Confidence CTF 2016 – Blackbox

CONFidence CTF to zawody organizowane przez Dragon Sector – najlepszą polską drużynę według rankingu CTFime.org, która zdobyła mistrzostwo świata w 2014 roku. Finały, jak co roku, odbyły się na konferencji CONFidence w Krakowie i były poprzedzone internetowymi kwalifikacjami. Również jak co roku przybyły drużyny z całego świata, by zmierzyć się ze sobą. W trakcie zawodów trafiliśmy na nietypowe zadanie, które udało nam się po długiej walce rozwiązać. Tak nam się spodobało, że postanowiliśmy je opisać w ramach Strefy CTF.

CTF	Confidence CTF 2016 http://2016.confidence.org.pl/ctf/
Waga CTFtime.org	44.69 (https://ctftime.org/event/308)
Liczba drużyn (z niezerową liczbą punktów)	22
System punktacji zadań	Od 50 punktów (bardzo proste) do 500 punktów (trudne).
Liczba zadań	26
Podium	1. Tasteless (Międzynarodowy) – 2700 pkt. 2. P4 (Polska) – 2250 pkt. 3. 9447 (Australia) – 1900 pkt.
Zadanie	Blackbox (Reverse 500p)

O ZADANIU

Reversing code on a known architecture is barely a challenge, how about a custom one? Try your luck/skills with this.

Blackbox było jednym z najtrudniejszych zadań na CTFie (zostało wycenione przez twórcę na 500 punktów, czyli maksymalną możliwą liczbę) i wymagało dość nietypowego podejścia. Nie było tu na przykład analizy kodu na poziomie Asemblera (ani tym bardziej C) – zamiast tego najpierw trzeba było tego Asemblera się domyślić, a jedyne, co mieliśmy do dyspozycji, to plik wykonywalny dla wymyślonej architektury.

Musieliśmy sami najpierw domyślić się, gdzie znajduje się kod maszynowy w pliku, później podzielić go na opkody oraz przede wszystkim domyślić się znaczenia każdego z nich – zaczynając od tych najprostszych (dodawanie, odejmowanie, skoki), a kończąc na tych bardziej zwariowanych (alokacja pamięci, czytanie z zasobów). Ten artykuł jest historią opowiadającą, jak można wyciągnąć coś z niczego i przeprowadzić analizę nieznanych danych, zaczynając kompletnie od zera.

W ramach ciekawostki, rozmawiając po konferencji z organizatorami, dowiedzieliśmy się, że autor zadania podczas tworzenia go zainspirował się towarzyszem ze swojej drużyny. Osoba ta w jednym z wcześniejszych konkursów dostała nieznaną binarkę i rozwiązała zadanie, analizując ją w całości w podobny, „blackboxowy" sposób. Jak się później okazało – niepotrzebnie, bo architektura była po prostu mało popularna, ale standardowa. Ale w końcu to jest właśnie najlepsze w CTFach, że nie liczy się sposób rozwiązania zadania, a jedynie ostateczny wynik.

ANALIZA PROGRAMU

W zadaniu otrzymaliśmy niewielki (1316 bajtów) plik binarny. Polecenie `file`, użyteczne w takiej sytuacji, dało nam informację, że jest to plik ELF, zazwyczaj używany w charakterze plików wykonywalnych.

```
$ file task.bin
task.bin: ELF 32-bit LSB executable, *unknown arch 0x539*
version 1 (SYSV), statically linked, interpreter /lib/ld-
dragon.so, corrupted section header size
```

Nie był on jednak w żadnej typowej architekturze, jak x86 czy np. ARM. Program `file` nie rozpoznał go i podał jedynie jej numer – `0x539` (czyli `1337` w systemie dziesiątkowym). Domyśliliśmy się zatem, że jest to własna architektura, wymyślona na potrzeby CTFa – co za tym idzie, nie było do niej publicznie dostępnego deasemblera czy emulatora. Zadanie polegało więc na domyśleniu się znaczenia poszczególnych opkodów (instrukcji) i późniejszej analizie kodu.

