Jak napisać własny slave? - część 1

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

Na pewno znakomita większość, jeżeli nie wszyscy użytkownicy, przynajmniej słyszeli o WHDLoad, dzięki któremu można uruchamiać gry z dysku twardego. Mało tego, możemy też z gry wyjść z powrotem do systemu. Wielu przeszła też myśl nie tylko o graniu za pomocą wyżej wspomnianego pakietu, a o możliwości stworzenia własnego patcha do gry. Mimo, że WHDLoad wspiera coraz więcej gier i być może kiedyś ziści się marzenie autora, że w końcu nie będzie gry uruchamianej z dyskietki, to wiele gier jeszcze jest do spatchowania. Zamiennie będę używał słowa naprawianie bądź ulepszanie, gdyż ja WHDLoad pojmuje nie w kategorii programu - degradera - umożliwiającego granie z dysku twardego, ale jako potężne narzędzie zarówno do ulepszania jak i środowiska do programowania bądź portowania gry. Czymże jest wspomniane patchowanie gry? Jest to nakładanie na oryginalną zawartość swoich danych. Podmieniać możemy kod, jak i dane - wszystko zależy od tego, co chcemy osiągnąć. Jeśli tylko chcemy zmienić grafikę na własną, to musimy sobie zdawać sprawę, że nie możemy robić tego w dowolny sposób. Na przykład, jeśli w grze jest obrazek tytułowy o wymiarach 128x100x4, to nie możemy w prosty sposób podmienić na grafikę w 256x200x8. Jasne jest, że taki patch jest możliwy, ale wymaga o wiele więcej pracy. Z kodem jest inaczej, o czym się wkrótce przekonamy. Podstawowym wymaganiem jest znajomość asemblera 68000. Po cichu zakładam, że czytelnik jest obeznany z Amigą i nie straszne mu rejestry hardware. Przyda się także wiedza z zakresu programowania pod systemem.

W wybranym przez nas miejscu instalujemy pakiet deweloperski dla WHDLoad. Najważniejszy dla nas będzie katalog autodoc, gdzie opisane są funkcje dostępne dla programisty i pliki z katalogu include, czyli WHDLoad.i. W nim to zdefiniowane mamy potrzebne stałe, tagi, struktury i makra. Dodatkowo w whdmacros.i można znaleźć także przydatne makra.

Podczas pracy nad slave, potrzebne będą również inne narzędzia. Do inżynierii odwrotnej, czyli deasemblacji pliku wynikowego do źródła w asemblerze, przyda się Resource 6.06 bądź darmowa IRA. Dla mnie osobiście bezkonkurencyjny jest pierwszy z nich. IRA działa z poziomu Shella, co może odstraszać początkujących, ale myślę, że warto się z nim zapoznać. W naszym kursie skupimy się tylko na Resource. Aby bezproblemowo rozpakowywać pliki, warto zainstalować pakiet xfd wraz z xfdDecrunch - znakomicie ułatwi nam to pracę. Oczywiście zdarzyć się może, że xfdDecrunch nie rozpakuje pliku mimo, że faktycznie został on spakowany zmodyfikowanym narzędziem. Wtedy mamy dwa wyjścia. Po pierwsze, spróbować naprawić spakowany plik, co wymaga dużo większej wiedzy, ale czasem przy odrobinie szczęścia wystarczy zmienić ciąg ASCII i ponownie użyć xfdDecrunch. Drugim wyjściem jest rozpakowanie ręczne, czyli wyłuskanie procedury depakującej, zmiana jej i uruchomienie, a następnie przegranie już rozpakowanych danych. Przyda się też DropHunk (dostępny na Aminecie) czy też podobna pchełka, pokazująca informacje o pliku wykonywalnym. Daje to ogólny obraz co w danym pliku siedzi. Dodatkowo można zaopatrzyć się w rippera modułów muzycznych, tak by wiedzieć gdzie (w jakiej lokalizacji) znajduje się muzyka z gry, co przyda się przy deasemblacji. Na końcu warto wspomnieć, że przy tworzeniu slave będziemy się posiłkować asemblerem Barfly.

