Bachelor Thesis

May 18, 2025

Understanding arbitrary code execution

A case study on Pokémon Emerald

Sasha Toscano

Abstract

Abstract goes here ... You may include up to six keywords or phrases. Keywords should be separated with semicolons.
Keywords:

Advisor:
Carlo Alberto Furia
Co-advisor:
Marc Langheinrich

Approved by the advisor on Date:

Contents

1 Introduction

In the world of software, security is one of the most critical concepts as it is the foundation of trust in the software we use every day. Let it be something as simple as our browser remembering our passwords, or as complex as a bank's online banking system, we need to be able to trust that the software we use is secure and that our data is safe and well protected. This is especially important in this day and age where we rely on these types of software for everything: from communication, to banking, and even to entertainment.

The software we use is often complex and interconnected, making it difficult to ensure what is secure and what isn't. In many cases, these programs are often developed quickly and with limited resources, and as a result, security is often an afterthought in the development process, leading to oversights or the introduction of vulnerabilities that can be exploited by attackers.

One of the most common types of these vulnerabilities is arbitrary code execution (ACE), which allows a malicious user to execute some arbitrary code on a target system or on a specific application. This can happen because software or computer systems in general are not capable of differentiating between some generic text and actual commands, without some proper protections put in place.

This then becomes a significantly serious issue as it can lead to unauthorized access to sensitive information, data loss, and even complete system compromise. These types of ACE vulnerabilities can be found in a wide range of software: including operating systems, web applications, and even video games. In fact, some very dangerous ACE vulnerabilities have been found in video games like *Super Mario World*, where they have been used to allow users to literally "program" new games into the system mid-run.

In this thesis, we will explore the concept of ACE using a case study of the game *Pokémon Emerald* (2004) to demonstrate how such vulnerabilities can be exploited. We will then examine the current state of ACE-related attacks and discuss modern techniques that will be used to mitigate them. We will also look at some of the most common techniques used to protect against these vulnerabilities, and how they can be prevented.

2 Arbitrary code execution

I'd like to start with a simple explanation of what arbitrary code execution (ACE in short) is: *updog*. And what is updog, you may ask? *Not much, what's up with you?*.

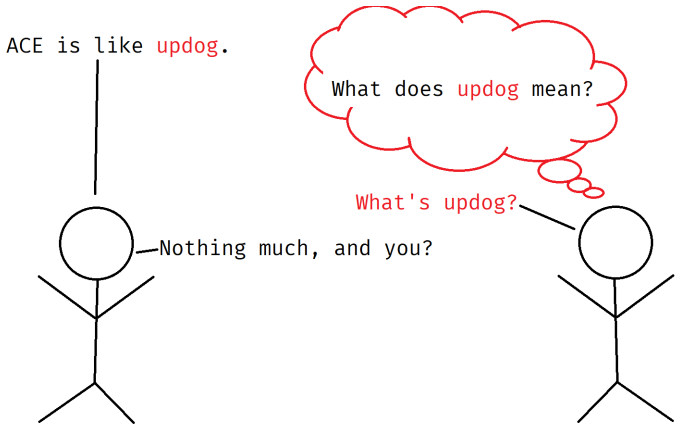

Figure 1. A graphical illustration of the *updog* joke

This joke, illustrated in Figure 1 (credits to Smoovers[1]) is an extremely simple but direct way of explaining what ACE is. In this case the joke is self-explanatory, but the idea behind it is that we end up saying something (*what's up dawg?*) that is not what we intended to say (*what does updog mean?*). This is extremely similar to what happens during ACE in the real world, where we end up executing code that we did not intend to execute, albeit with dramatically more complicated consequences.

To get a bit more into the proper definition of things: ACE is a type of flaw or vulnerability that allows an attacker to execute some (generally speaking) malicious code on a target system. Without proper protections and preventions put in place, an attacker can easily exploit these to gain access to sensitive information, take control of the system, or even cause damage to the system itself. The severity of this incidents can vary greatly: in the past these exploits have gone from something low-level like allowing gamers to better their speedrunning performances (where speedrunning

[1] https://www.youtube.com/watch?v=-XmXYCXX7y4

is the act of playing a video game with the goal of completing it as fast as possible), to something a fair bit more complicated, like leaking sensitive kernel memory on real world systems, as was the case in Retbleed[2].

