging.
- `typedef struct { uint32_t word[80]; } package;`  
Represents the internal 80-word schedule buffer used per 512-bit compression chunk.

## 3 Storage Engine: Content-Addressable Storage (CAS)

Shogun-Sync implements an on-disk version control hierarchy stored under:

`%APPDATA%/TheCreativeAssembly/Shogun2/.sync/`

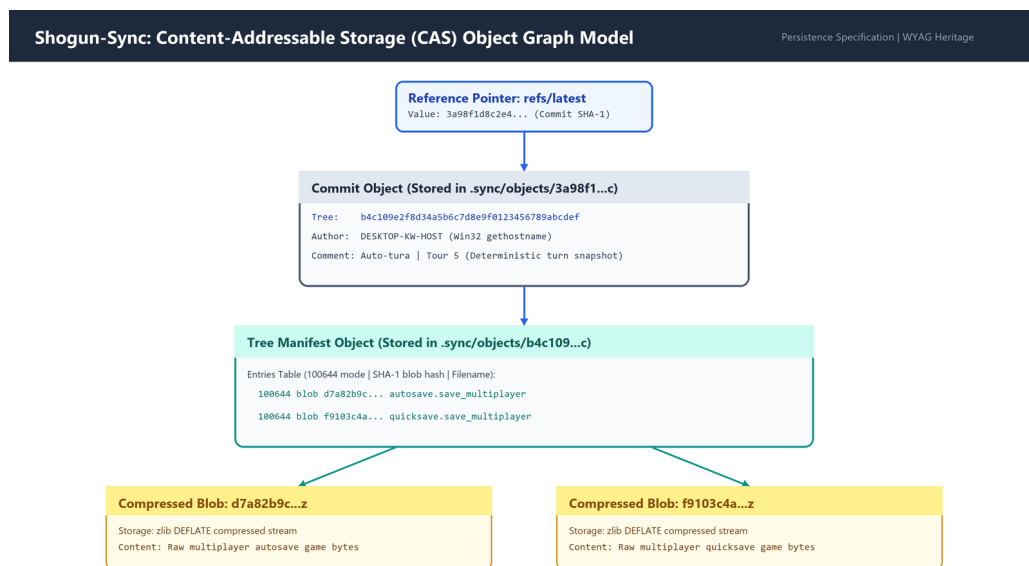


Figure 3: Object Graph of the Content-Addressable Storage engine showing the Commit Object pointing to a Tree Manifest, which references compressed Blobs.

### 3.1 Object Types & Disk Hierarchy

- **Blobs (`.sync/objects/<hash>.z`):** Compressed raw save file contents. The object hash is computed over uncompressed file bytes.
- **Trees (`.sync/objects/<hash>.m`):** Compressed manifests listing tracked saves. Entries are stored formatted as:

```
blob <40-char-hash> <save_name.save_multiplayer> \n
```

- **Commits (`.sync/objects/<hash>.c`):** Atomic state manifests recording tree hash, author, and description:

```
Tree <tree-hash> Author <hostname> Comment <comment-string>\n
```

- **References (.sync/refs/latest):** Text reference file storing the 40-character hash of the current active commit object.

### 3.2 C API Reference: kw\_conf.h / kw\_conf.c

- **bool init()**  
Creates the local repository directory structure (.sync, objects, refs). Applies the Win32 FILE\_ATTRIBUTE\_HIDDEN attribute to .sync. Returns 0 on success, 1 on failure (Error 007).
- **bool getVariables()**  
Resolves the Windows profile paths via %APPDATA% and %LOCALAPPDATA%. Initializes Winsock2 (WSAStartup) and resolves the local machine hostname. Returns 0 on success, 1 if paths are missing (Error 005, Error 006).
- **bool scan(mapTree \*\*mt, int \*x)**  
Scans the multiplayer save directory using native Win32 APIs (FindFirstFileA, FindNextFileA) matching \*.save\_multiplayer. Two-pass design: when \*x == 1, counts discovered saves; when \*x == 0, allocates and populates the mapTree array. Returns 0 on success, 1 on scan error (Error 008).
- **bool add(char hashTree[41])**  
Iterates through discovered saves, computes SHA-1 checksums, checks existence in .sync/objects/, compresses missing blobs via zlib into <hash>.z, constructs the tree manifest buffer, compresses it into <hash>.m, and returns the manifest hash in hashTree. Employs a 5-step retry loop with 100 ms sleep delays to handle transient game write locks.
- **bool commit(char comment[])**  
Orchestrates commit creation: calls add(), constructs the commit manifest string, hashes and compresses it into <hash>.c, and updates refs/latest. Returns 0 on success, 1 on error (Error 018).
- **void test(char saveName[])**  
Testing routine. Loads a target save file, computes its SHA-1 digest, and prints the result.

