LACTF-2024 - Reversing aplet321

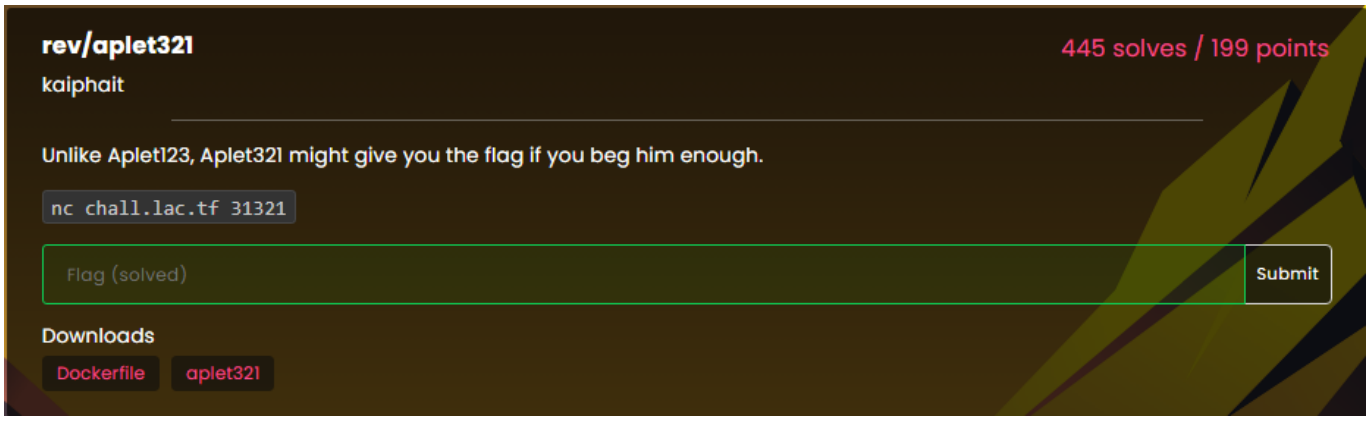
github.com/anre18

21.02.2024

Contents

1 Challenge-Introduction

The following Write up is intended as an short overview for this years LACTF-Reversing-Challenge **aplet321**.
The Event contained a bunch of CTF-Challenges from all typical Categories from Rev to misc. For more information about the event, look here on the Official Website. All the Events-Challenges can be found on the Official LACTF Github Repostitory.
I myself attended the Event online as a member of our University's IT-Security team, and primarily focused on the Reversing and Pwn Challenges (of which I intend to publish a few write-ups in the next days)



The Challenge itself starts with a short introductory Text: *Unlike Aplet123, Aplet321 might give you the flag if you beg him enough.*
This is a great reference to the conclusion of this Challenge, as well as to the pwn Challenge called *aplet123*.

After that, the connection information follows for the remote connection, which will be needed later to solve the challenge at the end.
The only remaining information we get, are the Challenge Executable, called **applet321** and one Dockerfile for setting up the Challenge (not strictly needed).

1.1 Running the Binary

To get a grasp on what we're dealing with, we simply run the binary.
We are greeted with an introductory message informing us that we are dealing with *"aplet321"*, and then asked if they can help.
Now we are prompted to enter Text. After entering a simple Message, we just get the reponse: *so rude*.

```
└─$ ./aplet321
hi, i'm aplet321. how can i help?
I'd like to get the Flag
so rude
```

2 Reversing

2.1 Initial-Checks

To get an more in-depth understanding about the given executable we start with an simple file check.

```
└─$ file aplet321
aplet321: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=b6322155d8e3d5e
cbc678a2697ccce38be0e7c10, for GNU/Linux 3.2.0, not stripped
```

This gives us the information, that we're dealing with an simple **unstripped 64-Bit ELF Executable** in **little-Endian**.
As we are dealing here with an typical Reversing, and not an Pwning Challenge, we can simply skip the normal checks for Binary-Security Mechanisms, as we already now, that the Binary isn't **stripped**, and therefore should be relatively straight forward to Reverse-Engineer with an "Decompiler" used by Ghidra, IDA, or similar tools.
I'm also gonna skip the other typical checks for **strings, function-calls** and so on, as they wont lead to anything more useful.
Therefore, I'm going to continue with the static code analysis, as this will give us a complete picture of the program we're dealing with.

