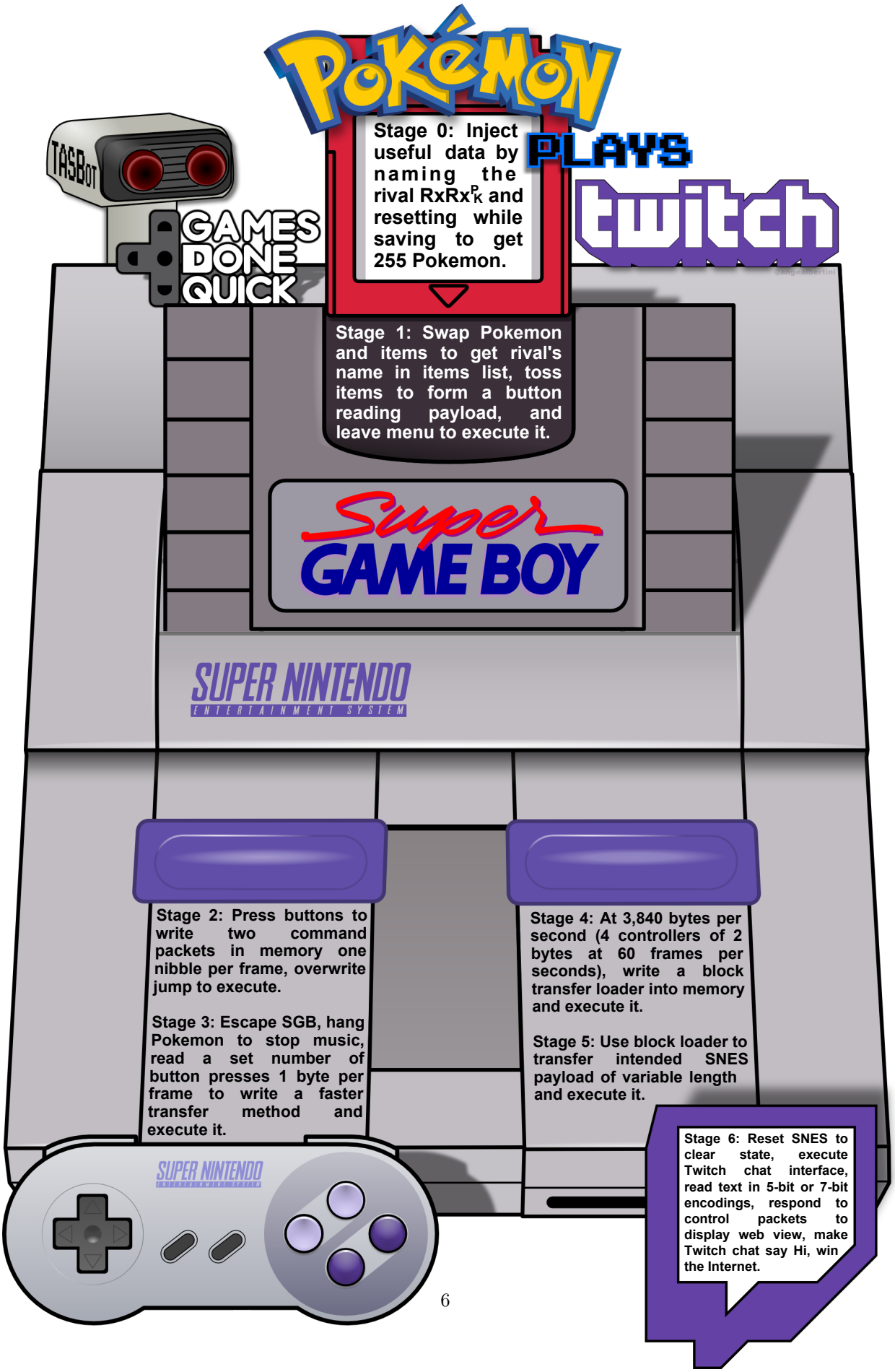

Pokémon
PLAYS
twitch
TASBot
GAMES DONE QUICK
Stage 0: Inject useful data by naming the rival RxRxPK and resetting while saving to get 255 Pokemon.
Stage 1: Swap Pokemon and items to get rival's name in items list, toss items to form a button reading payload, and leave menu to execute it.
Super GAME BOY
SUPER NINTENDO ENTERTAINMENT SYSTEM
Stage 2: Press buttons to write two command packets in memory one nibble per frame, overwrite jump to execute.
Stage 3: Escape SGB, hang Pokemon to stop music, read a set number of button presses 1 byte per frame to write a faster transfer method and execute it.
Stage 4: At 3,840 bytes per second (4 controllers of 2 bytes at 60 frames per seconds), write a block transfer loader into memory and execute it.
Stage 5: Use block loader to transfer intended SNES payload of variable length and execute it.
Stage 6: Reset SNES to clear state, execute Twitch chat interface, read text in 5-bit or 7-bit encodings, respond to control packets to display web view, make Twitch chat say Hi, win the Internet.

3 Pokémon Plays Twitch

by Allan Cecil (dwangoAC), Ilari Liusvaara (Ilari) and Jordan Potter (p4plus2)

For the Awesome Games Done Quick (AGDQ) 2015 charity marathon we exploited a chain of unmodified Nintendo game console components consisting of a Pokémon Red Game Boy cartridge in a Super Game Boy running in a Super Nintendo. We plugged the latter into custom hardware posing as a normal controller. In this *seven*-stage exploit, we corrupted a save file to give ourselves 255 Pokémon, swapped Pokémon, and tossed items to plant shellcode. We committed a series of atrocities using documented command packets and ultimately broke into the Super Nintendo's working RAM, where we wrote our own chat interface to display live contents of the Twitch chat and even a representation of a defaced website.

3.1 TAS'ing a Game to execute Arbitrary Code

TASVideos[2] hosts Tool-Assisted Speedruns of games that are created using an emulator with speed controls such as slow motion and frame advance, along with the ability to save and restore the state of the game (or, rather, of the entire console) at any time. TAS movie files consist of a list all of the button presses sent to the console every frame from the time it is powered on until the game is beaten. It aids our poor human reflexes, but it can do a lot more—like arbitrary code execution!

The first run on the site to use this ability to execute arbitrary code to jump to the credits of a game was Masterjun's Super Mario World run. Later, Bortreb used arbitrary code execution in a run of Pokémon Yellow, marking the first time external content was added to an existing game.

In late 2013, dwangoAC worked with Ilari and Masterjun to present a run at AGDQ 2014 that programmed the games Snake and Pong into Super Mario World on a real console using a replay device (also known as a "bot") from True.[3] This was a huge success and was covered by Ars Technica, but we knew that we could do even more, which ultimately led us to the project described in this article.[4]

3.2 The Game Choice

We started with an end-goal of executing arbitrary code on a Super Nintendo (SNES) using a Super Game Boy (SGB) cartridge as the entry point. We initially planned to use Pokémon Yellow based on Bortreb's exploit in that game, but quickly discovered that the SGB detection routine used by Pokémon Yellow is extremely broken and only worked on a real SGB by pure chance.[5] After looking at other options, we chose to use the Pokémon Red version, which uses a more reliable SGB detection routine that gets us access to the full SGB palette, a custom border, and consistent timing benefits we'll discuss later.[6] Using Pokémon

[2] `http://tasvideos.org`

[3] `http://truecontrol.org`

[4] It should also be noted that all recent AGDQ events have directly benefited the Prevent Cancer Foundation which was a huge motivator for several of us who worked on this project. The block we presented this exploit in at AGDQ 2015 helped raise over $50K and the marathon as a whole raised more than $1.5M toward cancer research, making this project a huge success on multiple levels.

[5] In brief, the detection routine is extremely sensitive to how many DMG clock cycles various operations take; the emulator is likely slightly inaccurate, which causes the detection to fail, but from looking at the behavior it seems like it "just works" on the real hardware. This is sheer luck, and the game developers likely never even knew it was so fragile.

[6] If the SGB BIOS does not find the special codes in the DMG game header that indicate it's an SGB-enabled game (`$146` equal to `$03`), it locks up the command channel until the game is reset, rendering any SGB based exploitation impossible. See `http://gbdev.gg8.se/wiki/articles/The_Cartridge_Header` for more details.