## 4 Compression Subsystem & The Save Pre-Compression Paradox

Shogun-Sync embeds **zlib (v1.3.2)** to compress repository objects and serialize network transfer payloads.

### 4.1 Empirical Finding: The Save Pre-Compression Paradox

During implementation and benchmarking, an interesting empirical property of *Total War: Shogun 2* saves was identified:

- A standard save file (autosave.save\_multiplayer) occupies **2,188,140 bytes** on disk.
- Compressing this file via zlib deflate yields **2,186,557 bytes**.
- Total storage reduction is **0.07%** (less than 2 KiB), while consuming noticeable CPU cycles.

The Creative Assembly's Warscape engine already compresses save game streams internally before writing them to disk. Attempting to compress already compressed binary data produces virtually zero gain.

**Architectural Decision:** Zlib compression is retained across the project because:

1. Text manifests (.m and .c files) achieve over 75% compression ratio, reducing repository disk footprint.
2. It provides a uniform container format for all repository objects (.z, .m, .c).
3. It standardizes binary network framing during Auto-Resync save transfers.

## 4.2 C API Reference: kw\_compress.h / kw\_compress.c

- `void kw_compress(uint8_t *buff, size_t ML, char fileName[])`  
Compresses an in-memory buffer of size ML using `deflateInit` with `Z_DEFAULT_COMPRESSION`. Streams output into `fileName` in 16 KiB chunks (`CHUNK = 16384`). Handles cleanup via `deflateEnd`.
- `bool kw_decompress(uint8_t *buff, size_t ML, char fileName[])`  
Decompresses a zlib stream buffer of size ML and writes restored data to `fileName` using `inflateInit` and `inflate`. Returns 0 on success, 1 on stream error (`Error 010`, `Error 011`).
- `bool kw_decompress_file(char inFileName[], char outFileName[])`  
Loads a compressed file from disk into RAM and passes the buffer to `kw_decompress`.

## 5 Filesystem Watcher & Turn Synchronization

Multiplayer campaign progression is turn-based. At the conclusion of each turn, the game engine executes a state serialization sequence and updates the `autosave.save_multiplayer` file.

### 5.1 Engine Reverse Engineering & Runtime Memory Analysis

To determine the precise mechanism by which game state is serialized and synchronized across multiplayer peers, the target process was subjected to dynamic memory analysis, thread call stack tracing, and instruction disassembly. This low-level investigation directly dictated the non-invasive architectural design of Shogun-Sync.

#### 5.1.1 Process Topology & Execution Environment

Runtime process inspection revealed that `Shogun2.exe` operates as a 32-bit (x86) bootstrap launcher. The entire campaign simulation engine, state machine, battle mathematics, and persistence subsystem are encapsulated within the core dynamic library `Empire.Retail.dll`.

Register analysis during turn transitions confirmed a standard 32-bit x86 execution state:

- **General Purpose Registers:** 32-bit wide (EAX, EBX, ECX, EDX, ESI, EDI).
- **Stack Pointers & Frame Bounds:** ESP and EBP managing local stack frames with 4-byte pointer alignment.
- **Calling Conventions:** Extensive use of `__thiscall` (passing the object pointer in ECX) and `__cdecl` for runtime helper functions.

This finding established the fundamental requirement that all external components interacting with the process space (including `shogun_overlay32.dll` and the remote thread injector) must be strictly compiled for the 32-bit x86 ABI.