2.2 Static Code Analysis

To staticly Analyse the Binary, I went straight up to Ghidra. The following contains the already decompiled Code, as well as the renamed(by me) Variable Descriptors.

```
1  int main(void)
2  {
3    int substring_check;
4    size_t user_input_Length;
5    char *user_Input_Reference;
6    int please_Counter;
7    int pretty_Counter;
8    char user_input;
9    char acStack_237 [519];
10
11   setbuf(stdout,(char *)0x0);
12   puts("hi, i\'m aplet321. how can i help?");
13   fgets(&user_input,0x200,stdin);
14   user_input_Length = strlen(&user_input);
15   if (5 < user_input_Length) {
16     please_Counter = 0;
17     pretty_Counter = 0;
18     user_Input_Reference = &user_input;
19     do {
20       substring_check = strncmp(user_Input_Reference,"pretty",6);
21       pretty_Counter = pretty_Counter + (uint)(substring_check == 0);
22       substring_check = strncmp(user_Input_Reference,"please",6);
23       please_Counter = please_Counter + (uint)(substring_check == 0);
24       user_Input_Reference = user_Input_Reference + 1;
25     } while (user_Input_Reference != acStack_237 + ((int)user_input_Length - 6));
26     if (please_Counter != 0) {
27       user_Input_Reference = strstr(&user_input,"flag");
28       if (user_Input_Reference == (char *)0x0) {
29         puts("sorry, i didn\'t understand what you mean");
30         return 0;
31       }
32       if ((pretty_Counter + please_Counter == 0x36) && (pretty_Counter -
             please_Counter == -0x18)) {
33         puts("ok here\'s your flag");
34         system("cat flag.txt");
35         return 0;
36       }
37       puts("sorry, i\'m not allowed to do that");
38       return 0;
39     }
40   }
41   puts("so rude");
42   return 0;
43 }
```

The Program primarily consists of an, relative short, main function. The first few lines are simple **Variable initializations**, followed by the **First Text-Prompt** in *line 12*.

After that our Text-Response, from stdin, is saved inside an Variable, for up to 0x200(dec512) Characters. (*Line 13*)
This input then gets **length-checked** against the **Value 5**. If our input is longer than that, the program **continues**, otherwise it will straight up **end** and prompt the *"so rude"* response, we're already familiar with. (*Lines 14-15*)
Therefore we **definitely** want to **input more that 5 Characters**. The following Part assumes, that that's the

case.

The program continues with multiple Variable assignments (*lines 16-18*), which will come up later.
After that a loop begins, which iterates over our input(*lines 19-25*).
This loop checks for the sub-strings "**pretty**"(*Line 20*) and "**please**" (*Line 22*), and counts them up (*Lines 21, 23*), within the previously assigned Variables.
When the Loop finishes, the program checks the Variable containing the **Amount** of **"please"** (*Line 26*). When its **zero**, we get the typical "so rude" Message, ending the program, when not, the Program continues.

After that, a check, for the substring **"flag"**, follows(*Line 27*). When unsuccessful (aka. we **didn't insert the String "flag"** as part of our input), we get an different **Error** Message: *sorry, i didn't understand what you mean*, followed by the stop of the Program (*Lines 29-30*).

So, currently we already know, that for an successful run (up to this point), we atleast need to insert more than **5 Characters**, and part of our input has to **contain** the words *please* and *flag*, which combined already satisfies the 5 Character check.

Now the Program starts with the final and most important check:

- The **Sum** of *prettys pleases* needs to be exactly **0x36(dec54)**
- AND: The **difference** of *prettys* and *please* needs to be exactly **-0x18(dec -24)**

Or as formula:

$$Pretty's + Please's = 54$$
$$Pretty's - Please's = -24$$

As we can see we've got an simple Linear system, we could solve in multiple ways (It's simple math). Nevertheless when solved, we get: **15** for the amount of prettys, and **39** for the amount of pleases needed, to satisfy this requirement.

When satisfied, the Program will congratulate us with a Text prompt, followed by the **printed Flag**. (*Lines 33-34*)

If we **fail** the check, we just get another **Error** Message, followed by the Program **ending**. (*Line 37*)

3 Retrieving the Flag

After the Static-Code-Analysis we finally know everything needed, to get the Flag printed:

- We need to use **more than 5** Characters (*Line 15*)
- The String **flag** has to be part of the input (*Lines 27-28*)
- The String **pretty** has to be used exactly **15** times
- The String **please** has to be used exactly **39** times

- And all of this within **0x200(dec512) Characters** (*Line 13*)

The first requirement is automatically fulfilled by the others, and given the requirements, the last one should also be easily achievable.
All we have to do, is to craft a String containing the word flag, the word pretty, 15 times, and the word please 39 times. And we should get the Flag.
One such String could look like this:

```
prettyprettyprettyprettyprettyprettyprettyprettyprettyprettyprettyprettyprettyplease
pleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepl
easepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleaseple
asepleasepleasepleasepleasepleasepleaseflag
```

When we establish a connection to the remote endpoint using the provided netcat connection and input our string, we successfully retrieve the flag.

```
└─$ nc chall.lac.tf 31321
hi, i'm aplet321. how can i help?
prettyprettyprettyprettyprettyprettyprettyprettyprettyprettyprettyprettyprettypleasepleasepleasepleasepleasepleasep
leasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepleasepl
easepleasepleasepleasepleasepleasepleasepleasepleasepleaseflag
ok here's your flag
lactf{next_year_i'll_make_aplet456_hqp3c1a7bip5bmnc}
```

4 Conclusion

Overall this Challenge was quite simple, but still fun. It didn't require the deepest Programming knowledge, and was quite short and contained. It also did not need any kind of specially crafted code to extract the Flag.

Therefore, it was a great introductory Challenge for the more Complicated Challenges later in the CTF, and I would recommend it to anyone who'd like to get their first experience into Reversing. It was really Beginner friendly and I'd hope to see Challenges like this one again, for LACTF-2025.