Red also has another added benefit in that the entire game has been skillfully disassembled.[7]

3.3 The Emulator

When we started this project in August 2014, the only emulator capable of emulating an SGB inside of an SNES at a low enough level for our needs was the BSNES emulator. Unfortunately, although BSNES is very accurate at emulating an SNES, it doesn't do a very good job of emulating an SGB. The Gambatte Dot-Matrix Game Boy (DMG) emulator is likewise very accurate, but is unable to emulate an SGB on its own. Ilari was able to create a hybrid emulation core using BSNES to emulate the SNES↔DMG interface chip while using Gambatte for DMG emulation. This was a considerable undertaking, but in the end the emulator was very usable, albeit somewhat slow, as properly emulating the synchronization between the SNES CPU and the DMG CPU is a challenge. Ilari continued to provide emulator development and scripting support throughout the project.

3.4 The Hardware

We didn't just want to exploit a game in the sandbox of a console emulator and call it a Proof of Concept. We wanted to do the job properly and create an actual exploit that would work on real hardware. Only one member of our team (dwangoAC) had all of the required hardware in one place, namely a SNES console, a SGB cartridge, a copy of Pokémon Red, and the replay device posing as a controller, also known as a "bot."[8] Because we wanted to stream data from an attached computer, we opted to use an older, serial-over-USB connected device, namely True's NES/SNES Replay Device. This choice of hardware had a few limitations but worked out well for the project in the end.

Figure 1 – The legendary TASBot

3.5 The Plan

We were initially unsure what kind of payload to create once we had gained the ability to execute arbitrary code on the SNES. Initially we investigated methods of showing crude video, but abandoned it after spending far too much time failing to increase the datarate and running into limits with the processing speed of the SNES's 65C816 CPU. An IRC discussion about Twitch Plays Pokémon[9] led dwangoAC and p4plus2 to brainstorm what it would take to incorporate similar elements into our payload, and the concept of *Pokémon Plays Twitch* was hatched—where a Pokémon character enters a Twitch chat room and "plays" Twitch. In the end, we took it to the next level by giving Red a voice in a chat interface on the SNES and giving TASBot, the robot holding the replay board, the ability to speak through *espeak* and argue with Red. There's much more to say about that, but let's first get to the point where we can execute arbitrary code!

[7] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/pokered-master.zip`

[8] The term "bot" was originally used because it's as if you have a robot playing the game for you; dwangoAC later attached one of these replay devices to a R.O.B. robot as shown in Figure 1 and after presenting Pong and Snake on SMW, the name TASBot came to be associated with the combination as described at `http://tasvideos.org/TASBot`.

[9] A way of crowdsourcing gameplay by parsing commands sent over IRC.

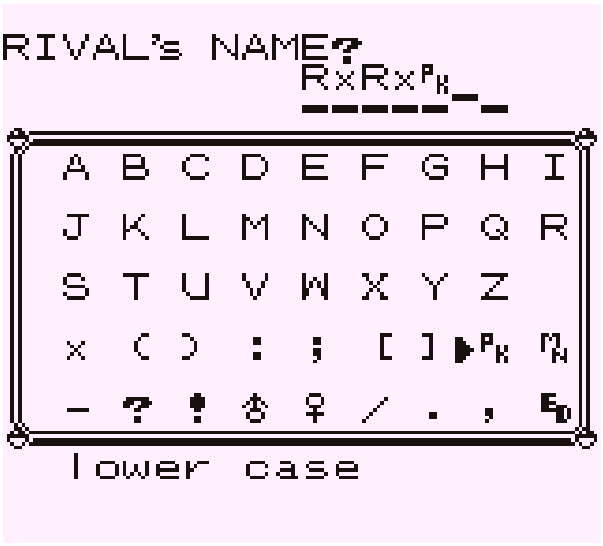


Figure 2 – A strange rival

3.6 Stage 0: Corrupting a save game.

(3–7 bytes per minute.)

We start the game by creating a save file, giving ourselves the default name of Red and naming our rival RxRx as shown in Figure 2. We then save the game as in Figure 3, but reset the console directly after it starts writing to the cartridge's SRAM. There is checksumming on most of the values in the save file but at least one value has no checksum at all, namely the byte at the start of the "party data" that stores the number of Pokémon that have been caught. By some chance, this value in SRAM (at `0xAF2C`, or `0x2F2C` when paged) is initially set to `FF`, so we wait long enough for valid name data and a save file header to be written before resetting. It is possible to do this with human reflexes as the window is approximately 20 ms but we opted to run a wire from our replay device to pin 19 on the expansion port on the underside of the SNES. This allowed us to trigger a reset by shorting the pin to ground, as shown in Figure 3.[10]

3.7 Stage 1: Writing Z80 assembly by swapping Pokémon and tossing items.

(30 bytes per second.)

After loading the game but before changing anything, the initial state of the GBBUS memory map is as follows:[11]

```
0xD163  Number of Pokemon caught,
        corrupted to 0xFF in Stage 0.
0xD164  Pokemon IDs (1 byte each),
        corrupted to 0xFF.
0xD16A  Sentinel byte (0xFF)
0xD16B  Pokemon Data (44 bytes each);
        all are corrupted to 0xFF.
0xD273  Pokemon original trainers;
        all are corrupted to 0xFF.
0xD2B5  Pokemon nicknames;
        all are corrupted to 0xFF.
0xD2F7  Pokemon owned bitmap (19 bytes);
        all zeroes.
0xD30A  Pokemon seen bitmap (19 bytes);
        all zeroes.
0xD31D  Number of items; initially 0
0xD31E  Items array; each entry is 2 bytes,
        an item ID and item count.
        After the last item, there is an FF.
        (Initially located at 0xD31E.)
0xD347  Money as Binary-Coded Decimal.
        (Initially 00 30 00, $3000.)
0xD34A  Rival's name. (Set during Stage 0,
        initially
        91 F1 91 F1 E1 50 00 00 00 00 00.)
0xD355  <misc data>
0xD36E  Map level script pointer.
        (Initially B0 40.)
```

We want to adjust some of these values to create a payload described in the next section, and the game conveniently provides three ways to arrange the state of memory.

- Swapping Pokémon: The game implements moving Pokémon around the list by swapping the raw contents of entries in the ID, Data, Original trainer, and nickname tables, which happens to offset data by an odd amount. Since we have 255 Pokémon instead of the 141 the game was originally limited to we can swap

[10] As with many exploits, the seed for this came from Bortreb's Pokémon Yellow exploit, which itself came from earlier discoveries of others. Masterjun adapted the exploit for Pokémon Red using the BizHawk DMG emulator and dwangoAC took this information and made the Stage 0 and Stage 1 movie file in LSNES.

[11] The same values can be found in the GBWRAM region at an offset of `-0xC000`, so the value for `0xD163` in GBBUS (which isn't visible in the LSNES memory editor) can instead be found at `0x1163` in GBWRAM. GBBUS addressing is used throughout for consistency with existing resources such as the pokered disassembly.

[12] This means the Pokémon data now extends past end of WRAM, and in fact the WRAM should in effect loop around, although this isn't used.

around the contents of anything in WRAM above `0xD164`.[12]

- Tossing items: Throwing away unwanted items decrements the second byte in an item's two-byte ID / Quantity pair. Unfortunately, there are some items that can't be tossed, either because the game prevents tossing them or because doing so softlocks or crashes the game.
- Swapping items: Items can be swapped around in the list of items, which normally just swaps the item data. If you swap two of the same item, the game tries to merge them by adding their counts and then shifting the item list. The shift adjusts the item count and writes a new sentinel item ID. (It doesn't touch either the item count in that slot or the old sentinel.)

Since we don't have any items, let's get some! Swapping the first Pokémon with the tenth causes the `FF`'s located at `0xD16B` through `0xD196` to be written to `0xD2F7` through `0xD322`. Per the memory map, the number of items is located at `0xD31D` and this is changed along with lots of other nearby addresses from `00` to `FF`, which causes the game to think we have 255 items. We eventually enter the item menu and proceed to toss a number of safe items, but—because we can only ever decrement the quantity byte in each item's ID/Quantity two-byte pair—we have to go back and swap Pokémon to make what was once an ID into an item count and vice versa.

In order to avoid softlocking the game, we have to walk through the sequence in a very particular order. There are several items that the game refuses to toss, some that crash the game if you try to toss them, some that can only be thrown once—after which all items afflicted with this condition can no longer be tossed. Some will crash the game simply by being in the menu even if you never even select them.

To work around these pitfalls, we prepare memory by doing several Pokémon and item swaps followed by an initial round of tossing, we go back to swap Pokémon in a way that realigns memory so we can now toss what used to be item IDs. We swap several Pokémon to relocate the Stage 1 code and create a swath of 0's in front of it, and at the very end we swap two identical items to shift memory two spaces back. That's a lot to take in in one sentence, so Figure 4 diagrams the complete list of changes we make showing the value changes as anchored initially from GBBUS `0xD349`.

