Lab 7 – Difficult 2

Author: user 6 aka Horia Mut

This lab showed us the power of binary instrumentation with PINtools.

This document focuses on cracking the D2_6 program and its password as this is the one we would like to evaluate.

Table of Content

Tools Required 3
Style 3
How To Crack 4
The 1st Problem - int3 5
The 2nd problem - rdtsc 6
Binary Instrumentation 7
Installation 7
Beating the code 8
Appendix 1 13
MyPinTool.cpp 13

Tools Required

In order to crack the program and follow along you will need the following software:

- gdb
- Python Exploit Development Assistance
- file
- readelf or ltrace
- objdump
- PIN Tools

Style

Command line inputs are presented as follows:

```
# tar -xf pin-3.0-76991-gcc-linux.tar.gz
# cd pin-3.0-76991-gcc-linux
```

Commands and their outputs are presented as such:

```
# echo "Hello World"
Hello World
```

They may also be presented as screenshots.

Assembly instructions are presented in the following manner:

```
80486c0:    83 ec 0c        sub    esp,0xc
80486c3:    83 39 02        cmp    DWORD PTR [ecx],0x2
80486c6:    8b 59 04        mov    ebx,DWORD PTR [ecx+0x4]
80486c9:    75 5c           jne    8048727 <fputs@plt+0x2e7>
```

When we wish to draw attention to a particular part of a code block or output, the relevant content is set in bold:

```
80486d3:    89 d7           mov    edi,edx
80486d5:    f2 ae           repnz scas al,BYTE PTR es:[edi] ; here we check the length
80486d7:    83 f9 e6        cmp    ecx,0xfffffe6
```

How To Crack

We already know that we will need to instrument the binary in order to succeed.

First, we want to know more about the file:

```
# file D2_6
D2_6: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=4695ad353a8a7023d0c69adb9545f33239375636, stripped
```

The executable is stripped therefore debugging symbols are not available. This means that the function "main" will not be found if we disassemble the executable.

Let's disassemble the executable:

```
# objdump -M intel -d D2_6 > disassembled.txt
# less disassembled.txt
```

Reading the disassembly output we can see the following checks:

```
80486c0:    83 ec 0c        sub    esp,0xc
80486c3:    83 39 02        cmp    DWORD PTR [ecx],0x2
80486c6:    8b 59 04        mov    ebx,DWORD PTR [ecx+0x4]
80486c9:    75 5c           jne    8048727 <fputs@plt+0x2e7>
```

Check if a password argument was entered.

```
80486cb:    8b 53 04        mov    edx,DWORD PTR [ebx+0x4]
80486ce:    31 c0           xor    eax,eax
80486d0:    83 c9 ff        or     ecx,0xffffffff
80486d3:    89 d7           mov    edi,edx
80486d5:    f2 ae           repnz scas al,BYTE PTR es:[edi]
80486d7:    83 f9 e6        cmp    ecx,0xffffffe6
80486da:    74 27           je     8048703 <fputs@plt+0x2c3>
```

