Achieving Arbitrary Code Execution on the Nintendo DS using Buffer Overflow Vulnerabilities

James Borgars

June 8, 2021

1 Introduction to the Nintendo DS

The **Nintendo DS** is a video game console created by Japanese company Nintendo which was released in 2005 in Europe. It was hugely popular, having sold over 150 million units worldwide.[1]

Due to the age of original Nintendo DS model (which will also be referred to as the NDS), the security is not seen as very effective by today's standards. To run homebrew (software made by an independent developer that was not authorised or published by the console manufacturer), a "passthrough" method was required. These can range from hardware devices such as adapters or by sending a payload from a computer wirelessly to the NDS using a feature known as **DS Download Play**. A list of discovered passthrough methods with explanations as to their attack vectors and workings was posted by Damian Yerrick.[2]

Firmware was updated several times by Nintendo when they manufactured later units of the NDS, which can be identified through making the device freeze in a certain way, as explained in the following forum post[3] :

```
Insert an authentic Nintendo DS Game Card into SLOT-1, or
a Game Boy Advance cart into SLOT-2, of your Nintendo DS.
Turn on DS (hold Select+Start if you have autostart enabled
in the settings).
Enter Pictochat.
Enter any chat room.
Eject the Game Card or GBA Cart from the DS.


v1:  Pictochat hangs
v2:  two grayish blue screens
v3:  two dark green screens
v4:  two golden yellow screens
v5:  two magenta screens (DS lite units have this)
v6:  two dark blue screens (un-confirmed new DSes have this)
iQue:  two dark green screens
```

From the DSi model onwards, Nintendo was able to release firmware updates to console owners who connected their console to WiFi. This allowed Nintendo to patch firmware bugs that could be exploited to load arbitrary code. We are going to be exploiting the original Nintendo DS model for the purposes of this project.

2 Preparation

This project is inspired from the paper titled *Exploiting DS games through stack smash vulnerabilities in save files* by CTurt.[4] These type of vulnerabilities do indeed allow for arbitrary code execution but are limited by the SD Card or NAND access,[5] meaning that any exploit that is found is limited to the payload stored in the save. These exploits would not be able to execute a sizeable amount of code (due to the maximum file size of a save file) as well as being difficult to modify the payload for an average user, rendering these exploits useless in a lot of scenarios.

The tools used for this project will be similar to that used by CTurt, however a much larger selection of tools will be used for this project.

The development team **devkitPro** maintains an ARM compiler (The payload will run on the DS' ARM9 processor) that also comes with the library **nds.h**, which makes it possible to convert some C/C++ code into ARM assembly.

In terms of emulation, I going to be using two different programs: **DeSmuME Dev Build** and **No$GBA Debugger**, I felt two emulators necessary due to discovering the following when testing them ahead of this project: DeSmuME had much more accurate emulation and had some good debugging features, however register values and memory offsets were incorrect. No$GBA had apparent bugs during emulation and lacked features such as searching for ASCII strings in memory, but provided accurate memory addresses and register values during testing.

Python 3 will be used to create scripts in aiding the patching of savefiles, as almost every game has checksums stored within its savedata in order to detect tampering and corruption, however we will be attempting to figure out what checksum is used in order to avoid these forms of tamper detection.

Hex Editors are also vital in editing savedata, as I will be moving back and forth betweeen a Linux distribution and a Windows partition, I will be using **wxHexEditor** for Linux and **HxD** for Windows. Finally, the last necessary thing to mention is the game that is going to be the target for this project: **WordJong DS**, specifically the 2007 USA release, as the save data is not cross-compatible with later releases of the game. The reasons why WordJong DS was chosen are the following:

1. The game has functionality to store a name - meaning it has to store user input.
2. This name was stored in plaintext ASCII within the save and was null terminated.
3. Large savedata size - the raw save file comes in at exactly 8 KiB, this allows for a relatively large payload for the type of exploit.
4. It seems like there is only one checksum for the entire savedata.

3 Reversing the Checksum

By creating multiple saves for the game, it was evident that if you changed the name of your profile (and nothing else), two areas of the save would change:

1. The plaintext ASCII string of the name
2. The first four bytes of the save - this is a very strong indicator of being the checksum

Having tested checksum reversal of another game, **Driving Theory Training**, before starting this project, I became familiar with a group of checksum generators known as **Cyclic Redundancy Checks** (or CRCs), this was present in a custom 16-bit implementation (known as a CRC-16) for Driving Theory Training. Unfortunately, Driving Theory Training was not exploitable even after reversing the checksum, due to checking the values of certain bytes and a very small save file size.

As the checksum for this save file was four bytes, I strongly suspected it may be an implementation of CRC-32, which is the most common form of CRC. When reversing the checksum of Driving Theory Training, I came across the program **RevEng CRC**, which became very helpful in finding out the variables involved in a specific CRC implementation, especially if it is not part of the group of most popular implementations. I used RevEng CRC with WordJong save files, having given it four instances of save data as well as their corresponding checksums. Instantly it returned with one matching CRC implementation: known as **CRC-32/JAMCRC**.

