CS390R: Reverse Engineering GoldenEye 007

Stuart Lustig , Daniel Melanson , Jared Dalkas , Jon Rubio

Github: https://github.com/daniel-melanson/cs390r-group-h

Overview **2**
- GoldenEye 007 2
- N64 Device and Architecture 2
- Emulators 2
- The GoldenEye 007 N64 ROM 3

Analyzing Similar Targets **4**
- Purpose of Looking at Other N64 Games 4
- Legend of Zelda: Ocarina of Time and Arbitrary Code Execution 4
- Reflections from Analyzing LoZ: OoT 5

Original Steps to Reverse Engineer the Game **6**
- Inspecting the Decompilation 6
- Looking at Known Bugs 6

Fuzzing GoldenEye 007 ..... **8**
- Overview 8
- BizHawk API 8
- Checking For Crashes 9
- Randomizing Input 9

Save File Manipulation **10**
- Modifying N64 Game Saves 10
- What Save Data is Stored? 10
- How Is Save Data Stored in Memory? 12
- How Are Checksums Used in the Code? 12
- Generating Our Own Checksums 13
- Using Our Program to Modify Save Files 13
- Potential of Custom Save Files 13

Biggest Challenges **14**
- Static Code Analysis 14

Future Goals **16**
- Investigating Known Bugs 16
- Fuzzing 16
- Save File Manipulation 16

Sources **16**

Overview

Our group shares an interest in exploiting retro games. We watched some videos showcasing exploits done on Zelda on the Nintendo 64 (N64) that leveraged arbitrary code execution by exploiting the heap and save file names. This got us excited to attempt something similar.

We decided to focus on 007 Goldeneye for the N64. We were able to get the game running and attached to a debugging environment. Coincidentally, we found a Github page with someone attempting to reverse engineer the MIPS assembly to C. Although it is not fully reverse engineered, we were able to understand enough of the game logic and found interesting functions that were helpful in our venture..

The original goal for the project was to see if there are any vulnerabilities in the code that we can use to our advantage. Ideally, we would have liked to port similar exploits found in Zelda into Goldeneye. As we progressed and delved deeper into the mechanics of Goldeneye, it became apparent that this was unlikely due to the differences in each game. Using topics we learned in class, we were able to succeed in other ways such as reverse engineering save files and implementing a fuzzer.

GoldenEye 007

Goldeneye 007 is a first-person shooter released for the N64 in 1997 that was developed by an inexperienced team, Rare, over the course of two years. It quickly became one of N64's most beloved titles, taking third place in total sales for the console. It closely follows the plot of the 1995 James Bond film, offering an engaging single-player campaign and a popular multiplayer experience for up to four players. Goldeneye 007 left a lasting impact on the FPS genre and remains one of the most replayed video games in gaming history.

N64 Device and Architecture

The N64 is powered by a 64-bit NEC VR4300 CPU running at ~90 MHz. This MIPS III processor provided substantial processing power for its time. Despite being 64-bit, the console's games generally used 32-bit data-operations. The 32-bit operations were sufficient to generate 3D scenes, but executed faster and required less storage space. This was important as the storage devices could not hold much data. And there is only 4MB of RDRAM located on the board.

The game cartridges (paks) store the instruction set for the N64 games. This includes the game engine and its own bootloader. The game cartridges are mapped directly into the console's normal address space. This gives the N64 superior read/write times when compared to other CD-based consoles.

N64 controllers have a distinct three-pronged design, which supports an analog control stick, directional pad, and ten buttons. Additionally, there was a port for controller paks. These controller specific storage devices held around 32 KB of data. Typically, these paks were used to store user save data. Similar to the game pak, these devices were memory mapped into the normal address space of the device.

Emulators

Analyzing N64 games requires an emulator to run the console's instruction set. Our team ended up using two different N64 emulators to achieve different tasks. BizHawk is an open-source multi-platform emulator with a focus on tool-assisted speedruns. This allowed our team to programmatically interact with a running instance

of our target to send input. In addition to BizHawk, our team used Project64. Project64 is an open-source N64 specific emulator. Our team leveraged Project64's load and save functionality to reverse engineer save files.