Ponieważ plik był w znanym formacie – ELF, wykorzystaliśmy program `readelf` do odczytania jego nagłówków. Oto wynik tej komendy:

```
$ readelf -hl task.bin
ELF Header:
 Magic:   7f 45 4c 46 01 01 45 00 00 00 00 00 00 00 00 00
 Class:                             ELF32
 Data:                              2's complement, little endian
 Version:                           69 <unknown: %lx>
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              EXEC (Executable file)
 Machine:                           <unknown>: 0x539
```

```
Version:                           0x1
Entry point address:               0x19100
Start of program headers:          52 (bytes into file)
Start of section headers:          0 (bytes into file)
Flags:                             0x0
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         4
Size of section headers:           0 (bytes)
Number of section headers:         0
Section header string table index: 0

Program Headers:
  Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR      0x000034 0x00019034 0x00019034 0x00060 0x00060 R E 0x4
  INTERP    0x0000c4 0x000190c4 0x000190c4 0x00012 0x00012 R   0x1
      [Requesting program interpreter: /lib/ld-dragon.so]
  LOAD      0x000000 0x00019000 0x00019000 0x00448 0x00448 R E 0x1000
  LOAD      0x000448 0x0001a000 0x0001a000 0x000dc 0x000dc R   0x1000
```

Z gąszczu informacji wyciągnąć można niektóre interesujące nas fakty, jak na przykład to, że plik korzysta z ABI (ang. *Application Binary Interface*) uniksowego – zatem jeśli pojawiłyby się jakieś wywołania systemowe, można by znaleźć ich znaczenie w ogólnie dostępnych tablicach.

Przydatne są także adresy sekcji. Pierwsze dwie to typowe dla ELFów nagłówki, opisujące odpowiednio położenie nagłówków sekcji oraz nazwę interpretera, którym należy otwierać dany plik. Na ogół jest nim `ld` (na przykład dla programu `/bin/ls` interpreter to `/lib64/ld-linux-x86-64.so.2`). W konkursowej binarce jest to jednak niestandardowy plik `/lib/ld-dragon.so`, co potwierdza nasze przypuszczenia co do architektury.

Kolejna sekcja mapuje początkowe `0x448` (czyli 1096) bajtów na adres `0x19000` z prawami RE (*Read, Execute*) – prawdopodobnie jest to zatem część wykonywalna, najbardziej nas interesująca. Ostatnia sekcja zaczyna się od bajtu `0x448` i ma wielkość `0xdc` (dziesiętnie 220) – jest to więc druga część pliku, aż do jego końca. Nie jest ona wykonywalna, a jedynie do odczytu, można więc na tym etapie wnioskować, że jest to część zawierająca stałe (odpowiednik *.rodata* ze standardowych ELFów).

W wyniku powyższej komendy znajduje się jeszcze jedna ciekawa informacja: adres, w którym program rozpoczyna wykonywanie (ang. *Entry Point*), to `0x19100`, czyli 256-ty bajt sekcji wykonywalnej.