These vulnerabilities can be exploited through various means, including buffer overflows, code injection, and other techniques; however in the context of this case study, we will be looking at a specific example of ACE in the world of video games, specifically in the game *Pokémon Emerald* where the techniques used to exploit the vulnerabilities are **Arbitrary memory write** and **out-of-bounds writes**, which is where existing mechanics are used to place crafted instructions into writable areas.

3 Pokémon Emerald: A Case Study of Arbitrary Code Execution

In this section, we explore how players can manipulate the inner workings of *Pokémon Emerald* to execute their own code within the game, a phenomenon that occurs through ACE. This process is made possible by chaining together several in-game glitches that interact in unintended ways as seen in figure 2, ultimately allowing the player to influence memory and control the game's behavior.

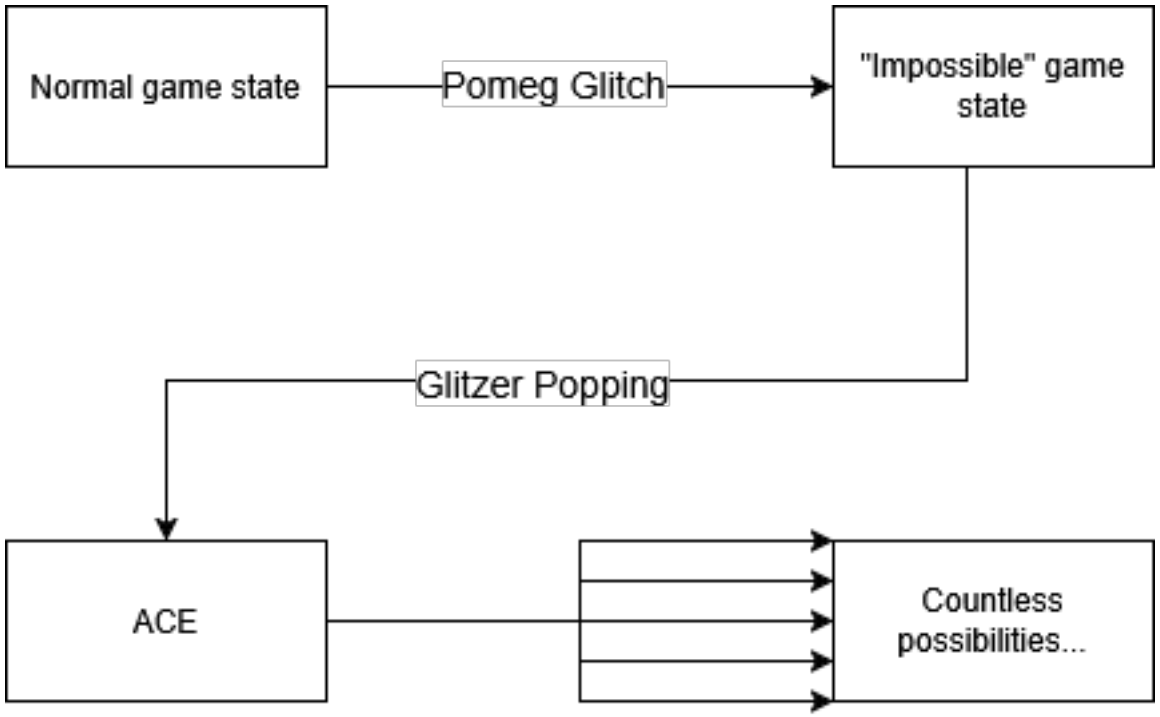

Figure 2. A scheme illustrating the process to achieve ACE

The starting point for this chain is the *Pomeg glitch* (Chapter 3.1), a glitch that allows the player to reduce a Pokémon's HP below zero outside of battle, causing the game to enter an invalid state. When attempting to initiate a battle under these conditions, the game creates a placeholder Pokémon to avoid crashing, unknowingly opening the door to deeper exploitation. By examining this placeholder in battle, players can set up conditions to trigger another glitch known as *Glitzer Popping* (Chapter 3.2), which involves renaming PC storage boxes in a precise way. This process, along other steps, ultimately leads the game into treating the renamed data as if it were valid Pokémon data, and eventually as executable code. This setup causes memory to be misread and reinterpreted and with careful planning, players can direct the game to areas of memory that contain useful data, or even custom payloads, by taking advantage of how the game processes corrupted or invalid Pokémon entries. While this avoids the need for external tools or hacking devices, rendering this types of manipulations doable on any type of hardware, the level of control granted is surprisingly deep, even able to altering game logic and spawning items or Pokémon at will. This can become a serious issue, as, although technically impressive and often used for speedrunning, this type of ACE could have lead to a monetary loss for the Pokémon Company. This is because, when the game originally came out in 2004, the company used to hold special in-person events where some rare Pokémon were given out to players (and this used to be the only way to get these Pokémon), as such, for anyone to be able to "generate" them through a couple of glitches it would be a problem for the company, as the players did not need to attend this highly publicized events to get the Pokémon.

[2]https://comsec.ethz.ch/research/microarch/retbleed/

3.1 The Pomeg glitch