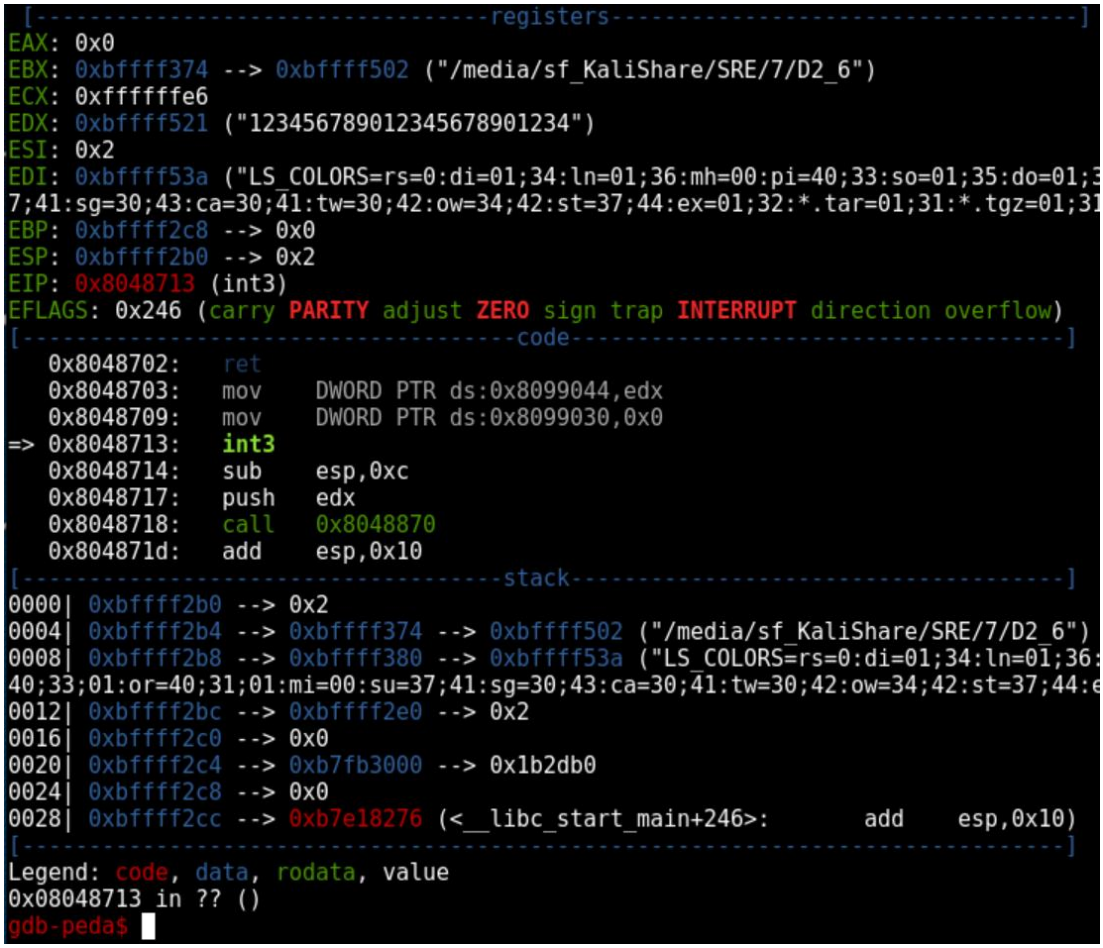
Check if it is of length 0xFF-0xE6 = 0x19 = 25. This means that the password is 24 characters long.

We can now try and run gdb and debug the program.

```
root@kali:/media/sf_KaliShare/SRE/7# gdb D2_6
GNU gdb (Debian 7.11.1-2) 7.11.1
Copyright (C) 2016 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 "i686-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 D2_6...(no debugging symbols found)...done.
gdb-peda$ set args 123456789012345678901234
gdb-peda$ start
```

The 1st Problem - int3

Running the debugger, we will encounter the following:



We have an int3 instruction. This instruction will send a SIGTRAP signal if we are debugging the program. Try to use a breakpoint, setting it much further along in the code, then call xuntil. It will fail and stop at the next int3 instruction. This is problematic, and will not allow us to use gdb to navigate the code easily.

We can also see that we call a function:

```
8048718:    e8 53 01 00 00          call   8048870 <fputs@plt+0x430>
```

