Report for login at 2024-06-11

Summary

Enumeration

Binary Information

File Name	Path	Format	Bit
login	app/testFile/login	ELF	32-bit

Security of the Binary

Basic Security Features			
Linked	**Stripped**	**RELRO**	**Canary**
dynamically linked	no	partial	no

Advanced Security Mechanisms		
NX	**PIE**	**RPath**
no	no	no

Security Meta-Information		
RunPath	**Symbols**	**Fortify Source**
no	yes	no

Strings

- Enter admin password:
- pass
- Correct Password!
- Incorrect Password!
- Successfully logged in as Admin (authorised=%d) :)
- Failed to log in as Admin (authorised=%d) :(
- login.c
- .note.gnu.build-id

Vulnerable Functions

- gets
- printf

Libraries

- linux-gate.so.1
- libc.so.6
- /lib/ld-linux.so.2

Assembly Code

```
xor ebp, ebp
pop esi
mov ecx, esp
and esp, 0xfffffff0
push eax
push esp
push edx
call 0x80490b3
add ebx, 0x2f70
lea eax, [ebx - 0x2d40]
push eax
lea eax, [ebx - 0x2da0]
push eax
push ecx
push esi
mov eax, 0x8049192
push eax
call 0x8049070
hlt
mov ebx, dword ptr [esp]
ret
nop
nop
nop
nop
```

```
nop
ret
nop
nop
nop
nop
nop
nop
nop
nop
mov ebx, dword ptr [esp]
ret
nop
nop
nop
nop
nop
nop
mov eax, 0x804c028
cmp eax, 0x804c028
je 0x8049110
mov eax, 0
test eax, eax
je 0x8049110
push ebp
mov ebp, esp
sub esp, 0x14
push 0x804c028
call eax
add esp, 0x10
leave
ret
lea esi, [esi]
nop
ret
lea esi, [esi]
lea esi, [esi]
```

```
nop
mov eax, 0x804c028
sub eax, 0x804c028
mov edx, eax
shr eax, 0x1f
sar edx, 2
add eax, edx
sar eax, 1
je 0x8049158
mov edx, 0
test edx, edx
je 0x8049158
push ebp
mov ebp, esp
sub esp, 0x10
push eax
push 0x804c028
call edx
add esp, 0x10
leave
ret
lea esi, [esi]
ret
lea esi, [esi]
cmp byte ptr [0x804c028], 0
jne 0x8049180
push ebp
mov ebp, esp
sub esp, 8
call 0x80490e0
mov byte ptr [0x804c028], 1
leave
ret
lea esi, [esi]
ret
lea esi, [esi]
lea esi, [esi]
```

```
nop
jmp 0x8049120
lea ecx, [esp + 4]
and esp, 0xfffffff0
push dword ptr [ecx - 4]
push ebp
mov ebp, esp
push ebx
push ecx
sub esp, 0x10
call 0x80490d0
add ebx, 0x2e57
mov dword ptr [ebp - 0xc], 0
sub esp, 0xc
lea eax, [ebx - 0x1ff8]
push eax
call 0x8049060
add esp, 0x10
sub esp, 0xc
lea eax, [ebp - 0x12]
push eax
call 0x8049050
add esp, 0x10
sub esp, 8
lea eax, [ebx - 0x1fe1]
push eax
lea eax, [ebp - 0x12]
push eax
call 0x8049030
add esp, 0x10
test eax, eax
jne 0x804920c
sub esp, 0xc
lea eax, [ebx - 0x1fdc]
push eax
call 0x8049060
add esp, 0x10
```

```
mov dword ptr [ebp - 0xc], 1
jmp 0x804921e
sub esp, 0xc
lea eax, [ebx - 0x1fca]
push eax
call 0x8049060
add esp, 0x10
cmp dword ptr [ebp - 0xc], 0
je 0x804923b
sub esp, 8
push dword ptr [ebp - 0xc]
lea eax, [ebx - 0x1fb4]
push eax
call 0x8049040
add esp, 0x10
jmp 0x8049250
sub esp, 8
push dword ptr [ebp - 0xc]
lea eax, [ebx - 0x1f80]
push eax
call 0x8049040
add esp, 0x10
mov eax, 0
lea esp, [ebp - 8]
pop ecx
pop ebx
pop ebp
lea esp, [ecx - 4]
ret
nop
push ebp
call 0x80492c1
add ebp, 0x2d9a
push edi
push esi
push ebx
sub esp, 0xc
```

```
mov ebx, ebp
mov edi, dword ptr [esp + 0x28]
call 0x8049000
lea ebx, [ebp - 0xf0]
lea eax, [ebp - 0xf4]
sub ebx, eax
sar ebx, 2
je 0x80492b5
xor esi, esi
lea esi, [esi]
sub esp, 4
push edi
push dword ptr [esp + 0x2c]
push dword ptr [esp + 0x2c]
call dword ptr [ebp + esi*4 - 0xf4]
add esi, 1
add esp, 0x10
cmp ebx, esi
jne 0x8049298
add esp, 0xc
pop ebx
pop esi
pop edi
pop ebp
ret
lea esi, [esi]
ret
mov ebp, dword ptr [esp]
ret
```

