# PicoCTF 2014 Writeups

Reed Koser

December 12, 2014

## 1 Police Records

In order to solve this problem, I needed to do two things: pull all of the badgeids from the picoctf server and then find the duplicate. This is a bit tricky because the badge server "encrypts" all communications with an xor key dictated by the server. Fortunately, both the encryption and the protocol is symmetric so all of the hard work of handling communication with the server has already been done for us in the provided source code.

Using the provided encryption/decryption routines, I wrote a short python client that iterated through badge IDs and printed them out. Once I knew that the badge IDs went from 0-1000 I extended the client to keep track of known badge ids and print out duplicates, solving the challenge.

## 2 Obfuscation

In order to solve obfuscation, the first step I took was to disassemble the binary. True to its name, it promised to be an exercise in frustration should I attempt to discern the function of the program statically. Instead, I used the linux `perf` tool to acquire information about the runtime of the program. The `perf stat` command can provide instruction-count resolution, and by maximizing the instruction count I was able to find an input which the binary accepted. It so happened that there was more than one key which the binary accepted and the one that my timing attack found first was not the key the game expected. However, a quick message to the mods and we had Obfuscation marked as solved.

## 3 Baleful

To solve Baleful, I began by running `strings` on the binary. The hint mentions that the binary is packed with a common packer, and the output of `strings` confirms this. Specifically, there was a string that says:

```
$ Info: This file is packed with the UPX executable packer http://upx.sf.net $
```

which gives away the name of the packer.

After unpacking the binary, I was confronted with thousands of machine code instructions. At this point a normal person would likely try to step through the program with a debugger, but after a few minutes of that I realized I wasn't going to get anywhere without understanding how the code was structured, so I rolled up my sleeves and wheeled out the nuclear weapon of binary analysis problems: IDA.

The first revelation came when I realized that the program was structured as a virtual machine. After this realisation I decided that the best solution would be to write a recursive descent disassembler (possibly not the most efficient solution, but source is available upon request![1]), in order to continue my quest through the land of machine code. Once I knew the general program flow, it was simply a matter of filling in opcodes, and in short order I had decoded the first section of code the virtual machine executes:

```
00001000 |18 01 00 3a 10 00 00    | mov   r00 0000103a
00001007 |18 01 01 db 1e 00 00    | mov   r01 00001edb
0000100e |18 00 03 00             | mov   r03 r00
00001012 |18 01 05 42 cf 74 01    | mov   r05 0174cf42
00001019 |1b 04 03                | mov   r04 [r03]
0000101c |06 00 04 04 05          | xor   r04 ( r04 ^ r05 )
00001021 |1c 03 04                | mov   [r03] r04
00001024 |02 01 03 03 04 00 00 00 | add   r03 ( r03 + 00000004 )
0000102c |17 00 01 03             | test  ( r01 - r03 )
00001030 |14 19 10 00 00          | jge   00001019
00001035 |0e 3a 10 00 00          | jmp   0000103a
0000103a |0e c0 1b 00 00          | jmp   00001bc0
```

This is a relatively obvious xor "encryption" decoder, and clearly I need the normal version of the code. At this point I had figured out where in memory the instruction pointer and main memory were stored, so I set a conditional breakpoint in `gdb` for when the instruction pointer had advanced into the encrypted opcodes and then dumped the VM's main memory again. Once I had the full code, it was a relatively simple mater to figure out what the code was doing. The first thing that jumped out out me was the code like this:

```
00001142 |0f 3f 10 00 00          | call  0000103f
00001147 |18 01 00 69 00 00 00    | mov   r00 00000069
0000114e |0f 3f 10 00 00          | call  0000103f
00001153 |18 01 00 6f 00 00 00    | mov   r00 0000006f
0000115a |0f 3f 10 00 00          | call  0000103f
0000115f |18 01 00 6e 00 00 00    | mov   r00 0000006e
00001166 |0f 3f 10 00 00          | call  0000103f
```

Since the function at `0x103f` is just a dispatch to a `ccall` that prints the character in `r00`, these code segments are what outputs to the user. After finding and labeling all of these output functions, it was a simple (if time consuming) matter to crunch backwards through the code to figure out what conditions caused the program to print "Congratulations!..." The most restrictive (and therefore interesting) seemed to be this loop:

```
000017d9 |18 00 1e 02             | mov   r1e r02       # some sorta loop
000017dd |04 01 1e 1e 04 00 00 00 | imul  r1e ( r1e * 00000004 )
000017e5 |18 00 00 0a             | mov   r00 r0a
000017e9 |02 00 1e 1e 00          | add   r1e ( r1e + r00 )
```

[1] I would include it here, but it makes the report four times longer without adding any substantive content

```
000017ee |18 00 03 1e              | mov   r03 r1e
000017f2 |1b 04 03                 | mov   r04 [r03]
000017f5 |05 01 00 03 02 04 00 00  | idiv  r00 ( r03 / 00000402 )
000017fd |00                       | nop
000017fe |18 00 1e 03              | mov   r1e r03
00001802 |04 01 1e 1e 04 00 00 00  | imul  r1e ( r1e * 00000004 )
0000180a |18 00 00 05              | mov   r00 r05
0000180e |02 00 1e 1e 00           | add   r1e ( r1e + r00 )
00001813 |18 00 03 1e              | mov   r03 r1e
00001817 |1b 03 03                 | mov   r03 [r03]
0000181a |06 00 04 04 03           | xor   r04 ( r04 ^ r03 )
0000181f |18 00 1e 02              | mov   r1e r02
00001823 |04 01 1e 1e 04 00 00 00  | imul  r1e ( r1e * 00000004 )
0000182b |18 00 00 09              | mov   r00 r09
0000182f |02 00 1e 1e 00           | add   r1e ( r1e + r00 )
00001834 |18 00 03 1e              | mov   r03 r1e
00001838 |1b 03 03                 | mov   r03 [r03]
0000183b |18 00 1e 04              | mov   r1e r04
0000183f |18 00 00 03              | mov   r00 r03
00001843 |17 00 1e 00              | test  ( r1e - r00 )
00001847 |15 51 18 00 00           | jne   00001851
0000184c |0e 58 18 00 00           | jmp   00001858
00001851 |18 01 01 01 00 00 00     | mov   r01 00000001
00001858 |02 01 02 02 01 00 00 00  | add   r02 ( r02 + 00000001 )
00001860 |18 00 03 01              | mov   r03 r01
00001864 |18 00 1e 02              | mov   r1e r02
00001868 |18 01 00 1e 00 00 00     | mov   r00 0000001e
0000186f |17 00 1e 00              | test  ( r1e - r00 )
00001873 |11 d9 17 00 00           | jlt   000017d9      # loop end
```

which (after staring at it for awhile) is another XOR decryption loop. The previous two hundred or so opcodes fill in two buffers, one of which this loop decrypts using the other as the key. Once I figured that out, I used `gdb` to dump the memory in those two buffers and a bit of python magic to xor the two and spit out the flag.