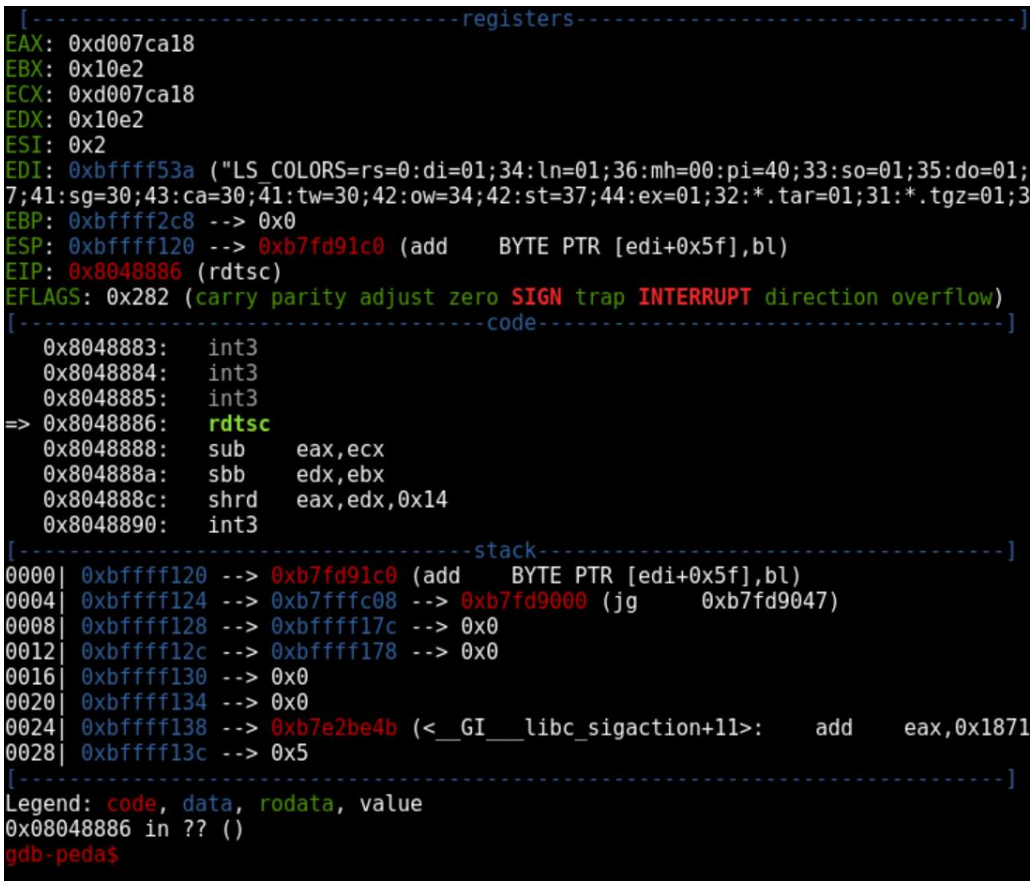
It is highly likely that this function is our password check function. The function is big and returns at the following address:

```
80977be:    c3                      ret
```

That's 80851 instructions later! If you read the code, you will notice a lot of int3 instructions that are mixed with one of your worst enemies that is described on the next page.

The 2nd problem - rdtsc

A bit farther along in the check function we have another new instruction: rdtsc.



This instruction Reads the Time-Stamp Counter and saves the result in EDX:EAX. Reading the disassembled output, we see that this result is used, meaning that time is a factor in the flow of the program.

We can read the function from the end up and we will see that the last call to rdtsc is done at the following address:

80972bf:	0f 31	rdtsc

The last part of the function now uses the input argument and checks the password according to a hard to understand algorithm. Moreover, the password depends on the number of ticks that have happened while the program is running, so we cannot debug it without changing the password.

We have two enemies in this program: time, and the inability to debug the code fast.

In order to surmount these issues, we need to instrument our binary.

Binary Instrumentation

Installation

We now need to use the PIN Tools provided by intel at:

https://software.intel.com/en-us/articles/pintool/

These tools only work on Intel processors IA32 and intel64 and are provided for Windows, Linux, Android and their Intel Xeon Phi.

In order to get the tools, we can do the following:

```
# wget https://software.intel.com/sites/landingpage/pintool/downloads/pin-3.0-76991-gcc-linux.tar.gz
# tar -xf pin-3.0-76991-gcc-linux.tar.gz
```

We will now need to build the tools.

```
# cd pin-3.0-76991-gcc-linux
# cd source/tools/ManualExamples
# make all TARGET=ia32
```

You may encounter errors such as:

```
make: warning: Clock skew detected. Your build may be incomplete.
```

This may happen if your system goes in stand-by while make is running (this happened on my Kali Linux VM). The files are no longer in sync with the internal clock. You can fix this by updating the timestamp of all files within the pin folder:

```
# cd ../../..
# find . -exec touch {} \;
```

Then try to build the tools again. Once the build is successful we should test it:

```
# ../../../pin -t obj-ia32/inscount0.so -- /bin/ls
```

On my Kali Linux VM (Linux kali 4.7.0-kali1-686-pae), the output was the following:

```
# ../../../pin -t obj-ia32/inscount0.so -- /bin/ls
A: Source/pin/vm_ia32_l/jit_region_ia32_linux.cpp: XlateSysCall: 37: Sysenter is supported on IA32 only and the expected location is inside Linux Gate

NO STACK TRACE AVAILABLE
Detach Service Count: 1394
Pin 3.0
Copyright (c) 2003-2016, Intel Corporation. All rights reserved.
```

This issue appears on when running PIN on IA32 binnaries on Debian kernels higher than 4.3. (source)

We therefore changed to an older Ubuntu running laptop which executed the PIN tools without any issues.