In Pokémon Emerald, the player can use a glitch called *Pomeg glitch* to put the game in an impossible state. To explain this glitch first it would be wise to understand how the game roughly works. In Pokémon Emerald the player can be in two states: the exploration state, where they can go around the map with a team of up to six Pokémon which can be caught in, and the battle state, that is where the fighting happen. Any pokemon caught after the sixth gets sent in the PC which is a storage system that allows the player to store Pokémon that they do not want to carry with them. The Pokémon have their own stats, which improve based on their levels and what other Pokémon they fight. The statistic involved with the Pomeg glitch is *HP* (Hit Points), which is the amount of health a Pokémon has. If a Pokémon's HP reaches 0, it faints and cannot be used until it is revived and if all of them reach 0, we get a game over and this can only happen when the game is in the battle state.

The HP statistic is calculated through the following formula:

$$\text{HP} = \left(\frac{(2 \times \text{Base} + \text{IV} + \left(\frac{\text{EV}}{4}\right)) \times \text{Level}}{100} \right) + \text{Level} + 10 \tag{1}$$

This formula gives us the maximum HP of a Pokémon, which is the maximum amount of health it can have. In the case of the *pomeg glitch*, the only relevant variable is the EV value: this is because the EV value is the only statistic that can be directly modified based on the actions of the player. It is a number between 0 and 255, and it increases every time a Pokémon defeats another, but most importantly it can also be decreased by using certain items. In the case of the *pomeg glitch*, the player can use a specific item called *Pomeg berry* to lower the EV value of a Pokémon by 10. However, if, for example, the Pokémon's current HP was at 1, when updating the newly calculated HP statistic, both the current HP and max HP get lowered and as such, the game could set the current HP value to 0, or, even worse, something below it, like underflowing to $2^{16} - 1$ or 65535 HP (due to the HP statistic being an unsigned two-bytes integer). Now, to use a *Pomeg berry* the player needs to not be in the combat state, and this can create some issues because the game is only capable of handling game overs (which happens when all of the player's Pokémon reach 0 HP) if the player is in a combat state. The problem arises in the situation where the Pokémon whose HP is being updated is the only one with $HP > 1$, because the player may ends up with a party of only dead Pokémon, as can be seen in figure 3.

Figure 3. A technically impossible state

This is not a problem right as it happens, however if the player is to start a battle, the game will try to look for a Pokémon to fight with, and it will find none, because all of them are dead. So after unsuccessfully looking for a Pokémon, the game will then send out a **?-like** Pokémon (commonly known in the Pokémon community as **Decamark**) as can be seen in Figure 4, which is be used to prevent game crashes.

At this point, the player can try to: fight, run away or use an item, all options which would result in a white out (the game over mechanic). But how can this lead to ACE? This is where the glitch known as *Glitzer popping* comes into play.