Kilka słów o samym procesie patchowania - nie ukrywam, że w niektórych przypadkach jest to bardzo żmudna i czasochłonna praca. Żmudna w tym sensie, że ilość czasu spędzonego na testowaniu patcha może spowodować, że na daną grę przez jakiś czas nie spojrzymy. Czasochłonna, bo aby sprawdzić czy patch w pełni działa, trzeba grę ukończyć, a gdy rozwiązanie nie działa, powoduje często zawieszenie komputera i tylko reset może pomóc. Z tego też między innymi powodu mój blat Workbencha jest minimalistyczny, aby system startował bardzo szybko.

Zabieramy się do działania

Najpierw wybierzemy grę. Na początek wystarczy produkcja mieszcząca się w jednym pliku, którą można bez problemu załadować z poziomu Shella. Wydawać by się mogło, że przecież wystarczy kierować się rozmiarem pliku wykonywalny i w ten sposób mamy łatwego patcha do zrobienia. Odpowiedź nie jest już taka oczywista, bo wszystko zależy od tego, co napotkamy przeglądając, czyli deasemblując owy plik. Dla naszych celów wybrałem AppleHunt - grę, którą bez

Jak napisać własny slave? - część 1

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

problemów możemy pobrać z Aminetu.

W wybranym przez nas katalogu tworzymy nowy o nazwie AppleHuntWHD i w nim będziemy pracować. Tutaj będzie docelowy plik z rozszerzeniem slave i ikona służąca do uruchamiania gry. Będąc w nim, tworzymy kolejne katalogi. Najważniejszy z nich to data - tu bedą przechowywane pliki z gry. Następnie icons, zawierający różne ikony - standardową, dla pakietu newicons itd. W katalogu sources znajdzie się kod źródłowy naszego slave'a, a podkatalog RCS służy do wersjonowania. Dalej katalog texts, gdzie zwykle trzymam pliki tekstowe z różnymi informacjami na temat zarówno gry, jak i techniczne detale, tak abym po przerwie, mógł w miarę szybko się zorientować, co zostało zrobione, gdzie utknąłem. Ostatni katalog to versions, zawierający podkatalogi crack i original. Tam przechowuję różne wersje gry - jak widać są tam też wersje scrackowane. Z tymi ostatnimi to różnie bywa, jeśli chodzi o patchowanie, bo zdarza się, że kod różni się od oryginalnej wersji i trzeba się napracować, aby dodać obsługę takiej wersji. Przy pracowaniu nad kolejnymi patchami takie tworzenie całej tej struktury katalogów jest nudne i męczące, dlatego ja posiłkuje się skryptem dosowym.

.KEY NAME,DIR .BRA { .KET } .DEF DIR="DH2:WHD" ;default directory .DEF NAME="Empty" ;default name ; ECHO "WHDLoad template" IF EXISTS {DIR} PCD {DIR} SET NAMEWHD{$$} {NAME}WHD IF NOT EXISTS $NAMEWHD{$$} MAKEDIR $NAMEWHD{$$} CD $NAMEWHD{$$} MAKEDIR data MAKEDIR icons MAKEDIR sources sources/RCS MAKEDIR texts MAKEDIR versions versions/crack versions/original ECHO "$NAMEWHD{$$} zrobione." ELSE ECHO "Nazwa *"$NAMEWHD{$$}*" istnieje." ENDIF PCD ELSE ECHO "Nie ma takiego katalogu *"{DIR}*" ." ENDIF