Code Analysis

Pseudo C Code

main.c

```
Revised Code:



/* WARNING: Function: __x86.get_pc_thunk.bx replaced with
injection: get_pc_thunk_bx */

undefined4 main(void)

{
  int comparisonResult;
  char adminPasswordInput[6];
  int isAuthorized;
  undefined *stackPointer;

  stackPointer = &stack0x00000004;
  isAuthorized = 0;
  puts("Enter admin password: ");
  gets(adminPasswordInput);
  comparisonResult = strcmp(adminPasswordInput,"pass");
  if (comparisonResult == 0) {
    puts("Correct Password!");
    isAuthorized = 1;
  }
  else {
    puts("Incorrect Password!");
  }
  if (isAuthorized == 0) {
    printf("Failed to log in as Admin (authorised=%d) :
(\n",0);
  }
  else {
    printf("Successfully logged in as Admin
(authorised=%d) :)\n",isAuthorized);
  }
  return 0;
}
```

Description

Explanation:

The provided code snippet is a simple C program that prompts the user to enter an administrator password and checks if the entered password is correct. If the correct password "pass" is entered, it prints "Correct Password!" and sets the authorization flag, **isAuthorized**, to 1. If the password is incorrect, it prints "Incorrect Password!". Depending on the value of **isAuthorized**, it then prints whether the login attempt as an admin was successful or not.

Security issues: 1. Use of **gets**: The function **gets** is unsafe because it does not check the size of the input, which can lead to buffer overflow vulnerabilities. An attacker could exploit this to execute arbitrary code. 2. Hardcoded password: The password "pass" is hardcoded into the source code, which is not secure. A better approach would be to securely hash and store passwords. 3. Lack of input validation: There is no validation on the user input, making it vulnerable to buffer overflow. A safer alternative, such as **fgets**, which limits the number of characters read, should be used. 4. Plain-text comparison: Using **strcmp** to compare passwords in plain text is insecure. Passwords should be hashed and the comparison should be done on the hashed values.

_start.c

```
/* WARNING: Function: __i686.get_pc_thunk.bx replaced with
injection: get_pc_thunk_bx */

void processEntry _start(undefined4 initParam1,undefined4
initParam2)
```

```
{
  undefined stackBuffer[4];

  __libc_start_main(main, initParam2, &stack0x00000004,
__libc_csu_init, __libc_csu_fini, initParam1, stackBuffer)
  ;
  do {
                    /* WARNING: Do nothing block with
infinite loop */
  } while( true );
}
```

Description

The function **processEntry** named **_start** is the initial entry point for the program. It takes two parameters **initParam1** and **initParam2**. Inside the function, there is a declaration of a short stack buffer named **stackBuffer** that only allocates four bytes. The function **__libc_start_main** is then called, which initializes the main program by passing it the **main** function, **initParam2**, the **stack0x00000004** stack address, initialization, and finalization routines (**__libc_csu_init** and **__libc_csu_fini** respectively), and other parameters including **initParam1** and the stack buffer. The code then enters an infinite loop where nothing is executed.

Security-wise, there is a potential risk with the use of a small stack buffer **stackBuffer** of only 4 bytes which can easily be overrun, leading to stack corruption and potential exploitation through buffer overflow vulnerabilities. Functions like **__libc_start_main** can accept unsafe parameters leading to various vulnerabilities if not handled correctly. Additionally, the infinite loop can lead to the program hanging and not performing any useful operation which is typically not ideal for production code.

_init.c

```
/* WARNING: Function: __x86.get_pc_thunk.bx replaced with
injection: get_pc_thunk_bx */

int initialize_context(EVP_PKEY_CTX *context)

{
  undefined *function_pointer;

  function_pointer = PTR___gmon_start___0804bffc;
  if (PTR___gmon_start___0804bffc != (undefined *)0x0) {
    function_pointer = (undefined *)(*(code
*)PTR___gmon_start___0804bffc)();
  }
  return (int)function_pointer;
}
```

Description

The provided function named **_init** has been renamed to **initialize_context** to reflect that it initializes some context, which in this case is passed as an **EVP_PKEY_CTX** pointer. The variable **puVar1** has been renamed to **function_pointer** to clearly indicate that it is intended to store a function pointer.

The function checks if **PTR*gmon*start_0804bffc** is not null, and if it is not, it executes the function pointed to by **PTR*gmon*start_0804bffc** and assigns the result to **function_pointer**. The function then returns the integer representation of **function_pointer**.

There are several potential security concerns in this code. First, the function does not validate the context passed to it; if **context** were to be used within the function and were null or invalid, this could lead to undefined behavior or crashes. Secondly, the function uses a pointer without verifying that it indeed points to a valid function, which could result in executing malicious code if the pointer were tampered with.

Finally, returning the integer representation of a possibly null or function base address pointer without checking may cause further issues in calling code, potentially resulting in crashes or unexpected behavior. Famous pitfalls relate to pointer handling, including buffer overflows or improper pointer casting, which are common vulnerabilities in C programming.

ChatGPT Analysis

Exploit

Fuzzing

Exploit success with these input :

- pass
- login.c

Buffer Overflow

Format String

Credits

This report was generated using automated tools and the expert analysis of security researchers.