Wstęp

Poniższy tekst jest przykładem przedstawiającym w jaki sposób można “naginać” pewne reguły gier wbrew woli jej twórców. W przygodówce Primordia (jeżeli ktoś nie grał, a lubi point’n’click to polecam) mamy kilka zakończeń, które wynikają z posiadanego przez nas ekwipunku. Otóż podczas gry możemy zdobyć oraz stracić pewne przedmioty, które będą nam potrzebne w kluczowym końcowym fragmencie fabuły. W przypadku ich braku zamykają się przed nami pewne ścieżki zakończenia całej historii. Ponieważ całość mnie wciągnęła, stwierdziłem, iż gra zasługuje na jedyne słuszne dobre zakończenie (⊙‿⊙) . Niestety w wyniku swojej nieuwagi straciłem pewien potrzebny mi do tego przedmiot. Rozwiązania są dwa: cofamy się do dość dalekiego zapisu gry i powtarzamy tytuł od nowa albo grzebiemy w jej pamięci (͡° ͜ʖ ͡°)ﾉ⌐■-■ .

Kwestie organizacyjne

Po pierwsze ten materiał nie będzie zawierał spojlerów. Pokażę jak oszukując grę dodać do ekwipunku postaci jakiś przedmiot, ale nie będzie to jeden z tych przedmiotów “ważnych” dla zakończenia fabuły. Dodaje się je analogicznie, a jedyne co musicie sami wybadać, to liczbowy identyfikator tego przedmiotu w grze. Po drugie będę korzystał z wersji linuksowej gry, ale poniższy opis powinien być pomocny również dla osób, które chcą zrobić podobny numer na wersji windowsowej.

Najpierw zwiad

Zanim zacznie się walczyć z przeciwnikiem trzeba go poznać. Polecam uruchomić grę w konsoli lub po prostu zajrzeć w jej logi. Dlaczego? Otóż są spore szanse na to, iż gra “przedstawi nam” silnik na jakim bazuje. W przypadku odpalenia Primordii w konsoli można zobaczyć na samym początku interesującą nas informacje:

```
Running Primordia
AGS: Adventure Game Studio v3.3 Interpreter
Copyright (c) 1999-2011 Chris Jones and 2011-20xx others
ACI version 3.3.0.1132
```

Wiemy już więc, iż gra działa na silniku Adventure Game Studio. Szybkie poszukiwania w Google pozwalają ustalić nam dodatkowe dwa fakty związane z tym silnikiem:

- Używa języka skryptowego, którego dokumentacja dostępna jest tutaj.
- Silnik jest open source i jego źródła dostępne są publicznie tutaj.

Należy pamiętać, iż to wszystko to jedynie elementy związane z silnikiem, a nie stricte sama grą. Kody źródłowe samej gry nie są publicznie dostępne. Być może da radę wydobyć skrypty z gotowej gry, ale w naszym przypadku niewiele to nam pomoże, bo chcemy jedynie

dodać do naszego ekwipunku jeden przedmiot, a nie modyfikować całą rozgrywkę. Przeglądając dokumentacje poszukajmy jakieś funkcji związanej z dodawaniem elementów do ekwipunku. Ja wybrałem `AddInventory`. Musimy teraz znaleźć jej implementacje w kodzie silnika. Możemy ręcznie przeszukiwać pliki, załadować całość do jakiegoś IDE i tak jej poszukać lub użyć (będąc w folderze ze źródłami z Gita) polecenia grep:

```
grep -nri addinventory *
```

To polecenie przeszuka wszystkie pliki tekstowe pod kątem ciągu znaków “addinventory” (ignorując przy tym ich wielkość). Wśród wyników wyszukiwania moją uwagę zwrócił wynik wyglądający następująco:

```
Engine/ac/character.cpp:119:void
Character_AddInventory(CharacterInfo   *chaa,  ScriptInvItem  *invi,
int addIndex) {
```

Analizując kod źródłowy wspomnianej funkcji dość łatwo można zauważyć, iż informacje o wyposażeniu danej postaci są przechowywane w typie `CharacterInfo`. Szybki powrót do konsoli i znowu skanujemy kod poleceniem `grep`, ale tym razem poszukujemy informacji o owym typie. Z wyników wyszukiwania można dostrzec:

```
Common/ac/characterinfo.h:52:struct CharacterInfo {
```

Otwieramy plik z definicją tej struktury i bingo. Mamy strukturę z tablicą typu short, w której trzymane są informacje o aktualnym wyposażeniu. Na podstawie analizy kodu można zauważyć, iż każdy przedmiot występujący w grze ma swoje przypisane pole w tej tablicy. Jeżeli postać posiada dany przedmiot to jest tam wpisane 1, a w przeciwnym wypadku 0.