Beating the code

We now need to do two things:

1) Fake the time-stamp counter
2) Get the correct values that we test against

Getting the correct values

In order to get the correct values, we need to see what tests are done. Keep in mind that the result of a call will be in the EAX register. A successful call under Linux will return 0, therefore, it is highly likely that the EAX register must contain 0 when the function returns.

We can see that there are plenty of XOR and OR instructions being run. Some of the XOR instructions use a register as the 1st operand and a memory address as the 2nd operand, such as:

```
80972e3:    32 4a 07        xor    cl,BYTE PTR [edx+0x7]
```

This instruction uses an offset and the address contained by the EDX register. The result is XORed with the CL register (left side of ECX) and stored in the same register. This means that we take the 7th character of our password and we XOR it with the content of CL.

A bit further along we have another XOR instruction that uses 2 registers:

```
80972f3:    31 ca           xor    edx,ecx
```

Here the previous result is XORed against another value. In this case, the content of EDX (the 1st operand) is the correct content, and the result of the operation must be 0 in order for the check to pass. This means that:

```
password[7] = content_of(EDX)
```

We now need to determine, for each XOR instruction that is done between two registers, if the content in the 1st operand or the 2nd one is the one that's correct. If any of the 2 registers have been changed, that is the one that contains the correct character of the password.

We have provided as an appendix, the final instructions of the password checking function with comments as to which operand contains the correct content. The rest of the instructions were omitted as they only confuse.

In order for our PIN tool to get the correct password we have the following code for each XOR instruction:

```
/* ================================================================== */
/* XOR operation between 2 registers                                  */
/* ================================================================== */
/* Here we verify if the password retrieved and xored before is valid.*/
static VOID xor_analyse_reg_to_reg(VOID *ip, CONTEXT* ctxt, ADDRINT op1, ADDRINT op2)
{
        xor_reg2reg_ins_counter++;
        printf("Password xor regs = 0x%08x, 0x%08x\n",op1,op2);

        // Here we have some problems.
        // A number of operations may have happened,
        // and we are checking too see if the values are still the same.
        switch(last_index)
        {
                // 1st operand is the correct one.
                case 0x7:
                case 0x12:
                        password[last_index] = (UINT8) op1;
                        break;
                // 2nd operand is the correct one.
                default :
                        password[last_index] = (UINT8) op2;
                        break;
        }
}

/* ================================================================== */
/* XOR operation between a register and a memory offset               */
/* ================================================================== */
/* Here we retrive the password entered and we verify it against another */
/* value retrieved or computed before.                                */
static VOID xor_analyse_pass_index(VOID *ip, CONTEXT* ctxt, ADDRINT op1, UINT32 index)
{
        xor_reg2off_ins_counter++;
        //printf("Indexing reg = 0x%08x index=0x%08x\n",op1,index);

        last_index = (UINT8)index ;
        // Special trick for the index 0, this will be the correct password.
        if(last_index == 0)
        {
                password[last_index] = (UINT8) op1;
                printf("Password xor regs = 0x%08x, 0x%08x\n",op1,index);
        }
}
```

Beating time

In order to beat the clock we will need to modify the content of the EDX:EAX register after the rdtsc instruction has been run.

We can check to see if the instruction we are at is an RDTSC instruction as follows:

```
if (INS_IsRDTSC(ins))
```

We can then insert the calls to modify the contents of EDX and EAX respectively **after** the instruction has been run:

```
INS_InsertCall(ins, IPOINT_AFTER, (AFUNPTR)SwizzleEdx,IARG_PTR, v, IARG_RETURN_REGS,
REG_GDX, IARG_END);
INS_InsertCall(ins, IPOINT_AFTER, (AFUNPTR)SwizzleEax,IARG_PTR, v, IARG_RETURN_REGS,
REG_GAX, IARG_END);
```

SwizzleEdx and SwizzleEax being our custom functions that change the content of the 2 registers.

