Zaciemnianie kodu, czyli jak ein**g6aty**a9**zcć**u$**ind**ź@**urtu**

Pani w szkole zawsze powtarzała, że kod musi być ładny i prosty do zrozumienia, bo to ułatwia pracę nam i innym. Nie myślała jednak o tym, że „inni" nie ograniczają się tylko do ludzi z naszego zespołu, ale obejmują także wszystkich próbujących sprawdzić, w jaki sposób działa nasz algorytm wyboru najlepszego kapelusza na podstawie zawartości naszej szafy i ostatnich wpisów na blogach modowych, lub też w jaki sposób sprawdzamy, czy wprowadzony klucz licencyjny jest poprawny. W niniejszym artykule chcielibyśmy zaprezentować metody obfuskacji, które sprawią, że analiza naszego kodu nie sprowadzi się do prostego użycia deasemblera/dekompilatora, ale pochłonie znacznie więcej czasu, pieniędzy, krwi i potu atakującego.

TYTUŁEM WSTĘPU

Obfuskacja jest tematem bardzo obszernym, wciąż pojawiają się kolejne publikacje dotyczące nowych technik lub ich modyfikacji, istnieje też wiele narzędzi (darmowych i płatnych) pozwalających nam na automatyczne zaciemnienie naszego kodu. Tekst, który czytelnik ma przed sobą, nie będzie gotową receptą na napisanie obfuskatora do dowolnego języka, ma bardziej na celu zaprezentowanie dostępnych technik i zasad ich działania.

Należy również pamiętać, że tak jak z innymi metodami zabezpieczania kodu – mają one na celu spowolnienie ataku, a nie jego uniemożliwienie. Oprócz technik zaciemniania równolegle rozwijane są algorytmy mające na celu automatyczne odwrócenie tego procesu, które same w sobie są również bardzo ciekawym tematem (i źródłem nowych pomysłów używanych do zaciemniania).

Obfuskacja ma miejsce na jednym z trzech etapów:

- Transformacje kod źródłowy -> kod źródłowy
- Transformacje kod pośredni -> kod pośredni
- Transformacje kod maszynowy -> kod maszynowy

Wszystkie mają ten sam cel – zmienić wygląd programu tak, aby zrozumienie zasad jego działania zajmowało dużo czasu, jednocześnie nie zmieniając jego funkcjonalności. W czym tkwią różnice?

Transformacje kod źródłowy -> kod źródłowy

Pozwalają nam na zamianę naszego czytelnego kodu (z którego pani w szkole byłaby dumna) na kod bardzo trudny do czytania. Plusem takiego rozwiązania jest to, że po takiej transformacji możemy użyć dowolnego kompilatora, gdyż kod dalej jest poprawny, jedynie wygląda bardziej egzotycznie. Dodatkowo, mimo tego, że kod wynikowy ma dziwną strukturę, możemy go debugować, widząc źródło, co jest dużą pomocą, gdyż błędy w kodzie mogą pojawiać się dopiero po zaciemnieniu i wynikać z błędów przekształceń w obfuskatorze (czasem przekształcenie zadziała, mimo że nie powinno, i np. zmienna zostanie zainicjalizowana złą wartością), możemy też trafić na błędy kompilatora, który w ferworze optymalizacji naszego pokręconego kodu zapomni, że coś nie zawsze musi być ładnie poukładane w pamięci, i wyemituje opkody powodujące losowe błędy (zabawa przy debugowaniu gwarantowana). Poprawianie tego typu błędów sprowadza się albo do poprawy kodu błędnie funkcjonującego elementu, albo do zmiany struktury naszego źródła tak, aby robiąc to samo, wyglądała nieco inaczej. Na przykład jeśli widzimy, że zmienna przyjmuje dziwne wartości początkowe po zaciemnieniu, możemy spróbować rozłożyć jednolinijkową deklarację i inicjalizację zmiennej na dwie osobne operacje.

Jest też kilka minusów – aby wykonać taką transformację, nasz obfuskator musi rozumieć składnię danego języka (np. C), czyli musimy użyć parsera, który zwykle jest nietrywialny. Jako, że nie będziemy (raczej) sami pisać parsera, użyjemy jednego z dostępnych, a te często wspierają język w konkretnej wersji (np. C99, ale już nie C11). Dodatkowo kompilatory wspierają różne dodatki specyficzne dla danego kompilatora lub też mają różną składnię dla tych samych elementów (na przykład wstawki asemblerowe w gcc i MSVC), co też może być problemem albo dla parsera, albo dla naszego obfuskatora, który nie będzie rozumiał, że `__pragma` to to samo co `#pragma`, tylko może być w dziwnych miejscach. Kolejnym problemem jest to, że kod, który przekształcimy, musi zostać skompilowany, a kompilatory stosując coraz bardziej zaawansowane techniki optymalizacji, są często w stanie usunąć bądź uprościć niektóre fragmenty kodu, które właśnie z trudem wygenerowaliśmy.

Wszyscy lubimy przykłady, więc oto jeden z nich: wyobraźmy sobie sytuację, w której chcemy zamaskować łańcuch znaków używany w kodzie. Używanym językiem jest C, kompilator to clang (jedna z ostatnich wersji), system OSX. Plan jest następujący:

- mamy funkcję `decrypt_string`, która przyjmuje tablicę bajtów (pierwszy z nich to długość stringa), XORuje bajt po bajcie ustalonym kluczem i wynik zwraca w nowo zaalokowanym buforze. Wygląda tak:

```
char* decrypt_string(const char* string){
  unsigned int len = string[0] + 1;
  char* res = calloc(len, 1);
  for (int i=1; i<len; ++i){
    res[i-1] = string[i] ^ 0xaf;
  }
  return res;
}
```