Wiemy już gdzie trzymane są informacje o ekwipunku. Pora przejść do kolejnej fazy.

Planowanie ataku

Gra zostanie podłączona pod debugger GDB, w którym radośnie będziemy mogli jej grzebać w pamięci. Napotkamy jednak pewne problemy, jak np. położenie struktury w pamięci wirtualnej procesu. Po pierwsze trzeba więc wypracować metodę na pozyskiwanie tego adresu w pamięci. Drugim problemem będzie ustalenie gdzie jest początek tablicy w strukturze. Można bawić się z obliczaniem offsetu, ale pokażę bardziej eleganckie rozwiązanie.

Wróćmy do dokumentacji języka skryptowego silnika gry. Zwróćcie uwagę, iż każda implementacja funkcji z języka skryptowego zaimplementowana w C++ pobiera trochę więcej argumentów, niż wersja “wyjściowa” dla deweloperów gry. Takim argumentem jest np. zawsze na pierwszym miejscu wskaźnik na poszukiwaną przez nas strukturę. Wystarczy więc ustawić breakpoint w GDB w chwili gdy jedna z takich funkcji będzie wywoływana, odczytać wskaźnik i zacząć rozwiązywać drugi wspomniany wcześniej problem (położenie

tablicy). Zatrzymajmy się jednak tutaj na chwilę, bo rodzi się nam tutaj kilka podproblemów. Dla jednych rozwiązanie ich będzie "oczywistą oczywistością", a dla innych coś zupełnie nowego dlatego postaram się je w miarę dobrze opisać.

Przecież nie masz kodu źródłowego. Na czym chcesz zrobić breakpoint?

Jedno z kłamstewek głoszonych przez niektórych prowadzących zajęcia na uczelniach oraz części książek o programowaniu w C/C++ głosi, iż: *"Po skompilowaniu kod źródłowy zostaje zamieniony w formę niezrozumiałą dla człowieka, a gotowy plik wykonywalny nie zawiera kodu źródłowego"*. Owszem, gotowy plik niewiele przypomina kod C/C++ z edytora, ale owa tajemna "forma", do której zostaje przetłumaczony to zwykły język assembly w żaden sposób nie zabezpieczony przez ciekawymi oczami. Dowolną natywną aplikację możecie załadować do debuggera w stylu GDB i analizować jej działanie. Są też bardziej wyspecjalizowane w takiej analizie programy jak IDA (wariant #nabogato) lub darmowy radare2 (#cebulamotzno). Oczywiście są techniki utrudniania takiej analizy, ale to już materiał na całkiem osobny artykuł. Mając w każdym razie dostęp do kodu assembly w GDB ustawię breakpoint, który zatrzyma wykonywanie kodu w chwili wywołania wybranej przeze mnie funkcji.

Ale skoro masz jedynie goły kod assembly, to jak znajdziesz położenie danej funkcji?

Nawet jeżeli dana aplikacja jest kompilowana jako "Release" przy użyciu kompilatora GCC bez żadnych symboli debugowania, to w jej środku pozostają pewne "resztki" jak np. nazwy funkcji (ta reguła jest zależna od kompilatora i np. w Visual C++ już nie występuje). Do debugowania "na całego" to za mało, ale do ustalenia gdzie funkcja jest wywoływana w pamięci wystarcza. Jako pocieszenie dla zmartwionych tym faktem programistów dodam, że można i tych pozostałości się pozbyć. W Linuksie służy do tego polecenie `strip`. Takie czyszczenie jest używana przez demoscenowców robiących intra 4k, 8k, 64k itp, gdzie każdy kB jest na wagę złota. Małe sprzątanie pozwala zyskać trochę dodatkowego miejsca w skompilowanym intrze (oczywiście jest to jeden z etapów przygotowywania intra, ale to też materiał na osobny artykuł).