The GoldenEye 007 N64 ROM

Below is a hash of the specific ROM dump our team analyzed.

```
$ md5sum GoldenEye 007 (USA).z64
70c525880240c1e838b8b1be35666c3b  GoldenEye 007 (USA).z64
```

Analyzing Similar Targets

Purpose of Looking at Other N64 Games

Before looking for exploits in GoldenEye 007, we wanted to look into other N64 games to see what types of exploits appear. The idea was that once we found an exploit or type of exploit, we could try to relate it to GoldenEye 007 and see if there was any way to recreate it in a different game.

Legend of Zelda: Ocarina of Time and Arbitrary Code Execution

The majority of our time spent analyzing similar targets was looking into how arbitrary code execution works in the Legend of Zelda: Ocarina of Time. A great explanation of the arbitrary code execution (ACE) exploit can be found in this video.

So, how does ACE in OoT work? First, we note that all memory exists in the same address space, including save files, in any N64 game. Also, all memory is executable. From class, we know that this means that if we can point the instruction pointer to user-controlled data, we can execute any MIPS instructions we want, just like inputting a shellcode.

The next observations are specific to the programming of Legend of Zelda games for the N64. In LoZ, each object that is interactable in some way is called an “actor”. These objects are stored on the actor heap. They are loaded and unloaded every time the main character, Link, switches between rooms. Link can switch between rooms seamlessly by walking through certain doorways, and all actors specific to the old room are unloaded. Then, all actors specific to the new room are loaded by allocating space for them on the actor heap.

The type of bug that allows ACE to work in OoT is called Stale Reference Manipulation (SRM). SRM is basically the same as a use-after-free vulnerability which we learned about in CS390R. The specific way that the exploit uses SRM is by Link picking up an object and then deleting that object on the actor heap and replacing it with a different object. For example, Link can pick up a pot and switch rooms at the same time, using a different exploit called a super-slide, which allows Link to start picking up the pot, switch rooms, and then pick up the pot. The reference to the pot is stored in Link’s memory on the actor heap once he starts picking up the pot, so in the new room he has a reference to whatever actor is in the same memory where the pot used to be. In the image below, we can see the result; Link is holding an invisible pot over his head.

Once this step is completed, we can change some of the values of whatever is stored in the location that the pot should have been stored. For example, if Link turns and walks around he changes the X and Y coordinates and the rotation of the pot. However, if the SRM is set up correctly, a user could use this to change the code of another actor, for example changing the contents of a chest or the behavior of an NPC.

In the example in the video, the SRM exploit is used to change the code of an NPC to jump the instruction pointer to the save file name. The save file name is chosen upon the first creation of the game save (see image below). The video explains how setting this save file name to specific MIPS instructions can allow many different exploits, by running some custom code and then redirecting execution back to normal code.

For example, there was a save file name that would disable Link's gravity:

Reflections from Analyzing LoZ: OoT

The first thing we learned was that anything in program memory can be executed. We also learned that save files are part of the program's address space. We did not learn much else that might be useful to GoldenEye 007, because the vulnerabilities in OoT relied heavily on the vulnerability to SRM which is specific to OoT.

We did note that the vulnerability to ACE required that a super-slide be used, which was a known bug in LoZ. This suggested to us that if we were looking for exploits in GoldenEye 007, known bugs would be a good place to start.

Original Steps to Reverse Engineer the Game

Inspecting the Decompilation

Our team found a community-managed GoldenEye 007 decompilation project on Github. This project took the original MIPS-III instruction set and separated it into two main modules: library code and game engine code.

The library code contained source files for utility functions regarding the operating system, libc, input/output, and audio. Luckily, this code was fully decompiled and in some cases it even came from source. However, our team found this code to be irrelevant for our project and mostly ignored its implementation.

The actual game engine decompilation is less organized. Large chunks of related instructions are grouped together in C files with helpful names. Unfortunately, the majority of the contents of the C files are macros containing the raw MIPS instructions. But there are some useful constructs inside the decompilation. Each MIPS instruction, and all global variables, are prefixed with a comment to the address inside the ROM. This came in handy when we were debugging a running instance of the ROM. Our team was able to search addresses inside the source code to see which instructions manipulated them. Below are some examples of such:

```
glabel debug_menu_processor
/* 0C50D0 7F0905A0 27BDFFA0 */  addiu $sp, $sp, -0x60
/* 0C50D4 7F0905A4 3C038003 */  lui   $v1, %hi(g_DebugScreenshotRgb)
/* 0C50D8 7F0905A8 8C636FFC */  lw    $v1, %lo(g_DebugScreenshotRgb)($v1)
/* 0C50DC 7F0905AC AFBF0014 */  sw    $ra, 0x14($sp)
/* 0C50E0 7F0905B0 AFA40060 */  sw    $a0, 0x60($sp)
/* 0C50E4 7F0905B4 AFA50064 */  sw    $a1, 0x64($sp)
/* 0C50E8 7F0905B8 AFA60068 */  sw    $a2, 0x68($sp)
/* 0C50EC 7F0905BC 1060000D */  beqz  $v1, .L7F0905F4
```

```
//D:80036BA0
u32 D_80036BA0 = 0;
//D:80036BA4
s32 g_DebugMenuOffsets[] =
{
        8, 0x13, 0x1E, 0x2B,
    0x32, 0x39, 0x45, 0x4D,
    -1
};

//D:80036BC8
struct mcm_layout g_DebugMenuPositions[] = {
```