The primary purpose of all this swapping and tossing is to create a better way to craft our own

[13]The swap where j. is swapped with j. involves the pairs `00 00` and `00 F4`, but they turn into `00 63` and `00 91` because we abuse the fact that the game assumes a quantity of `00` is the same as `FF` and you can only have ninety-nine of any given item in one slot. This results in `FF` + `F4` = `1F3` which is larger than the sum mod 256 dec., at which point the game stores 63 in one

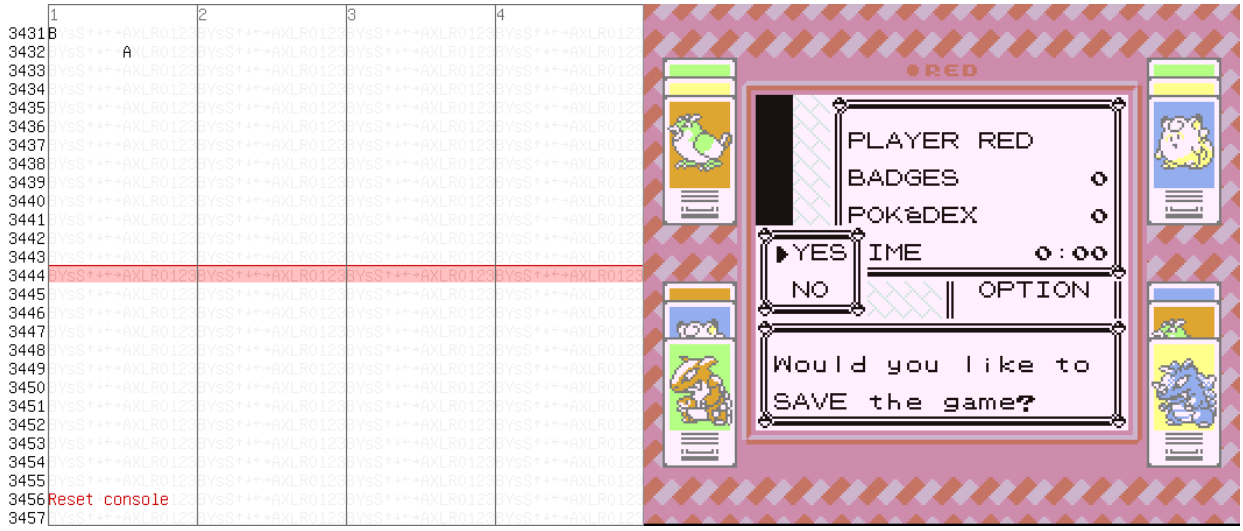


Figure 3 – Corrupting a save game by pressing A to save, then resetting 24 frames later.

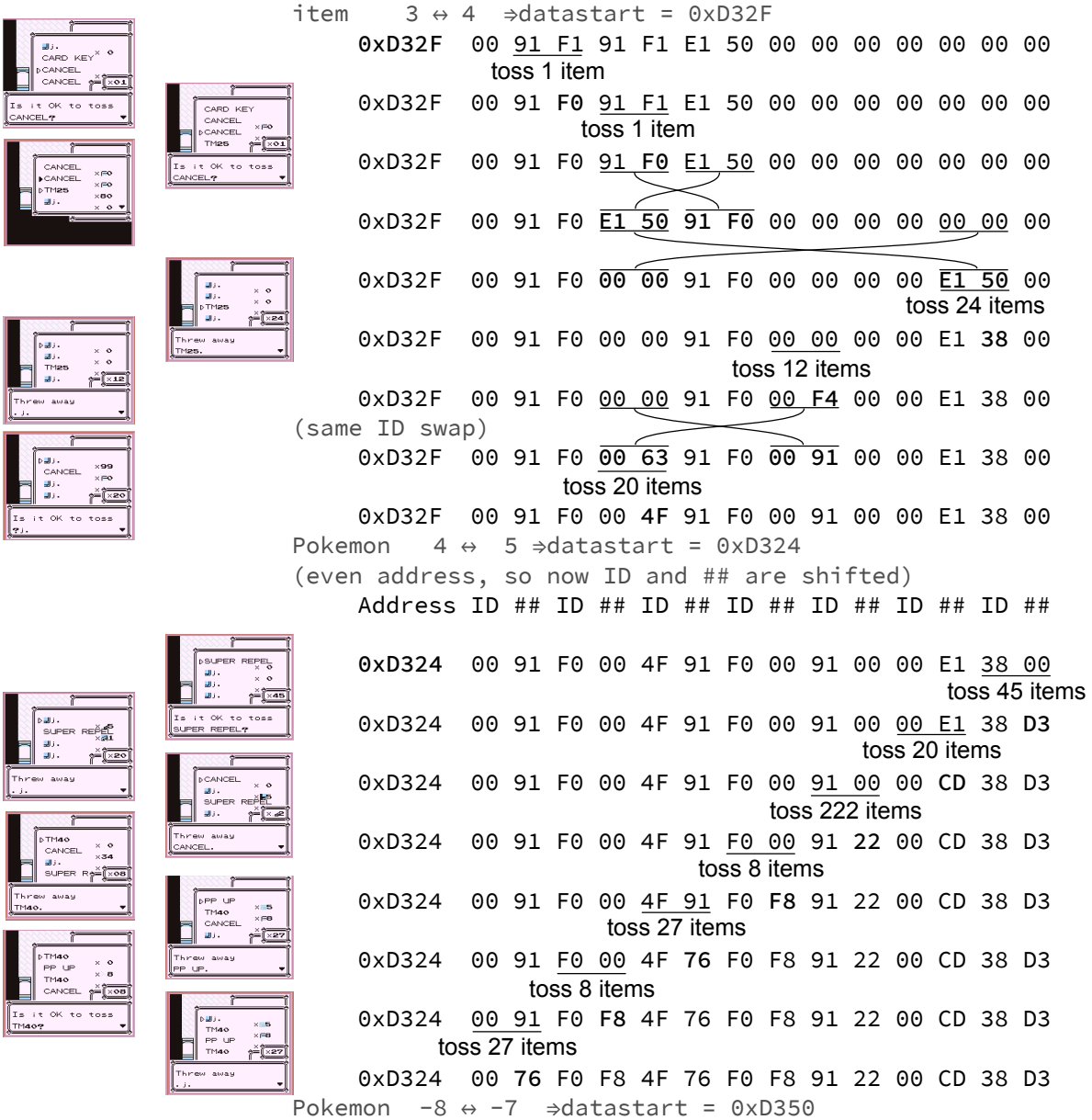


Figure 4 – Pokémon and items are re-arranged in memory to create shellcode.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

Items with these IDs can be tossed
Game refuses to toss items with these IDs
Trying to toss items with these IDs crashes the game
Items with these IDs are initially tossable, but tossing any makes game to refuse to toss more
Just trying to show these IDs freezes the game

Figure 5 – Item IDs can double as Z80 opcodes.

code—as it would be quite tedious to use this method to do anything longer.[13] Here's a disassembly of what we've been able to write so far, starting from 0xD361.

```
0xD362  00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
```

LR35902 shellcode at 0xD361:

```
Player's starting money
        30 00     JR NC,0          // nop
        76        HALT             // wait for frame
        F0 F8     LDH A, (0xF8)    // load input
        4F        LD C,A           // save in C
        76        HALT             // wait for frame
        F0 F8     LDH A, (0xF8)    // load input
        91        SUB C            // decode opcode
        22        LD (HL+),A       // stage2[HL++] = A
        00        NOP
        CD 38 D3  CALL 0xD338      // call stage2
```

Everything up to this point has been the process of writing Stage 1, but now it's time to walk through executing it, although some of the shortcuts we took require a bit of explanation.

First, the reason 0xD361 contains 30 is because the beginning of the Stage 1 data is mostly copied from the field that holds the rival name—which happens to be directly preceded by the player's money. Of this quantity we see the last two out of three bytes represented here in BCD format; the full value 00 30 00 starts at 0xD360. It would take extra effort to eliminate the 30 00 portion, but because that sequence is effectively a NOP, we leave it be.

