What This Video Assumes

- This video assumes you've watched Parts 1-5 of the Video Series and met the assumptions in Parts 1-5 of the Video Series. You'll also need to understand the content of the previous videos thoroughly.
- That you have some version of Python 3 installed. I've got everything working with the latest version of Python, Python 3.10.7. You may download Python 3 here: https://www.python.org/downloads/
- That you've installed the patching plugin into IDA Pro. You may download it here: https://github.com/gaasedelen/patching/releases/tag/v0.1.2
- To install, simply drag and drop the 'patching' folder extracted from the release file into the IDA Pro 'plugins' folder.

Video Contents

- Special Thanks To
- What Are Code Caves?
- Two Methods of Finding Free Space for a Code Cave
- Branching to the Code Cave
- Writing Your Custom Code in the Code Cave
- Turning Your Work into an Atmosphere Code
- Loading Custom Values into Memory

Special Thanks To

- Eiffel2018 for his thread on how to make Code Caves, as well as Patjenova's comments for using multiple 00 bytes in MAIN to insert instructions into. You may find the thread here: https://gbatemp.net/threads/tutorial-2-making-code-cave.603446/
- Tomvita and Snakes of the Edizon SE and tomvita Discord servers for helping me to understand code caves, branches, and installation of the 'patching' plugin.

What Are Code Caves?

- A Code Cave is a blank space in the executable file of a game that we exploit to write our own instructions in. By using blank space, we can write our own instructions without overwriting other parts of game code. This can prevent a lot of glitches and allow us to make our own functions.
- In the previous videos, you were limited to the space of code you had around you and if there was a BLR or LDR present. With code caves, we bypass both limitations. Now, we can write a bunch instructions at once even if space didn't previously allow for it and we can move custom values into a register without any problems.

Example Code Cave

- For this video, we'll be using Pokemon Mystery Dungeon DX as an example. Our instruction of interest controls what Pokemon is recruited and we don't have enough space to insert a MOV instruction to change the value to what we'd want it to be.
- The instruction of interest is located at MAIN+0x02627BA0. It is in a function with two instructions.
- Here is the whole function:
  STR W1, [X0, #0xBC] <- Store Value of Pokemon to Recruited Pokemon Address
  RET <- Return from Function

- The STR instruction stores what Pokemon is being recruited and Register W1 contains the value of the Pokemon being stored. We want to change Register W1 to contain a Pokemon of our choice, but we don't have enough space to place a MOV instruction. We'll fix that.

- Here's what I'd prefer the function be:
  MOV W1, #0x1E8 <- Set Register W1 to 0x1E8, which is Primal Groudon.
  STR W1, [X0, #0xBC] <- Store Primal Groudon to Recruited Pokemon Address
  RET <- Return from Function

Example Code Cave (Cont)

- Basically, what we're going to do is find free space to hold all the code we'd like to execute. Then, we'll change the original STR instruction to a B (Branch) instruction, which will jump to the free space where we stored our code. Finally, we'll write our custom code into that free space. Our function will get executed, then after the code finishes, the game will return to the original function. Basically, we'll be doing this:

Function:
1. B SpeciesHack <- Branch/Jump to Our Code Cave/Function
5. RET <- Return from Function

SpeciesHack:
2. MOV W1, #0x1E8 <- Set Register W1 to 0x1E8, which is Primal Groudon.
3. STR W1, [X0, #0xBC] <- Store Primal Groudon to Recruited Pokemon Address
4. RET <- Return from Function

- The instructions are numbered because that's the order those instructions will execute in. Keep in mind this is an example and we could really make a code cave for any instruction for any purpose. The functions could also theoretically be named anything, as the label doesn't usually matter.
- The purpose of the next few slides, Method 1 and Method 2, will be methods to find free space so we can write our code.