Looking at Known Bugs

As mentioned earlier, we learned from analyzing the Legend of Zelda: Ocarina of Time, and somewhat from our past knowledge on video game exploits, that known bugs are often related to exploits. As such, we looked into known bugs. We did this by going through wiki pages such as the Goldeneye 007 wiki and by watching YouTube videos (Goldeneye 007 Glitches, Goldeneye 007 More Glitches).

We were looking for bugs specifically that seem like they could be connected to an exploit, could cause a crash, or were generally interesting to us in some way. One known bug that causes a crash was on the level

Silo, where it is known that “Throwing an object after throwing a Plastique or a Remote mine on a Circuit board then picking it up will crash the game.” Another bug we wanted to look into was where “Destroying all of the ceiling monitors simultaneously in the monitor room will cause projectiles (Throwing Knives, mines, Rockets, etc.) to be stuck in the air when thrown” on a couple of levels. These two seemed interesting because the first causes a crash, and the second seems like something in the code modifies other code or data that it should not be modifying.

Sadly, we did not have enough time to fully look into what causes these bugs. The main reason was that most of the code handling projectiles was still in MIPS assembly in the decompilation. Looking into these bugs also could have led us down a rabbit hole, since they may rely on many different specifics of the code, MIPS processors/assembly, and the N64. So we felt it best to move on to other topics.

Fuzzing GoldenEye 007

Overview

We implemented fuzzing by writing a Python script (fuzz.py) which generates a random series of controller inputs. The Python script generates the inputs, opens the template Lua script (fuzz_template.lua), writes this template to a new Lua script at a specified INPUT_PATH, with the controller inputs inserted as the contents of a local variable called frame_input_states, launches the emulator, waits for an exit code to determine if a crash has occurred, and writes the offending input to a timestamped file in the case of a crash.

The Lua script loads the Goldeneye ROM and a savestate, sets the controller inputs for every frame, checks if the game has crashed by checking if the framecount has advanced, exits with error code 12345 on a crash, or 0 if execution has ended normally.

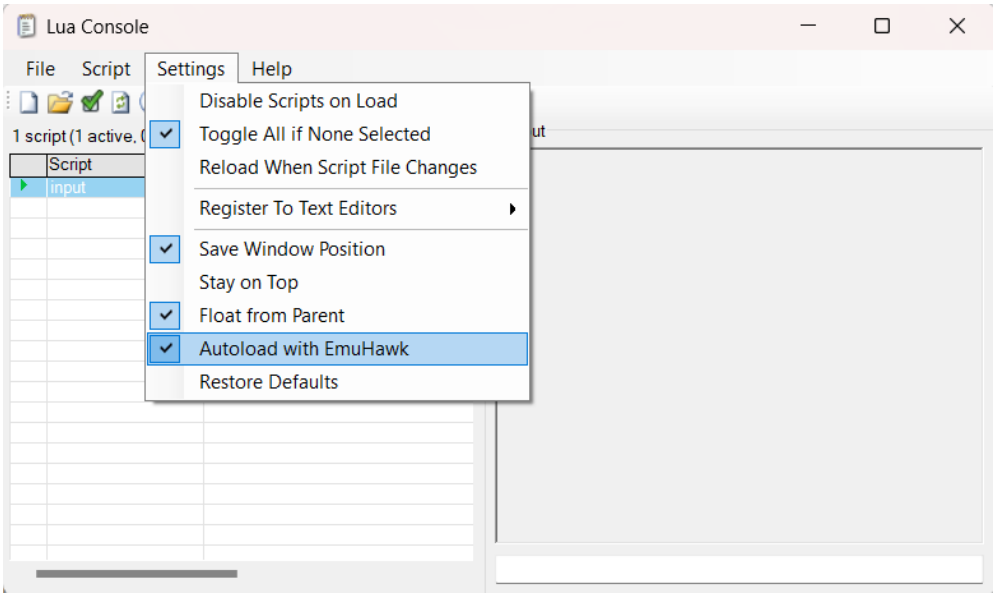
A C harness was written for an earlier draft, with the idea being for a Python script to generate inputs, which would be loaded into the harness, which would in turn write the Lua script and launch the emulator, but we decided to simplify the implementation.