### 5.1.2 Call Stack Tracing of the Serialization Pipeline

Setting execution breakpoints on file handle creation APIs during end-of-turn processing captured the active call stack hierarchy within `Empire.Retail.dll`:

```
Empire.Retail.dll+E6757 → Empire.Retail.dll+EE2CA → Empire.Retail.dll+12A9DDB →
Empire.Retail.dll+12A730B → Empire.Retail.dll+BD5E17 → Empire.Retail.dll+F228D
```

Disassembly of the stack frame setup at `Empire.Retail.dll+61D6F180` revealed the internal sequence responsible for save path formatting and file descriptor opening:

```
push    ebp
mov     ebp, esp
sub     esp, 1Ch
push    esi
mov     esi, ecx
push    dword ptr [ebp+0Ch]
lea     eax, [ebp-1Ch]          ; Stack buffer receiving wide string path
push    eax                    ; L"...\\save_games\\automatyczny zapis.save"
call    Empire.Retail.dll+61D6EEC3
mov     esi, eax
lea     edi, [ebp-10h]         ; Mode string buffer
; EDI receives descriptor: L"rbs" (Read Binary Stream)
push    100h
lea     eax, [ebp-04h]
push    dword ptr [ebp-10h]    ; Mode string L"rbs"
push    dword ptr [ebp+08h]    ; Save path buffer pointer
push    eax
call    Empire.Retail.dll+61D76A72 ; Dispatches to kernel32.dll!CreateFileW
```

The trace proved that save serialization is initiated synchronously by the campaign coordinator thread. The game engine constructs fully qualified Unicode (UTF-16LE, `wchar_t`) path strings on the stack pointing to `%APPDATA%/TheCreativeAssembly/Shogun2/save_games_multiplayer/`, requests exclusive binary file streams with mode descriptor `L"rbs"`, and flushes the entire serialized campaign state to disk before permitting turn handover.

### 5.1.3 Memory Watchpoint Analysis & Path Traversal Loop

Dynamic memory watchpoints placed on the heap buffer holding active save path strings (e.g., memory address `0x1782149E`) isolated the string evaluation routine located at `Empire.Retail.dll+F0177`:

```
66440172: je      Empire.Retail.dll+F018A
66440174: mov     cx, [eax]
66440177: cmp     cx, [edx]          ; EDX points to heap string (0x1782149E)
6644017A: jne     Empire.Retail.dll+F0186
6644017C: add     eax, 02h           ; Advance 16-bit wchar_t pointer
```

Register inspection at the moment of evaluation:

```
EAX=0x34CF5416, EDX=0x1782149E, EBX=0x6157DAC4, ESI=0x34CF5444, EDI=0x0000005E,
EBP=0x00000BF9, ESP=0x012FC264, EIP=0x6644017A
```

This confirmed that path resolution uses an iterative 16-bit character comparison loop traversing UTF-16 strings to perform path normalization, duplicate checks, and validation against existing save archives prior to writing.

### 5.1.4 Architectural Decisions Derived from Memory Findings

The empirical discoveries from runtime memory inspection directly established four implementation decisions in Shogun-Sync:

### 1. Kernel Watcher vs. Dynamic Memory Hooks:

Memory tracking revealed that game state pointers and save buffers are allocated on dynamic heap pages (e.g., 0x1782149E, 0x34CF5416, 0x7F56C994) whose addresses vary significantly across process launches, memory fragmentation, and game updates. Direct memory hooking of unexported internal engine pointers would introduce instability and crash vulnerabilities. Conversely, observing that the engine reliably flushes the complete serialized binary stream to disk led to implementing native kernel filesystem notifications via `FindFirstChangeNotificationA`, avoiding risks associated with reading volatile internal memory.

### 2. I/O Contention Mitigation (Retry Backoff):

The disassembly confirmed that `Empire.Retail.dll` maintains an active file handle during the multi-megabyte compression and disk writing phase. If an external synchronization utility attempts an immediate synchronous read upon the first notification event, the operation fails with a sharing violation (`ERROR_SHARING_VIOLATION`). To resolve this, `kw_conf.c` introduces a 500 ms stabilization sleep upon event trigger, combined with a 5-step retry loop (with 100 ms delays via `Sleep(100)`), ensuring that file reading only begins after the engine has completely closed its write stream.

