Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

April 3, 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 the ability 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 in detail, using a case study of the game *Pokémon Emerald (2004)* to illustrate how these vulnerabilities can be exploited and to understand the importance of creating state of the art protections against this type of attacks. After which we will discuss the state of the art in ACE vulnerabilities and how they can be mitigated. 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?*.

This is an extremely simple but direct way of explaining what ACE is (credits to Smoovers for the joke). 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 (don't like this, gotta rephrase), 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 allowing gamers to better their speedrunning performances (where speedrunning is the act of playing a video game with the goal of completing it as fast as possible), to leaking sensitive kernel memory on real world systems, as was the case in Retbleed.

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 - case study

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 full party of dead Pokémon. This is not a problem at the second 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 to fight with, the game will then send out a **?????????** (commonly known in the Pokémon community as **Decamark**), 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). The other option is the more interesting one, where a glitch referred to *Glitzer Popping* can happen.

3.2 Glitzer Popping

Glitzer popping is a technique that allows the player to execute arbitrary code in the game

3.3 What actually goes on behind the scenes

4 State of the art chapter

4.1 What is the situation today

4.2 Current studies on security and improvements on protections

To insert a figure use the following command:

Figure 1. The caption of my figure

5 Conclusion

The experimental result goes here ... [1].

---

[1]https://www.usi.ch

6   Future work

7   Summary

Future works goes here.