To reduce the number of bytes that needed to be modified, we used several clever tricks. The code that jumps to this point sets HL to the jump target address, and HL is a canonical pointer register that can be written to. We reused 0xD36E (the map level script pointer) as the loop jump address; upon exiting the menu, the map level script pointer is loaded and called, so it loads the value in 0xD36E into HL and jumps to it.

```
1041   LD HL, 0xD36E
1044   LD A,(HL+)
1045   LD H,(HL)
1046   LD L,A
1047   LD DE, 0x104C
104A   PUSH DE
104B   JP (HL) ; [D36E]
```

Stage 1's purpose is to read the buttons being held down on the controller and write them somewhere, eventually executing what we've written using this slightly more efficient method than twiddling with Pokémon and items. At a high level, this code will read a byte from the controller on one frame, read another byte from the controller on the next frame, subtract the two, store the result at a given memory offset and repeat, successively storing values one byte at a time in order in memory, and ultimately executing said bytes.

The game's NMI (Non-Maskable Interrupt) routine writes a bitmap of the current buttons being held down during each frame (mapped as the buttons ABsSRLUD from lowest to highest bit) to 0xFFF8, and HALT is used to wait for the next frame. Unfortunately, the SGB BIOS cancels out LEFT+RIGHT or UP+DOWN if they are pressed simultaneously and instead converts those bits to 0's. To work around it, our short routine reads two frames and combines the values in a way that can yield arbitrary bytes. Because of restrictions on

item and $190 mod \texttt{FF} = 91$ is stored as the remainder in the other.

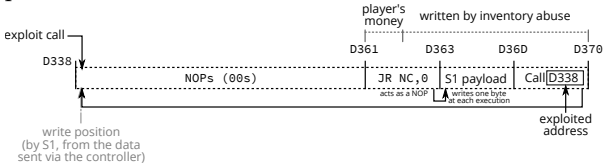
[14] There is no working way to ADD the two reads because we can't write that opcode. Due to byte restrictions, we can't use JP either, but CALL is close enough. See Figure 5.

which bytes we can create, we use `LD C,A` to store the first value and then `SUB C` to combine them.[14]

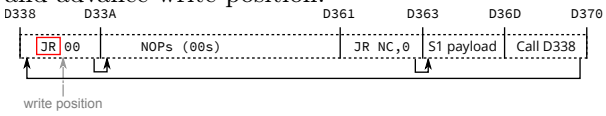
Using this method, we write the following data to `0xD338`, which is executed every frame; that is to say, it is repeatedly executed even before it is fully written!

```
1 18 27 <= first jump
  3E 1C CD AF 00 21 4D D3 CD EB 5F 2E 58 00 CD
     EB 5F 18 FE 79 00 18 00 06 AD 12 42 30
    FB 40 91 18 42 00 00 18 00 00 00 <=
    Stage 2 payload
3 18 D7 <= second jump
```

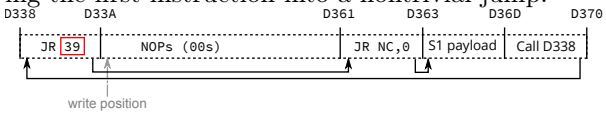
The memory range from `0xD338` to `D360` contains only `00`'s and forms a cascade of harmless `NOP` instructions. This is critical, because this entire section is executed every time a byte is written; this also means we have to be extremely careful with what we write, to avoid executing an incomplete Stage 2 that causes a crash. The solution is to write a jump instruction of `18 27` into the first two bytes—which will skip execution of Stage 2 while it is being constructed. The sequence `18 27` can't be entered in one frame, but fortunately the incomplete form, `18 00`, is effectively a `NOP` instruction. This gives us the full range from `0xD33A` to `0xD360` where we can write whatever we want with impunity, and `HL` points to `0xD33A`.



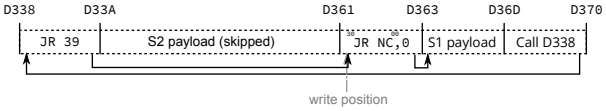
We write `0x18` (`JR x`) into current write position and advance write position:



We write `0x27` into current write position, turning the first instruction into a nontrivial jump.

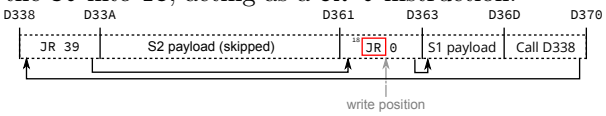


We write the Second Stage to `D33A`–`D360` which is jumped over and not executed. This takes 39 iterations through the loop.



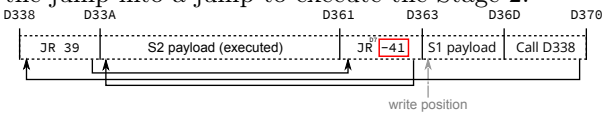
After this, we somehow need to jump to the newly completed Stage 2. The `HL` now points to `0xD360` and the next byte we poke is `18`, which turns the first instruction in the Stage 1 code into `JR 0`, which is still effectively a `NOP`.

We write `18` (`JR x`) to current position, turning the `30` into `18`, acting as a `JR 0` instruction.



We write `D7` into `0xD362`, which modifies the instruction to be `JR -41`, which jumps to `0xD33A`, the start of the second payload. After one more call into `0xD338` and the subsequent jump to `0xD360`, the execution jumps to the Second Stage.

We write `D7 (-41)` to current position, turning the jump into a jump to execute the Stage 2:



One last note before moving on to what Stage 2 will do for us: as with most things in this exploit, entering the Stage 2 payload isn't as straightforward as it should be, and this time it's because the SNES and the DMG run at different clock speeds and framerates. It takes 351,120 cycles for the DMG to run one frame, and 357,366 for the SNES to run one frame. Each side polls the inputs once per their frame, and the SNES side updates the inputs that the DMG side reads once per frame. Since each SNES frame takes slightly longer, the SNES regularly skips updating inputs for one full DMG frame, causing the input to be duplicated.[15]

This clock skew slip happens about every 56 DMG frames. (Sometimes it's 57 frames between slips due to slipping.) It takes a full 86 frames to input the Stage 2 sequence because there are 39 bytes of payload plus 4 bytes total for prologue and epilogue jump instructions, and each byte takes 2 frames to enter as a result of working around L+R and U+D combinations being nulled out. This means we have to cope with at least one clock skew slip and because 86 isn't that much bigger than 2*56

[15]This has implications even outside of TAS'ing: If you hold a button for a single frame you risk that input being lost (if the previous frame had no buttons being pressed, that single frame's press could be overwritten with the no buttons pressed frame from before) or your buttons could be held for an extra frame (even though you let go, you hit right before the skew so your buttons are sent for an additional frame). Both of these behaviors could cause a talented realtime player to question their abilities as they wouldn't have any idea that the console had been the cause of their input being wrong.

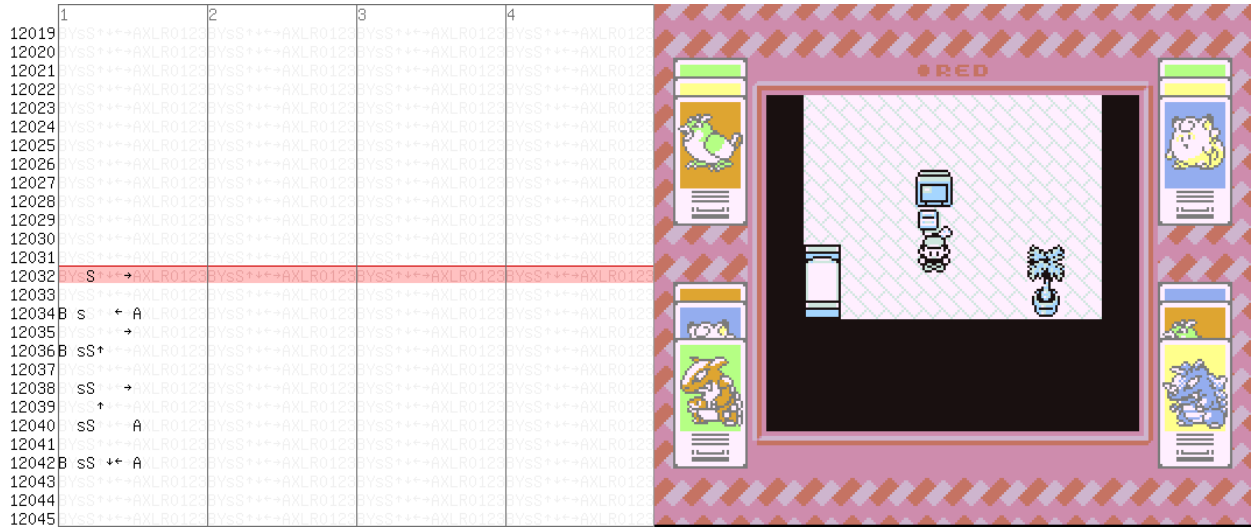


Figure 6 – Sending payload (combos injected by first controller)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

from http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html

Figure 7 – Z80 opcodes that can be sent through SGB input filtering.

the slip position must be relatively near the middle to avoid having to deal with two slips.[16]

3.8 Stage 2: Sending packets to escape SGB from very little space.

We have just 39 bytes to work with in the Stage 2 payload we just wrote and we need to make the most out of every last byte. Fortunately, Pokémon Red already contains a routine that sends a command packet into the SNES. The catch is the code to send that packet is in another ROM bank (`0x1C`) that we need to switch to. While the ROM bank can be switched by a single write, the game NMI routine (which runs every frame) does not save the bank - it switches to one stored in another memory address instead. Two writes are needed to reliably change the bank which would take too much space; however, the common part of ROM (mapped regardless of the bank) has a function that does something, then switches banks and returns. That function makes for a very useful gadget! The entry address for this function is `0x00AF`, with register `A` holding the bank number.

