Aplikacja EFI – Snake

Kacper Grzelakowski

1. Wprowadzenie

Celem projektu była implementacja klasycznej gry Snake jako aplikacji działającej w środowisku UEFI. Realizacja zadania wymagała odejścia od standardowych bibliotek systemowych na rzecz bezpośredniej komunikacji z firmwarem poprzez protokoły UEFI.

W opracowanej implementacji użytkownik korzysta z menu sterowanego strzałkami, które posiada opcje **PLAY**, **HALL OF FAME** i **QUIT**. Pierwsza z nich umożliwia klasyczną rozgrywkę w Snake, gdzie sterując klawiszami **WASD,** ustalamy kierunek poruszania się węża. Z każdym zjedzonym pokarmem pojawia się nowy pokarm w losowym miejscu, a wąż nie tylko się wydłuża, ale również przyspiesza. Po kolizji (z sobą samym lub z granicami planszy) otrzymujemy komunikat o tym, jaki wynik uzyskaliśmy i możliwość podania 3 liter jako identyfikatora do rankingu wyników dostępnym w sekcji Hall of Fame.

Głównym wyzwaniem tego projektu była konfiguracja narzędzi, które umożliwiają symulację środowiska pracy firmware'u. Zastosowanie odizolowanej emulacji nie tylko przyspieszyło proces wytwarzania aplikacji, ale również zapewniło bezpieczeństwo fizycznego sprzętu.

Standard UEFI narzuca szereg wymagań, które musiały zostać uwzględnione w tym projekcie, m.in.:

- Obsługa partycji wyłącznie w systemie plików FAT.
- Wymóg stosowania nowoczesnego standardu GPT.
- Aby aplikacja została automatycznie zlokalizowana i uruchomiona przez firmware, musi znajdować się w lokalizacji **/EFI/BOOT/** pod nazwą **BOOTX64.EFI.**

2. Konfiguracja środowiska

Jako środowisko programistyczne wybrano system operacyjny Linux (dystrybucja Fedora). Wybór ten wynika ze względu na wsparcie dla niskopoziomowych narzędzi do manipulacji systemami plików oraz łatwą dostępność zestawu narzędzi potrzebnych do tego projektu w porównaniu do systemu operacyjnego Windows.

2.1. Narzędzia do emulacji

Do wykonania tego projektu kluczowe było wykorzystanie narzędzi, które potrafią symulować środowisko, w którym można taką aplikację tworzyć:

- Emulator systemu (QEMU): Odpowiada on za symulacje warstwy sprzętowej (hardware). Zamiast operować na fizycznych podzespołach, emulator tworzy wirtualny procesor, pamięć RAM itp. Wykorzystanie QEMU zapewnia izolację środowiska, co eliminuje ryzyko uszkodzenia własnego komputera. Dodatkowo sprawia to, że przy każdej zmianie nie musimy fizycznie restartować maszyny, tylko odpalić ponownie emulację. Kluczową zaletą QEMU jest wsparcie dla debugera GDB, co pozwala na łatwą inspekcję pracy programu.

- Emulator firmware'u UEFI (OVMF): Sam emulator sprzętu po uruchomieniu nie wie jak „rozmawiać" z partycjami dysku ani jak wyświetlać grafikę. OVMF (Open Virtual Machine Firmware) dostarcza niezbędną implementację standardu UEFI dla maszyn wirtualnych. To on udostępnia aplikacji protokoły takie jak GOP (Graphics Output Protocol) oraz Simple File System Protocol, umożliwiając odnalezienie i uruchomienie pliku gry na partycji ESP.

