Bypassing Checks

Problem: Can you find the flag in this file?

Given: ByPassingChecks

Hint: Gdb is amazing

Introduction: Bypassing Checks is a medium level reversing challenge designed to improve students reverse engineering skills by learning to bypass simple checks found in binaries. Learning to bypass simple checks is useful because you can then control the execution flow of a program without actually meeting those requirements. This writeup assumes you have basic knowledge with the debugger and disassembler of your choice.

Steps:
(1) This writeup will be solved using GDB Debugger and Hopper disassembler, but you're free to use whatever tools you're comfortable with. Let's begin by performing a basic analysis against the binary using the 'file' and 'strings' command.

```
[qijun@glap reverse]$ file BypassingChecks
BypassingChecks: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamicall
y linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID
[sha1]=f911c033504688a5bc9fb9e88e6f8b5b4bf11c63, not stripped
```

There are two pieces of information that are important at the moment. First, this file is a 64-bit elf binary. Second, this binary is 'not stripped' meaning this binary was compiled with debugging info.

(2) Next we take a look at the output from the strings command. The screenshot is only a partial output. The majority of the output produced by strings is rather uninteresting but I've gone ahead and highlighted what could be an important piece of information later. Notice that there's two strings somewhere in this binary named 'check_key' and 'flag_data'. It's possible those two strings could be of use later when you begin reversing the binary.

```
__libc_start_main@@GLIBC_2.2.5
__data_start
__gmon_start__
__dso_handle
FLAG
_IO_stdin_used
__libc_csu_init
KEY_LEN
_end
_start
__bss_start
main
_Jv_RegisterClasses
__isoc99_scanf@@GLIBC_2.7
__TMC_END__
_ITM_registerTMCloneTable
check_key
_init
flag_data
```

(3) Then, we execute the binary and see what this program does. First you'll need to chmod the binary so you can execute it on your system 'chmod 755 binaryName'. Upon execution the program asks for us to input a key. I tried '12345' the program then output "Wrong" and quit. Trying several other inputs resulted in the same behavior. It becomes pretty clear you need to figure out the magic key, or bypass the check that compares your key with the correct key.

```
[qijun@glap reverse]$ chmod +x BypassingChecks
[qijun@glap reverse]$ ./BypassingChecks
Key: fdjak
Wrong
```

(4) Load the binary into Hopper. Let's start by taking a look at main() to try and figure out what this executable is doing. I'll also highlight a very useful feature of Hopper, although I believe IDA-Pro and Radare2 have similar functionalities. Below is a snippet from main()

```
00000000004007e1        mov     byte [ss:rbp+var_44], 0x54
00000000004007e5        mov     byte [ss:rbp+var_43], 0xc8
00000000004007e9        mov     byte [ss:rbp+var_42], 0x7e
00000000004007ed        mov     byte [ss:rbp+var_41], 0xe3
00000000004007f1        mov     byte [ss:rbp+var_40], 0x64
00000000004007f5        mov     byte [ss:rbp+var_3F], 0xc7
00000000004007f9        mov     byte [ss:rbp+var_3E], 0x16
```

It seems as though one of the first things main() performs when the program is executed is to move a large chunk of hexadecimal characters onto the stack. This could quite possibly be the program generating the flag for us. Use a Hex to ASCII converter to find out for yourself!

All of this assembly gets a tad tedious to deal with though doesn't it? Let's have Hopper do some heavy lifting for us and generate some high level pseudo code for the main() function.

That's much better! Using the pseudo code generated by Hopper it's much easier to get a better understanding of a program's functionality. In this case the main() function executes a printf() statement that outputs "Key:" to the screen, then takes our user input via a call to scanf(). Once user input is read a conditional if() calls check_key() to validate our input against the correct key. If check_key() returns anything other than 0 interesting_function() will be executed. Otherwise a simple puts() function is executed. I'm going to go out on a limb and say that puts() function is responsible for printing "Wrong!" to the screen.

Since we now know there's a function responsible for validating our input, let's take a look at the function and see if we can't figure out how our input is validated. Below is pseudo code generated by Hopper for the check_key() function.

```
function main {
    var_18 = *0x28;
    rbx = rsp;
    printf("Key: ");
    __isoc99_scanf(0x4009f2, (rsp - (sign_extend_32(0x14) + (0x10 - 0x1)) / 0x10 *
0x10) + 0x0);
    if (check_key((rsp - (sign_extend_32(0x14) + (0x10 - 0x1)) / 0x10 * 0x10) + 0x0)
!= 0x0) {
            interesting_function(0x54);
    }
    else {
            puts(0x4009f5);
    }
    rax = 0x0;
    rcx = var_18 ^ *0x28;
    COND = rcx == 0x0;
    if (!COND) {
            rax = __stack_chk_fail();
    }
    return rax;
}
```

```
function check_key {
    var_8 = arg0;
    var_10 = 0x0;
    for (var_C = 0x0; var_C <= 0x4; var_C = var_C + 0x1) {
            var_10 = var_10 + *(int32_t *)(var_8 + sign_extend_32(var_C) * 0x4);
    }
    if (var_10 == 0xdeadbeef) {
            rax = 0x1;
    }
    else {
            rax = 0x0;
    }
    return rax;
}
```

Inside of the check_key() function two variables are created, var_8 and var_10. Then the program enters a for() loop which appears to take the data from var_8 and place it into var_10. Next the program enters a conditional if() statement which compares the data in var_10 to 0xdeadbeef. If this comparison returns true the value of the $rax register is set to 1, otherwise the $rax register is set to 0. Then the function returns the value of the $rax register.

It should be clear now that your input needs to evaluate true against 0xdeadbeef to get the proper return value from the check_key() function. However, 0xdeadbeef is a magic number in hexspeak for an uninitialized segment of memory. This means there isn't a valid key to compare your input against, so no matter what you enter the program will never give us the flag.

(5) But there's still hope! Hypothetically if you did input the correct key, the value of $rax register would be set to a value of 1 to indicate the correct key was input by the user. Why don't you use GDB to set a breakpoint at the address 0x400700 right before the function returns and set the value of $rax to 1 manually? Try it out.

```
[qijun@glap reverse]$ gdb ./BypassingChecks
GNU gdb (GDB) Fedora 7.10.1-31.fc23
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./BypassingChecks...(no debugging symbols found)...done.
(gdb)
```

(6) Next you want to set a breakpoint at address 0x400700 to pause the execution of the program right before the check_key() function returns. This is done so you can set the value of $rax to 1 before the function returns.

```
(gdb) b *0x400700
Breakpoint 1 at 0x400700
(gdb)
```

Then you want to run the program inside of gdb so the breakpoint we just set is hit. Type ‘run’ inside of gdb to execute a program. The program will execute as it normally would outside of gdb until we hit our breakpoint.

```
(gdb) run
Starting program: /home/qijun/teaching/ctf/txctf-wr
s
Key: 12345

Breakpoint 1, 0x0000000000400700 in check_key ()
(gdb)
```

(7) Before you continue the program's execution, you want to set the value stored in the $rax register to 1. Faking the program into thinking you entered a valid key. You can manually set the value stored inside a register with the command ‘set $register = value’. Additionally, you can inspect the current value stored in a register with the command ‘i r register’.

```
(gdb) set $rax=1
(gdb) i r rax
rax            0x1      1
(gdb)
```

After you’ve set the value contained in the $rax register to 1 you’re free to continue the programs to get the flag...