Method 1: Search for 00s in Executable

- First off, make a backup of your game's MAIN file and IDA Pro database file. Then, open the MAIN file of your game. Then click 'View' in the ribbon, go to "Open subviews", and then click 'Segments.' You should see several segments, like .text, .rodata, .data, .bss, and maybe more.
- Look at the segments that have both an R under the R column and an X under the X column. The area could potentially have more letters like an L or W, but as long as these areas have an R and X and not a period in its place, that's a potential area of the game we can use for free space.
- Go back to 'IDA View-A.' Go to 'Search' in the ribbon and click "Sequence of bytes." Now consider to yourself how many instructions you'll have to use in order to make your cheat code. For each instruction you want to write, type "00 00 00 00" in the box that says 'String' plus another "00 00 00 00" on the end. The extra "00 00 00 00" on the end is space for a RET instruction, which we'll use to end our newly created function/code cave. For instance, if you wanted to write two instructions, you'd search for "00 00 00 00 00 00 00 00 00 00 00 00", or twelve '00' bytes, four for each instruction. Eight of those bytes go to the two instructions and the last four bytes go to a RET instruction.
- If you do not put a RET instruction as I suggested at the end of your function/code cave, the game WILL crash as the game expects to see a RET, as otherwise, the function will never end and no other instructions in the game will get executed.
- Set the search to "Find all occurrences" and the search type to 'Hex'. Now, click 'OK'. Give the program time to free space. You may get several results or you may get none. If you do get results, you can only use the '00' bytes found in segments that have both an R and an X, as this allows us to write memory and execute code. I always see the .text segment work for free space, so consider using that. Writing to areas without R and X will result in the game crashing.
- The occurrences tab may say something like 'Instruction' and then ALIGN 0x10, ALIGN 0x20, or even DCB or DCQ. Do not worry about this. If these areas even popped up and were in the appropriate segments, they're valid free space. Copy down the address, go to 'IDA View-A' again, and then jump to the address reported.
- When you see the instruction reported by the occurrences tab, right click it and press 'Undefine.' After pressing Undefine, make sure there are as many DCB 0 lines as there were '00' bytes that you had. For instance, I searched twelve '00' bytes, so I should expect to see twelve DCB 0 lines.
- Go to the at the beginning of the DCB 0 lines, press Ctrl+N. The zeros should change. Then do this for each line of DCB 0 until they're no longer zero, making sure that whatever address you press Ctrl+N on is a multiple of 0x4.
- After finishing, go to original address you jumped to and press C on the keyboard. If it asks you to directly convert to code, say Yes. You'll now notice you have NOPs where the code used to be. Save your database. We'll come back to it.
- If you don't find any matches based on the amount of instructions you want, you'll need to consider shortening the code you want to write or using Method 2, which I will get to.

Method 2: Unused .text Space

- Go to the end of .text and see if there's something that says % 0x?, where ? Is any number in decimal. If that's there, that's free space you can utilize. Right click on it, then click 'Undefine'. Depending on how big 0x? was, there will be just as many % 1 lines. Each one of these % 1 lines is an unused byte. For example, if our MAIN file said 0x1B8, there'd be exactly 440 (0x1B8) lines. Considering each instruction is 4 bytes, and there's 440 lines with one byte each, we can write up to 110 instructions in that amount of free space. The main file I use in this video says % 0x1B8, but the amount of free space will vary from game to game. You may not even have this at the end, suggesting you can only use Method 1.
- For each instruction you want to write, press Ctrl+N on the keyboard on the first % 1 byte and then do it to the next few bytes. I want to write two instructions and a RET on the end, so I'll click Ctrl+N three different times. After you're finished, go to the top of the bytes you just changed and press 'C' on the keyboard. The bytes you changed will turn into NOPs.
- Once again, ending your code cave/function with a RET is necessary, as we have to allow the game to come back from our custom code and start running its own code again. You only need one RET in a single function.

Branching to the Code Cave

- Right-click the first NOP you made in your free space. Then click 'Rename' and give it a name. I want my function to be named SpeciesHack, so that'll be the name I give the first NOP as that'll be where our function starts.
- Go back to the STR instruction, right-click it, then press Assemble. In the Assembly box, Type B and then your label. In this case, I'd type:

  B SpeciesHack

- Copy the bytes from the 'Patching' window. Then, right-click the STR instruction, highlight 'Patching', then click "Change byte...". Delete the first four bytes then paste the copied bytes in. Afterwards, click 'OK'. If B and your label shows up, you did everything correctly. Otherwise, you might need to choose another label name or even a different free space location. This is because sometimes a label is similar to one already in the executable or the code cave is too far from the instruction we're branching from.

Writing Your Custom Code in the Code Cave

- Go to where your labeled NOP is. If you don't remember where it is, you can type in the label when clicking "Jump to address" and IDA Pro will take you there.
- For the first instruction, right-click the labeled NOP and click 'Assemble'. Type in your code into the 'Assembly' box, copy the bytes, right-click the NOP, press 'Patching', delete the first four bytes, and then paste the bytes you copied in and press 'OK'. Do this to each NOP you made an instruction for until you've written all of your instructions out.

