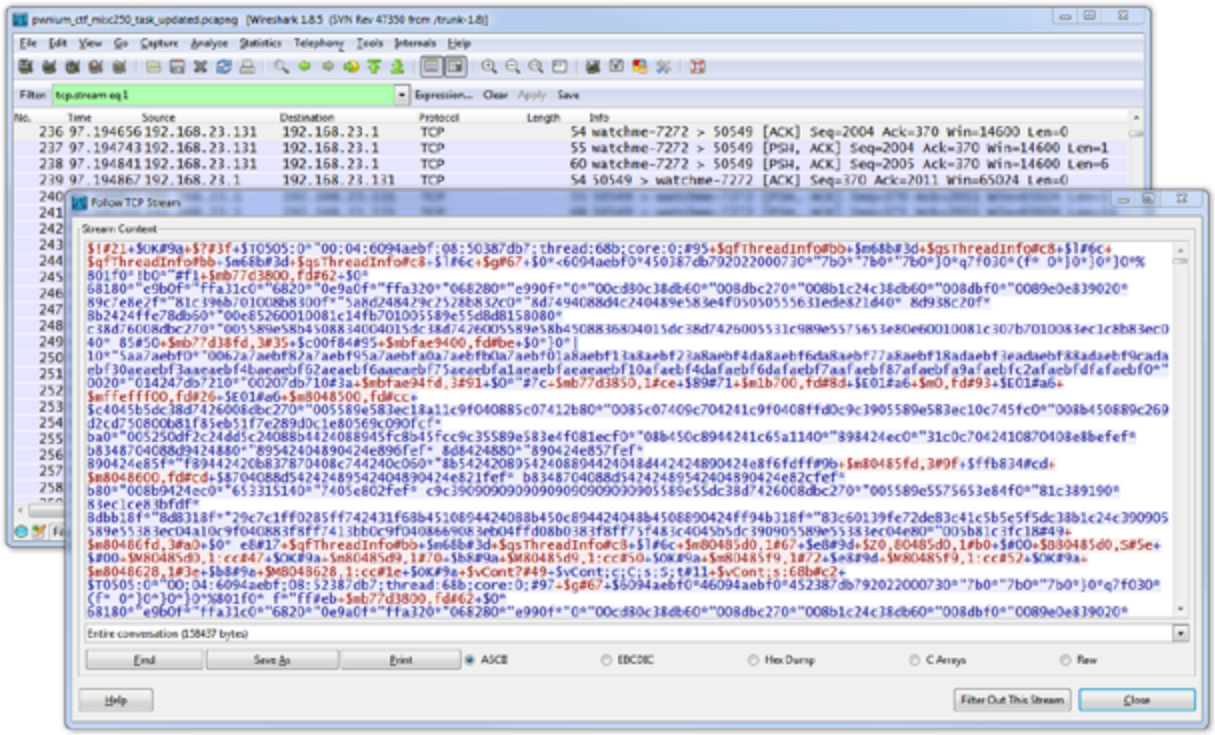
Gynvael Coldwind

Zdobyć flagę… Pwnium CTF 2014 – Remote KG

Średnio co około dwa tygodnie gdzieś na świecie odbywają się komputerowe Capture The Flag – zawody, podczas których kilku-, kilkunastoosobowe drużyny starają się rozwiązać jak najwięcej technicznych zadań z różnych dziedzin informatyki: kryptografii, steganografii, programowania, informatyki śledczej, bezpieczeństwa aplikacji internetowych itd. W serii „Zdobyć flagę…" co miesiąc publikujemy wybrane zadanie pochodzące z jednego z minionych CTFów wraz z jego rozwiązaniem.