[16]The movie we used was 2(prologue)+5(banksetting)+6(packetsend)+5(packetsend)+1(nop-for-slip)+2(hang)+11(packet1)+7(packet2)+2(unused)+2(epilogue)=43 bytes. We later discovered a different method where the smallest possible extended payload would have been 2(prologue)+5(banksetting)+6(packetsend)+2(hang)+13(packet)+2(epilogue)=30 bytes which is still too much to input without a slip due to our data rate being restricted to one nibble at a time, although the packet data's `0x00` portion could potentially be used for this purpose.

[17]It could be possible to use just one, by putting the NMI routine in a memory-mapped SGB packet register, but we decided not to, as we would need full exploit abilities just to test if this method actually works because the emulator isn't accurate enough to test with.

We need to send two separate command packets, described below.[17] The packets aren't a full 16 bytes in length like they appear to be, but 11 and 7 bytes; the tails of the packets are ignored, so we let the packet payloads overrun into whatever happens to be next. After sending the packets, we have no use for the DMG anymore, so we hang the Z80 by entering a tight loop.

The following Stage 2 assembly code is loaded into `0xD33A`—`D360`.

```
; The gadget takes a new bank number in A.
3E 1C      LD A, #$1C
; Call the bankswitch gadget.
CD AF 00   CALL $00af
; The address of the first packet to send.
21 4D D3   LD HL, packet1
; Call packet send routine.
CD EB 5F   CALL $5feb

; The low byte of address of the 2nd packet.
; used to compensate input slipping.
2E 58      LD L, 0x58
00         NOP
; Call packet send routine.
CD EB 5F   CALL $5feb

18 FE      JR -2       ; Hang the DMG.

packet1: ; 0xd34d
  DB 0x79, 0x00, 0x18, 0x00, 0x06, 0xad,
     0x12, 0x42, 0x30, 0xfb, 0x40

packet2: ; 0xd358
  DB 0x91, 0x18, 0x42, 0x00, 0x00, 0x18,
     0x00, 0x00, 0x00
```

Originally, the `LD L, 0x58 ; NOP` sequence was `LD HL, 0xD358` but we discovered that the transfer routine leaves the upper eight bits of the address in the `H` register at the end of the transfer. The transfer end of the packet at `0xD34D` will be `0xD35D`, so the `H` register will be `D3`, which is exactly the value we want for the next packet, so we can save one byte by just loading the `L` register. The saved byte can taken to be `NOP (00)`.

The repeated input can land on two inputs of the same byte, or the last input of one byte and first input of next. The latter is much better, since for any byte pair, it is possible to construct three valid inputs. However, the first is much worse: The byte will be forced to `00`, and even more unfortunately, the frame rules always cause the duplication to occur in a bad way. The `00` freed from only loading L is close enough to the middle that this byte can be targeted for duplication. It turned out that the emulator doesn't emulate the input slipping quite accurately and we (dwangoAC) had to do a lot of tedious trial and error testing to time the input correctly.[18] The offset between emulator and real hardware turned out to be eight frames, which we adjusted by adding eight frames of no input into the file sent to the bot prior to exiting the menu.

3.9 Exploiting DMG→SGB command packets for gaining a foothold on SNES

The Super Game Boy command packet protocol has two nifty commands for gaining control of the SNES. `0x79` writes arbitrary data to an arbitrary memory location, while `0x91` sets the NMI vector and jumps to an arbitrary address. Both commands are real, documented command packets; they are not undocumented debug commands.

Since the Stage 2 executing on the DMG is so small we needed to minimize the number of packets required. The SNES's controller registers are memory-mapped I/O registers that automatically update each video frame when enabled. It is possible to execute code from those registers but it isn't particularly easy to do so, largely because it is very unsafe to execute anything from those registers when they are in the middle of an update. (There are all sorts of intermediate stages.)

The solution is to find some way for the SNES CPU to waste time during that update elsewhere. The NMI vector and the NMI handler are perfect for this: when enabled, it starts running just before the register starts updating, so we just need an NMI handler that wastes somewhere between roughly 4 and 260 scanlines so it hits after the current NMI returns but before the next NMI starts. Scanning descriptions of various SNES I/O registers, a useful one seems to be `$4212`, which has bit 7 set when the console is performing a vertical retrace. The NMI occurs immediately after the vertical retrace starts and the retrace lasts for about 40 scanlines, so waiting for `$4212` bit 7 to clear works out perfectly. Since the retrace bit is bit 7 and the SNES CPU happens to be in a mode where the `A` regis-

[18] Each blind test took about 5 minutes, as we had to play back the entire movie before reaching the point where we could determine if it worked and we weren't entirely certain it would work at all, but eventually we discovered the correct offset.

[19] Based on the setting of a flags register bit that selects between an 8– and 16–bit `A` register.

ter is 8 bits wide,[19] numbers with bit 7 set show as negative, so it's trivial to branch on those using `BMI` instruction. Handily enough, the `LDA` instruction that loads the memory address into the `A` register sets the condition flags, so we can just loop around that one instruction using `BMI`.

After the loop, we must return from the NMI. This is done using the `RTI` instruction, so the final NMI handler looks like:

```
1 loop:
  AD 12 42 LDA $4212  ;Read 0x4212.
3 30 FB    BMI loop   ;Loop while bit 7 is set.
  40       RTI        ;Return from NMI.
```

This handler trashes the `A` register, which is generally considered bad style, but we can get away with doing that.

We send two packets; the first one writes six bytes (`AD 12 42 30 FB 40`) into the memory address `0x001800`. This is the NMI routine.

```
  79                ; Write Memory
2 00 18 00          ; Target Address
  06                ; Size
4 AD 12 42 30 FB 40 ; Content
```

Figure 8 – Inception

The second one jumps to `0x004218` (which is the start of the controller registers), with the NMI vector set to `0x001800` (which points to the routine we just wrote).[20]

```
91          ; Jump
2 18 42 00  ; Jump Target
  00 18 00  ; NMI Vector
```

3.10 Stage 3: From stable loop in autopoller registers to loading payloads.

(480 bytes per second; 60 payload bytes per second.)

We have transferred control flow to controller registers, but we aren't done just yet. The controller registers are only eight bytes in size, and normally not all bits are even controllable. However, there are some tricks we can play to control all the bits. First, even though a standard SNES controller only has 12 buttons, the autopoller reads all 16 bits. Normally the last four bits are controller type identification bits. Since those bits are read from the controller, the controller can set those bits to whatever it likes, including changing those bits every frame. Second, the last four bytes of the register are read from the second data line that is normally not connected to anything unless there is a multitap device. It isn't possible to just connect a multitap device whenever we like as the game will softlock. Fortunately, it is possible to just connect the second controller so that it shares all the other pins (+5V, ground, latch and clock), but use the second data pin instead the first.

These two tricks allow controlling all 128 bits in the controller registers which gives us 8 bytes of data per frame. While this is a huge improvement over our Stage 1 effective data rate of a nibble per frame it still only amounts to a datarate of 300 bytes per frame because three of those 8 bytes need to be used for looping in the controller registers, leaving only five bytes usable. (Although, as you'll see, only one byte of payload data can be sent per frame.)