Turning Your Work into an Atmosphere Code

- First off, we'll go to the STR instruction we replaced with a Branch. Right-click it and click 'Assemble'. Look at the bytes for the value and copy them down. Once they're copied down, reverse the order of the bytes.
- After that, use Code Type 0 to write those bytes to the instruction address.
- Now, go to where your free space is and get the addresses for those. Click 'Assemble' on each instruction one-by-one, getting the bytes, reversing them, and placing them into Code Type 0 like we did for the Branch instruction. When you've gotten done with each instruction, the code is finished and can be given out to the general public.
- You may wonder how Atmosphere is storing the labels we wrote for our code cave. Truth is, Atmosphere isn't storing that at all and neither is the byte code we copied. The Branch instruction branches to a place relative in memory, so by placing the label name into the 'Patching' plugin, it calculated the relative locations of the current instruction and the code cave and spat out a number that'd get us from the instruction to the code cave. This is why you'll notice the byte code changed for Method 1 versus Method 2 in the tutorial, as the branch address was different, and thus, the calculated relative address was also different. So, if we look back at our branch and place it into ARMConverter, you'll see it may look weird. That's ok, as even if it looks weird, the branch will still work appropriately.

Loading Custom Values into Memory

- There are often times when MOV cannot accommodate the value you'd like to store. For instance, try using MOV X0, #0x12345678. It won't work. Your specified value may be some16-bit, 32-bit, Float, 64-bit, or Double value that MOV doesn't work with.
- Likewise, it can also be annoying to make an ASM Code if the people using it cannot customize it. For instance, in order for the person using our Recruitment Pokemon Modifier to choose a Pokemon other than Primal Groudon, they'd have to use ARMConverter and come up with the assembly code themselves. You can't just have a person who has no idea about atmosphere code going to ARMConverter, typing MOV W1, #0x???, and then expect them to have any idea what to do next.
- This is where the LDR instruction comes into play. Here's a previously unmentioned form of LDR:
  LDR ?n, Label

- Where ?n is W, X, S, or D and n is any number from 0-30.
- The label is basically the same as we did labels for branches. The Patching plugin will calculate a relative location in memory based on the difference from the labeled location and the instruction location. Then, LDR will load the value from that address into the specified register.

Loading Custom Values into Memory (Cont)

- What we're going to do is use Method 1 or Method 2 to find a free space in memory and then write a value of our choice into that free space.
- If we want to load a custom 8-bit, 16-bit, 32-bit value, or float value from free space, we'll need to find a free space address that ends in 0x4 or 0xC.
- If we want to load a custom 64-bit or double value from free space, we'll need to find a free space address that ends in 0x0 or 0x8.
- We have to look for addresses with these endings, because otherwise, the value won't be loaded from the label properly.
- After you find the appropriate free space address, jump to it in memory. You will have to right-click and click 'Undefine' in order to access the appropriate bytes.
- Now, right-click the address, highlight 'Patching', then click "Change byte..." Remove the first four bytes then reverse the bytes of your custom value. Afterwards, paste it in the window then click 'OK'. For instance, if I wanted to write 0x12345678, I'd place 78 56 34 12 into the window.
- Right-click the address and click 'Rename'. Give it a unique name. Then, you can place LDR ?n, Label into your code cave and if the code chances to LDR ?n, =0xYourValue, it worked.
- To make my previous code customizable to the user's liking, I'll replace the MOV instruction I wrote with LDR W1, Label. I'll then copy the bytecode for the corresponding instruction and then replace the Code Type 0 that stored the MOV instruction.

- Finally, I'll write the address we're storing our custom value in and the custom value as a Code Type 0 in Atmosphere. This is the address the LDR instruction will be loading from and can be changed to the user's liking. Code Type 0 will be done slightly differently than usual here to accommodate how LDR works. Code Type 0 will never be written this way otherwise.

- All custom 8-bit, 16-bit, 32-bit, and Float values will be written with the following:
  040000XX XXXXXXX4 YYYYYYYY or 040000XX XXXXXXXC YYYYYYYY.

- All custom 64-bit and Double values will be written with the following:
  080000XX XXXXXXX0 YYYYYYYY YYYYYYYY or 080000XX XXXXXXX8 YYYYYYYY YYYYYYYY.

Sections That Transfer Over to Other Consoles

- Code caves, although found by different methods on different systems, are used by hackers and even malware creators quite a bit.
- For example, the Nintendo Wii C2 "Insert ASM" Gecko Code Type finds an empty space in memory, branches to it, writes custom ASM code, then branches back to game code.