### 3. Decoupled IPC Telemetry:

Rather than attempting to write telemetry or HUD state into engine memory structures, Shogun-Sync establishes an independent named shared memory mapping (`Local\ShogunSyncIPC`). The injected overlay (`shogun_overlay32.dll`) reads synchronization status exclusively from this decoupled Win32 buffer, isolating the game engine from external state mutations.

### 4. Save Reload Injection via Win32 Window Messages:

Rather than risking memory corruption by attempting to call unexported campaign loading routines inside `Empire.Retail.dll`, Shogun-Sync injects standard keyboard shortcut sequences (`VK_F9` and `Ctrl + L`) into the game's message queue via `PostMessageA`, instructing the engine to trigger its native quickload routine safely.

## 5.2 Kernel Notification Architecture

Directory monitoring is established using:

```
FindFirstChangeNotificationA(savePath, FALSE,
    FILE_NOTIFY_CHANGE_LAST_WRITE);
```

Inside `watch()`, the thread executes non-blocking event multiplexing:

1. Polls network sockets via `pollNetwork()` (50 ms timeout).
2. Checks directory state using `WaitForSingleObject(hd, 0)`.
3. Upon receiving `WAIT_OBJECT_0`, sleeps for 500 ms to guarantee complete file flushing by the game engine.
4. Executes `commit("Auto-tura")`.
5. Increments the turn counter `tour++`.
6. Dispatches turn verification messages over the WebSocket channel.

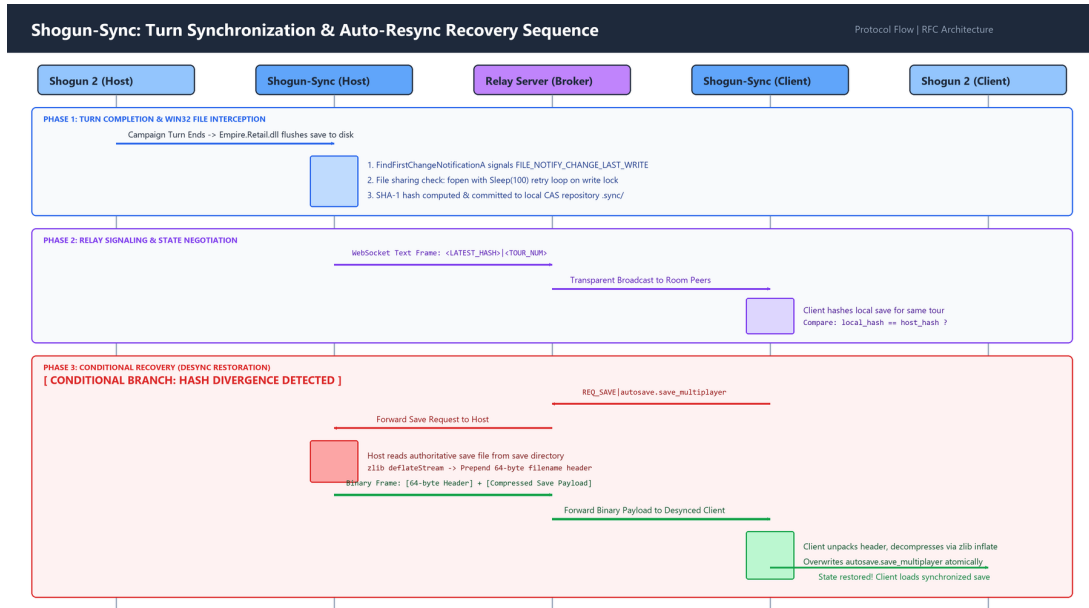


Figure 4: Turn synchronization sequence diagram illustrating normal hash validation, desynchronization detection, and binary save restoration.

