

1. Gra rozpoczyna się próbą uruchomienia komputera. Zmodyfikuj grę, aby przy próbie wpisania hasła do dysku zamiast gwiazdek wyświetlały się znaki hasła. Opisz w sprawozdaniu w jaki sposób dokonano modyfikacji.

Początkowo oczywiście trzeba znaleźć miejsce w kodzie, które odpowiada za wypisywanie znaków gdy użytkownik wpisuje hasło. Dzięki temu, że mamy dołączone symbole do zadania i funkcje mają dosyć wymowne nazwy można szybko zlokalizować większość funkcji które mają coś wspólnego z komputerem. Zatem wyszukałem w IDA wszystkie które zawierają infix **PC**, w ten sposób znalazłem funkcję *pc\_putch*. Jednocześnie pamiętałem o tym, że szukam prawdopodobnie jakiegoś przypisania symbolu \* (0x2A) który na szczęście wystąpił w tej funkcji, co mnie tylko utwierdziło w tym, że znalazłem odpowiednie miejsce i teraz wystarczy przekazać tam znaki wpisywanego hasła. Uznałem zatem, że warto sprawdzić co wywołuje tę funkcję bo musiała być ona odpowiedzią na jakieś zdarzenie. Używając IDA można wyszukać cross-references przez kliknięcie X na funkcji, to przenosi nas do *pc\_keypress*. W niej możemy znaleźć kilka odwołań do rejestru *r9d* który jest porównywany do spacji (0x20) oraz carriage return (0xD), co mnie nakierowało, że mogą być tam znaki hasła, następnie upewniłem się używając x64dbg, że faktycznie tak jest i zamieniłem przypisanie \* (0x2A) na rejestr *r9w* tzn.: *mov word ptr ds:[rcx+rdx\*1], 0x2A* na *mov word ptr ds:[rcx+rdx\*1], r9w* (z *nopem*) Miałem na początku pewne wątpliwości co do wielkości rejestrów, lecz po prostu sprawdziłem i jest okej.

2. Gra weryfikuje poprawność wprowadzonego hasła do dysku. W sprawozdaniu opisz algorytm, za pomocą którego hasło jest weryfikowane. Na podstawie zebranych informacji, odnajdź poprawne hasło.

Po wyszukaniu w IDA funkcji z infixem **PASSWORD** wyskakują trzy funkcje z czego jedna z nich ma znaczącą nazwę *pc\_check\_luks\_password*. Na początku do rejestru *r9d* zapisywana jest "pewna" wartość a następnie rejestr ten jest porównywany do 8 – jeśli równość nie jest spełniona to jest wykonywany *xor al, al* więc pewnie było to coś w stylu *return false*. Po chwili analizy z wykorzystaniem x64dbg można zauważyć, że ta "pewna" wartość to jest liczba wpisanych znaków hasła, w ten sposób dowiadujemy się, że hasło musi być długości 8. Przechodząc dalej mamy wyzerowanie rejestru *ecx*, który jest używany jako zmienna do kontrolowania pętli. Tak wnioskuję, ponieważ dalej jest wykonywane *inc ecx* i *cmp* z rejestrem *r9d* równym 8. Następnie wykonane jest załadowanie adresu do rejestru *r8*, gdzie adres ten zapewne wskazuje na bufor z hasłem. Dodatkowo przenosimy wartość 0x91 do rejestru *edx*, jest on dalej wykorzystywany następująco:

1) Sprawdzamy czy wartość w rejestrze *dl* jest równa 1 – jeśli tak, to sprawdzamy czy pierwszy bajt pod adresem z rejestru *r8* jest liczbą tzn. najpierw standardowo konwertujemy z chara na inta odejmując '0' i sprawdzamy czy wynik jest mniejszy równy 9. Jeśli tak, to jest wykonywany skok do *return false*.

2) Na końcu ciała pętli robimy bitowe przesunięcie w prawo o 1 rejestru *edx*.