BizHawk API

BizHawk is an emulator built primarily for Tool Assisted Speedrunning, which involves pre-recording and playing back precise inputs in order to pull off tricks which would be unviable for a human player. As such, Bizhawk comes with Lua scripting functionality and a fairly extensive API.

At the start of the Lua script, the rom is opened with client.openrom, and a save state is loaded with savestate.load. Both of these functions take a file path as input, which is inserted into the template Lua script by the Python script. Changing these paths requires changing the ROM_PATH or SAVE_STATE_PATH variables in fuzz.py.

While there is no way to load a Lua script into Bizhawk from the command line, they can be preloaded into the Lua Console and set to run automatically on startup. As such, for the fuzzer to work, the input script must have been manually loaded in the Lua Console, and the “Autoload with EmuHawk” setting must be enabled.



Checking For Crashes

The BizHawk Lua API unfortunately does not provide a simple way to check if the ROM has crashed, and the handful of other retro game fuzzing projects we found online all involved trying to crash the emulator, rather than the game which is being emulated. As such, a workaround was required.

The API does come with a function emu.framecount, which returns an integer value representing the number of frames which have been rendered during the current session. Not all ROM crashes will result in this. We tested this by using the known crash in the Silo level, and found that the framecount ceased to increment after crashing.

A crash is not the only time that the framecount might not increment, it can also happen intermittently on lag frames, or for several frames in a row during a loading screen. To prevent false positives, the Lua script keeps track of the number of consecutive frames which have gone by without the framecount advancing in the variable repeated_framecount, and checks that against an epsilon value MAX_REPEATED_FRAMECOUNT. A crash is only reported if repeated_framecount is greater than or equal to MAX_REPEATED_FRAMECOUNT. An epsilon value of 100 frames was found to work for Goldeneye, but would need to be re-tuned for fuzzing of other titles.

Randomizing Input

The BizHawk Lua API does not feature an interface that could be used to calculate instruction coverage. So input is randomly generated for each frame. The input generator will select a random amount of frames, in the range of 100 to 1000, to generate input for. Then for each frame, compose a randomized input state for each of the 10 buttons, 4 d-pad directions, and analog stick. Buttons are either pressed or not; however, the analog stick has an orientation represented at signed byte weights for horizontal and vertical direction. Input states are then converted from a Python run-time data structure to a Lua source code string.

Once the inputs have been inserted into the Lua script, execution is fairly straightforward. For every input_state in frame_input_states, we check the current framecount, use joypad.set to set the input for the next frame to be rendered, advance the gamestate by one frame with emu.advance, and then check for a crash.

Save File Manipulation

Modifying N64 Game Saves

N64 Controller Paks

The N64 has a feature called controller paks. This feature improves the memory-saving features of many N64 games. Instead of saving storage directly to the cartridge, it saves storage into controller paks (pictured above). Some games even require the use of controller paks. The paks can be swapped between controllers and edited, meaning we could actually perform exploits in real life by modifying the game's save data. We were under the assumption that this worked for GoldenEye 007 when we presented, but the game actually does not support this feature and stores all of its saves on the cartridge.

Modifying N64 Game Saves

Although the fantasy of being able to plug-and-play any game save we would want into any real N64 is quite enticing, in reality we would actually have to have our hands on the cartridge in order to modify the game saves stored on it. For example, cartridge readers such as this one can be used to overwrite the save file in a GoldenEye 007 game cartridge (or you can even make your own!). So it is actually viable to modify the game save of a real GoldenEye 007 cartridge, meaning any exploits or bugs we might find related to modifying game saves could be used on a real N64 with a real cartridge.

What Save Data is Stored?