3.2 Glitzer Popping

The *Pomeg data corruption glitch*, most commonly known as the *Glitzer popping*, is a glitch that allows the player to execute some arbitrary code by renaming the boxes to a specific string, which will then be interpreted as some ARM-Assembly code that will be executed when the game tries to read the name of the box. The player can rename the boxes in many different ways, obtaining different results spanning from obtaining event-exclusive items or Pokémon

Figure 4. The decamark Pokémon

(for Pokémon Emerald many events were held in-person at some specific locations), to modify player information or game flags, all the way to being able to execute custom scripts (like creating your own events). "But how do we get the game to read the name of the box to the point where it gets executed?" is the most logical follow-up question. The answer is simple: we need to use the *Pomeg glitch*.

3.2.1 The setup

First things first, this thesis is not a tutorial on how to achieve this glitches, as there are countless fantastic other guides about replicating them (like this video made by Youtuber PapaJefé[3], or this guide made by Mickaël Laurent[4]), rather, my purpose is to illustrate and figure out why and how does ACE happen, using Pokémon Emerald as a basis for it.

To actually start executing some arbitrary code we need two things: the first one is the previously illustrated *Pomeg glitch*, where the player is able to setup an impossible game state, and the second one needs the Boxes in which the Pokémon that don't fit in the player's team are stored. These are 14 different boxes in which the Pokémon are stored, and they can be renamed by the player. The names of these boxes are stored in a specific area of the memory, and this is where *Glitzer popping* comes into play: as previously stated it is a technique that allows the player to execute some arbitrary code by renaming the boxes to a specific string, which will then be interpreted as some code that will be executed when the game tries to read the name of the box (an example can be seen in Table 1).

Box	Name
Box 1	(VTTnFMBn)
Box 2	(EEENJRo)
Box 3	(EE5. . . q)
Box 4	(EFF. . .)
Box 5	(KT?nTR?n)
Box 6	(EEEIP?n)
Box 7	(EE'FQm)
Box 8	(EmFlo)
Box 9	(yLRom"Ro)
Box 10	(EEEFGEn)
Box 11	(EE . . . q)
Box 12	(E. . . -n)
Box 13	(FQRn. . . Rn)
Box 14	(EEEt ?n)

Table 1. Example box names required to teleport to map ID `0B10`

The game is not capable of handling the situation where the player has a Decamark in their team, and as such when the player tries to open their team information in battle it will unsuccessfully try to read the Pokémon's data from the memory, leaving instead a blank Pokémon.

[3] https://www.youtube.com/watch?v=45kwOEhKbUg
[4] https://e-sh4rk.github.io/ACE3/

So when the player moves the cursor to the first slot in their team, the cursor actually underflows and goes to slot 256 (since a team is made up of a maximum of 6 Pokémon) and as such, instead of scrolling through slots 1-6, we player has access to slots 255 and above (as the cursor is technically out of bounds). However, slots outside the first 6 aren't meant to be Pokémon slots, so the game accesses random blocks of RAM data and treats them as Pokémon, even though they are not. In this case the 255th party slot is actually the PC Pokémon data and continuing to scroll upwards allows the player to actually go over memory addresses reserved for various in-game data.

Each time the party Pokémon selection pointer moves to a new party slot, an anti-cheat verification routine is triggered for the selected "Pokémon" (because the game thinks it is, but it's actually not). If the checksum of the selected data block (interpreted as a Pokémon) is invalid, the game modifies it into a Bad Egg, which is the result of the anti-cheating protocol, a bad egg is what the game sees as a "cheated" or invalid Pokémon. This transformation involves setting the Egg Status flag to 1 and enabling two additional bits, which designate the Egg as a "Bad" Egg. Since the memory blocks being interpreted as party Pokémon are not genuine Pokémon structures, their checksums will almost always be invalid—unless the slot is empty.

3.2.2 Pokémon data structure

The Egg Status flag can reside in one of four different locations within a Pokémon's data structure. It is part of one of the four substructures that make up a Pokémon which can be seen in figure 5 (credits to Bulbapedia[5]), and the order of these substructures is determined by the Pokémon's Personality Value (PID modulo 24). Because these substructures are encrypted using the Pokémon's PID and the Trainer ID (TID), setting the Egg Status flag to 1 can result in either a bit value of 1 or 0, depending on the result of `PID` XOR `TID`. In contrast, the two bits that define a "Bad" Egg are always located at fixed positions, and they are consistently set to 1 when a checksum is invalid. These bit modifications are what cause memory corruption in RAM. However, since only three bits are affected within a block of 100 bytes, the resulting corruption is minimal.