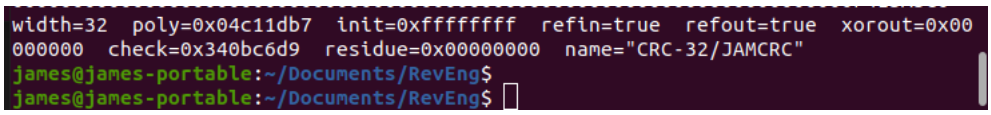

Figure 1: The output of RevEng CRC when it was given the save files

After knowing this I wrote a Python script using the **pwntools** and **binascii** libraries to calculate read a save file, and calculate the correct checksum for that file.

4 Exploring Attack Vectors

Now that the checksum has been rendered useless in regards to checking for valid code, it is now time to explore what is possible with regards to savefile editing. I was happy to see that after writing a large amount of 0xAA consecutively after the ASCII string of the user's name, the game did not detect corruption, meaning that there are a very limited number (or possibly zero) value/range checks of certain bytes. This led to the corruption of data used for the Awards section of the game, and I also discovered a crash within that section, as it tries to load the user's name on one of the pages, therefore causing a Buffer Overflow. After, inspecting the buffer overflow, I could see that the register R3, R4, R5, and most importantly, R15 (the Program Counter) were overwritten by save data. At this point I knew I had found a viable entry point for Arbitrary Code Execution.

Using both emulators, I got more familiar with the memory layout that was present in the game and managed to calculate the correct memory offset of the bytes that overflow the Program Counter, I could then set those bytes to a memory location just after its own offset, therefore allowing execution of code that is located just after the overflow itself.

5 Payload Creation

With this exploit, a payload would have to be stored in binary within the save file. Using the some tools from the devkitPro Compiler, it is possible to convert C code down into ARM9 assembly and then into the binary representation of the same assembly instructions. However, the premise of this

paper is to show how to achieve Arbitrary Code Execution in the first place and less about what the code actually does. This factor combined with the difficulty of learning ARM assembly and the time necessary with both the assembly language and the required programs, meant developing a complex payload was simply not worth it. Luckily, in CTurt's report,[4] he posts a small piece of ARM assembly code that makes the DS screen change colour. This is enough to show a Proof of Concept of Arbitrary Code Execution.

Using a Makefile, we can convert the ARM9 assembly down into binary, this can then be appended after the string which overflows its buffer, followed by running the new save file through the Python Script to patch the CRC. We can now trigger the added code by going to Page 2 of the Awards Section of the game.

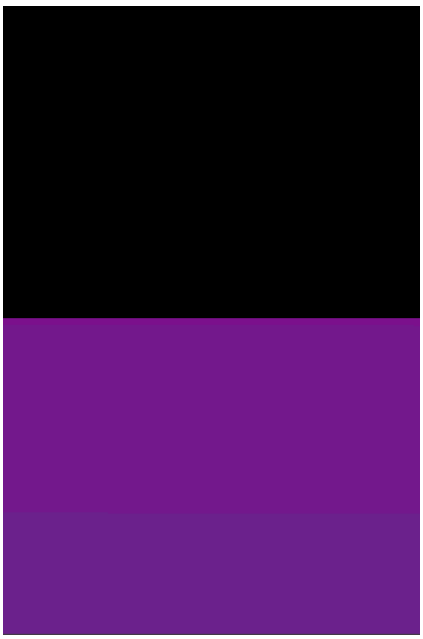

Figure 2: A screenshot of an emulator running the payload.

6 Conclusion

In summary, I pursued this project in order to delve into areas of computing that I am not familiar with, and can finish this project by saying that I feel accomplished having done this, and have learned how to adapt to very new and complex situations in regards to programming and understanding code. It would be nice to further this project by executing more complex code from the exploit in the future, however I need to refine my skills in other areas before attempting to pursue programming in ARM32 assembly.

I will upload some of my code, including the Proof of Concept save file, a 'stub' save file (has the exploit but no payload), as well as the patching Python script, to my GitHub (`https://github.com/Borgars`), if anyone wishes to look further, and develop their own code/exploits with my findings as a template.

Overall, this project left me with not only a sense of accomplishment, but also some tangible fruits for my labour in the form of a functioning exploit, as well as a sense of surprise that "smashing the stack" was/is still a viable exploitation technique in the 21st Century.

References

[1] Nintendo. Consolidated sales. `https://www.nintendo.co.jp/ir/library/historical_data/pdf/consolidated_sales_e1603.pdf`, 2016. Accessed: 2021-06-04.

[2] Damian Yerrick. Nintendo DS passthrough methods. `https://pineight.com/ds/pass/`, 2005. Accessed: 2021-06-04.

[3] seikene. Find your DS firmware. `https://acidmods.com/forum/index.php?topic=15112.0`, 2008. Accessed: 2021-06-04.

[4] CTurt. Exploiting DS games through stack smash vulnerabilities in save files. `https://cturt.github.io/DS-exploit-finding.html`, 2015. Accessed: 2021-06-05.

[5] DSiBrew. DSi exploits. `https://dsibrew.org/w/index.php?title=DSi_exploits`, 2020. Accessed: 2021-06-05.