## 6 Networking & Relay Protocol Specification

Shogun-Sync establishes encrypted transport sessions over an external, standalone WebSocket relay broker. The relay infrastructure operates as an independent network router: it manages peer room isolation, authentication validation, and packet forwarding without persisting, modifying, or inspecting save game payloads. The client connects to the secure endpoint:

```
wss://<relay-broker-host>/ws/<room>?password=<pass>&action=<action>
```

### 6.1 Framing Specification

| Frame Type   | Format / Header          | Description & Wire Payload                                                                                                                                                                                                                                                            |
|--------------|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Text Frame   | <HASH> <TOUR>            | Turn sync message. 40-character commit hash followed by pipe delimiter and turn number (e.g. d262c2... 5).                                                                                                                                                                            |
| Text Frame   | REQ_SAVE <filename>      | Resync request sent by Client upon hash mismatch. Identifies the save file to fetch from Host.                                                                                                                                                                                        |
| Binary Frame | 64-byte Header + Payload | Authoritative save payload sent by Host to Client: <ul style="list-style-type: none"> <li><b>Bytes 0–63:</b> Target save filename null-terminated ASCII string (max 63 chars + \0).</li> <li><b>Bytes 64+:</b> Zlib-compressed binary data of the authoritative save file.</li> </ul> |

### 6.2 Path Traversal Defense

To block arbitrary file exfiltration, `sendSaveToClient()` sanitizes filenames:

```
if (saveName == NULL || strstr(saveName, "..") != NULL ||
    strchr(saveName, '/') != NULL || strchr(saveName, '\\') != NULL) {
```

```

    printf("Error 016: Invalid packet received\n");
    return;
}

```

### 6.3 C API Reference: kw\_conn.h / kw\_conn.c

- `void connectToRoom(char roomName[], char password[], char action[])`  
Initializes Mongoose manager (`mg_mgr_init`), sets boot timestamp, builds endpoint URL, and connects via `mg_ws_connect`.
- `void pollNetwork()`  
Polls sockets via `mg_mgr_poll(&mg, 50)`.
- `void sendPackage(char msg[], size_t leng)`  
Sends text frame over active WebSocket handle `c`.
- `void sendSaveToClient(const char *saveName)`  
Reads save, compresses via zlib, prepends 64-byte filename header, and transmits binary frame.
- `void eventManager(struct mg_connection *conn, int ev, void *evData)`  
Handles `MG_EV_CONNECT` (TLS initialization), `MG_EV_WS_OPEN`, `MG_EV_WS_MSG` (text signaling and binary save inflation), and teardown events.

## 7 In-Game Overlay & Direct3D Hooking Subsystem

To provide continuous synchronization telemetry without requiring alt-tabbing or dual monitors, Shogun-Sync features an in-game graphical HUD injected directly into the running *Total War: Shogun 2* game process (`Shogun2.exe`).