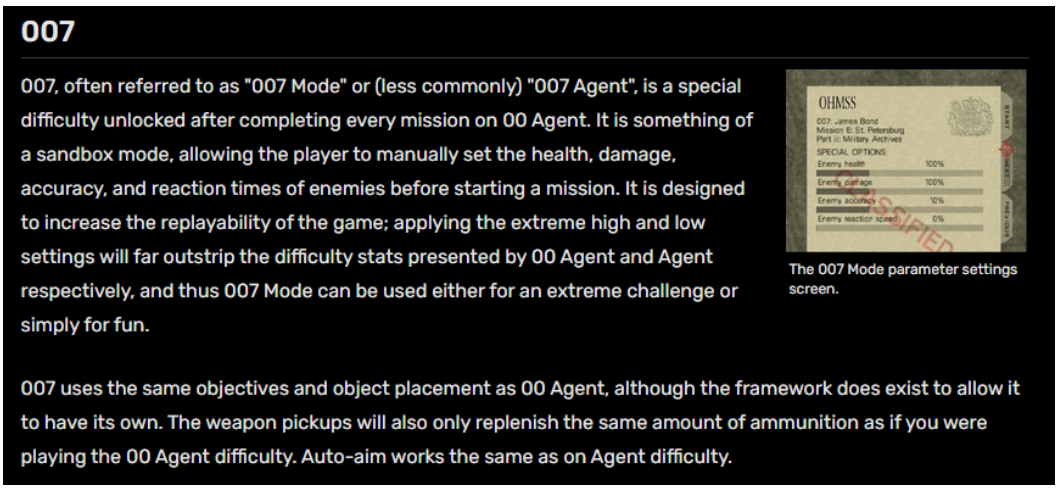
First, we can see that upon launching the game, there are four save files, contained in folders.

Upon inspecting the decompilation, we find that the save_data struct stores all the save data for one folder of memory:

```
typedef struct save_data
{
  s32 chksum1;
  s32 chksum2;
  u8 completion_bitflags;
  u8 flag_007;
  u8 music_vol;
  u8 sfx_vol;
  u16 options;
  u8 unlocked_cheats_1;
  u8 unlocked_cheats_2;
  u8 unlocked_cheats_3;
  char padding;
  u8 times[(SP_LEVEL_MAX-1) * 4];
} save_data;
```

We then looked into how this data is used in the program, and found the following:

- chksum1 and chksum2 are used to validate the save data upon launching the game (will be discussed later)
- completion_bitflags is at least used to check what state the folder is in (empty, etc.). We think it has some other uses but avoided looking too much into it
- flag_007 is nonzero if the player has unlocked the 007 difficulty mode:



007

007, often referred to as "007 Mode" or (less commonly) "007 Agent", is a special difficulty unlocked after completing every mission on 00 Agent. It is something of a sandbox mode, allowing the player to manually set the health, damage, accuracy, and reaction times of enemies before starting a mission. It is designed to increase the replayability of the game; applying the extreme high and low settings will far outstrip the difficulty stats presented by 00 Agent and Agent respectively, and thus 007 Mode can be used either for an extreme challenge or simply for fun.

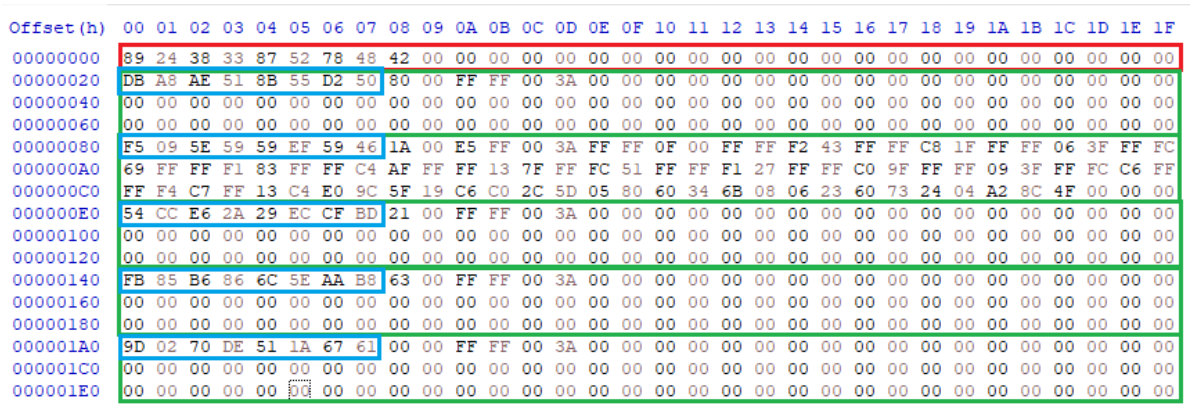
The 007 Mode parameter settings screen.