Można na to spojrzeć w ten sposób: zapiszemy wartość 0x91 jako 0b10010001 – teraz lepiej widać, że cała powyższa część algorytmu oznacza nic innego jak to, że na pierwszej, piątej i ostatniej pozycji nie może być liczby. Po wyjściu z pętli jest wykonywana instrukcja *call hash\_password* a następnie jej wynik jest porównywany z pewną stałą. Wynik tego porównania jest kolejnie wykorzystywany w instrukcji *setz al*, która zapala bajt pod warunkiem, że *cmp* ustawiło ZF = 1. Wnioskujemy zatem, że musimy znaleźć hasło, którego hash będzie równy powyższej stałej, więc musimy poznać jaką działa ta funkcja hashująca. Na początku rozpisywałem sobie krok po kroku co robi ta funkcja, lecz opiszę jedynie kluczową obserwację tzn. w pewnym momencie jest wykonywana instrukcja: *xor rax, qword ptr ds:[r10+rdx\*8+0x9730]*

Używając x64dbg podejrzalem jaka wartość znajduje się pod adresem  $r10+rdx*8+0x9730$ , np. dla pierwszego obrotu pętli i hasła "hasloooo" dostajemy: 0x8E43C87E03060C18.

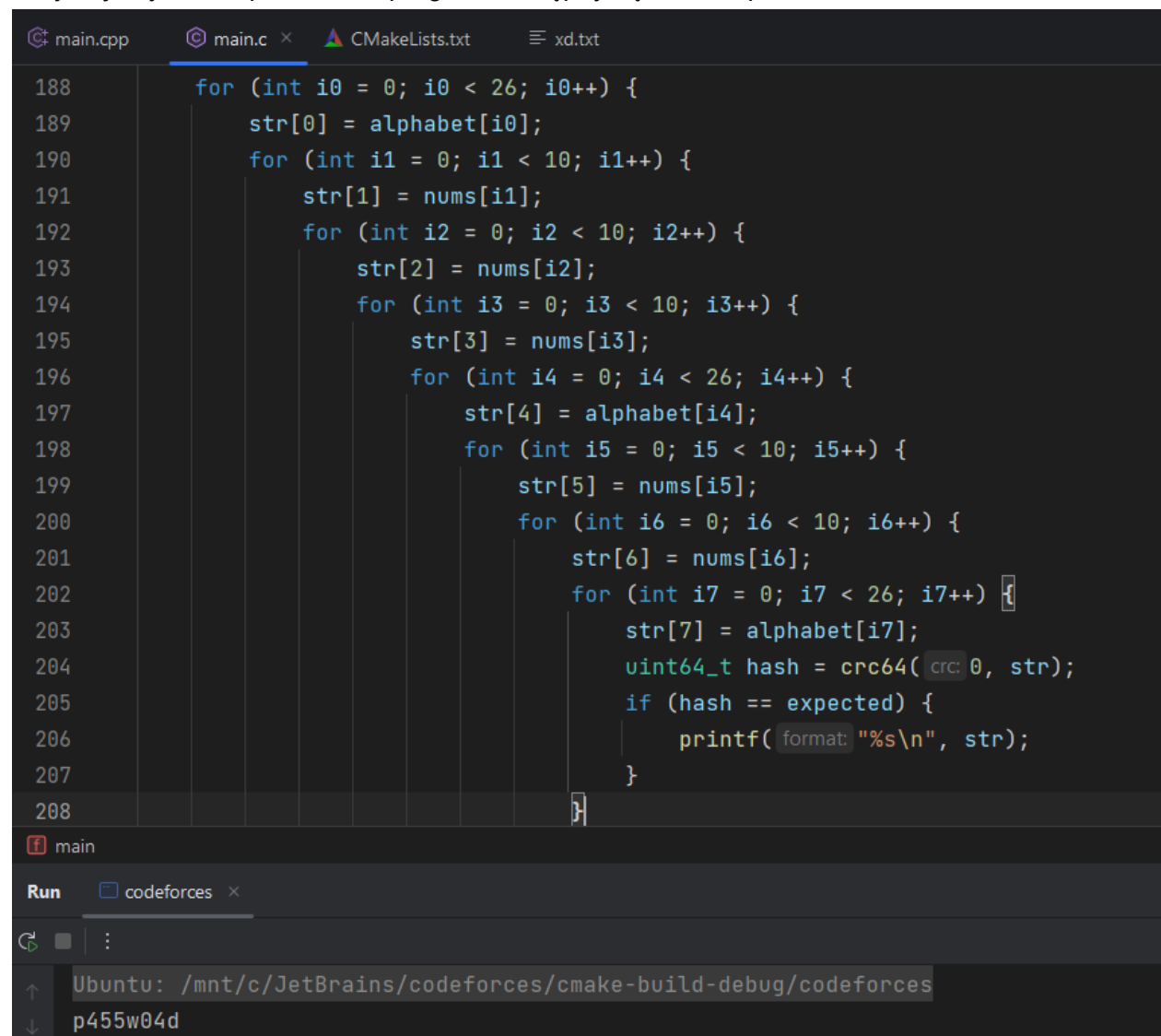
Wyszukując tą wartość w Google od razu wyskakuje kilka implementacji algorytmu CRC64, po chwili analizy można się przekonać, że dokładnie ten sam algorytm był zastosowany do hashowania naszego hasła. Podsumowując zebrane informacje wiemy że:

1) Hasło ma 8 znaków

2) Hasło na pierwszej, piątej i ósmej pozycji nie ma liczby, a można wpisywać jedynie liczby i małe litery (o czym można się przekonać próbując wpisać cokolwiek innego)

3) Na końcu podane hasło jest hashowane za pomocą algorytmu CRC64 i porównywane do stałej

Z tymi informacjami można spróbować odwrócić hasha (o ile się w ogóle da) lub napisać program który je próbuje znaleźć brute forcem. Warto dodać, że drugi warunek nie gwarantuje, że na reszcie pozycji będą liczby a tylko na wybranych będą litery ale logicznie jest tak założyć :) Z tym śmiałym założeniem mamy  $10^5 * 26^3$  możliwych haseł więc całkiem sensowne ograniczenie do brute force'a. W ten sposób otrzymujemy hasło: p455w04d, program dostępny będzie w zipie.



```
188     for (int i0 = 0; i0 < 26; i0++) {
189         str[0] = alphabet[i0];
190         for (int i1 = 0; i1 < 10; i1++) {
191             str[1] = nums[i1];
192             for (int i2 = 0; i2 < 10; i2++) {
193                 str[2] = nums[i2];
194                 for (int i3 = 0; i3 < 10; i3++) {
195                     str[3] = nums[i3];
196                     for (int i4 = 0; i4 < 26; i4++) {
197                         str[4] = alphabet[i4];
198                         for (int i5 = 0; i5 < 10; i5++) {
199                             str[5] = nums[i5];
200                             for (int i6 = 0; i6 < 10; i6++) {
201                                 str[6] = nums[i6];
202                                 for (int i7 = 0; i7 < 26; i7++) {
203                                     str[7] = alphabet[i7];
204                                     uint64_t hash = crc64(0, str);
205                                     if (hash == expected) {
206                                         printf("format: \"%s\\n\", str);
207                                     }
208                                 }
209                             }
210                         }
211                     }
212                 }
213             }
214         }
215     }
```

main

Run codeforces

Ubuntu: /mnt/c/JetBrains/codeforces/cmake-build-debug/codeforces

p455w04d

3. Ciągłe zmuszanie do uruchomienia komputera przez mamę za każdym restartem gry jest nieco irytujące, prawda? Zmodyfikuj grę w taki sposób, by jej początkowy stan był taki jak po próbie uruchomienia komputera z mamą. Opisz w jaki sposób ustalana jest aktualna "faza" gry.