Pobieramy archiwum z grą z Aminetu i kopiujemy go do odpowiedniego katalogu, a z rozpakowanej wersji, kopiujemy tylko plik AppleHunt do katalogu data. Mimo, że nie zaczęliśmy jeszcze właściwej pracy, to już wiemy, że mamy do czynienia z plikiem wykonywalnym. Co to oznacza dla nas? To, że plik składa się z odpowiednich hunków i że w nim zawarte są zarówno dane, jak i kod. Warto w tym miejscu na szybko przejrzeć AppleHunt za pomocą FileMastera korzystając z opcji ShowHex czy też użyć innego narzędzia pozwalającego zobaczyć plik w hex i w ascii. Przeglądając plik zwracamy uwagę czy przypadkiem nie widać ciągów, które normalnie można przeczytać, po pewnym czasie wystarczy tylko szybkie spojrzenie, aby stwierdzić, że gra została spakowana. Ten szybki podgląd ma dać nam podstawę do użycia narzędzia rozpakowującego xfdDecrunch - nikt nie broni nam używać tego narzędzia zawsze, nawet gdy mamy do czynienia z rozpakowanym plikiem. W naszym przypadku AppleHunt został spakowany PowerPackerem, o czym informuje nas xfdDecrunch.

>c:xfdDecrunch AppleHunt xfdDecrunch 1.9 (09.09.1999) © 1994-99 Georg Hörmann, Dirk Stöcker Reading "AppleHunt" (116224 bytes)... Decrunching "PowerPacker 4.0" file... Writing "AppleHunt" (233352 bytes)...

W wyniku plik główny podwoił swój rozmiar i pozostał w dalszym ciągu plikiem wykonywalnym. A co byłoby, gdyby to był plik binarny? Skąd byśmy to wiedzieli? xfdDecrunch by nas poinformował, że po rozpakowaniu pliku następuje skok do określonej lokacji. Tylko po co rozpakowywać plik, skoro będzie zajmował więcej miejsca? Przecież wystarczy go odpowiednio spatchować. Tak, to prawda, że można w ten sposób podejść do sprawy. Ja robię to w ten sposób dlatego, aby było możliwe użycia packera wspieranego przez WHDLoad. Wówczas plik może być krótszy, bo nie ma procedury depakującej i sam proces rozpakowywania jest dla nas transparentny. Poza tym procedury rozpakowujące mogą powodować, że trzeba w najlepszym przypadku wyłączyć cache w procesorach 68060. Ja nie znam się dobrze na depakowaniu, a tym bardziej nie wiem co się może dziać na takim procesorze i wolę ten problem ominąć, zamiast ślęczeć nad kodem, który tak naprawdę ma mało wspólnego z grą. Czyli w dalszym ciągu będziemy pracować nad rozpakowaną wersją AppleHunt. Informacje od xfdDecrunch warto odnotować w katalogu texts, umieszczając tam stosowny plik. A jak to zrobić? Wystarczy przekierować wyjście na plik podczas ponownego uruchamiania narzędzia, robiąc to na spakowanej wersji pliku. Skoro mam plik wykonywalny, to przyjrzyjmy się bliżej, sprawdźmy jak jest z

Jak napisać własny slave? - część 1

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

hunkami. Robimy DropHunk AppleHunt.

Hunk layout of AmigaDOS load file "AppleHunt" (233352 bytes filesize) --- Starting with hunk #0 ---------- HUNK_HEADER Number of hunks: 1. First hunk to load #0, last hunk to load #0. -> Hunk #0 requires a storage of 232344 bytes. » Hunk will be forced to CHIP-memory. HUNK_CODE 232344 bytes HUNK_RELOC32 239 long relocation(s) for hunk #0 --- End of hunk #0 ---------- Done. DropHunk's rating: Less than or equal to 4 relocations per 1000 bytes. »» Yes, nothing against it; strong compiler used.

Przeanalizujmy wynik. Widać tu, że autor gry nie napracował się, aby umieścić osobno dane i kod. Plik wykonywalny zawiera także tabelę relokacji, czyli przesunięcia niezbędne w procesie relokacji programu do dowolnego adresu. Przeskanujmy jeszcze AppleHunt ripperem muzycznym. Odnajdziemy moduł muzyczny 'glenn aage beate' o długości 123536 bajtów a przesunięcie względem początku gry wynosi $19c00 = 105472, czyli po szybkiej kalkulacji moduł znajduje się prawie na końcu pliku.