Their code is the following:

```
/* ================================================================== */
/* Fake EDX register content                                          */
/* ================================================================== */

static UINT32 SwizzleEdx()
{
        // Get the content of EDX:EAX, mask the content of EAX,
        // set it as an unsigned long long if needed, then shift by 32 bits.
        _edx = (_edx_eax & 0xffffffff00000000/*ULL*/) >> 32;
        return _edx;
}

/* ================================================================== */
/* Fake EAX register content                                          */
/* ================================================================== */

static UINT32 SwizzleEax()
{
        // Get the content of EDX:EAX, mask EDX.
        _eax = _edx_eax & 0x00000000ffffffff/*ULL*/;
        // Set the next count to be + 100 ticks.
        _edx_eax+=100;
        return _eax;
}
```

The rest of the code for out PIN tool is located in the appendix.

Running the PIN Tool and the Executable

In order for the password to be cracked without understanding and translating the assembly algorithm, (and playing with time, did you see Dr. Strange? There's a rule against f**kin with time.) we will need to run the PIN tool and our executable.

Out executable is located here:

```
~/sre/7/D2_6
```

Our pin tool is located here:

```
pin-3.0-76991-gcc-linux/source/tools/MyPinTool/MyPinTool.cpp
```

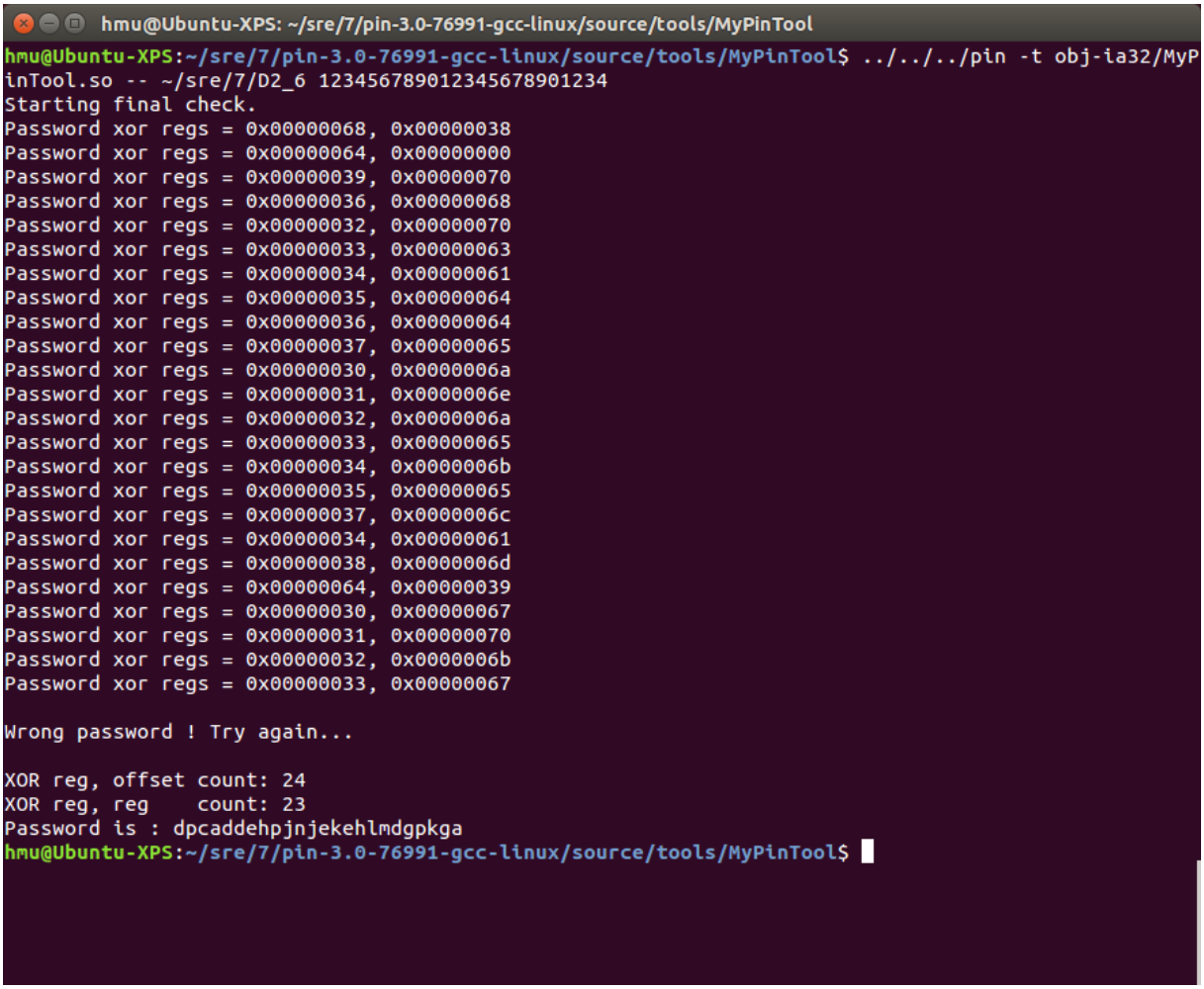
Assuming we are in this folder, we can build our pin tool by running:

```
# make obj-ia32/MyPinTool.so
```

