Kontrola integralności, czyli jak się upewnić, czy 2+2 nadal wynosi 4

W życiu każdego programu nadchodzi moment, kiedy musi opuścić przytulne miejsce na dysku programisty i wyruszyć w świat, podbijając serca użytkowników swoją funkcjonalnością i interfejsem. Świat jest jednak niebezpiecznym miejscem, przepełnionym ludźmi, którzy będą chcieli zobaczyć, co jest w środku, jak działa i czy przypadkiem nie mogą mieć tego za darmo. Zakładając, że nasz program nie jest z gatunku Open Source, możemy „nauczyć" go kilku technik obrony. Jedną z nich jest tytułowa kontrola integralności, pozwalająca na wykrycie popularnych sposobów analizy i modyfikacji oprogramowania.

ZANIM ZACZNIEMY

Kontrola integralności jest dość dużym zagadnieniem i poruszenie wszystkich tematów i technik w jednym artykule jest z góry skazane na porażkę. Poniższy tekst ma za zadanie przybliżyć podstawy i przyjmuje następujące ograniczenia:

- kompilatory: gcc i clang;
- przykładowe programy kompilują się i działają na systemach OSX i Linux, a co za tym idzie, po lekkich modyfikacjach również na iOS'ie i Androidzie (dla kodu pisanego przy użyciu NDK). Wszystko powinno również działać w przypadku programów kompilowanych przy pomocy narzędzi Google Native Client (NaCL), jako że są one oparte o LLVM, tak jak i clang;
- skompilowany plik wykonywalny nie zawiera relokacji, czyli jest skompilowany jako PIE (Position Independent Executable), lub jest biblioteką dynamiczną. Biblioteki statyczne, a co za tym idzie również np. statyczne frameworki dla systemu iOS, nie będą działać.

Dlaczego pomijam tu Windowsa? Ogólnie prezentowane poniżej metody działają w identyczny sposób i na tym systemie, jednak pojawia się kilka problemów, które wymagają zastosowania nieco innych rozwiązań. Głównym problemem są relokacje – aplikacje w Windowsie, w przeciwieństwie do linuxowych zawsze kompilowane są tak samo, przez co zawierają relokacje, aby system mógł je ładować pod różnymi adresami w pamięci (np. z powodu ASLR). W przypadku Linuxa istnieje możliwość skompilowania kodu tak, aby relokacji nie było. Przykładowo linuxowe biblioteki dynamiczne domyślnie kompilowane są w ten sposób. W czym przeszkadzają nam relokacje? Ich istnienie powoduje zmianę niektórych bajtów kodu wykonywalnego podczas ładowania binarki. Jako że sumy kontrolne sprawdzają, czy w kodzie nic się nie zmieniło, można się domyślić, jaki efekt będzie miało stado bajtów przyjmujących inną wartość przy każdym uruchomieniu. Można te miejsca znaleźć i odpowiednio korygować wynik sumy, ale wykracza to poza zakres tego artykułu.

Gwoli ścisłości – w starszych Windowsach (XP), gdzie ASLR nie zawsze działał, lub go nie było, tablice relokacji mogły być pomijane i biblioteka/aplikacja zakładała, że zostanie załadowana pod domyślny adres, czyli podczas ładowania żadne bajty nie były zmieniane i suma kontrolna byłaby stabilna.

CO POTRAFI NASZ PROGRAM?

Przyjmijmy prostą wersję, która sprawdza wprowadzony klucz i jeśli jest on poprawny, zwraca „super tajną informację". Program wygląda następująco:

Listing 1. 00_base.c – podstawowy program

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <stdint.h>
4.
5.  int verifyKey(const char* key){
6.    uint8_t i = 0;
7.    uint8_t v1 = 0;
8.    uint8_t v2 = 0;
9.
10.   while (key[i] && i<10){
11.     if (key[i] >= 'a' && key[i] <= 'f'){
12.       ++v1;
13.     }
14.     else if (key[i] >= '0' && key[i] <= '9'){
15.       ++v2;
16.     }
17.     ++i;
18.   }
19.
20.   if (!key[i] && i == 10 && v1 == 2){
21.     return v1+v2;
22.   }
23.
24.   return -1;
25. }
26.
27. int main(int argc, char** argv){
28.   if (argc < 2){
29.     printf("Bez klucza nic nie powiem\n");
30.     return -1;
31.   }
32.
33.   if (verifyKey(argv[1]) == 4){
34.     printf("Super tajna informacja\n");
35.   }
36.   else{
37.     printf("Dalej nic ci nie powiem\n");
38.   }
39.   return 0;
40. }
```

Nasz finezyjny algorytm sprawdzający klucz znajdujący się w funkcji `verifyKey` przechodzi znak po znaku przez ciąg przekazany jako parametr przy uruchomieniu. W zmiennej `v1` zlicza wystąpienia liter z przedziału `[a-f]`, w `v2` ląduje natomiast ilość liczb z przedziału `[0-9]`.