Growth			Attacks			EVs & Condition			Miscellaneous		
	size (bytes)	offset (bytes)		size (bytes)	offset (bytes)		size (bytes)	offset (bytes)		size (bytes)	offset (bytes)
Species	2	0	Move 1	2	0	HP EV	1	0	Pokérus status	1	0
Item held	2	2	Move 2	2	2	Attack EV	1	1	Met location	1	1
Experience	4	4	Move 3	2	4	Defense EV	1	2	Origins info	2	2
PP bonuses	1	8	Move 4	2	6	Speed EV	1	3	IVs, Egg, and Ability	4	4
Friendship	1	9	PP 1	1	8	Special Attack EV	1	4	Ribbons and Obedience	4	8
Unused	2	10	PP 2	1	9	Special Defense EV	1	5			
			PP 3	1	10	Coolness	1	6			
			PP 4	1	11	Beauty	1	7			
						Cuteness	1	8			
						Smartness	1	9			
						Toughness	1	10			
						Feel	1	11			

Figure 5. Pokémon Data Structure by Substructure

The substructure order, as previously said, is defined by a Pokémon's personality value (which is a random unsigned 32-bit integer that is generated at the exact moment a Pokémon gets 'created' in the game) modulo 24, the result is then mapped to the structure seen in table 2 where G stands for growth, A is attacks, E is EV & condition and M is miscellaneous.

[5]https://bulbapedia.bulbagarden.net/wiki/Pokémon_data_substructures_(Generation_III)

00. GAEM	06. AGEM	12. EGAM	18. MGAE
01. GAME	07. AGME	13. EGMA	19. MGEA
02. GEAM	08. AEGM	14. EAGM	20. MAGE
03. GEMA	09. AEMG	15. EAMG	21. MAEG
04. GMAE	10. AMGE	16. EMGA	22. MEGA
05. GMEA	11. AMEG	17. EMAG	23. MEAG

Table 2. Substructure order based on PID%24

An additional layer of randomness is introduced by the DMA (Direct Memory Access) system, which is another built-in anti-cheat mechanism that shifts the RAM addresses of numerous data structures whenever the player performs actions such as entering a battle, going through a doorway, or opening the in-game Bag. The DMA remaps memory addresses using a translation table of multiple double-words. Any value subjected to DMA can occupy up to 32 different memory addresses, each spaced by 4 bytes (a double-word).

Importantly, party Pokémon are not affected by DMA, which means the memory addresses of the six standard party slots remain fixed. However, any data read from memory locations beyond the sixth slot *is* subject to DMA. Since each party Pokémon occupies 25 double-words, and DMA remapping allows up to 32 double-word shifts, each double-word in a slot beyond the sixth could potentially be placed at an address susceptible to corruption via the Egg Status bit alterations. Due to the variability in both RAM content and corruption locations, these elements may interact unpredictably, sometimes preventing a given double-word from being corrupted by the Egg-related flags.

Through the use of carefully planned strategies (like the *Glitzer Popping*), it becomes possible to intentionally corrupt specific values within the data of a Pokémon, while simultaneously ensuring that surrounding data remains unaffected, thus enabling a form of highly targeted or pinpoint corruption. This capability allows for the precise manipulation of a Pokémon's PID (Personality ID) and/or TID (Trainer ID) within the PC storage system, without altering the remainder of the Pokémon's structured data. Since both the PID and TID serve a crucial role in encrypting Pokémon's four internal substructures, with each one containing different categories of the Pokémon's information, the corruption of these values causes a dramatic shift in the Pokémon's checksum, which functions as a form of data integrity verification.