Przystąpmy do pisania naszego slave'a. Plan jest bardzo prosty.

A. Załadować AppleHunt.
B. Spatchować.
C. Uruchomić.

Aby poprawnie załadować plik wykonywalny, przeważnie używamy do tego celu funkcji z biblioteki dos o nazwie LoadSeg. W rejestrze D1 podajemy wskaźnik na string zawierający nazwę pliku i jeśli operacja się powiodła, to otrzymamy wskaźnik BCPL do listy segmentów. Dalej to przetwarzamy BCPL na normalny wskaźnik i uruchamiamy kod. Ponieważ WHDLoad zatrzymuje system, więc nie mamy do niego dostępu i musimy sobie jakoś z tym poradzić. Mamy trzy możliwe rozwiązania załadowania pliku wykonywalnego.

1. skorzystać z OSEmu i funkcji LoadSeg z dos.library
2. skorzystać z KickEmu (Kickstart 1.3 bądź 3.1) i funkcji LoadSeg z dos.library
3. skorzystać z funkcji resload_LoadFile i resload_Relocate.

Dwa pierwsze rozwiązania bazują na emulacji systemu podczas pracy WHDLoad. Pierwszy z nich to najwcześniejsza próba poradzenia sobie z problemem patchowania gier, które w niewielkim stopniu wykorzystują OS, na przykład alokują pamięć, ładują pliki poprzez dos.library. Niewątpliwą zaletą jest mały rozmiar OSEmu, który to można dodatkowo odchudzić, lecz tu trzeba większej wiedzy. Minusem jest, że skrypt instalacyjny dorzuca dodatkowy plik do katalogu data. Największą bolączką OSEmu jest częściowa emulacja OS i co za tym idzie, że podczas testów po wielu godzinach patchowania, uruchomianiu i grania, dowiadujemy się, że gra korzysta z funkcji systemu, której nie ma w OSEmu bądź jest zaimplementowana szczątkowo. KickEmu pozbawiony jest ostatniej wady OSEmu, ale za to wymagania pamięciowe dla naszej gry rosną. My skupimy się na opcji trzeciej.

Stwórzmy w ulubionym edytorze plik AppleHuntWHD.s i umieśćmy tam początkowy blok komentarza, tak aby było jasne.

;---------- ; Author Asman ; Version 1.0 ; History 2007-03-10 ; Requires - ; Copyright Public Domain ; Language 68000 Assembler ; Translator Barfly V2.9 ; ; Game AppleHunt ; ; Id ;----------

Ponieważ ja używam do wersjonowania plików programu RCS, to dziwny ciąg Id będący identyfikatorem zostanie

Jak napisać własny slave? - część 1

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

zastąpiony odpowiednim podczas pracy z RCS. Oczywiście można używać innego systemu wersjonowania plików bądź też nie używać żadnego. Ja gorąco zachęcam, bo niejednokrotnie RCS ratował mi skórę przed pisaniem slave'a od nowa, gdy okazało się, że zmiany dokonane przeze mnie wieszają grę. Pod blokiem komentarzy dodamy includy, etykietę RELEASE, która to na ten moment będzie zakomentowana i dyrektywy dla Barfly (jeśli używasz innego, to nie dodawaj ich) dla WHDLoad

```
INCDIR Includes: INCLUDE WHDLoad.i INCLUDE whdmacros.i ;RELEASE IFD BARFLY OUTPUT "Ram:AppleHunt.slave" IFD RELEASE BOPT O+ ;enable optimizing BOPT OG+ ;enable optimizing ELSE BOPT O- ;disable optimizing BOPT OG- ;disable optimizing ENDC ;END IFD RELEASE BOPT ODd- ;disable mul optimizing BOPT ODe- ;disable mul optimizing BOPT w4- ;disable 64k warnings BOPT wo- ;disable optimize warnings SUPER ENDC ;END IFD BARFLY
```