CTF	Pwnium CTF 2014 http://ctf.pwnium.tn/
Waga CTFtime.org	5 (https://ctftime.org/event/150)
Liczba drużyn (z liczbą punktów większą od 1)	428
System punktacji zadań	Od 1 (proste) do 300 (umiarkowane) punktów.
Liczba zadań	24
Podium	1. oxAWESoME (Izrael) – 3186 pkt. 2. Plaid Parliament of Pwning (USA) – 3036 pkt. 3. Rdot.org (Rosja) – 2936 pkt.
Przedstawione zadanie	Remote KG (RE 250)

Tabelka: Informacje o CTF oraz o zadaniu

O CTFIE

Okres letni na scenie CTF to swoisty sezon ogórkowy – rozgrywane są jedynie pojedyncze turnieje, i to w stosunkowo dużych odstępach czasu. Ma to swoje zalety – drużyny mogą odpocząć, a także przygotować się do największego wydarzenia w roku – odbywających się na początku sierpnia finałów DEF CON CTF w Las Vegas, w stanie Nevada (któremu poświęcimy kolejny odcinek naszej serii).

Tunezyjski Pwnium CTF był miłym przerywnikiem w trakcie CTFowej „suszy". Turniej ten odbył się w tym roku po raz pierwszy i składał się zasadniczo z zadań nietrudnych, choć kilka z nich wymagało większego nakładu pracy (np. w jednym z zadań otrzymywało się uszkodzony kod QR, który należało „ręcznie" zdekodować).

Zadanie, które opisujemy w niniejszej części „Zdobyć flagę...", warte było 250 punktów i było jednym z najbardziej złożonych problemów podczas tego turnieju. Należy dodać, że zadanie zostało udostępnione niedługo przez zakończeniem Pwnium CTF i nie zostało rozwiązane przez żadną drużynę w czasie trwania zawodów. Nasza drużyna była dość blisko rozwiązania, ale niestety dotarliśmy do niego już po oficjalnym zakończeniu.

REMOTE KG

W tym zadaniu otrzymaliśmy plik z zapisem ruchu sieciowego w formacie PcapNg odczytywanym m.in. przez popularne narzędzie do analizy ruchu sieciowego – Wireshark. Celem, jak zwykle, było zdobycie flagi.

Czytelnicy, którzy chcieliby spróbować swoich sił z zadaniem, mogą znaleźć wspomniany plik pod poniższym adresem: http://gynvael.coldwind.pl/ctf-mirror/

Po otwarciu pliku wyświetliliśmy listę połączeń TCP (*Statistics → Conversations → TCP*), by spróbować wywnioskować, na czym się skupić. Okazało się, że organizatorzy postanowili uprościć sprawę, nazywając jeden z portów „watchme" (z ang. obserwuj mnie).

Kolejnym krokiem było wyświetlenie przekazywanych danych (*Follow Stream*). Przesyłane dane okazały się być nieznanym nam protokołem tekstowym. Kilka pierwszych pakietów wyglądało następująco (patrz również ilustracja na samym początku artykułu):

```
$!#21
+$OK#9a
+$?#3f
+$T0505:0*"00;04:6094aebf;08:50387db7;thread:68b;core:0;#95
+$qfThreadInfo#bb
+$m68b#3d
```

Przeglądając komunikacje dalej, natrafiliśmy na kilka charakterystycznych fragmentów, które podpowiedziały nam, z czym możemy mieć do czynienia:

```
$mbfae9400,fd#be
+$0* 7f039b95aebf00207db714247db780840408f4ef7e…
```

oraz

```
$vCont;s:68b#c2
```

Konkretniej, litera „m" wraz z liczbą zapisaną heksadecymalnie w zapytaniu, oraz stosunkowo duża ilość danych również zapisanych heksadecymalnie w odpowiedzi, skojarzyła nam się z zażądaniem przesłania fragmentu pamięci („m" od „*memory*"), natomiast „Cont" oraz „s" brzmiały jak „*continue*" oraz „*step*".

Mając powyższe słowa kluczowe oraz nazwę zadania („**Remote** KG"), domyśliliśmy się, że patrzymy na zapis sesji zdalnego debugowania nieznanej aplikacji (choć „KG" w nazwie zadania wskazywało na „*KeyGen*", czyli generator kluczy).

Najpopularniejszym debuggerem oferującym możliwość zdalnego debugowania jest GDB (GNU Debugger); zaczęliśmy więc od przejrzenia specyfikacji używanego przez niego protokołu[1], co okazało się być strzałem w dziesiątkę.