W tym momencie mamy już sporo metadanych, możemy więc przystąpić do sprawdzania zawartości poszczególnych sekcji (zwłaszcza dwóch ostatnich). Poniżej prezentujemy fragmenty pliku przedstawione w postaci szesnastkowej.

```
00000150  57 02 00 02 00 57 03 00  01 00 55 01 00 00 00 69  |W....W....U....i|
00000160  03 00 00 00 14 02 00 00  00 57 03 00 02 00 57 05  |.........W....W.|
00000170  00 01 00 19 01 00 02 00  57 04 00 02 00 0c 80 02  |........W.......|
00000180  00 00 57 01 00 05 00 57  02 00 04 00 09 03 00 00  |..W....W........|
00000190  00 0c c0 01 00 00 57 01  00 04 00 0c 1c 02 00 00  |......W.........|
000001a0  57 02 00 04 00 57 03 00  01 00 55 01 00 01 00 69  |W....W....U....i|
000001b0  04 00 00 00 69 fc 00 00  00 05 06 00 00 00 05 05  |....i...........|
000001c0  00 00 00 05 04 00 00 00  57 06 00 02 00 37 03 00  |........W....7..|
000001d0  03 00 57 02 00 01 00 27  02 00 03 00 1f 02 00 03  |..W....'........|
000001e0  00 25 03 00 01 00 41 03  00 00 01 4c c2 ff ff ff  |.%....A....L....|
000001f0  37 03 00 03 00 37 04 00  04 00 57 02 00 01 00 27  |7....7....W....'|
00000200  02 00 03 00 23 05 00 02  00 31 05 00 ff 00 27 04  |....#....1....'.|
00000210  00 05 00 05 01 00 00 00  57 01 00 03 00 31 01 00  |........W....1..|
00000220  03 00 27 01 00 06 00 23  01 00 01 00 31 01 00 ff  |..'....#....1...|
00000230  00 27 04 00 01 00 31 04  00 ff 00 09 01 00 00 00  |.'....1.........|
```

```
00000440  00 00 00 10 00 00 00 00  01 00 00 00 18 a0 01 00  |................|
00000450  01 00 00 00 3c a0 01 00  02 00 00 00 4c a0 01 00  |....<.......L...|
00000460  68 65 6c 6c 6f 20 63 72  75 65 6c 20 77 6f 72 6c  |hello cruel worl|
00000470  64 2c 20 68 6f 77 20 61  72 65 20 79 6f 75 3f 0a  |d, how are you?.|
00000480  00 00 00 00 67 69 62 20  70 69 6e 20 70 6c 73 3f  |....gib pin pls?|
00000490  0a 00 00 00 89 00 00 00  3f 50 ea 9d 7b f4 51 54  |........?P..{.QT|
000004a0  bb e2 56 53 85 7e 44 90  7d a9 0d 9c cd e7 f9 5a  |..VS.~D.}......Z|
000004b0  12 0c f2 21 b7 02 f8 3b  9d 7c 92 d9 96 af d6 97  |...!...;.|......|
000004c0  25 7e b5 04 a5 16 83 21  06 f6 48 2c b7 c6 90 b2  |%~.....!..H,....|
000004d0  64 bf e5 a0 a8 8b e0 74  f2 82 1c 7b f3 49 74 33  |d......t...{.It3|
000004e0  e2 3a 89 3b f2 1d 4a 19  c7 3f eb 65 76 8d 1e ed  |.:.;..J..?.ev...|
000004f0  20 78 b6 75 2e 31 73 fa  6e 3a 4b 0c 93 8a 22 7f  | x.u.1s.n:K...".|
00000500  89 f2 4a 10 59 c2 fd ad  cd 81 03 0f cf 1f fc 1b  |..J.Y...........|
00000510  7d 63 8a 69 a3 fe 81 98  38 69 a3 04 b6 00 cc f6  |}c.i....8i......|
00000520  de 00 00 00                                       |....|
```

Pierwszy screen pochodzi z sekcji wykonywalnej, natomiast drugi z sekcji z danymi tylko do odczytu. W tej ostatniej widać tekst w języku angielskim, co pokazuje, że jesteśmy na dobrym tropie. Przed napisami znajdują się dane binarne. Zakładając, że są to 32-bitowe liczby całkowite zapisane, zaczynając od najmniej znaczącego bajtu (ang. *little endian*), są to kolejno: `1`, `0x1a018`, `1`, `0x1a03c`, `2`, `0x1a04c`. Co druga z tych liczb jest niewielka, więc prawdopodobnie są to jakieś flagi. Pozostałe liczby są zbliżone do adresu tej sekcji (`0x1a000`), zatem można przypuszczać, że są to adresy kolejnych „zasobów" zawartych w pliku. I rzeczywiście, pierwsze dwa wskazywały na kolejne napisy („hello …" oraz „gib …"). Trzeci zawierał jednak adres niewiele nam mówiących danych binarnych. Zapamiętaliśmy ten fakt, gdyż było dość prawdopodobne, że były to zaszyfrowane bądź skompresowane dane.

Jeśli chodzi o analizę pierwszej sekcji, rzuca się w oczy wzór w dumpie pamięci – ukośne „kreski" z drukowalnych znaków. Dzieje się tak z uwagi na pewną regularność w pięciobajtowych kawałkach pamięci. Oznaczało to najpewniej, iż pojedyncza instrukcja w tej architekturze ma stałą długość, wynoszącą właśnie pięć bajtów. Mogliśmy się domyślać, że pierwszy z nich (ten wyróżniający się) oznacza typ instrukcji (jak np. `add` czy `call`), a pozostałe cztery mogą zawierać jej argumenty (np. numer rejestrów czy stałą, która ma zostać wykorzystana do obliczeń). Mogliśmy dzięki temu rozpocząć analizę od wypisania na ekran wszystkich instrukcji z pliku razem z ich parametrami. Napisaliśmy do tego celu krótki skrypt w Pythonie:

```
import sys
data=open(sys.argv[1], "rb").read()
data=data[0x100:0x448] # Executable section.
for i in range(len(data)/5):
  ins=data[5*i:5*(i+1)]
  dump=ins.encode("hex")
  print dump[:2]+"\t"+dump[2:]
```