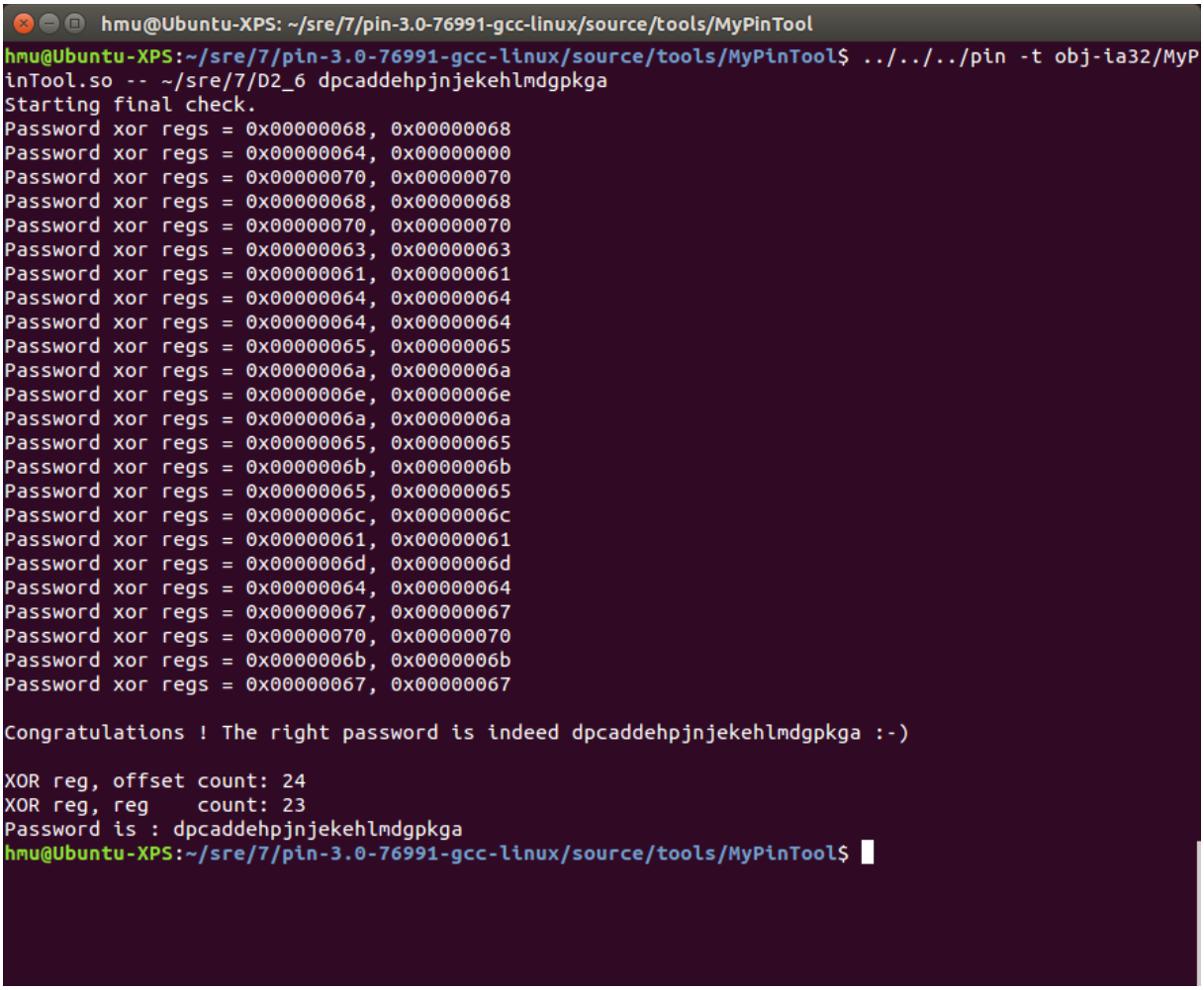
We can then run the pin tool as such:

```
# ../../../pin -t obj-ia32/MyPinTool.so -- ~/sre/7/D2_6 12345678901234567890123 4
```