Słowem wyjaśnienia dyrektywy te nakażą asemblerowi stworzenie pliku wynikowego AppleHunt.slave i umieszczenie go w RAM:. Dodatkowo, jeśli etykieta RELEASE zostanie zdefiniowana, to nastąpi optymalizacja kodu. Aby ułatwić sobie nieco pracę, zdefiniujemy macro, pomocne przy pisaniu wywołań procedur z WHDLoad, dodamy etykiety mówiące o zapotrzebowaniu naszego slave'a na pamięć. Na początek 0.5 MB Chip wystarczy, gdyby to się okazało za mało, to zwiększymy do 1 MB. I dalej mamy strukturę opisująca slave'a.

```
; ; \1 - nazwa procedury bez przedrostka resload_ ; callwhd MACRO move.l _resload(pc),a2
jsr resload_\1(a2) ENDM CHIPMEMSIZE = $80000 FASTMEMSIZE = 0 slv_Version = 15 slv_Flags = 0
slv_keyexit = $59 ;F10 IFNE slv_Version-15 FAIL wersje ponizej 15 nie sa wspierane ENDC ;END IFNE
slv_Version-15 slv_base SLAVE_HEADER ;ws_Security + ws_ID dc.w slv_Version ;ws_Version
dc.w slv_Flags ;ws_flags _basemem dc.l CHIPMEMSIZE ;ws_BaseMemSize dc.l 0 ;ws_ExecInstall
dc.w _start-slv_base ;ws_GameLoader dc.w slv_CurrentDir-slv_base ;ws_CurrentDir
dc.w 0 ;ws_DontCache _keydebug dc.b 0 ;ws_keydebug _keyexit dc.b slv_keyexit ;ws_keyexit
_expmem dc.l FASTMEMSIZE ;ws_ExpMem dc.w slv_name-slv_base ;ws_name
dc.w slv_copy-slv_base ;ws_copy dc.w slv_info-slv_base ;ws_info IFD BARFLY IFND .passchk
DOSCMD "WDate >T:date" .passchk ENDC ;END IFND .passchk ENDC ;END IFD BARFLY
slv_CurrentDir dc.b "data",0 slv_name dc.b "Apple Hunt",0 slv_copy dc.b "19xx xxx",0 slv_info dc.b "installed &
fixed by Asman",10 dc.b "Version 1.0 " IFD BARFLY INCBIN "T:date" ENDC ;END IFD BARFLY dc.b
0 EVEN
```

Przed samą strukturą dodałem warunek sprawdzający na wypadek przypadkowej zmiany wersji na wyższą. Sama struktura zaczyna się od makra, które to odnajdziemy w pliku WHDLoad.i. Tworzy ono zabezpieczenie przed uruchomieniem slave'a jako pliku wykonywalnego, dodatkowo dodaje napis "WHDLoadS" za tym kodem. Następne ważne pole to wersja struktury - my będziemy korzystać z wersji 15. Potem są pomocne flagi dla WHDLoad, jak na przykład flaga mówiąca o tym, żeby wyczyścić pamięć zerami przy starcie. ws_GameLoader to różnica między początkiem struktury, a kodem slave'a, czyli przesunięcie. Kolejne pole to ws_CurrentDir, które jest przesunięciem do podkatalogu, w którym znajdują się pliki gry. W tym przypadku będzie to katalog "data". Pola ws_keydebug i ws_keyexit zawierają kody raw klawiszy dla wyjścia z gry, przy czym pierwsze z nich powoduje dodatkowo zrzut pamięci, a miejsce zrzutu określa zmienna CoreDumpPath w pliku WHDLoad.prefs. Potem jest ws_ExpMem zawierające ilość pamięci dodatkowej - w zależności co mamy na pokładzie naszej maszyny, będzie to w pierwszej kolejności pamięć typu Fast a potem Chip. Dalej mamy pola ws_name, ws_copy i ws_info - są to przesunięcia do nazwy gry/dema, informacji o produkcji (czyli rok, autorzy/wydawca) i dodatkowych informacji. Zwyczajowo umieszcza się tu również podziękowania za oryginalne obrazy do gry i znajdują się tu też dane o autorze, wersji i dacie wypuszczenia slave'a.