Fragment jego wyjścia:

```
0c   1c020000
57   02000400
57   03000100
55   01000100
69   04000000
69   fc000000
05   06000000
05   05000000
05   04000000
57   06000200
37   03000300
57   02000100
27   02000300
1f   02000300
25   03000100
41   03000001
4c   e2ffffff
37   03000300
37   04000400
57   02000100
27   02000300
23   05000200
31   0500ff00
```

Jak można się zorientować, większość instrukcji ma dość charakterystyczny rozkład bajtów składających się na argumenty: pierwszy bajt niezerowy, ale niewielki (zazwyczaj z przedziału `01-06`), drugi zerowy, trzeci także `01-06` i czwarty ponownie zerowy. Można założyć, że są to dwie niewielkie 16-bitowe liczby całkowite. Ponieważ pojawiały się one w większości instrukcji, zapewne oznaczały one rejestry, na których dana instrukcja operowała. Na przykład „`27 02000300`" oznaczało instrukcję o kodzie `0x27`, operującą na rejestrach drugim i trzecim.

Posortowaliśmy instrukcje według pierwszych bajtów i sprawdziliśmy dla każdej z nich, jakie pojawiają się w niej argumenty. Jeśli zawsze były one niewielkie, to zakładaliśmy, że powyższy tok rozumowania jest poprawny (tak było dla większości opkodów). Niektóre miały nieco inny format, na przykład instrukcja `0x09` wystąpiła w następujących formach:

```
09   02000000
09   03000000
09   04000000
09   05000000
09   06000000
```

Pierwszy bajt rzeczywiście zawsze był w odpowiednim zakresie, jednak trzeci (odpowiadający za drugi rejestr) był zawsze wyzerowany. Wywnioskowaliśmy więc, że ta instrukcja przyjmuje tylko jeden parametr, który również jest numerem rejestru. Instrukcji w takim formacie również było kilka typów. Były też instrukcje, których argument zawsze był równy zero (czyli de facto prawdopodobnie nie przyjmowały żadnego parametru).

Po odpowiednim oznakowaniu wyżej wymienionych typów instrukcji pozostała nam zaledwie garstka opkodów o nieznanym formacie. Na przykład opkod `0x0c`:

```
0c   1bfeffff
0c   1c020000
0c   80020000
0c   8a020000
0c   ad020000
0c   defeffff
0c   e0010000
```

Widać, że argument do tej instrukcji jest jedną liczbą zapisaną w *U2* (kodzie uzupełnienia do dwóch). Poszczególne argumenty w systemie dziesiątkowym to kolejno: `-485`, `540`, `640`, `650`, `685`, `-290` i `30`. Zauważyć można, że wszystkie z tych liczb są podzielne przez pięć, a ponadto są porównywalne z długością kodu wykonywalnego, równą `840`. Można z tego wywnioskować, że będzie to instrukcja skacząca w inne miejsce w programie. Ponieważ niektóre z tych argumentów są ujemne, nie może to być skok absolutny, lecz względny (czyli podany jako przesunięcie od obecnego położenia). Opkodów tego rodzaju również było kilka – nie wiedzieliśmy więc, które z nich oznaczają skoki (odpowiednik `JMP`), a które wywołania funkcji (odpowiednik `CALL`), a także które są warunkowe, a które nie. Niemniej sam fakt posiadania wszystkich adresów będących celami skoku był bardzo przydatny, gdyż pozwalał on na ustalenie wysokopoziomowej struktury programu – na przykład wyodrębnienie funkcji, pętli czy bloków warunkowych.