Specifically, to loop successfully in the controller registers we need to wait for the NMI induced interrupt in order to avoid the NMI happening at an unpredictable instruction (because the NMI trashes `A`) and then jump to the start of the controller register. Then there is issue that NMI is not initially

[20] We considered putting the NMI code into the SGB packet receive buffer, which is a memory-mapped I/O register (and presumably can be executed by the CPU). We decided against this since the SGB emulation in BSNES is quite questionable and we didn't know if it would work, largely due to the difficulty of testing it.

enabled, even if the handler is set, so the first frame has to enable the NMI handler. Fortunately, this can be done rather compactly:

```
loop:
A9 81       LDA #$81
8D 00 42    STA $4200   ; Set 0x4200 = 0x81 (
    autopoller enabled, IRQ disabled, NMI
    enabled)
CB          WAI
80 F8       BRA loop
```

Since the code is idempotent, this is good time to switch from sending input in once per frame to sending input in once per latch poll. The way the SGB BIOS polls the controllers is completely crazy, often polling more than once per frame, polling too many bits, trying to poll but leaving the latch held high, etc. Because this is a somewhat common problem even in other games, the bot connected to the controller ports has a mode where it synchronizes what input to send based on the edge of each video frame (i.e. 60ths of a second in a polling window) by keeping track of how much time has elapsed; if the game asks for input more than once on the same frame we give it that frame's input again until we know it is time for the next frame's polls, which means we can follow the polling no matter how crazy it is. The obvious tradeoff is that this mode is limited to 8 bytes per frame with 4 controllers attached, so we need to switch the bot's mode to one that is strictly polling based, sending the next set of button presses on each latch. Making that transition can be a bit glitchy considering it was added as a firmware hack but because this piece of code is idempotent we can just spam the same input several times as we only need it to hit in the range. This happens from frame 12117 to 12212 in the movie.

We now have a stable loop in the controller registers that we can use to poke some code into RAM. The five bytes per frame is enough to write one byte per frame into an arbitrary address in first 8kB of the SNES's RAM:

```
LDA #$xx
STA $yyyy
```

This assembles to five bytes, `A9 xx 8D yy yy`. Finally, after the writes, we can use `JML` (four bytes) to jump to the desired address. Since the DMG is still playing some annoying tunes, the first order of business is to try to crash it. Writing `00` to the clock control/reset register at `0x6003` should do the trick by stopping the DMG clock, and in fact this works in the LSNES emulator, but on a real console the annoying tunes keep playing until the DMG corrupts itself enough to crash completely.[21]

3.11 Stage 4: Increasing the datarate even further.

(3840 bytes per second.)

One byte per frame is rather slow as it would take us several minutes to write our payload at that speed so we poke the following routine (Stage 4) that reads 8 bytes per frame from the autopoller registers and writes it sequentially to RAM, starting from `0x1A00` until `0x1B1F` into address `0x19000`.

```
SEP #$30     ;Set 8-bit A and X/Y
LDA #$01     ;Set 0x4200 = 0x01
             ;(autopoller en, NMI dis)
STA $4200
REP #$10     ;Set 16-bit X/Y, keep A 8-bit.
LDY #$1A00   ;Load address to write to.
wait_vblank_start:
LDA $4212    ;Wait until vblank starts.
BPL wait_vblank_start
wait_vblank_end:
LDA $4212    ;Wait until vblank ends, so the
             ;new controller value arrives.
BMI wait_vblank_end
LDX #$4218   ;Start address of controller reg
    .
LDA #$00     ; MVN copies 16-bit amount of
    bytes, even with A being 8 bit.
XBA          ; So ensure that the high bits are
    zero.
LDA #$07     ; A = 7, copy 8 bytes.
PHB          ; MVN changes the data bank
    register, so save it.
MVN $7E,$00     ; Copy the 8 bytes from 0
    x4218 to RAM. Y is auto-incremented.
PLB          ; Restore the data bank register.
CPY #$1B20      ; Have we reached 0x1820?
BNE wait_vblank_start  ; If no, wait a frame
     and read again.
JML $7E1A08     ; Jump to read payload.
```

As machine code, `e2 30 a9 01 8d 00 42 c2 10 a0 00 1a ad 12 42 10 fb ad 12 42 30 fb`

[21] It's not a surprise that it behaves differently in the emulator, as the SGB emulation accuracy in BSNES is questionable in a lot of places; it's possible that the emulator is triggered on a different edge of the clock than real hardware or something similar. Regardless, on real hardware the DMG eventually crashes in a way that makes it stop producing sound and while it's about the equivalent of driving a car into a brick wall instead of hitting the brakes it at least gets the job done.

```
a2 18 42 a9 00 eb a9 07 8b 54 7e 00 ab c0
20 1b d0 e4 5c 08 1a 7e.
```

Why jump to eight bytes after the start of the payload? It turns out that code loads some junk from what is previously in the controller registers on the first frame, so we just ignore the first few bytes and start the payload code afterwards. Eight bytes per frame still isn't fast enough, so the routine this code pokes into RAM is another loader routine that uses serial controller registers to read eight bytes eight times per frame, for total of 64 bytes per frame.

Let's take a look at the Stage 5 payload:

```
; 0000 => Current transfer address.
; 0002 => Transfer end address.
; 0004 => Blocks to transfer.
; 0006 => Current transfer bank.
; 0008 => 0: Transfer not in progress.
;         1: Transfer in progress.
; 000C => Blocks transferred.
; 0010 => Jump vector to next in chain.
; 0020-0027 => Buffer
; 0080-00BF => Buffer.

Start:
NOP         ; 8 NOPs, for the junk at start.
NOP
NOP
NOP
NOP
NOP
NOP
NOP
SEI
LDA #$00    ; Autopoll off, NMI and IRQ off.
STA $4200

REP #$30        ; 16-bit A/X/Y.

LDA #$0000      ; Initially no transfer.
STA $0008

frame_loop:

SEP #$20
not_in_vblank: ; Wait until next vblank ends
LDA $4212
BPL not_in_vblank
in_vblank:
LDA $4212
BMI in_vblank
REP #$20

LDA #$0008
STA $0004
LDA #$0000
STA $000C

rx_block:
LDA #$0001
STA $4016
LDX #$0003
latch_high_wait:
DEX
BNE latch_high_wait
STZ $4016
LDX #$0004
latch_low_wait:
DEX
BNE latch_low_wait

LDA #$0000
STA $0020
STA $0022
STA $0024
```