Among the known corruption methods, the two specific bits responsible for creating so-called "Bad Eggs" are not suitable for intentional and controlled Pokémon data corruption, as they invariably fail to preserve the checksum, rendering the resulting data unstable and largely unusable. In contrast, a more nuanced method (corruption via the Egg State Flag) offers a much more promising avenue, as it allows for manipulation while keeping the checksum intact. This method alters the checksum by a multiple of 0x4000, and because the checksum is stored as a 16-bit word, any change that corresponds to an even multiple leaves the checksum effectively unchanged. Although there exist certain rare conditions under which this multiple might be odd, thus compromising the checksum, these conditions can be easily identified and prevented, allowing for a consistently reliable corruption method that keeps the data within safe bounds.

Since the PID governs the order in which the four data substructures are stored and subsequently read by the game, corrupting it directly alters this order, resulting in the game interpreting one category of data as another: for example, reading what is meant to be the Moves substructure as if it were the EVs substructure. This misinterpretation of substructure order becomes a powerful tool for altering multiple facets of a Pokémon's attributes such as its species, held item, experience points, individual values (IVs), effort values (EVs), friendship level, obedience flag, origin data, and of course, its moveset, simply by assigning specific values to those fields prior to the corruption process. Although there are theoretically 24 different ways to reorder the four substructures, only 10 permutations are actually possible within the game's mechanics when doing a corruption. These ten permutations are collectively referred to as Corruption Types, and each one produces unique effects based on how the substructure data is reorganized, meaning that the specific impact of a PID corruption is entirely determined by which Corruption Type is applied.

However, even when using the Egg State Flag corruption technique, which successfully preserves the Pokémon's checksum and alters the substructure order as intended, the change to the encryption key (caused by modifying either the PID or TID) can still introduce undesirable side effects. These side effects include transforming the Pokémon into an Egg, assigning glitched or unusable moves to the second and fourth move slots, or disrupting other internal values that affect gameplay. The presence of a corrupted Pokémon in an Egg state poses a significant drawback, as the act of hatching resets many of its attributes and, in the case of certain Glitch Pokémon, can result in game crashes or freezing. Furthermore, corrupted move slots can restrict the player from using, viewing, reordering, or replacing the affected moves, thus limiting the Pokémon's functionality and strategic use.

To mitigate these problems and produce a clean, stable result, an advanced technique involves performing two successive *Glitzer Popping* procedures, first corrupting the PID, and then the TID (or vice versa) in a way that ensures

the Pokémon ends up with a valid checksum, a desired change in substructure order, and, importantly, a restoration of its original encryption key through the reversal of the PID or TID alteration. This dual-step approach ensures that the corrupted Pokémon retains its intended modifications while eliminating residual glitched values, effectively enabling the precise and consistent corruption of nearly any Pokémon without the drawbacks typically associated with such manipulations.

Returning to the main topic, it's actually possible to switch some important bytes of a Pokémon's own data structure (IVs and the species ID) leading to the creation of a stable "glitch Pokémon", stable because the checksum doesn't fail due to the corruption being controlled. In this case the corruption of the "Bad egg" will be minimal and as such the egg will end up being able to be hatched normally. This glitchmon, upon being hatched, will try to play its sprite animation (as every Pokémon does when being hatched *Pokémon Emerald*); certain glitchmons have these animations data actually stored in the RAM, close the the Boxes data. When trying to play this sprite animation, the game will try to figure out which sprite animation to play by moving the cursor to the address listed in the Pokémon's data, however, due to the Pokémon itself being corrupted, the address ends up being outside the normal table of the animation sprites, and due to how low-level programming languages work the program doesn't fail if it reads something that is outside of its intended bounds, it actually reads something "random". However, this "random" data ends up pointing at the beginning of the boxes names, and is actually the code that we have written in the boxes names.

3.2.3 **Running the code**

When a Pokémon egg hatches, a small animation gets played, and the game will then try to read the Pokémon data from the memory. This is where the code gets executed: when the game tries to read and then execute the code corresponding to the animation, instead it ends up reading and trying to execute the name of the boxes. This is ultimately where, by naming the boxes in a specific way (such as seen in table 1), the game will execute this code, which can stretch from generating an unavailable Pokémon all the way to being able to run your own custom events.

As seen in table **??**, the code that gets executed ultimately ends up actually being a series of ARM assembly instructions, which are then executed by the game. The game is capable of executing these instructions because it is built on top of an ARM architecture, which is a type of CPU architecture that is commonly used in mobile devices and embedded systems. The ARM architecture is a low-level programming language, which is what allows the game to execute these instructions directly, and they are executed in a specific order and they are written in a specific format that is then decoded by the CPU and executed.