SEPARACJA PAKIETÓW

Zanim jednak przeszliśmy do wgłębiania się w samą sesję debuggera, zdecydowaliśmy się najpierw wyekstrahować dane poszczególnych pakietów TCP i zapisać je do oddzielnych plików – takie podejście ułatwia późniejszą pracę z pakietami, jak i ich analizę. Do problemu można podejść na wiele sposobów, od skorzystania z biblioteki obsługującej PcapNg (w tym wypadku należy pamiętać, że PcapNg zawiera ramki ethernetowe, które transportują pakiety protokołu IP, które z kolei transportują pakiety protokołu TCP, a dopiero dane zawarte w niektórych pakietach TCP są dla nas interestujące), aż do wyeksportowania pakietów lub danych strumienia TCP bezpośrednio z Wireshark w postaci pliku XML lub zestawu tablic stałych w składni języków C/C++. My zdecydowaliśmy się na ostatnią opcję.

Dla przykładu, kilka pierwszych wyeksportowanych pakietów wygląda następująco:

```
char peer0_0[] = {
0x24, 0x21, 0x23, 0x32, 0x31 };
char peer1_0[] = {
0x2b };
char peer1_1[] = {
0x24, 0x4f, 0x4b, 0x23, 0x39, 0x61 };
```

Chcieliśmy również, aby w docelowych plikach oprócz danych znalazł się również kierunek transmisji. Ostatecznie do separacji pakietów użyliśmy poniższego skryptu, stworzonego w języku Python:

```
import re
with open("export.c") as f:
  d = f.read().replace("\n","")
p = r"char peer([^_]+)_([^[]+)\[\] = {([^}]+)}"
d = re.findall(p, d)

for (i, p) in enumerate(d):
  data = ''.join(map(lambda x:chr(int(x, 16)),
    p[2].replace("0x", "")
      .replace(" ", "")
      .split(",")))
  fn = "packets/%.5u.packet" % i
  with open(fn, "w") as f:
    f.write("%s\n%s" % (p[0], data))
  print i
```

Po uruchomieniu powyższego skryptu otrzymaliśmy (w katalogu „packets") 6368 plików o nazwach od „00000.packet" do „06367.packet". Jeśli chodzi o kierunek transmisji zapisany na samym początku każdego pliku, to w naszym przypadku „0" oznaczało dane wysłane od GDB w kierunku serwera debugującego aplikację, a „1" odpowiedź od serwera do GDB. Mając rozdzielone pakiety, przeszliśmy do ich analizy.

GDB REMOTE SERIAL PROTOCOL

Protokół GDB okazał się być stosunkowo prosty i oparty na schemacie żądanie-odpowiedź. Każde żądanie rozpoczyna się od znaku dolara ($), po którym następuje właściwe polecenie, zazwyczaj w postaci jednego znaku (np. „m" oznacza „odczytaj pamięć"), oraz jego parametry (np. adres i liczba bajtów). Pakiet zakończony jest znakiem hash (#) oraz sumą kontrolną.

Serwer potwierdza odbiór polecenia pojedynczym znakiem plus (+), następnie je wykonuje, po czym odsyła odpowiedź skonstruowaną identycznie jak pakiet żądania (czyli $odpowiedź#sumakontrolna). Interpretacja treści odpowiedzi zależy od polecenia.

Większe partie danych są dodatkowo skompresowane algorytmem RLE (*Run-Length Encoding*), który stosowany jest jedynie w miejscach, w których ma to sens (pozostałe dane są przepisywane wprost). Fragment danych faktycznie skompresowany jest oznaczany znakiem gwiazdki, po którym następuje liczba powtórzeń zapisana w postaci znaku ASCII o kodzie wyrażającym faktyczną liczbę powtórzeń plus 29. Drobnym zaskoczeniem był dla nas fakt, że sam znak do powtórzenia (znak przed gwiazdką) z ciągu wejściowych danych należy również przepisać do danych wynikowych, tak, jakby liczba powtórzeń była większa o jeden.