Tak więc zatrzymamy wykonywanie funkcji i ze stosu odczytamy kolejne argumenty funkcji?

Kolejne kłamstewko/archaizm na jaki można natrafić. Tak naprawdę, to gdzie są umieszczane argumenty danej funkcji jest trochę bardziej skomplikowane i opowieści z wykładów w stylu "wszystkie argumenty funkcji są odkładane na stos" możecie położyć na półce w domu gdzieś pomiędzy "Czerwonym Kapturkiem", a książkami historycznymi. Stos jako miejsce składowania argumentów miał większe znaczenie w 32-bitowych systemach, ale dzisiaj już większość ma 64-bitowe oprogramowanie. Kwestię przekazywania argumentów do funkcji opisuje ABI. W naszym wypadku nie musimy jednak pochłaniać całej dokumentacji od ABI, a jedynie zapoznać się z takim ciekawym artykułem na Wikipedii, gdzie interesujący nas fragment został streszczony (ponieważ kompilatory w ramach optymalizacji robią czasami dziwne rzeczy, to warto dla pewności zerknąć w kod assembly, ale to już kolejny osobny temat). Ponieważ działam na wersji Linuksowej, to interesuje mnie

jedynie System V AMD64 ABI, gdzie pierwszy argument funkcji leci do rejestru RDI, a nie na stos. Tak więc po zatrzymaniu działania gry przez breakpoint należy odczytać wartość tego rejestru.

Mamy adres struktury, ale co z jej odczytaniem?

Jak wspomniałem można bawić się z obliczaniem offsetu, żeby uzyskać adres tablicy z listą przedmiotów, ale możemy również jak cywilizowany człowiek odwołać się bezpośrednio do pola w strukturze. Nie mamy symboli debugowania, żeby to zrobić? Nic nie szkodzi. Sami je sobie wygenerujemy (͡° ͜ʖ ͡°)ﾉ⌐■-■ .

Zaczynamy atak

Po pierwsze zróbmy sobie kopię kodu assembly do wygodnego przeglądania w jakimś edytorze. Przechodzimy w konsoli do miejsca, gdzie znajduje się plik wykonywalny z grą i dajemy:

```
objdump -S Primordia.bin.x86_64 > dump.txt
```

To nam zrzuci kod assembly gry z pliku tekstowego. Mając ten zrzut wracamy tylko na chwilę do dokumentacji języka skryptowego silnika i wybieramy sobie jakąś funkcję, na którą zastawimy pułapke. Ja wybrałem `Walk`, a używając na kodzie silnika wyszukiwania w przedstawiony wcześniej sposób ustaliłem, iż ta funkcja w C++ jest zaimplementowana pod nazwą `Character_Walk`. Teraz otwieramy nasz zrzut z assembly w edytorze i wyszukujemy frazę `Character_Walk` i wyszukujemy gdzie jest wywoływana. W moim przypadku jej wywołanie występuje pod adresem 0x4c9d75 (ta skrajnie lewa kolumna w zrzucie oznacza adres w pamięci). Adres może ulec zmianie jeżeli posiadacie inną wersję gry niż moja. Cała linijka wygląda następująco:

```
4c9d75: e8 c6 ff ff ff callq 4c9d40 <_Z14Character_WalkP13CharacterInfoiiii>
```

Od lewej: wspomniany adres w pamięci, wygląd polecenia w “surowej” formie szesnastkowej, mnemonik polecenia wraz z argumentem (tutaj adres pod który znajduje się wywoływana funkcja przez instrukcję `callq`, “śmieciowa” nazwa, która pozostała w skompilowanym pliku. Jak widać została ona “nieco” zniekształcona na potrzeby linkera, który na ich podstawie określa np jakiego typu argumenty przyjmuje funkcja i szuka z czym połączyć dane odwołanie w procesie linkowania aplikacji (obiecuje sobie, że kiedyś zagłębię się jak wygląda dokładnie cały proces budowania aplikacji, ale chwilowo mam inne pilniejsze sprawy).