007 uses the same objectives and object placement as 00 Agent, although the framework does exist to allow it to have its own. The weapon pickups will also only replenish the same amount of ammunition as if you were playing the 00 Agent difficulty. Auto-aim works the same as on Agent difficulty.

- music_vol, sfx_vol control volume levels
- options controls a variety of options
- the cheat unlocks show which cheats have been unlocked by scoring low times on certain difficulties of specific levels or by pressing certain buttons in a specific order
- Lastly, times stores the times that the player took to finish each level. The times are stored in seconds by using 10 bits for each time

How Is Save Data Stored in Memory?

The first 32 bytes are fixed, shown bound in a red box below. It starts with a checksum hash for the entire save file and the remaining bytes are used for padding. Then there are 5 save files in a row shown bound in a green box. The first 4 are the folders for each slot, with a 5th used for copying between slots. Each game slot starts with a checksum that is generated at save creation/update, shown bound in a blue box. In total, the entire save file adds up to 512 bytes.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00000000	89	24	38	33	87	52	78	48	42	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	DB	A8	AE	51	8B	55	D2	50	80	00	FF	FF	00	3A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	F5	09	5E	59	59	EF	59	46	1A	00	E5	FF	00	3A	FF	FF	0F	00	FF	FF	F2	43	FF	FF	C8	1F	FF	FF	06	3F	FF	FC
000000A0	69	FF	FF	F1	83	FF	FF	C4	AF	FF	FF	13	7F	FF	FC	51	FF	FF	F1	27	FF	FF	C0	9F	FF	FF	09	3F	FF	FC	C6	FF
000000C0	FF	F4	C7	FF	13	C4	E0	9C	5F	19	C6	C0	2C	5D	05	80	60	34	6B	08	06	23	60	73	24	04	A2	8C	4F	00	00	00
000000E0	54	CC	E6	2A	29	EC	CF	BD	21	00	FF	FF	00	3A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	FB	85	B6	86	6C	5E	AA	B8	63	00	FF	FF	00	3A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0	9D	02	70	DE	51	1A	67	61	00	00	FF	FF	00	3A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

How Are Checksums Used in the Code?

When the game starts, it validates the first checksum that is stored in the red box above. If the checksum fails, then the entire save is deleted and a new one is created. It then verifies that each game slot also has a valid checksum, if it does not, it resets that particular slot. After these checksums are validated, the save file is loaded into the game. Below are snippets of the validation process found in the reverse engineered code.

```
// bad checksum, create a new save and replace damaged one.
if (!checksumOK)
{
    smallSave NewSave = blankSmallSave;
    joyChecksum = NewSave;
    fileWriteSmallSave(&joyChecksum);
}
```

```
if (!checksumOK2)
{
    fileResetSave(&saves[i]);
}
```

Generating Our Own Checksums

After learning how checksums were validated for the save files, we tried to see if we could insert arbitrary data into a save file and created a tool that generates the appropriate hashes. We successfully built a C program that did exactly this. In our program, we skipped the first 32 bytes that were generated at file creation since it did not rely on the rest of the data. We then read in each game slot data, skipping the first 8 bytes, and fed it into the algorithms implemented below. This gave us the hash needed to fill in the first 8 bytes for the appropriate slot.

```
13    uint64_t SubHash(uint32_t hi, uint32_t lo);
14    uint64_t Hash(unsigned char *data);
```

We built the program with several different options:

- Fill in each byte with a specific byte given as input. For example, we can fill every byte with 0x00-0xFF.
- Fill in each byte with a random byte.
- Create hashes for game data that was modified using a hex editor.

Link to the program we built to generate hashes: [007Savefile_hasher.c]

Using Our Program to Modify Save Files

An example of the program being run on a save that was hex edited can be found here: GoldenEyeSavesCS390rDemo.mp4

Potential of Custom Save Files

We tested different modified save files to see if we were able to get any undefined behavior. After testing the options we implemented, the randomizer piqued our interest. We were able to modify a save file filled with randomized bytes, get into the start screen with the save file loaded, and once we opened a save folder the game would crash. Interestingly this would not occur every time we randomized the data, with a small chance that a save folder will load with random times for each level that was generated by the program. This leads us to believe that save file manipulation with the addition of a fuzzer, would be useful in loading states with either incremental or randomized data changes to achieve more interesting results.