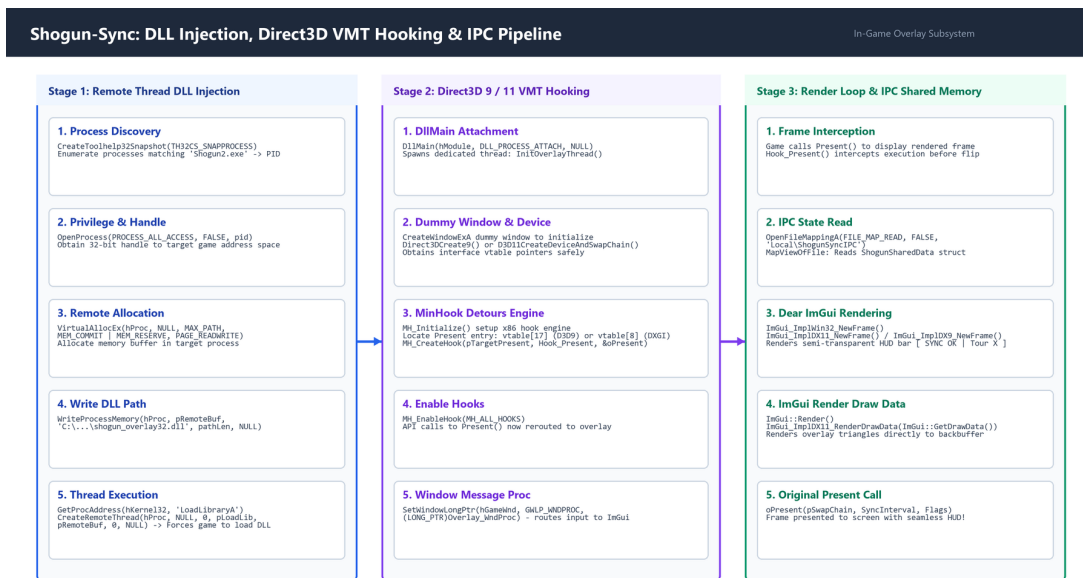


Figure 5: DLL Injection and DirectX Hooking architecture showing process memory allocation, remote thread execution of `LoadLibraryA`, Direct3D vtable detours via `MinHook`, and IPC shared memory exchange with the overlay HUD.

## 7.1 Process Enumeration & DLL Injection

The injection sequence operates entirely via native Win32 APIs:

1. **Process Enumeration:** Scans running processes via `CreateToolhelp32Snapshot` and `Process32Next` to locate the target PID of `Shogun2.exe`.
2. **Path Resolution:** Resolves the absolute path to `shogun_overlay32.dll` across the current working directory, executable folder, and `%LOCALAPPDATA%\ShogunSync`.
3. **Remote Allocation:** Opens the game process with `PROCESS_ALL_ACCESS` rights, allocates memory in the remote address space via `VirtualAllocEx`, and writes the full DLL path string using `WriteProcessMemory`.
4. **Thread Execution:** Resolves the 32-bit entry point of `LoadLibraryA` from `kernel32.dll` and spawns a remote thread with `CreateRemoteThread`.

## 7.2 Direct3D 9 & 11 Virtual Method Table (VMT) Detours

Upon attachment (`DLL_PROCESS_ATTACH`), the overlay library dynamically queries the active rendering API by inspecting loaded modules (`d3d9.dll` vs. `d3d11.dll`). It creates a temporary dummy window to safely retrieve interface vtables without interrupting game execution:

- **DirectX 9 Hooking:** Detours `IDirect3DDevice9::Present` (vtable index 17) and `Reset` (vtable index 16) using `MinHook` (`MH_CreateHook`).
- **DirectX 11 Hooking:** Detours `IDXGISwapChain::Present` (vtable index 8) and `ResizeBuffers` (vtable index 13).

## 7.3 Inter-Process Communication (IPC)

Telemetry and state synchronization are exchanged non-blockingly between the `shogun-sync` host daemon, the GUI frontend, and the injected DLL using a named Windows file mapping with read/write access:

```
OpenFileMappingA(FILE_MAP_ALL_ACCESS, FALSE, "Local\\ShogunSyncIPC");
```

The mapped region conforms to the binary structure:

```
typedef struct {
    uint32_t tour;
    uint32_t status;
    char status_msg[64];
    char room_code[8];
    bool isHost;
    bool triggerReload;
    char lastSaveName[64];
    uint32_t reloadCount;
    uint32_t lastReloadTick;
} ShogunSharedData;
```

To prevent race conditions and unintentional state corruption when multiple components (such as the CLI, GUI, or independent reload commands) access the shared memory region, `initIPC()` inspects the Win32 error code:

```
shogunHandle = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
    sizeof(ShogunSharedData), "Local\\ShogunSyncIPC");
bool alreadyExists = (GetLastError() == ERROR_ALREADY_EXISTS);
if (shogunHandle == NULL) {
    shogunHandle = OpenFileMappingA(FILE_MAP_ALL_ACCESS, FALSE, "Local\\
        ShogunSyncIPC");
    alreadyExists = true;
}
```

If `alreadyExists` is true, the buffer is mapped directly without zeroing out active session telemetry, ensuring that one-off commands (such as `shogun-sync reload`) can trigger in-game reloads while a background watch loop is executing.