Skoro znaliśmy już format większości instrukcji, postanowiliśmy poprawić poprzedni skrypt, tak aby wypisał kolejne instrukcje wraz z ich argumentami (choć oczywiście nie wiedzieliśmy jeszcze, co konkretne instrukcje robią). Kod:

```
import sys, struct

ops={
  0x57: "RR",  0x0c: "JMP", 0x50: "JMP", 0x48: "JMP",
  0x4c: "JMP", 0x10: "",    0x09: "R",   0x05: "R",
  0x37: "RR",  0x69: "I",   0x55: "RI",  0x25: "RI",
  0x23: "RR",  0x31: "RI",  0x41: "RI",  0x29: "RI",
  0x14: "???", 0x44: "JMP", 0x43: "RR",  0x27: "RR",
  0x1f: "RR",  0x18: "IR",  0x19: "RR"
}

data = open(sys.argv[1], "rb").read()
data = data[0x100:0x448] # Executable section.
for i in range(0, len(data), 5):
  ins = data[i:i + 5]
  dump = ins.encode("hex")
  op = ord(data[i])
  s="  " + str(i) + "\t" + "OP_" + dump[:2]
  type = ops[op]
  if type == "RR":
    s += " r" + dump[3] + ", r" + dump[7]
  elif type == "RI":
    s += " r" + dump[3] + ", 0x" + dump[8:] + dump[6:8]
  elif type == "IR":
    s += " 0x" + dump[4:6] + dump[2:4] + ", r" + dump[7]
  elif type == "I":
    s += " 0x" + dump[4:6] + dump[2:4]
  elif type == "R":
    s += " r" + dump[3]
  elif type == "JMP":
    off = struct.unpack("<i", ins[1:])[0]
    s += " jmp to " + str(i + 5 + off)
  elif type=="":
    s += ""
  else:
    s += " ??? " + dump
  print s
```

Nie jest to może piękny kod, ale pozwolił on nam na lepszą wizualizację tego, co już się dowiedzieliśmy o tej architekturze i naszym konkretnym programie. Poniżej znajduje się fragment wyniku działania tego programu:

```
  0      OP_14 ??? 1400000000
  5      OP_05 r1
  10     OP_0c jmp to 700
  15     OP_57 r3, r1
[...]
  695    OP_10
  700    OP_05 r2
  705    OP_05 r5
  710    OP_55 r2, 0x0000
  715    OP_23 r5, r1
  720    OP_31 r5, 0x00ff
  725    OP_41 r5, 0x0000
  730    OP_25 r1, 0x0001
  735    OP_25 r2, 0x0001
  740    OP_48 jmp to 715
  745    OP_29 r2, 0x0001
  750    OP_57 r1, r2
  755    OP_09 r5
  760    OP_09 r2
  765    OP_10
  770    OP_05 r6
[...]
```

Pierwsze kilka linijek opisuje pierwsze instrukcje, które się wykonują po uruchomieniu programu. Najpierw występuje jakaś nieznana instrukcja, następnie pewna czynność operująca na rejestrze `r1`, a potem instrukcja skacząca pod adres `700`. Ponieważ jest to dość daleki skok (cały program ma 840 bajtów), można się spodziewać, że są to osobne funkcje, a zatem `OP_57` to w rzeczywistości `CALL`, a instrukcje od `700` to jakaś funkcja. Skoro funkcja zaczyna się od `700`, to zapewne poprzednia funkcja kończy się instrukcję wcześniej. Ostatnią instrukcją funkcji jest w większości przypadków `RET` – roboczo możemy więc założyć, iż `OP_10`, pojawiający się w linijce 695, to właśnie `RET`. Kolejnym wystąpieniem tego opkodu jest linia 765, zatem jeśli to założenie jest prawdziwe, to rozważana funkcja zawiera swój kod w obrębie linii oznaczonych numerami `700-765`.

Wiele ABI posiada konwencje dotyczące wstawiania rejestrów na stos przy wchodzeniu do funkcji oraz ściągania ich przy wychodzeniu, aby uchronić ich zawartość przed niepożądaną zmianą. Tak też dzieje się w tym przypadku – pierwsze dwie instrukcje tej funkcji mają ten sam opkod operujący na pojedynczym rejestrze, co idealnie pasuje do instrukcji `PUSH`. Ostatnie dwie instrukcje również mają ten sam pierwszy bajt, a ponieważ operują na tych samych rejestrach co `PUSH` (w odwrotnej kolejności), jest duża szansa, że to `POP`.

Prolog i epilog funkcji już zatem znamy, a także wiemy, że funkcja może zmieniać dowolnie zawartość r2 i r5, bo i tak będą one przywrócone. Pierwszą nieznaną jeszcze instrukcją jest OP_55 r2, 0x0000. Nasuwa się myśl, że może to być przypisanie r2=0, chociaż nie jest to jeszcze nic pewnego. Dalej mamy pętlę, gdyż w linii 740 jest skok do 715. Najprawdopodobniej nie jest to pętla nieskończona, a w środku tej pętli nie ma żadnych innych skoków, zatem OP_48 to skok warunkowy. Sprawdźmy zatem, co dzieje się w tej pętli.

