

Pimp CrackMe doc

Autorzy: Mateusz „j00ru//vx” Jurczyk, Gynvael Coldwind//vx

2011-05-14

SPOILER: JEŚLI ZAMIERZASZ ROZWIĄZYWAĆ TO CRACKME, NIE CZYTAJ DALEJ

1. Wymagania sprzętowe

Dowolny procesor z rodziny Intel x86.

2. Wymagania systemowe

System operacyjny Microsoft Windows XP lub nowszy, w wersji 32-bitowej.

CrackMe NIE działa na systemach 64-bitowych, w trybie emulacji WOW64.

3. Cel CrackMe

Głównym celem zadania jest przedstawienie interesującej koncepcji dotyczącej utrudniania dynamicznej analizy działania konkretnej aplikacji. Pomysł opiera się na wykorzystaniu mechanizmu segmentacji w trybie chronionym, dostępnym na procesorach z rodziny Intel x86. Dzięki obserwacji, że system Windows udostępnia nieudokumentowany interfejs, umożliwiający tworzenie własnych, całkowicie nowych deskryptorów segmentów w obrębie LDT (*Local Descriptor Table*), możliwe staje się rozbicie całej logiki zabezpieczenia na mniejsze fragmenty kodu, posiadające i wywoływane w swoich własnych segmentach kodu (tj. segmentach, dla których adres bazowy jest różny od zera).

Ze względu na fakt, że sam mechanizm segmentacji jest (wyłączając pojedyncze zastosowania) uważany za przestarzały / nieistotny, wiele narzędzi wspomagających inżynierię wsteczną (takich jak najpopularniejsze debuggery: OllyDbg / ImmunityDbg) nie radzi sobie z analizą aplikacji, który w sposób aktywny wykorzystują ten *zapomniany* mechanizm.

Bezpośrednim celem użytkownika CrackMe jest odnalezienie poprawnego, 64-bitowego klucza wejściowego, po „wyklinaniu” którego aplikacja poinformuje o poprawnym rozwiązaniu zadania.

4. Mechanika zadania

Właściwe zadanie, z którym trzeba się uporać, aby złamać CrackMe, zostało zaimplementowane

w formie prostej maszyny wirtualnej. Procesor dysponuje znacznie ograniczoną ilością rejestrów ogólnego przeznaczenia, pamięci, oraz zbiorem instrukcji (patrz *Architektura maszyny wirtualnej*). Kawałki kodu odpowiedzialne za obsługę wykonania poszczególnych instrukcji zostały umieszczone w oddzielnych segmentach kodu, które są następnie wywoływane przez główny *dispatcher* procesora.

Oprócz wykorzystania samego mechanizmu segmentacji, w zadaniu obecne jest dodatkowe utrudnienie – jedna z instrukcji wirtualnego procesora jest odpowiedzialna za pseudo-losowe wymieszanie znaczenia zapisu dalszych instrukcji (*SHUFFLE*). Kolejność, w jakiej dokonywane są zamiany znaczenia poszczególnych *opcode'ów* jest ściśle związana z użytym generatorem liczb pseudolosowych, a w szczególności – z ziarnem, dostarczonemu generatorowi przez samego użytkownika (jako część wejściowego kodu). Oznacza to, że interpretacja znaczenia poszczególnych instrukcji wykonywanych na wirtualnym procesorze jest uzależniona od danych wejściowych.

Poprawne hasło, jakiego oczekuje się od użytkownika składa się z dwóch 32-bitowych części: ziarna generatora (starszy DWORD), oraz wartości, która jest weryfikowana na podstawie wyjścia kodu wirtualnej maszyny (składającego się z przekształceń arytmetycznych, logicznych, oraz bloków funkcji hashujących).

Przykładowy, poprawny kod wejściowy to D984 5C67 AED6 334B

5. Mapa przycisków (interfejs graficzny)

0	4	8	C
1	5	9	D
2	6	A	E
3	7	B	F

6. Architektura maszyny wirtualnej

- Tryb *little-endian*
- Dwa 32-bitowe rejestry ogólnego przeznaczenia: A, B
- Dostęp do pamięci operacyjnej o wielkości 0x10000 bajtów.
- Zbiór instrukcji

Instrukcja	Parametr 1	Parametr 2
SHUFFLE	-	-
EXIT	-	-
JMP	REL IMM32	-
JE	REL IMM32	-

JNE	REL IMM32	-
BACKUP	-	-
RESTORE	-	-
HASH	A (implicit)	-
MOV	[IMM32]	A
MOV	A	[IMM32]
MOV	B	[IMM32]
MOV	A	IMM32
MOV	A	B
XCHG	A	B
CMP	A	B
XOR	A	B
AND	A	B
AND	A	IMM32
OR	A	B
NOT	A	-
ADD	A	B
ADD	A	IMM32
ROL	A	IMM32
ROR	A	IMM32
SHL	A	IMM32
SHR	A	IMM32
MUL	A	B
MUL	A	IMM32

7. Rozwiązanie zadania

Rozwiązanie zadania składa się z dwóch etapów – odnalezienia poprawnej wartości ziarna, które w rezultacie spowoduje wygenerowanie poprawnego kodu maszynowego wirtualnej maszyny (w trakcie „mieszania” *opcode’ów* instrukcją SHUFFLE), oraz wyliczenie (znalezienie?) klucza, który po przetworzeniu przez kod maszyny wirtualnej zostanie pozytywnie zweryfikowany.

Weryfikacja polega na porównaniu aktualnej wartości rejestru A z wartością „wzorcową” za każdym razem, kiedy wykonywana jest instrukcja SHUFFLE. Pełni ona więc rolę zarówno „obfuskatora” dalszej części kodu, jak również momentu sprawdzania poprawności wyniku obliczeń. W całym (krótkim) kodzie maszyny wirtualnej występuje dokładnie 8 odwołań do instrukcji SHUFFLE – oznacza to, że otrzymany na wejściu klucz jest sprawdzany osiem razy w trakcie działania wirtualnej maszyny – jeśli wszystkie etapy weryfikacji zakończą się powodzeniem, klucz jest akceptowany jako poprawny.

W ogólności, w momencie n-tego wywołania SHUFFLE generator liczb pseudolosowych zostaje

na nowo zainicjalizowany liczbą, będącą $4 \cdot n$ najmłodszymi bitami *seeda* wprowadzonego przez użytkownika. Dzięki temu, odzyskanie pełnej, poprawnej wartości ziarna możliwe jest poprzez wyliczanie kolejnych, 4-bitowych części poszukiwanej wartości. W celu znalezienia odpowiedniej wartości pojedynczej części, możemy „przemieszać” kod za pomocą wszystkich możliwych 16 kombinacji, a następnie wybrać spośród nich te, które wydają się najbardziej prawdopodobne (np. kod powstały przy ich pomocy nie zawiera instrukcji EXIT).

Drugą część zadania można rozwiązać poprzez przetestowanie wszystkich możliwych wartości klucza, tj. rozwiązaniem siłowym. Prawdopodobny brak innych rozwiązań jest spowodowany faktem, że 32-bitowa wartość wejściowa jest przepuszczana przez ciąg funkcji haszujących, co w założeniu powinno skutkować brakiem możliwości odwrócenia jej działania.

Dzięki przedstawionemu podejściu, z jednej strony niemożliwe jest całkowite odgadnięcie poprawnego klucza (zakres liczby 64-bitowej jest zbyt szeroki), z drugiej użytkownik nie jest (w zamyśle) w stanie bezpośrednio wyliczyć oczekiwanej wartości na podstawie dostarczonego kodu.