4 Detecting and Preventing Arbitrary Code Execution

4.1 Detection

4.2 Prevention

Instruction	Opcode (Hex)	Character Used	Description
`sbc r12, lr, #0xD00`	E2CECED0	–	R12 = LR - 0xD00 - 1
`adcs r12, r12, #0x3000000`	E2BCC7C0	–	R12 += 0x3000000 (offset setup)
`-filler-`	BFBFBFFF	–	Padding
`bic r12, r12, #0xC8000000`	E3CCC4C8	–	Clear upper bits for alignment
`-filler-`	BFBFFF00	–	Padding
`ldrt r11, [r12, #0xA6]!`	E5BCB0A6	–	Load map ID address into R11
`-filler-`	BFFF0000	–	Padding
`mov r12, #'F'`	E3B0C04F	'F'	From Box 4 → Load base character
`-filler-`	FF000000	–	Padding
`adc r12, r12, #'K'`	E2ACCE4B	'K'	From Box 5 → Add next character
`adc r12, r12, #'T'`	E2ACCC54	'T'	From Box 5 → Add next character
`-filler-`	BFBFBFFF	–	Padding
`adc r12, r12, #'I'`	E2ACC049	'I'	From Box 6 → Final character added
`-filler-`	BFBFFF00	–	Padding
`strh r12, [r11, #4]`	E1CBC0B4	FKTI	Store constructed map ID
`-filler-`	BFFF0000	–	Padding
`mvn r12, #0xE1`	E3E0C0E1	–	Prepare to build opcode
`-filler-`	FF000000	–	Padding
`bic r12, r12, #0xED00000`	E3CCC6ED	–	Clear bits
`bic r11, r12, #0x1000000E`	E3CCB2E1	–	Prepares opcode for 'bx r0'
`-filler-`	BFBFBFFF	–	Padding
`adcs r12, pc, #0x30`	E2BFC1C0	–	Compute destination address
`-filler-`	BFBFFF00	–	Padding
`strt r11, [r12]!`	E5ACB000	–	Store 'bx r0' instruction
`-filler-`	BFFF0000	–	Padding
`adc r12, lr, #0xB0000`	E2AECAB0	–	Adjust base pointer
`-filler-`	FF000000	–	Padding
`sbc r12, r12, #0x30000`	E2CCCBC0	–	Subtract offset
`sbc r12, r12, #0xB00`	E2CCCEB0	–	More alignment
`-filler-`	BFBFBFFF	–	Padding
`adc r0, r12, #0xE8`	E2AC00E8	–	Final call to 'CB2_LoadMap2ENG'

Table 3. Explanation of the commands seen in Table 1

5 Summary/Conclusions

- https://www.sciencedirect.com/science/article/abs/pii/B9781597496537000104
- https://thesecmaster.com/blog/how-to-fix-the-cve-2021-40444
- https://www.okta.com/identity-101/arbitrary-code-execution/
- https://catonmat.net/ldd-arbitrary-code-execution
- https://www.wallarm.com/what/arbitrary-code-execution-vulnerabilities
- https://comsec.ethz.ch/research/microarch/retbleed/
- https://azeria-labs.com/writing-arm-assembly-part-1/
- https://arxiv.org/abs/2301.13760
- https://dl.acm.org/doi/10.1145/1920261.1920269
- https://glitchcity.wiki/wiki/Pomeg_glitch
- https://glitchcity.wiki/wiki/Pomeg_data_corruption_glitch
- https://projectpokemon.org/home/docs/other/notable-breakpoints-r31/
- https://bulbapedia.bulbagarden.net/wiki/Pomeg_glitch
- https://glitchcity.wiki/wiki/Arbitrary_code_execution
- https://problemkaputt.de/gbatek.htm#gbamemorymap
- https://tasvideos.org/6616S#GlitzerPopping
- https://bulbapedia.bulbagarden.net/wiki/Pok%C3%A9mon_data_substructures_(Generation_III)