Przykładowo, odpowiedź na żądanie odczytania trzech bajtów pamięci („000042.packet") wygląda następująco:

```
$0*"#7c
```

Dolar, hash oraz sumę kontrolną możemy odrzucić, co zostawia nam jedynie dane skompresowane RLE:

```
0*"
```

W powyższym przypadku znak 0 jest powtórzony pewną ilość razy, określoną przez cudzysłów. Kodem ASCII tego znaku jest 34; odejmując 29, dostajemy 5. Pamiętając, by przepisać również znak, który już wystąpił w danych, otrzymujemy sześć znaków 0 w zdekompresowanych danych (zapisanych heksadecymalnie, po dwie cyfry na bajt):

```
000000
```

Stworzona przez nas funkcja dekompresująca wygląda następująco:

Autor

```
def decode_rle(rle_data):
  o = []

  i = 0
  while i < len(rle_data):
    if rle_data[i] == '*':
      n = ord(rle_data[i+1]) - 29
      o.append(o[-1][0] * n)
      i += 2
      continue
    o.append(rle_data[i])
    i += 1

  return ''.join(o)
```

Wiedząc co nieco o protokole, przeszliśmy do implementacji skryptu, którego celem było…

WYDOBYCIE PAMIĘCI

Analizę sesji debuggera postanowiliśmy zacząć od wydobycia pamięci procesu z przesyłanych pakietów – wynikało to z założenia, że niezależnie od analizy pozostałych przesyłanych informacji i tak w pewnym momencie będziemy zmuszeni rzucić okiem na kod debugowanej aplikacji.

W celu wydobycia pamięci napisaliśmy prosty parser pakietów, który szuka odpowiedzi na żądania odczytu pamięci („m"), wyciąga z nich i dekompresuje dane, a następnie zapisuje je do pary plików nazwanych według następującego schematu:

```
NumerPakietu_Adres.bin
mem_Adres_NumerPakietu.bin
```

Powyższe nazewnictwo pozwala na szybkie sortowanie plików zarówno w kolejności chronologicznej, jak i na przeglądanie zmian w pamięci pod konkretnym adresem. Sam kod parsera przedstawia się następująco:

```
from struct import unpack

cmd_last = None
cmd_param = {}
mem_packet = 0

def decode_gdb(data):
  global cmd_last
  global cmd_param
  cmd_last = data[0]
  cmd_param.clear()

  if cmd_last != "m":
    return

  (addr, length) = map(lambda x: int(x, 16),
     data[1:].split(','))
  cmd_param["addr"] = addr
  cmd_param["length"] = length

  print "Read Memory: 0x%.8x (0x%x bytes)" % (
    addr, length)

def decode_remote(data):
  global mem_packet
  global cmd_last
  global cmd_param
  if cmd_last != "m":
    return

  if data[0] == "E":
    print "Memory read error (%s)." % data[1:]
    return

  membin = decode_rle(data).decode("hex")

  nm = "memory/%.3u_%.8x.bin" % (
     mem_packet, cmd_param["addr"])
  with open(nm, "wb") as f:
    f.write(membin)

  nm = "memory/mem_%.8x_%.3u.bin" % (
    cmd_param["addr"], mem_packet)
  with open(nm, "wb") as f:
    f.write(membin)

  mem_packet += 1

for i in xrange(6368):
  f = open('packets/%.5u.packet' % i, "rb")
  (side, data) = f.read().split("\n", 1)
  f.close()
  side = int(side)

  if data == "+": continue

  j = data.rindex("#")
  data = data[1:j]
  [decode_gdb, decode_remote][side](data)
```

Po uruchomieniu powyższego skryptu (do którego należy dokleić przedstawioną wcześniej funkcję `decode_rle`) otrzymaliśmy fragmenty przestrzeni adresów debugowanego procesu.

KOD W PAMIĘCI

Jak wspomnieliśmy wyżej, na tym etapie najciekawszym fragmentem pamięci dla nas był obraz głównego pliku wykonywalnego, który w systemach opartych o kernel Linux umieszczony jest najczęściej pod adresem 0x08048000. Posłużyliśmy się poniższym poleceniem, aby posklejać kolejne fragmenty pamięci z tego regionu i uzyskać obraz w jednym kawałku:

```
cat mem_08048000_477.bin mem_080480fd_478.bin mem_08048100_475.
bin mem_080481fd_476.bin mem_08048200_473.bin mem_080482fd_474.
bin mem_08048300_333.bin mem_080483fd_334.bin mem_08048400_108.
bin mem_080484fd_109.bin mem_08048500_005.bin mem_080485fd_006.
bin mem_08048600_007.bin mem_080486fd_008.bin mem_08048700_167.
bin mem_080487fd_168.bin mem_08048800_463.bin mem_080488fd_464.
bin mem_08048900_465.bin mem_080489fd_466.bin > elf
```

Należy w tym miejscu zwrócić uwagę, że obraz pliku wykonywalnego załadowanego do pamięci procesu nie jest identyczny z jego postacią na dysku, ale jest na tyle zbliżony, że nie utrudnia to analizy w znaczny sposób.