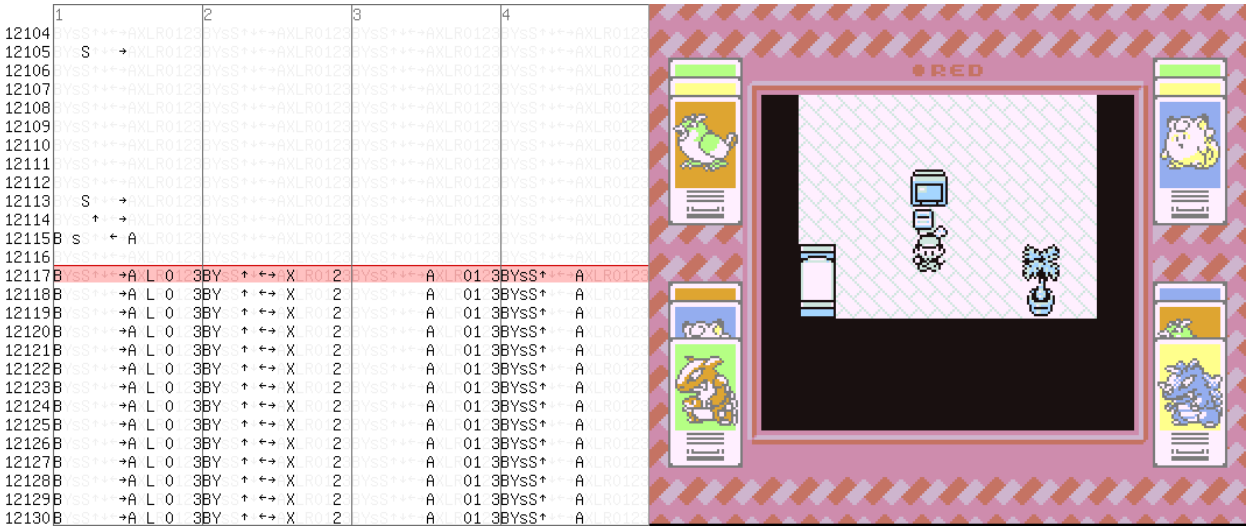


Figure 9 – Now using four controllers!

```
63  STA $0026
    
65  LDY #$0010
    read_loop:
67  LDA $4016
    PHA
69  ; Bit 0 => 0020, Bit 1 => 0024,
    ; Bit 8 => 0022, Bit 9 => 0026
71  BIT #$0001
    BNE b0nz
73  LDA $0020
    ASL A
75  BRA b0d
    b0nz:
77  LDA $0020
    ASL A
79  EOR #$0001
    b0d:
81  STA $0020
    
83  PLA
    PHA
85  BIT #$0002
    BNE b1nz
87  LDA $0024
    ASL A
89  BRA b1d
    b1nz:
91  LDA $0024
    ASL A
93  EOR #$0001
    b1d:
95  STA $0024
    
97  PLA
    PHA
99  BIT #$0100
    BNE b8nz
101 LDA $0022
    ASL A
103 BRA b8d
    b8nz:
105 LDA $0022
    ASL A
107 EOR #$0001
    b8d:
109 STA $0022
    
111 PLA
    BIT #$0200
113 BNE b9nz
    LDA $0026
115 ASL A
    BRA b9d
117 b9nz:
    LDA $0026
119 ASL A
    EOR #$0001
121 b9d:
    STA $0026
123
    DEY
125 BNE read_loop
    
127 ;Move the block from 0020 to its final place
    LDA $000C
129 ASL A
    ASL A
131 ASL A
    CLC
133 ADC #$0080
    TAY
135 LDX #$0020
    LDA #$0007
137 MVN $00, $00

139 ; Increment the counter at 000C,
    ; decrement the count at 0004.
141 ; If no more blocks, exit.
    LDA $000C
143 INA
    STA $000C
145 LDA $0004
    DEA
147 STA $0004
    BEQ exit_rx_loop
149 JMP rx_block
    exit_rx_loop:
151
    LDA $0008
153 BNE doing_transfer
    ; Okay, setup transfer.
155 LDA $0082
    CMP #$FF
157 BMI not_jump
    ; This is jump, copy the address.
159 STA $12
    LDA $0080
161 STA $10
    BRA out
163 not_jump:
    LDA $0080     ; Starting address.
165 STA $0000
    LDA $0082     ; Bank.
167 STA $0006
    LDA $0084     ; Ending address.
169 STA $0002

171 ; Self-modify the move.
    LDX #move_instruction
173 LDA $0006
    AND #$FF
175 STA $01,X

177 ; Enter transfer.
    LDA #$0001
179 STA $0008

181 ; See you next frame.
    JMP no_reset_transfer
183
    doing_transfer:
185
    ; Copy the stuff to its final place in WRAM.
187 LDY $0000
    LDX #$0080
189 LDA #$003F
    PHB
191 move_instruction:
    MVN $40,$00     ; Bogus bank, will be
```

```
    modified.
193 PLB
    TYA
195 STA $0000
    CMP $0002
197 BNE no_reset_transfer
    STZ $0008     ; End transfer.
199 no_reset_transfer:
    ; Next frame.
201 JMP frame_loop
    out:
203 JMP [$10]
```

3.12 Stage 5: Transfers of data in blocks with headers.

(3,840 bytes per second.)

This routine is rather complex, so let's review some of its trickier parts.

The serial protocol works by first setting the latch bit (bit 0) in `0x4016`, then clearing it, then reading the appropriate number of times from `0x4016` (port #1) and `0x4017` (port #2). Bit 0 of the read result is the first data line value, while bit 1 is the second data line value. After each read, the line is automatically clocked so the next bit is read. The two port latch lines are connected together; bit 0 of `0x4016` controls both.

The bot is slow, so we wait after setting/clearing the latch bit. We properly reassemble the input in the usual order of the controller registers, since we have CPU time available to do that. Since we read 16-bit quantities, port `0x4017` is read as high 8 bits, so the data lines there appear as bits 8 and 9.

To handle large payloads, the payload is divided into blocks with headers. Each header tells where the payload is to be written, or, if it is the last block, where to begin execution.

The routine uses self-modifying code: The source and destination banks in MVN are fixed in code, but this code is dynamically rewritten to refer to correct target bank.

3.13 Automating the Movie Creation

Since manually editing, recompiling and transforming inputs gets old very fast when iterating payload ROMs, tools to automate this are very useful. This is the whole reason for having Stage 5 use block headers. Furthermore, to not have one person doing the work every time, it's helpful to have a tool that even script-kiddies can run. The tool to do this is a Lua script that runs inside the emulator (The LSNES emulator has built-in support for running Lua scripts, with all sorts of functions for manipulating the emulator.)

```
1 dofile("sgb-arbitrarywrite.lua");

3 make_movie = function(filename)
          write_sgb_data("stage4.dat");
5         write_8bytes_data("stage5.dat");
          write_xfer_block(filename, 0x8000, 0
      x7E8000, 0x4000, 8);
7         write_xfer_block(filename, 0x10000,
      0x7F8000, 0x7A00, 8);
          write_jump_block(0x7E8051, 8);
9         print("Done");
  end
```

This code, the main Lua script, refers to four external files. "`stage4.dat`" contains the memory writes to load the Stage 4 payload from Section 3.11 while executing in the controller registers.

This file contains the Stage 4 payload, plus the ill-fated attempt to shut up the DMG. (As noted previously, it dies on its own later.) The first line containing `0x001900` is the address to jump to after all bytes are written.

2) "`stage5.dat`", which is the machine code corresponding to the Stage 5 loader.

3) A filename taken as a parameter, which is the payload ROM to use. As you can see, the Lua script fixes the memory mappings, but this is okay, as those are not difficult to modify.

The specified memory mappings copy a sixteen kilobyte byte region starting from file offset `0x8000` into `0x7E8000`, and the `0x7A00` byte region starting from offset `0x10000` into `0x7F8000`. (The first 32kB is assumed to contain initialization code for stand-alone testing, but we don't care about that.)

4) "`sgb-arbitrarywrite.lua`", which is just a function library.

```
--sgb-arbitrarywrite.lua
2 lo = function(a) return bit.band(a, 0xFF);
    end
  mid = function(a) return bit.band(bit.
    lrshift(a, 8), 0xFF); end
4 hi = function(a) return bit.band(bit.lrshift
    (a, 16), 0xFF); end

6 set8 = function(obj, port, controller, index
    , val)
    for i=0,7 do obj:set_button(port,
    controller, index + i, bit.test_all(bit.
    lshift(val, i), 128)); end
8 end
```

```
10
   add_frame=function(a, b, c, d, e, f, g, h,
       sync)
12   local frame = movie.blank_frame();
     frame:set_button(0, 0, 0, sync);
14   set8(frame, 1, 0, 0, b);
     set8(frame, 1, 0, 8, a);
16   set8(frame, 1, 1, 0, f);
     set8(frame, 1, 1, 8, e);
18   set8(frame, 2, 0, 0, d);
     set8(frame, 2, 0, 8, c);
20   set8(frame, 2, 1, 0, h);
     set8(frame, 2, 1, 8, g);
22   movie.append_frame(frame);
   end
24
   write_sgb_data = function(filename)
26   local jump_address = nil;
     local file, err = io.open(filename);
28   if not file then error(err); end
     for i in file:lines() do
30     if i == "" then
       elseif not jump_address then
32       jump_address = tonumber(i);
       else
34       local a, b = string.match(i, "(%w+)%s
         +(%w+)");
         a = tonumber(a);
36       b = tonumber(b);
         add_frame(0xA9, b, 0x8D, lo(a), mid(a)
       , 0xCB, 0x80, 0xF8, true);
38     end
     end
40   add_frame(0x5C, lo(jump_address), mid(
       jump_address), hi(jump_address), 0, 0, 0
       x80, 0xF8, true);
     file:close();
42 end
```

```
44 write_8bytes_data = function(filename)
     local file, err = io.open(filename);
46   if not file then error(err); end
     while true do
48     local data = file:read(8);
       if not data then break; end
50     local a, b, c, d, e, f, g, h = string.
       byte(data, 1, 8);
       add_frame(a, b, c, d, e, f, g, h, true);
52   end
     file:close();
54 end

56 write_xfer_block = function(filename,
       fileoffset, targetaddress, size, speed)
     local file, err = io.open(filename);
58   if not file then error(err); end
     file:seek("set", fileoffset);
60   while size % (8 * speed) ~= 0 do size =
       size + 1; end
     local endaddr = bit.band(targetaddress +
       size, 0xFFFF);
62   --Write the header.
     add_frame(lo(targetaddress), mid(
       targetaddress), hi(targetaddress), 0, lo
       (endaddr), mid(endaddr), 0, 0, true);
64   for i=2,speed do add_frame(0, 0, 0, 0, 0,
       0, 0, 0, false); end

66   --Write actual data.
     for i = 0,size/8-1 do
68     local data = file:read(8);
       if data == nil then data = string.char
       (0, 0, 0, 0, 0, 0, 0, 0); end
70     while #data < 8 do data = data .. string
       .char(0); end
       local a, b, c, d, e, f, g, h = string.
       byte(data, 1, 8);
72     add_frame(a, b, c, d, e, f, g, h, i %
```