Pora załatwić jeszcze sprawę symboli debugowania dla struktury z informacjami o graczu. Otwórzmy ponownie plik Common/ac/characterinfo.h i skopiujmy do osobnego pliku definicję struktury `CharacterInfo`. Następnie usuńmy końcowe metody, gdyż nie będą nam potrzebne, a wprowadzą dodatkowy zamęt związany z zależnościami. Na koniec do nowego

pliku skopiujmy dwie stałe użyte w strukturze, zmieńmy odrobinę nazwę struktury i voila. Mamy gotowy kod struktury:

```
#define MAX_INV             301
#define MAX_SCRIPT_NAME_LEN 20

struct MyCharacterInfo {
    int   defview;
    int   talkview;
    int   view;
    int   room, prevroom;
    int   x, y, wait;
    int   flags;
    short following;
    short followinfo;
    int   idleview;
    short idletime, idleleft;
    short baseline;
    int   activeinv;
    int   talkcolor;
    int   thinkview;
    short blinkview, blinkinterval;
    short blinktimer, blinkframe;
    short walkspeed_y, pic_yoffs;
    int   z;
    int   walkwait;
    short speech_anim_speed, reserved1;
    short blocking_width, blocking_height;
    int   index_id;
    short pic_xoffs, walkwaitcounter;
    short loop, frame;
    short walking, animating;
    short walkspeed, animspeed;
    short inv[MAX_INV];
    short actx, acty;
    char  name[40];
    char  scrname[MAX_SCRIPT_NAME_LEN];
    char  on;
};
```

Zapiszmy ją przykładowo w pliku hack_struct.h i utwórzmy obok prosty plik hack_struct.c:

```
#include "hack_struct.h"

struct MyCharacterInfo test;
```

Teraz pora na małą kompilacje:

```
gcc -g -c hack_struct.c
```

Pierwszy argument sprawi, iż zostaną dołączone do gotowego pliku symbole debugowania, a drugi sprawi, iż cykl budowania naszego kodu zostanie przerwany tuż przed procesem linkowania. Jeżeli wszystko przebiegło poprawnie, to na koniec otrzymujemy plik `hack_struct.o`.

Mając już wszystko przygotowane odpalamy GDB poleceniem:

```
gdb Primordia.bin.x86_64
```

Następnie musimy dodać nasz symbol debugowania. Zakładając, iż plik *.o znajduje się w tym samym miejscu, co plik wykonywalny gry całość dodawania symbolu wygląda następująco:

```
(gdb) add-symbol-file hack_struct.o 0
add symbol table from file "hack_struct.o" at
        .text_addr = 0x0
(y or n) y
Reading symbols from hack_struct.o...done.
```

Nasz symbol został załadowany na początku wspomnianego segmentu. Nie musimy się przejmować zbytnio pytaniem GDB czy na pewno chcemy załadować symbole w to miejsce, bo i tak nie mamy symboli debugowania od twórców.

Pora na breakpoint:

```
(gdb) break *0x4c9d75
Breakpoint 1 at 0x4c9d75
```

Czas zapiąć pasy i odpalamy ten cały grajdołek:

```
(gdb) run
```

Załadujmy dowolny stan gry i pokręćmy się trochę główną postacią (np. do jakieś lokacji obok), aż breakpoint zaskoczy (przy okazji polecam całość robić na grze odpalonej w trybie okna). Gdy to się stanie gra zatrzyma się, a debugger znowu przełączy się w tryb wydawania poleceń. Pora odczytać wartości rejestrów:

```
(gdb) info registers
rax            0x93     147
rbx            0x7fffffffcab0   140737488341680
rcx            0x397    919
```

```
rdx            0x93     147
rsi            0x1b     27
rdi            0x15554e0        22369504
rbp            0x3d94230        0x3d94230
rsp            0x7fffffffc520   0x7fffffffc520
r8             0x131    305
r9             0x536aff4        87470068
r10            0x0      0
r11            0x0      0
r12            0x7fffffffcc50   140737488342096
r13            0x7fffffffca00   140737488341504
r14            0xe0     224
r15            0x7fffffffc790   140737488340880
rip            0x4c9d75 0x4c9d75 <Sc_Character_Walk(void*,
RuntimeScriptValue const*, int)+37>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

Najbardziej interesuje nas RDI, który (jeżeli niczego nie pokręciliśmy po drodze) aktualnie wskazuje na położenie poszukiwanej przez nas struktury. Ponieważ nasze symbole debugowania zostały jakby wepchnięte na siłę, trzeba teraz w GDB zrzutować ten fragment pamięci na naszą strukturę i wyświetlić przykładowe pole:

```
(gdb) print ((struct MyCharacterInfo*)0x15554e0).scrname
$3 = "cEgo", '\000' <repeats 15 times>
```

Z tego co poszukałem na forach silnika AGS, mianem “cEgo” określa się tam postać gracza więc można powiedzieć, iż trafiliśmy w dobre miejsce (͡° ل͜ ͡°). Teraz zobaczmy te naszą tablicę zawierającą ekwipunek postaci.

```
(gdb) print ((struct MyCharacterInfo*)0x15554e0).inv
$5 = {0, 0, 0, 0, 1, 1, 1, 1, 0 <repeats 35 times>, 1, 0, 0, 0, 1,
1, 0 <repeats 252 times>}
```

Teraz wystarczy dodać decydujący cios w postaci modyfikacji wybranego pola w tablicy zyskując w ten sposób brakujący nam przedmiot. Na której znajduje się pozycji? Gdybym powiedział podchodziło by to pod spojler więc podpowiem jak to ustalić: załadujcie zapis gry z momentu, gdy posiadacie dany przedmiot, skopiujcie sobie gdzieś na boku zawartość powyższej tablicy. Potem załadujcie możliwie blisko moment po utracie tego przedmiotu i porównajcie, których pól wam brakuje i ustalcie w ten sposób metodą prób i błędów jego ID. Tutaj dodajmy sobie jakikolwiek losowy przedmiot i puśćmy przebieg gry dalej:

```
(gdb) set ((struct MyCharacterInfo*)0x15554e0).inv[8] = 1
(gdb) c
```

Pomimo naszej modyfikacji jednak nie widać nowego przedmiotu w ekwipunku. Co zrobić? Jak żyć? Gdzie z matką popełniliśmy błąd? Gdy trafiłem na taką barierę zacząłem przepatrywać w zrzucie assembly wszystkie funkcje związane z ekwipunkiem, aż moim oczom rzuciła się `_Z15update_invorderv()`. Już jej nazwa sugerował, że odświeża stan wyposażenia postaci. Miejsca, w których była wywoływana jeszcze bardziej potwierdzały tą teorię. Teoretycznie więc wystarczy teraz wymusić na silniku gry uruchomienie jej. Możemy to zrobić przez kombinowanie z akcjami w grze lub przez GDB. Przejdźmy do okna konsoli z odpalonym GDB i przy pomocy skrótu CTRL+C zatrzymajmy jeszcze raz wykonywanie się gry. Teraz wykonajmy ostateczne polecenie wymuszające wywołanie funkcji:

```
(gdb) call _Z15update_invorderv()
```

Odblokujmy dalsze działanie gry i spójrzmy jeszcze raz w ekwipunek. Pojawił się tam w magiczny sposób *REPAIR MANUAL*, którego wcześniej nie było. Kiedy już dodamy brakujący nam przedmiot można zapisać stan gry, wyłączyć ją, uruchomić już normalnie bez debuggera i cieszyć się takim zakończeniem historii jakie zechcecie.

Kontakt z autorem i licencja

Autorem powyższego tekstu jest Artur “Lacky” Łącki. W razie pytań można śmiało pisać na maila alacki93@gmail.com.

Przedstawiony powyżej tok postępowania teoretycznie można zastosować do innych gier, ale pamiętajcie o dwóch podstawowych faktach:

- Primordia jest grą skupioną na trybie jednego gracza. Modyfikując jej pamięć nie psuje innym zabawy. Pamiętajcie o tym zanim przyjdzie wam do głowy podbić wartość jakieś zmiennej, żeby osiągnąc lepszy wynik w grze.
- Większość współczesnych gier z trybem zabawy wieloosobowej ma zabezpieczenia wykrywające takie proste podmiany zmiennych. Próba zastosowania powyższej metody w grze typu CS:GO może dla was zakończyć się banem.