## 7.4 In-Game Save Reload Injection Subsystem

When desynchronization occurs during multiplayer campaign progression, the client automatically fetches the authoritative save file from the host, decompresses it via zlib, and flushes it to `save_games_multiplayer`. To eliminate the disruptive requirement of manually restarting the session, Shogun-Sync features an automated in-game save reload injection pipeline:

### 1. IPC Signal Dispatch:

Upon successful save restoration, `kw_conn.c` updates the shared memory mapping: setting `triggerReload = true`, copying the active save name into `lastSaveName`, and setting `status_msg = "Save injected - reloading..."`.

### 2. Window Message Detour (WndProc Hook):

Upon Direct3D initialization, the overlay captures the game's window handle (`g_hGameWnd`) and detours its window procedure via `SetWindowLongPtrA(g_hGameWnd, GWLP_WNDPROC, Hook_WndProc)`. The hook listens for:

- `VK_F10`: Toggles interactive ImGui menu mode (`g_bInteractive`), dynamically enabling keyboard and mouse capture.
- `VK_F9`: Triggers an immediate manual campaign save reload.

All unhandled window messages are transparently passed down to the original game window procedure via `CallWindowProcA`.

### 3. Synthetic Keystroke Injection (PostMessageA):

Inside `TriggerSaveReload()`, the overlay posts synthetic hardware keyboard events directly to the message queue of `Shogun2.exe`:

```
PostMessageA(g_hGameWnd, WM_KEYDOWN, VK_F9, 0x00430001);
PostMessageA(g_hGameWnd, WM_KEYUP, VK_F9, 0xC0430001);
PostMessageA(g_hGameWnd, WM_KEYDOWN, VK_CONTROL, 0x001D0001);
PostMessageA(g_hGameWnd, WM_KEYDOWN, 'L', 0x00260001);
PostMessageA(g_hGameWnd, WM_KEYUP, 'L', 0xC0260001);
PostMessageA(g_hGameWnd, WM_KEYUP, VK_CONTROL, 0xC01D0001);
```

This dual-trigger sequence dispatches both the quickload hotkey (`VK_F9`) and the Total War campaign load shortcut (`Ctrl + L`), prompting the Warscape engine to read the newly restored save file from disk.

### 4. HUD Banner & Reload Metrics:

Upon injection, the overlay displays an animated green banner notification (`[ SAVE INJECTED ]`) across the screen for 6,000 ms, updates `lastReloadTick`, increments `reloadCount`, and clears `triggerReload`.

## 7.5 Immediate-Mode HUD Rendering

Rendering is handled via Dear ImGui. In each hooked `Present()` call, the overlay polls the IPC region, configures a semi-transparent HUD window pinned to the upper-left viewport with `ImGuiWindowFlags_NoInputs` (ensuring zero click interception or mouse conflict with the game when idle), and renders:

```
Shogun-Sync | Tour:  X | [ SYNC OK ] | Room:  CODE (Host/Client) | Reloads:  N
                  | by KW
```

When interactive mode is toggled via `VK_F10`, the overlay opens an interactive control panel allowing players to view the last restored save name, verify total reloads, and click [ **Trigger Save Reload Now (F9)** ].

## 8 Interface & Operations Reference

### 8.1 Command-Line Interface (CLI)

| Command                   | Arguments                                 | Description                                                                    |
|---------------------------|-------------------------------------------|--------------------------------------------------------------------------------|
| <code>-gui, -g</code>     | None                                      | Launches the standalone graphical user interface.                              |
| <code>init</code>         | None                                      | Initializes local <code>.sync</code> metadata directories and CAS storage.     |
| <code>create</code>       | <code>&lt;password&gt;</code>             | Generates 6-character room code, starts host session, and enters watch loop.   |
| <code>join</code>         | <code>&lt;code_id&gt; &lt;pass&gt;</code> | Connects client to room by code and password, then enters watch loop.          |
| <code>commit</code>       | None                                      | Manually creates an immediate snapshot commit of current campaign saves.       |
| <code>-watch, -w</code>   | None                                      | Executes directory watch in offline standalone mode.                           |
| <code>-scan, -s</code>    | None                                      | Scans and counts multiplayer save files on disk.                               |
| <code>inject</code>       | <code>[dll_path]</code>                   | Injects DirectX HUD overlay into the running <code>Shogun2.exe</code> process. |
| <code>reload</code>       | <code>[save_name]</code>                  | Signals the in-game overlay via IPC to reload the target campaign save.        |
| <code>update, -u</code>   | None                                      | Checks for software updates and performs atomic in-place binary upgrade.       |
| <code>uninstall</code>    | None                                      | Cleanly removes Shogun-Sync binaries, shortcuts, and PATH configuration.       |
| <code>-version, -v</code> | None                                      | Displays software version ( <code>0.1.1-beta</code> ) and author.              |
| <code>-help, -h</code>    | None                                      | Displays usage syntax and command reference.                                   |