Najpierw do r5 jest przypisywany wynik jakiejś operacji z rejestrem r1. Ponieważ r1 jak dotąd się nigdzie nie pojawiło w tej funkcji, można przyjąć, że jest to jej argument. Do rejestru r5 nie zostało jeszcze nic zapisane, więc jego wartość na pewno będzie ignorowana w tej operacji (zakładamy, że w niezainicjowanym rejestrze znajdują się losowe dane). Kolejna instrukcja to OP_31 r5, 255. Ponieważ 255 to szesnastkowo 0xFF, jest spora szansa, że jest to instrukcja operująca na bitach, taka jak AND czy XOR. Dalej mamy OP_41 r5, 0. Większość operacji modyfikujących nie ma większego sensu z operandem równym zero (np. dodawanie), a bezpośrednie przypisanie zera też odrzucałoby dopiero co ustaloną i jeszcze niewykorzystaną wartość r5. Drogą eliminacji dochodzimy do wniosku, że najprawdopodobniej jest to instrukcja porównania ze stałą, tym bardziej że jesteśmy wewnątrz pętli, która powinna posiadać jakiś warunek stopu. Dalej mamy dwie instrukcje z różnymi rejestrami, choć obie mają ten sam opkod oraz jedynkę jako drugi operand. Ponieważ jesteśmy wewnątrz pętli, nasuwa się myśl, że może to być inkrementacja zmiennej iterującej – czyli OP_25 to zapewne ADD. Po końcu pętli mamy jeszcze instrukcje OP_29 r2, 1 oraz OP_57 r1, r2. Pierwsza z tych instrukcji wygląda podobnie jak ADD, chociaż ma inny opkod. Jest szansa, że jest to naprawienie błędu off-by-one po pętli, czyli odjęcie jedynki instrukcją SUB. Ponieważ r1 nie jest już ani razu użyte w tej funkcji, musi to być wartość zwracana z tej funkcji. Jest spora szansa, że jest to proste przypisanie rejestr-rejestr (MOV), wynikające z przyjętej konwencji wywołań.

Spójrzmy więc raz jeszcze, jak wygląda ta funkcja:

```
700    PUSH_R r2
705    PUSH_R r5
710    MOV_RI r2, 0x0000
715    OP_23 r5, r1
720    OP_31 r5, 0x00ff
725    CMP_RI r5, 0x0000
730    ADD_RI r1, 0x0001
735    ADD_RI r2, 0x0001
740    JMPxx jmp to 715
745    SUB_RI r2, 0x0001
750    MOV_RR r1, r2
755    POP_R r5
760    POP_R r2
765    RET
```

Wygląda na to, że jest to pętla, która w pseudokodzie miałaby następującą postać:

```
function f(arg) {
cnt = 0;
do {
   r5 = OP_23(arg);
   r5 = OP_31(r5, 255);
   arg += 1;
   cnt += 1;
} while(comparison(r5, 0));
cnt -= 1;
return cnt;
}
```

W tym momencie możemy rozpoznać tę funkcję – jest to po prostu strlen (funkcja zwracająca długość ciągu znaków). Mając tę informację, możemy wywnioskować, co robią niezidentyfikowane jeszcze instrukcje. OP_23 musiałby być instrukcją ładującą zawartość pamięci w danym miejscu (LOAD). Jeśli napis składa się z jednobajtowych znaków, to ma sens także OP_31 – jest to najprawdopodobniej AND, zwracający w tym wypadku wartość pamięci zrzutowaną na jednobajtowy typ. Wreszcie nieznany jak dotąd warunek skoku możemy zidentyfikować jako nierówność, czyli od teraz odpowiedni opkod jest znany jako JMP_NE (NE jak *not equal*).

W podobny sposób zbadaliśmy resztę programu. Oczywiście im więcej instrukcji mieliśmy już znane, tym prościej było odgadnąć resztę. Nie udało nam się jednak znaleźć znaczenia wszystkich opkodów, analizując program statycznie. Napisaliśmy więc zalążek emulatora tej maszyny wirtualnej, zawierający implementację już znanych opkodów. Stopniowo udało nam się domyślić znaczenia wszystkich z nich, kolejno jeden po drugim.