Żeby dowiedzieć się w jaki sposób ustalana jest aktualna faza gry, trzeba najpierw zastanowić się, czego się tak naprawdę szuka. Pierwszy pomysł jaki mi się nasunął to pewnego rodzaju maska bitowa, która by trzymała takie informacje. Z tą myślą z tyłu głowy uznałem, że w poszukiwaniu jej warto przejrzeć funkcje których nazwy wskazują na jakieś interakcje z mamą. Mamy całe 6 funkcji *mom\_ask\_pc* lecz jedynie w drugiej z nich jest wywoływana pewna magiczna funkcja *check* w której jest wykonywana instrukcja *bt* (bit test), co już może wskazywać na pewnego rodzaju testowanie flagi. Jak wyszukamy cross-references do adresu z jakiego sączytywane są wartości do rejestru to dostajemy odwołania do funkcji: *mark*, *clear* i *check*, których nazwy są całkiem wymowne. Domyślamy się zatem, że aktualna faza gry jest ustalana na podstawie wyżej wspomnianego bitsetu, gdzie funkcje *mark* i *clear* pozwalają manipulować fazą gry a *check* służy oczywiście do sprawdzenia aktualnej fazy. Oznacza to, że aby zmodyfikować początkowy stan gry możemy po prostu podejrzeć stan bitsetu po interakcjach z mamą i wykonać zwykłego *or'a* na tej zmiennej na początku funkcji *SDL\_main*, która obsługuje główną pętlę gry. Z pomocą x64dbg ustaliłem, że stan rejestru po wykonaniu wszystkich rozmów z mamą to *0x1747*, teraz wystarczy zastąpić jedno wywołanie *mark* w funkcji *SDL\_main* na: *or qword ptr ds:[0x00007FF7EFBC1AC8], 0x1747*

Jeszcze ważna uwaga – jeśli nie chcemy, żeby stan był taki jak po poprawnym wpisaniu hasła to musimy przekazać **0x747** a nie **0x1747**. Dlatego dałem dwa pliki wykonywalne *final.exe* oraz *final\_no\_pass.exe*. Można także ustawić breakpoint na funkcji *mark* i w ten sposób podejrzeć po kolei które bity są zapalane a następnie wywołać funkcję *call* z ustalonymi liczbami, ale początkowe rozwiązanie wydaje mi się bardziej odporne na jakieś pomyłki :)

4. Zmodyfikuj grę w taki sposób, aby dało się wygrać walkę z Garym. Postaraj się nie używać wyłącznie brutalnej siły, zmniejszenie wartości HP lub zwiększenie wartości ataku nie będzie oceniane pełną liczbą punktów. Zbadaj na czym polega nieuczciwa przewaga przeciwnika i wyrównaj szanse, stosując odpowiednią modyfikację.

Po kilku walkach z Garym można zauważyć, że nawet jak jest się już bardzo blisko pokonania przeciwnika to leczy się on w kolejnej turze i zostawia nas bez szans. Zatem jeśli udałoby się go zmusić zawsze do ataku, to mielibyśmy jakieś szanse na wygraną. Wyszukując funkcje których nazwy zawierają słowo **BATTLE** możemy znaleźć *battle\_keypress*, której nazwa już się dobrze kojarzy, ponieważ wcześniej kluczowy kod także był w *pc\_keypress*. Pomijając całą skomplikowaną logikę która leży w tej funkcji, na końcu możemy zobaczyć dwa znaczące wywołania tzn. *call cs:gEnemyStrategy* oraz *call who\_goes\_first*. Podglądając wartość zmiennej *cs:gEnemyStrategy* z pomocą x64dbg możemy zobaczyć, że zawiera ona wskaźnik do funkcji *strategy\_endless\_healing*. Wyszukując funkcje których nazwa zawiera słowo **STRATEGY** możemy znaleźć także strategię *simple* oraz *status*, zatem wystarczy teraz zamienić wywołanie *call cs:gEnemyStrategy* na *call strategy\_simple* i przeciwnik przestanie się "nieskończenie" leczyć. Można także było podmienić tę strategię mniej bezpośrednio tzn. tak aby *cs:gEnemyStrategy* zawierało wskaźnik do *simple*. Zauważmy, że zmienna ta jest ustawiana w funkcji *start\_battle*, chyba na podstawie wybranego obiektu trenera przeciwnika np. *gTrainer\_Gary\_Bulbasaur* ale to tylko dodałem jako dygresja, bo bezpośrednia podmiana jest łatwiejsza. Dodatkowo także można zmienić *who\_goes\_first*, żeby zawsze zaczynał pierwszym ale to już chyba byłoby użycie brutalnej siły więc nie dodawałem tej modyfikacji (choćby bardzo zwiększało winratio).