We will get the following result:



Running again but with the password prompted:



In order for the program to be cracked, the time factor must be controlled. This means that one needs to instrument the executable in order to pass the password check.

Appendix 1

MyPinTool.cpp

```
/*
 * This file contains an ISA-portable PIN tool for tracing memory accesses.
 */
#include <stdio.h>
#include <string.h>
#include "pin.H"
#include "instlib.H"
#include "time_warp.H"


FILE * trace;
static BOOL final_check_start = FALSE;
static BOOL final_check_stop = FALSE;
static UINT64 ins_counter = 0;
static UINT64 xor_reg2reg_ins_counter = 0;
static UINT64 xor_reg2off_ins_counter = 0;
static UINT8 password[25] ;
static UINT8 last_index = 0 ;

static UINT64 _edx_eax;
static UINT32 _eax;
static UINT32 _edx;


/* ===================================================================== */
/* Peek at the time result                                               */
/* ===================================================================== */
/*
static VOID test_time(VOID *ip, CONTEXT* ctxt)
{
        printf("EAX = %08x, EDX = %08x\n",PIN_GetContextReg(ctxt,
REG_EAX),PIN_GetContextReg(ctxt, REG_EDX));
}
*/
```

```
/* ================================================================== */
/* XOR operation between 2 registers                                  */
/* ================================================================== */
/* Here we verify if the password retrieved and xored before is valid.   */
static VOID xor_analyse_reg_to_reg(VOID *ip, CONTEXT* ctxt, ADDRINT op1, ADDRINT op2)
{
        xor_reg2reg_ins_counter++;
        printf("Password xor regs = 0x%08x, 0x%08x\n",op1,op2);

        // Here we have some problems.
        // A number of operations may have happened,
        // and we are checking too see if the values are still the same.
        switch(last_index)
        {
                // 1st operand is the correct one.
                case 0x7:
                case 0x12:
                        password[last_index] = (UINT8) op1;
                        break;
                // 2nd operand is the correct one.
                default :
                        password[last_index] = (UINT8) op2;
                        break;
        }
}

/* ================================================================== */
/* XOR operation between a register and a memory offset               */
/* ================================================================== */
/* Here we retrive the password entered and we verify it against another */
/* value retrieved or computed before.                                */
static VOID xor_analyse_pass_index(VOID *ip, CONTEXT* ctxt, ADDRINT op1, UINT32 index)
{
        xor_reg2off_ins_counter++;
        //printf("Indexing reg = 0x%08x index=0x%08x\n",op1,index);

        last_index = (UINT8)index ;
        // Special trick for the index 0, this will be the correct password.
        if(last_index == 0)
        {
                password[last_index] = (UINT8) op1;
                printf("Password xor regs = 0x%08x, 0x%08x\n",op1,index);
        }
}
```

```
/* ========================================================================== */
/* Fake EDX register content                                                  */
/* ========================================================================== */

static UINT32 SwizzleEdx()
{
	// Get the content of EDX:EAX, mask the content of EAX,
	// set it as an unsigned long long if needed, then shift by 32 bits.
	_edx = (_edx_eax & 0xffffffff00000000/*ULL*/) >> 32;
	return _edx;
}

/* ========================================================================== */
/* Fake EAX register content                                                  */
/* ========================================================================== */