Najwięcej czasu spędziliśmy nad opkodami MLC_* oraz LD_RES, które miały szczególnie nieoczywistą składnię (LD_RES ładował adres zasobu o zadanym indeksie do wybranego rejestru, przy czym w zależności od typu zasobu możliwe było, że dodatkowo do drugiego rejestru wstawiana była jego długość).

05	PUSH_R reg	Wrzuć podany rejestr na stos
09	POP_R reg	Pobierz dany rejestr ze stosu
0c	CALL_I imm	Wywołaj funkcję pod adresem imm
10	RET	Wróć z funkcji
14	LD_RES imm	Wczytaj zasób o podanym numerze
18	MLC_IR imm, reg	Zaalokuj pamięć o podanej wielkości i zapisz wskaźnik do rejestru
19	MLC_RR reg, reg	Zaalokuj pamięć o wielkości wczytanej z rejestru i zapisz wskaźnik do rejestru
1f	STORE reg, reg	Zapisz wartość z drugiego rejestru do miejsca w pamięci, na które wskazuje pierwszy rejestr
23	LOAD reg, reg	Pobierz wartość wskazywaną przez drugi rejestr i zapisz ją do pierwszego
25	ADD_RI reg, imm	Oblicz sumę parametrów i zapisz wynik do rejestru
27	ADD_RR reg, reg	Oblicz sumę parametrów i zapisz wynik do pierwszego rejestru
29	SUB_RI reg, imm	Oblicz różnicę parametrów i zapisz wynik do rejestru
31	AND_RI reg, imm	Oblicz iloczyn logiczny parametrów i zapisz wynik do rejestru.
37	XOR_RR reg, reg	Oblicz xor parametrów i zapisz wynik do 1-go rejestru
41	CMP_RI reg, imm	Porównaj parametry, zapisz wynik do flag procesora
43	CMP_RR reg, reg	Porównaj parametry, zapisz wynik do flag procesora
44	JMP_GE imm	Skocz, jeśli ostatnie porównanie dało wynik „większy lub równy”
48	JMP_NE imm	Skocz, jeśli ostatnie porównanie dało wynik „nierówny”
4c	JMP_LE imm	Skocz, jeśli ostatnie porównanie dało wynik „mniejszy lub równy”
50	JMP imm	Skocz pod podany adres
55	MOV_RI reg, imm	Zapisz stałą wartość do rejestru
57	MOV_RR reg, reg	Skopiuj wartość z drugiego rejestru do pierwszego
69	SYSCALL imm	Wykorzystaj funkcję systemową o podanym numerze, używajac linuxowego ABI

IMPLEMENTACJA EMULATORA

Skoro już wiedzieliśmy mniej więcej, jak wygląda zbiór instrukcji programu, zdecydowaliśmy się napisać do niego własny interpre-

ter. Zrobiliśmy to w dwóch celach: po pierwsze, żeby mieć pewność, że dobrze zrozumieliśmy działanie każdego opkodu, a po drugie – żeby móc wykonać program i dostać flagę.

Znając działanie każdego opkodu, implementacja własnego silnika wykonującego nie jest trudniejsza niż napisanie dowolnego innego interpretera/emulatora i sprowadza się do wielkiego switcha, który w zależności od typu opkodu wykonywał odpowiednie operacje na parametrach. Do implementacji wybraliśmy Pythona, bo jest to język, w którym wszyscy czujemy się dobrze, a ponadto nadaje się do szybkiego prototypowania na CTFach.