Above is a screenshot of a randomized save file. The folder with the 007 loads, the other 3 crash the game.

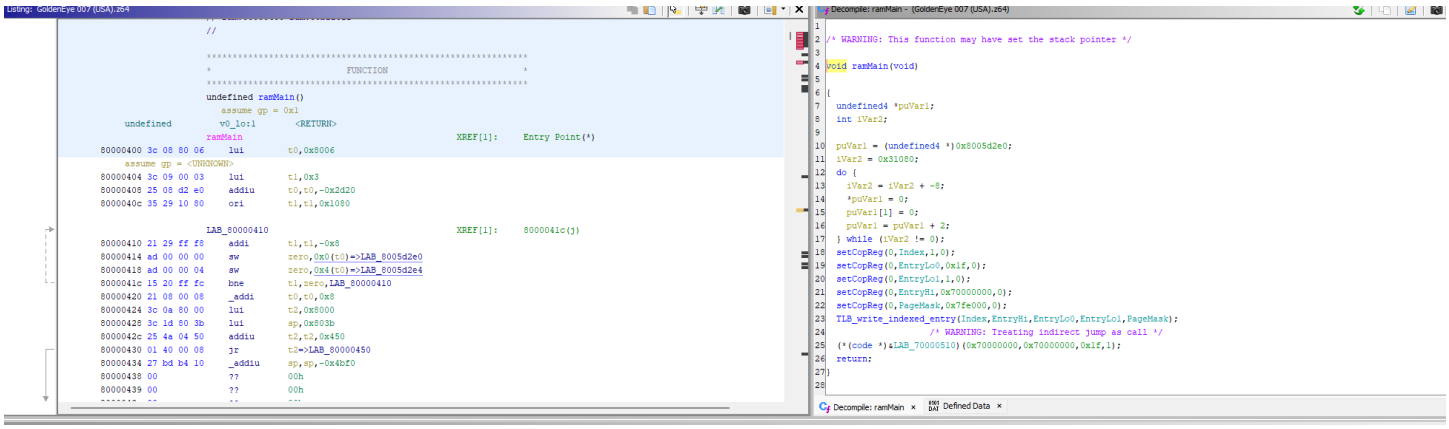
Biggest Challenges

Static Code Analysis

When we first discussed static code analysis, our goal was to use a tool to see if it can find game logic that could be exploited. One of the first issues we ran into was because the source code is not available to use and the reverse engineering project we found was not complete, we would have to use a static analysis tool built for MIPS assembly. Unfortunately, we were only able to find one that worked for 32-bit architecture, and given that the N64 uses a 64-bit architecture this would not work. Instead we came up with the idea that if we were able to decompile the rom into Ghidra, it could be helpful in using a static analyzer for the C programming language.

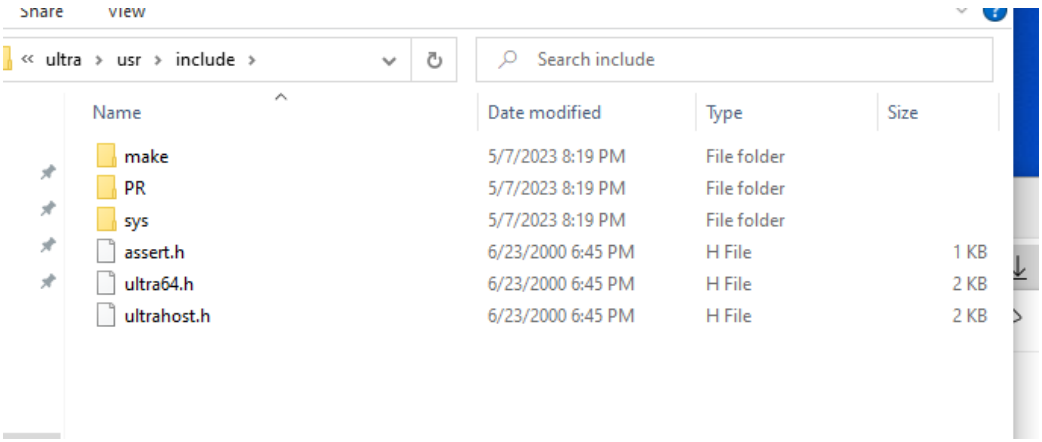
Importing the Project in Ghidra

We found a youtube video that explained how to use a N64 ROM loader plugin for Ghidra so it is able to decompile it into C code. We were able to get it set up properly and now able to see the code in a format we were familiar with. However, we ran into a problem where all the function calls, variable names, structs and other symbols used by the program were missing so most of the code was still a mess. We figured if we wanted to get this properly loaded in a static analysis tool, we would have to either have to fix these problems manually or find an alternate method.