### 8.2 Graphical User Interface (GUI)

The desktop application (`ShogunSync.exe`) provides an integrated Direct3D 11 / Dear ImGui interface partitioned into three tabs and a live activity console:

- **Multiplayer Lobby:** Selects session mode (Host, Client, Standalone Watch), manages room codes and passwords, starts/stops background worker threads, and displays live synchronization badges ([ `SYNCHRONIZED` ], [ `DESYNC DETECTED` ], [ `RESYNCING` ], [ `CONNECTING` ]) alongside total in-game reload counts.
- **Tools:** Triggers one-click repository initialization (`init`), manual save snapshots (`commit`), save directory scanning (`scan`), in-game overlay injection (`inject`) with live process detection of `Shogun2.exe`, and manual in-game save reload triggering (**Trigger In-Game Save Reload (F9)**).
- **Maintenance:** Provides integrated update verification and installation via WinINet (`updateApplication`), full system uninstallation (`uninstallApplication`), and license details.
- **Activity Log:** Real-time scrollable diagnostic stream outputting network transactions, turn updates, and error events.

### 8.3 Dedicated Installer & System Deployment (ShogunSyncSetup.exe)

To streamline distribution on end-user machines, the project includes a standalone Win32 installer (ShogunSyncSetup.exe) created using native Windows Common Controls (comctl32.dll).

The installer packages all compiled binaries within its own Portable Executable resource table:

- **Resource 101 (RCDATA):** shogun-sync.exe (CLI daemon and watch engine).
- **Resource 102 (RCDATA):** ShogunSync.exe (Windowless GUI launcher).
- **Resource 103 (RCDATA):** shogun\_overlay32.dll (32-bit DirectX hook library).

During installation, the utility:

1. Extracts all three binaries into the standard user local directory:

%LOCALAPPDATA%\ShogunSync\

2. Generates desktop and Start Menu shortcuts targeting ShogunSync.exe.
3. Registers the installation directory in the user PATH environment variable, enabling direct CLI execution from any command prompt.
4. Writes standard Windows uninstallation keys under:

HKCU\Software\Microsoft\Windows\CurrentVersion\Uninstall\ShogunSync

### 8.4 Diagnostic Error Codes

Comprehensive system diagnostic error codes (001–034) and complete operational troubleshooting matrices are documented in the companion *Error Codes Reference* document (errors.pdf).

## 9 References & Further Reading

- Polge, Thibault. *Write yourself a Git! (WYAG)*, 2020. <https://wyag.thb.lt/#intro>.
- National Institute of Standards and Technology (NIST). *FIPS PUB 180-1: Secure Hash Standard (SHA-1)*, 1995.
- Wikipedia contributors. *SHA-1: One iteration within the SHA-1 compression function*, Wikimedia Commons, <https://en.wikipedia.org/wiki/SHA-1>.
- Deutsch, P. and Gailly, J.-L. *ZLIB Compressed Data Format Specification version 3.3*, RFC 1950, 1996.
- Cesanta Software. *Mongoose Networking Library Documentation (v7.x)*, <https://cesanta.com/docs/>.
- Cornut, Omar. *Dear ImGui: Bloat-free Immediate Mode Graphical User Interface for C++*, <https://github.com/ocornut/imgui>.
- Tsuda, TsudaKageyu. *MinHook: The Minimalistic x86/x64 API Hooking Library for Windows*, <https://github.com/TsudaKageyu/minhook>.