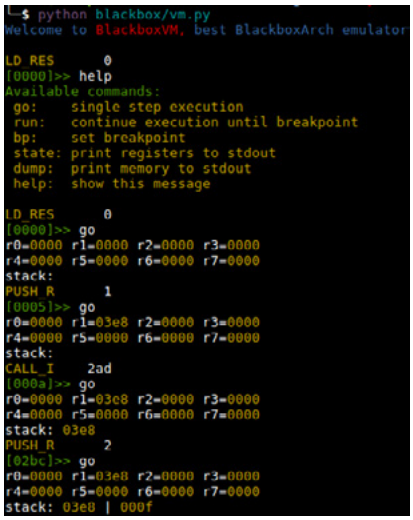
Sercem programu była wielka funkcja implementująca wszystkie potrzebne operacje:

```
def ins(op, args):
  global pc, regs, memory, free_ptr
  pc += 5
  op, a = parse_op(op, args)
  if op == "MOV_RR":
    regs[a[0]] = regs[a[1]]
  elif op == "CALL_I":
    stack.append(pc)
    pc += a[0]
  elif op == "JMP":
    pc += a[0]
  elif op == "JMP_NE":
    if cmp_flag != 0:
      pc += a[0]
  elif op == "JMP_LE":
    if cmp_flag == -1:
      pc += a[0]
  elif op == "JMP_GE":
    if cmp_flag == 1:
      pc += a[0]
  elif op == "RET":
    pc = stack.pop()
  elif op == "POP_R":
    regs[a[0]] = stack.pop()
  elif op == "PUSH_R":
    stack.append(regs[a[0]])
  elif op == "XOR_RR":
    regs[a[0]] ^= regs[a[1]]
  elif op == "SYSCALL":
    syscall(a[0])
  elif op == "MOV_RI":
    regs[a[0]] = regs[a[1]]
  elif op == "ADD_RI":
    regs[a[0]] += a[1]
  elif op == "LOAD":
    regs[a[0]] = memory[regs[a[1]]]
  elif op == "AND_RI":
    regs[a[0]] &= a[1]
  elif op == "CMP_RI":
    compare(regs[a[0]], a[1])
  elif op == "SUB_RI":
    regs[a[0]] -= a[1]
  elif op == "LD_RES":
    load_res(a)
  elif op == "CMP_RR":
    compare(regs[a[0]], regs[a[1]])
  elif op == "ADD_RR":
    regs[a[0]] += regs[a[1]]
  elif op == "STORE":
    memory[regs[a[0]]] = regs[a[1]]
  elif op == 'MLC_IR':
    regs[a[1]] = free_ptr * 1000 + 1000
    free_ptr += 1
  elif op == 'MLC_RR':
    regs[a[1]] = free_ptr * 1000 + 1000
    free_ptr += 1
  else :
    raise RuntimeError('unknown opcode')
```

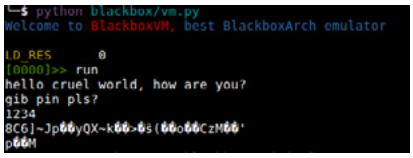
Rozwiązanie zadania *Blackbox* zostało nadesłane przez p4, polski zespół CTF-owy, który w chwili redagowania tego artykułu zajmował 3 miejsce w generalnej klasyfikacji CTFTime.org.

https://ctftime.org/team/5152
logo: http://i.imgur.com/tv6P8Nc.png (2000x2000)

Do tego dorzuciliśmy ładny interfejs oraz bardzo prosty debugger w celu uproszczenia analizy. Wynik naszych starań widać na screenie:

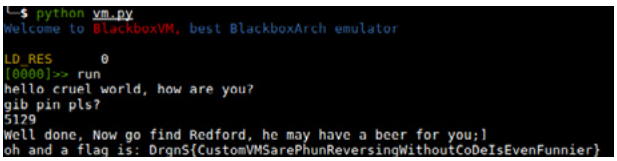


Mając interpreter, możemy nareszcie wykonać program! Niestety, to jeszcze nie koniec – program oczekuje od nas, że podamy mu pin:



Na powyższym screenie widać, co się dzieje, gdy wpiszemy 1234 jako PIN. Można się domyślać, że program deszyfruje dane, używając pinu jako klucza, i flagę otrzymamy, jedynie jeśli podamy prawidłowe hasło.

Co zrobić w tym przypadku? Wybraliśmy najprostsze, ale jednocześnie najpewniejsze rozwiązanie i po prostu uruchomiliśmy skrypt dla wszystkich możliwych wartości (atak ten jest znany jako *brute force*). Po kilku godzinach czekania udało nam się w końcu otrzymać dobry pin:



I dzięki temu otrzymaliśmy upragnioną flagę

```
DrgnS{CustomVMSarePhunReversingWithoutCoDeIsEvenFunnier}
```

PODSUMOWANIE

Opisane przez nas zadanie było dość nietypowe, jeśli chodzi o CTFy, ale przypadło nam do gustu. Mimo że architektura była wymyślona, to od zadania czuć było ducha „prawdziwego" reverse engineeringu. Zadanie składało się z dwóch części: wymagało od nas dość szczegółowego zrozumienia działania maszyny wirtualnej na niskim poziomie, ale potrzebowaliśmy także umiejętności programistycznych, aby napisać wydajny i bezbłędny interpreter.

Jarosław Jedynak, Adam Krasuski