static UINT32 SwizzleEax()
{
	// Get the content of EDX:EAX, mask EDX.
	_eax = _edx_eax & 0x00000000ffffffff/*ULL*/;
	// Set the next count to be + 100 ticks.
	_edx_eax+=100;
	return _eax;
}
```

```
/* ===================================================================== */
/* Instruction insertion                                                 */
/* ===================================================================== */
/* Is called for every instruction and instruments reads and writes.     */

VOID Instruction(INS ins, VOID *v)
{
	ins_counter++;
	//INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)test_time, IARG_INST_PTR, IARG_CONTEXT ,
IARG_END);

	// Last checks happen here: 0x80972d1
	if((FALSE == final_check_start))
	{
		if((ADDRINT)0x80972D1 == INS_Address(ins))
		{
			final_check_start = TRUE;
			printf("Starting final check.\n");
		}
	}
	else if (TRUE == final_check_stop)
	{
		// nop.
	}

	else
	// Only do this when we start the final checks.
	if(TRUE == final_check_start)
	{
		// Reached the end of the final check. Stop.
		if( (ADDRINT)0x80977B5 == INS_Address(ins))
		{
		final_check_stop = TRUE;
		}
		else
		// Dectect XOR Operation.
		if(XED_ICLASS_XOR == INS_Opcode(ins))
		{
			// Get the operands registers
			REG op1_reg = INS_OperandReg(ins, 0);
			REG op2_reg = INS_OperandReg(ins, 1);

			// Check if XOR is done between 2 registers, or with an offset. (Password).
			if(op2_reg != REG_INVALID())
			{
				// It's done between registers.
				INS_InsertCall(ins, IPOINT_BEFORE,
(AFUNPTR)xor_analyse_reg_to_reg, IARG_INST_PTR,
				IARG_CONTEXT , IARG_REG_VALUE ,op1_reg ,IARG_REG_VALUE
,op2_reg, IARG_END);
			}
```

```
                        else
                        {
                                // It's done between a register and a memory offset.
                                // Get the address with INS_MemoryDisplacement(ins).
                                INS_InsertCall(ins, IPOINT_BEFORE,
(AFUNPTR)xor_analyse_pass_index, IARG_INST_PTR,
                                IARG_CONTEXT , IARG_REG_VALUE ,op1_reg , IARG_UINT32
,INS_MemoryDisplacement(ins), IARG_END);
                        }
                }
        }

        // Check if we are calling the Read Time-Stamp Counter instruction.
        if (INS_IsRDTSC(ins))
        {
                // This will change the content of the registers and count as if 100 ticks passed
between calls.
                INS_InsertCall(ins, IPOINT_AFTER, (AFUNPTR)SwizzleEdx,IARG_PTR, v,
IARG_RETURN_REGS, REG_GDX, IARG_END);
                INS_InsertCall(ins, IPOINT_AFTER, (AFUNPTR)SwizzleEax,IARG_PTR, v,
IARG_RETURN_REGS, REG_GAX, IARG_END);
        }

}

/* ===================================================================== */
/* Print the results                                                     */
/* ===================================================================== */

VOID Fini(INT32 code, VOID *v)
{
        printf("XOR reg, offset count: %llu\n", xor_reg2off_ins_counter);
        printf("XOR reg, reg    count: %llu\n", xor_reg2reg_ins_counter);
        printf("Password is : %s\n", password);
        fclose(trace);
}

/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
        PIN_ERROR( "This Pintool prints a trace of memory addresses\n"
        + KNOB_BASE::StringKnobSummary() + "\n");
        return -1;
}
```

```
/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */
int main(int argc, char *argv[])
{
        if (PIN_Init(argc, argv)) return Usage();
        trace = fopen("d2_pin.out", "w");

        memset(password,0, sizeof(password));
        last_index = 0;
        INS_AddInstrumentFunction(Instruction, 0);
        PIN_AddFiniFunction(Fini, 0);
        // Never returns
        PIN_StartProgram();

        return 0;
}
```