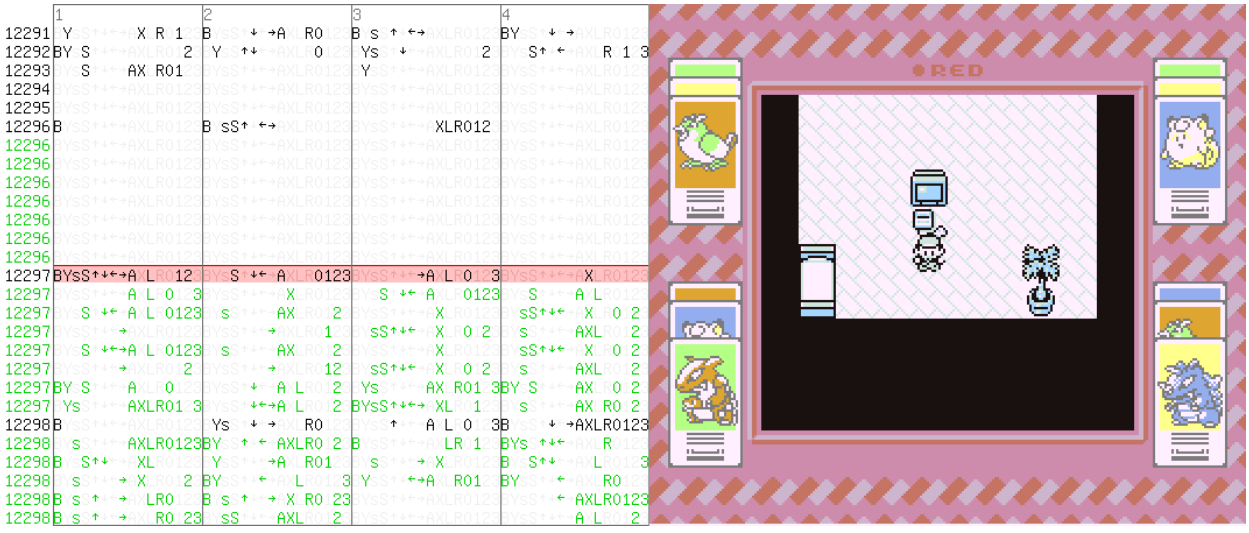


Figure 10 – Why should we wait for next frame? Go sub-frame! (in green)

```
    speed == 0);
  end
  file:close();
end

write_jump_block = function(address, speed)
  add_frame(lo(address), mid(address), hi(
    address), 1, 0, 0, 0, 0, true);
  for i=2,speed do add_frame(0, 0, 0, 0, 0,
    0, 0, 0, false); end
end
```

This script assumes that the loaded movie causes the SNES to jump into controller registers and then enable NMI, using the methods described earlier. It appends the rest of the stages and payload to the movie. Also, since it edits the loaded input, it is possible to just load state near the point of gaining control of the SNES and then append the payload for very fast testing. (Otherwise it would take about two minutes for it to reach that point when executing from the start.)

3.14 Stage 6: Twitch Chat Interface

After successfully transferring our payload, execution of the exploit payload (created by p4plus2) can officially begin. There are three primary states to the final payload: (1) Reset, (2) the Chat Interface, and (3) a TASVideos Webview.

3.14.1 The Reset

Because much of the hardware state is either unknown or unreliable at the point of control transfer we need to initialize much of the system to a known state. On the SNES this usually implies setting a myriad of registers from audio to display state, but also just as important is clearing out WRAM such that a clean slate is presented to the payload. Once we have a cleared state it is possible to perform screen setup.

In the initial case we set the tile data and tilemap VRAM addresses and set the video made to `0x01`, which gives us two layers of 4–bit depth (Layers 1 and 2) and a single layer of 2–bit depth, Layer 3.

Layer 1 is used as a background which displays the chat interface, while Layer 2 is used for emoji and text. Layer 3 is unused. A special case for the text and emoji however is Red's own text which is actually present on the sprite layer, allowing code to easily update that text independently.

3.14.2 The Chat Interface

Now that we have the screen itself set up and able to run we need to stream data from Twitch chat to the SNES. But we only have 64 bytes per frame available to support emoji as well as the alphabet, numbers, various symbols, and even special triggers for controlling the payload execution. This complexity quickly bogged down our throughput per frame, so we created special encodings for performance! On average the most common characters will be `a-z` in lower case, which conveniently fit into a 5–bit encoding with several more character to spare.

The SNES has both 16–bit and 8–bit modes, so in 16–bit mode we can easily process three characters with a bit to spare! But what about the rest of our character space? Well, we have a single bit remaining and can set it to allow the remaining characters to be alternatively encoded. The alternate encoding allowed for two 7 bit characters, with an additional toggle bit on the second character.

```
BXXXXXXX XXXXXXXX
if(E) goto special_encoding
if(!E) goto normal_encoding
    normal_encoding:
        0AAAAABB BBBCCCCC
        A = full character 1
        B = full character 2
        C = full character 3
    special_encoding:
        1XXXXXXX SXXXXXXX
        if(S) goto special_command
        if(!S) goto read_two_characters
        read_two_characters:
            1AAAAAAA 0BBBBBBB
            A = full character 1
            B = full character 2  (used for
    Red's text)
        special_command:
            1AAAAAAA 1BBBBBBB
            A = full character 1
            B = Command byte
```

Figure 11 – Twitch chat!

The most important command was `EE`, chosen very arbitrarily, which meant "transition state." The state transition would then toggle between the TASVideos website and chat interface. Also worth noting is that any character with a value of `00` was considered a null character and was not displayed for synchronization purposes.

3.15 The Website

The website itself is not very complicated, rather just interesting to mention to take advantage of mode `0x03` which allowed us to render a 256–color image, rather than the standard 16–color images from the prior section. The only caveat was that we had to make a quick tool to remove duplicate tiles to optimize the tile data to fit in VRAM. Background colors were controlled by tweaking the palette data rather than the image itself, as the SNES is very poor at manipulating raw tile data due to its planar pixel format.

3.16 Outside of the SNES

The bot was connected to the console through the controller ports and a single wire going to the reset pin on the expansion board, meaning that from an external perspective the hardware was completely unmodified. The bot itself was connected by a USB serial interface to a MacBook Pro running Linux. The source of the button presses being sent to the bot was in the form of a continuous bitstream representing the state of all buttons for each frame. Once the payload was fully written and the Twitch chat interface was complete the bitstream transitioned from being pre-created movie content to a bitstream in the format the chat interface payload needed it in, with 5-bit and 7-bit encodings for characters and emoji. This was controlled by the python scripts[22] that relied on a script to identify when Red, the player inside of the Pokémon Red game, said various things. The script also triggered things that TASBot, the robot holding the replay device, would say via the use of espeak, which allowed us to create a conversation between TASBot and Red.

Finally, as part of the script we predefined periods where we would "deface" the TASVideos website by changing it to different colors; this worked by showing an image on the SNES as well as literally defacing the actual website. Finally, the script was built with the ability to send commands to a serial-controlled camera, but truth be told we ran out of time to test it so we used a bit of stage magic to pretend like Twitch chat was interacting with the camera by typing directions to move it, and we had a helpful volunteer running the camera for us.

3.17 Live Performance

These exploits were unveiled at AGDQ 2015. They were streamed live to over 100,000 people on January 4th with a mangled Python script that didn't trigger the text for Red properly, then again on January 11th with the full payload. The run was very well received and garnered press coverage from Ars Technica[23] among others and resulted in substantially more interest in TASBot and the art of arbitrary code execution on video games than had existed previously. Most importantly, the TAS portions of the marathon where the exploit was featured helped raise over fifty thousand dollars directly to the Prevent Cancer Foundation. Overall, the project was a resounding success, well worth the substantial effort that our team put into it.

[22] `https://github.com/TheAxeMan301/PptIrcBot`
[23] Pokémon Plays Twitch: How a Robot got IRC Running on an Unmodified SNES by Kyle Orland