W końcu nadszedł czas na napisanie kodu slave'a:

Jak napisać własny slave? - część 1

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

```
GAME_STARTADR = $10000    _start:    ;zachowanie bazy resload      lea   _resload(pc),a1      move.l   a0,(a1)
;zaladowanie i rozpakowanie pliku      lea   game_name(pc),a0      lea   GAME_STARTADR,a1
callwhd   LoadFileDecrunch      ;relokacja pliku      lea   GAME_STARTADR,a0      sub.l   a1,a1
callwhd   Relocate      ;patchowanie gry      lea   _patch_base(pc),a0      lea   GAME_STARTADR,a1
callwhd   Patch      ;start gry      jmp   GAME_STARTADR
;-------------------------------------------------------------------------------   _patch_base      PL_START      PL_END
;-------------------------------------------------------------------------------
;%%%
```

Jak napisać własny slave? - część 1

Asman

(c) Polski Portal Amigowy (www.ppa.pl)

instrukcja zajmuje, bo zawsze możemy to sprawdzić. Także w Asm-One łatwo to sprawdzić, korzystając z komendy D adres czy też używać sztuczki i odjąć jedną etykietę od drugiej. Na razie nie będziemy wnikali dlaczego i co fixujemy, chodzi mi tylko o pokazanie mechanizmu. W pierwszej części przykładu pokrywamy odpowiednią ilością NOP instrukcje odpowiedzialne za ustawienie blthog i pustą pętlę. W drugiej zaś blok instrukcji zamieniamy na jsr fixClrAudX i odpowiednią ilość NOP. Ktoś sprytny może zauważyć, że pewne instrukcję się powtarzają i można zapisać je tak.

```
lea GAME_STARTADR,a0 lea ($86,a0),a1 REPT 5 move.w #$4e71,(a1)+ ;NOP ENDR ;END REPT 5
lea $1034(a0),a1 move.w #$4eb9,(a1)+ ;JSR pea fixClrAudX,(pc) move.l (sp)+,(a1) REPT 9
move.w #$4e71,(a1)+ ENDR ;END REPT 9
```

Za pomocą makr z WHDLoad.i możemy ten mały przykład zapisać w następujący sposób:

```
_patch_base PL_START ; remove blthog and empty loop PL_NOP $86,$a ; fix clr.w aud
PL_PSS $1034,fixClrAudX,$12 PL_END
```

Wstęp mamy za sobą. Stworzyliśmy bardzo prostego slave'a, który poza tym, że wczytuje poprawnie grę, to nic innego nie robi. Pełny plik można pobrać stąd. Po skompilowaniu otrzymamy plik AppleHunt.slave umieszczony w RAM. Zapomniałem wspomnieć, że ja korzystam z małego programiku WDate, który generuje aktualną datę i jest on w katalogu WHDLoad/Src/programs. Proponuje go skopiować do katalogu C bądź zmienić ścieżkę w źródle.

Spróbujmy odpalić naszego slave'a. Można to uczynić korzystając z ikonki, którą trzeba sobie przygotować albo wpisujac z shella:

```
WHDLoad slave=Ram:AppleHunt.slave preload
```

Naszym oczom, o ile mamy "odhaczony" w ustawieniach WHDLoad tryb expert, ukaże się komunikat o błędzie. Klikając w przycisk ShowReg zobaczymy dokładniejsze dane. Powodem błędu jest próba otwarcia biblioteki graphics.library. Bez EmuKicka nie będzie to możliwe, gdyż jak wiemy WHDLoad zatrzymuje system. Musimy więc zacząć łatać grę, ale pierwszy krok został już zrobiony. Próbując uruchomić pierwszy slave, przeprowadziliśmy w ten sposób pierwszy test.

www.ppa.pl