Po otwarciu otrzymanego obrazu w IDA Pro zlokalizowaliśmy funkcję `main` pod adresem 0x08048578. Po krótkiej analizie udało nam się wywnioskować nazwy kolejnych funkcji (szczególnie tych importowanych ze standardowej biblioteki języka C, jak `strlen` czy `scanf`), jak i potencjalną funkcję odpowiedzialną za wyliczenie flagi.

Ostateczna postać zdekompilowanej (za pomocą dekompilatora Hex-Rays) funkcji `main` prezentuje się następująco:

```
int main()
{
  int v0; // eax@1
  int v1; // ST20_4@1
  int v2; // ecx@1
  int result; // eax@1
  int v4; // [sp+24h] [bp-CCh]@1
  int v5; // [sp+88h] [bp-68h]@1
  int v6; // [sp+ECh] [bp-4h]@1

  v6 = v14;
  puts("Enter the username to get a serial:");
  scanf("%s", &v5);
  v0 = strlen((int)&v5);
  v1 = calc_flag(v0);
  sprintf(&v4, "%i-x075321-%d\n", v1, 6);
  printf("%s", &v4);
  scanf("%s", &v4);
  result = 0;
  if ( v14 != v6 )
    result = check_cookie(v2, v14 ^ v6);
  return result;
}
```

Jeśli chodzi o funkcję, którą nazwaliśmy `calc_flag`, a która znajdowała się pod adresem 0x08048534, jej zdekompilowana postać okazała się być dość krótka:

```
__int64 __cdecl calc_flag(int a1)
{
  double v1; // ST08_8@1

  v1 = (long double)(-880 * (554445 * a1 / 0x64u));
  return LODWORD(v1);
}
```

WYGENEROWANIE FLAGI

Okazało się, ze dalsza analiza nie jest konieczna – jak widać na powyższych listingach ostateczna flaga zależy jedynie od długości wpisanej nazwy użytkownika, a więc bez problemu mogliśmy wygenerować flagi dla wszystkich sensownych wariantów.

Zadanie to zrealizował poniższy kod:

```
#include <stdio.h>
int calc_flag(int l)
{
  double v1;
  v1 = (long double)(-880 * (554445 * l / 0x64u));
  return *(int*)&v1;
}

int main(void) {
  int i;
  for(i = 1; i < 10; i++) {
    printf("%i-x075321-%d\n", calc_flag(i), 6);
  }
  return 0;
}
```

Bibliografia

[1] *GDB Remote Serial Protocol.*
https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html

W efekcie jego wykonania otrzymaliśmy następujący zestaw kandydujących flag:

```
-805306368-x075321-6
-1610612736-x075321-6
33554432-x075321-6
-771751936-x075321-6
872415232-x075321-6
67108864-x075321-6
1711276032-x075321-6
905969664-x075321-6
-1744830464-x075321-6
```

Prawidłową flagą okazał się ciąg wyliczony dla długości równej 6.

PODSUMOWANIE

O ile samo zadanie nie należało do trudnych, tj. nie wymagało ani przesadnego wgłębiania się w protokół GDB, ani długich godzin spędzonych z disassemblerem, to było to bez wątpienia jedno z najbardziej oryginalnych problemów RE, z jakimi przyszło nam się zmierzyć. Należy też zaznaczyć, że w żadnym wypadku nie wyczerpało ono pomysłu analizy zdalnej sesji debugowania – można więc przypuszczać, że podobne zadania pojawią się i w przyszłych CTFach.

Rozwiązania zadania *Remote KG* zostały nadesłane przez Dragon Sector – jedną z Polskich drużyn CTFowych. *http://dragonsector.pl/*