Importing N64 Library Symbols into Ghidra

During our research of static analysis tools for the N64 and getting the ROM loaded into Ghidra, we found that Ghidra has an option to import header files and automatically load the information we need. Luckily, the N64's SDK was leaked a couple years back and we were able to get a copy of it. With this, we attempted to load the header files into Ghidra only to run into more issues during that process. Below are screenshots of the SDK files and one of the errors we encountered when importing them into Ghidra.



Why We Gave Up

Static analysis proved to be our biggest hurdle in exploiting Goldeneye 007. We came to the conclusion that the process thus far was time consuming and, even if we were successful in loading the headers, it would not be as useful as the reverse engineered code we had available to us. After coming to an agreement, we decided our efforts were better spent on other ideas. We decided that the time remaining should be used for manipulating save files and implementing a fuzzer, which indeed yielded better results.

Future Goals

Investigating Known Bugs

As mentioned in the section on known bugs, we would have liked to look into the bugs that we found interesting. We could have started this process by decompiling or understanding a couple of functions from MIPS assembly which cover thrown projectiles. Then, from there, we could debug the crash that happens in Silo, or we could try to find what code makes it so projectiles do not move (look for a potential jump instruction that passes the section where projectiles are moved, etc.)

Fuzzing

The implementation of our fuzzer was not optimal. In the future, our team should redesign the structure of input generation and feeding. However, creating a feature-rich fuzzer would require a significant amount of architecture.

Looking at the BizHawk source code makes it seem possible to develop a plugin that is able to calculate coverage of instructions. Additionally, instead of interfacing with the ROM through BizHawk's Lua API, our plugin would be able to directly interface with the ROMs input controller. Allowing us to completely remove the need to generate any sort of Lua scripts to feed input. We could then pair the input controller with a coverage-guided, grammar-based fuzzer (like Nautilus or AFL's Grammar-Mutator).

The plugin could initially prompt the user to input a series of save states (RAM snapshots) and a corpus of input in the custom syntax. Afterwards, the plugin needs to act as a middle man between the fuzzing backend (Nautilus/Grammar-Mutator) and BizHawk. It would need to translate the input from the backend into a series of API calls on the ROMs input device and feed back the reported coverage.

This process would be significantly faster than our current fuzzing implementation. With more control of the emulator, our team could develop a headless mode that skips unnecessary stages of emulation, such as drawing the ROM's video output to the host device, to boost performance.

Save File Manipulation

As mentioned in the section on the potential of custom save files, we could combine our findings of creating custom save files with valid hashes with a fuzzer to find the extents of what save files could be created and see if any cause interesting crashes.

Sources

Game Info: https://en.wikipedia.org/wiki/GoldenEye_007_(1997_video_game)
Project64 - Nintendo 64 Emulator: https://www.pj64-emu.com/
Speedrunning Community: https://rankings.the-elite.net/goldeneye
Goldeneye 007 Decomp Page: https://github.com/n64decomp/007
Goldeneye 007 Known Bugs: https://goldeneye.fandom.com/wiki/Game_bugs
Goldeneye 007 Glitches: https://www.youtube.com/watch?v=6mp0LZNwHkA&t=16s
Goldeneye 007 More Glitches: https://youtu.be/b6tkREWCHL8
Goldeneye 007 Save Time Editor: https://github.com/zeroKilo/GE64SaveEditorWV
ACE in OoT Explanation:- https://www.youtube.com/watch?v=RoEmGCNsbno
Big endian, byteswap, and little endian roms: http://jul.rustedlogic.net/thread.php?id=11769
N64/MIPS ASM: https://www.youtube.com/playlist?list=PLjwOF_LvxhqTXVUdWZJEVZxEUG5qt8fsA
N64 Decompiling: https://www.retroreversing.com/n64-decompiling
N64 SDK help: https://www.retroreversing.com/n64-sdk
N64 SDK download: https://ultra64.ca/resources/software
BizHawk Lua API: https://tasvideos.org/Bizhawk/LuaFunctions
BizHawk: https://github.com/TASEmulators