2.2. GNU-EFI

GNU-EFI to zestaw bibliotek i nagłówków służących do kompilowania aplikacji UEFI przy użyciu standardowego kompilatora GCC. Narzędzie to pozwala na przekonwertowanie pliku w formacie ELF (powstałego po kompilacji GCC) do formatu PE zgodnego z UEFI. To, w jaki sposób wykorzystamy to narzędzie, będzie widoczne dokładnie w kolejnym rozdziale dotyczącym kompilacji i budowania aplikacji. Komendy służące do zainstalowania tego narzędzia wyglądają następująco:

```
1  git clone https://github.com/ncroxon/gnu-efi.git
2  cd gnu-efi
3  make
```

2.3. Automatyzacja procesu kompilacji i budowania obrazu

Proces budowania projektu został w pełni zautomatyzowany za pomocą skryptu **compile.sh**, co przyspiesza proces testowania aplikacji oraz eliminuje ryzyko błędów manualnych. Skrypt ten działa zarówno, kiedy wprowadziliśmy zmiany do istniejącej już aplikacji, jak i wtedy, kiedy po raz pierwszy chcemy ją odpalić. Proces ten składa się z kilku etapów: kompilacja, linkowanie, konwersja do formatu .efi oraz modyfikacja obrazu dysku. Używana w poniższych komendach zmienna EFI_DIR oznacza ścieżkę do folderu gnu-efi.

2.3.1. Kompilacja, linkowanie i konwersja formatu

Kompilacja na podstawie pliku **main.c** tworzy nam plik obiektowy **main.o** za pomocą komendy:

```
1  gcc -I"$EFI_DIR/inc" -fpic -ffreestanding -fno-stack-protector -fno-stack-check \
2  -fshort-wchar -mno-red-zone -maccumulate-outgoing-args -ggdb \
3  -c main.c -o main.o
```

Kluczowe parametry kompilacji:

- **`-fpic`** - pliki wykonywalne UEFI PE muszą być relokowalne, ten parametr generuje kod niezależny od adresu
- **`-ffreestanding`** - wymusza kompilację w trybie bez udziału standardowych bibliotek systemu operacyjnego
- **`-fshort-wchar`** – zmienia rozmiar szerokich znaków na 16 bitów, co jest bardzo ważne, ponieważ zapewnia kompatybilność z protokołami tekstowymi UEFI
- **`-ggdb`** – już na etapie kompilacji generujemy pełne informacje debugowania w formacie GDB

Kolejnym etapem po otrzymaniu naszego pliku **main.o** jest **linkowanie**, które łączy pliki obiektowe w jeden plik wykonywalny **main.so**:

```
ld -shared -Bsymbolic -L"$EFI_DIR/x86_64/lib" -L"$EFI_DIR/x86_64/gnuefi" \
-T"$EFI_DIR/gnuefi/elf_x86_64_efi.lds" "$EFI_DIR/x86_64/gnuefi/crt0-efi-x86_64.o" \
main.o -o main.so -lgnuefi -lefi
```

Kluczowe parametry linkowania:

- **`-shared`** i **`-Bsymbolic`** – informują linker, aby utworzył plik **.so**.
- **`-L`** i **`-T`** – wskazują ścieżki do statycznych bibliotek GNU-EFI oraz do skryptu linkera (**.lds**). Skrypt ten definiuje układ sekcji w pliku binarnym, wymagany przez specyfikację UEFI.
- **`-lgnuefi`** i **`-lefi`** – linkowanie z biblioteką **gnuefi** jest obowiązkowe, ponieważ zawiera ona kod odpowiedzialny za relokację programu w pamięci. Biblioteka **efi** dostarcza pomocnicze definicje i funkcje ułatwiające korzystanie z usług systemowych UEFI.

Następnie możemy wykonać jedną funkcję, która się przyda później przy debugowaniu jaką jest: `objcopy --only-keep-debug main.so main.efi.debug`, co wyodrębnia symbole debugowania do pliku **main.efi.debug**.

Teraz następuje kluczowy etap konwersji. Plik .**so** jest w formacie ELF, którego UEFI nie potrafi uruchomić. Musimy go przekonwertować na format **PE** za pomocą narzędzia **objcopy**:

```
objcopy -j .text -j .sdata -j .data -j .rodata -j .dynamic -j .dynsym \
-j .rel -j .rela -j .rel.* -j .rela.* -j .reloc \
--target efi-app-x86_64 --subsystem=10 main.so main.efi
```

Kluczowe parametry:

- **`-j`** – wskazuje konkretne sekcje, które mają zostać przeniesione do finalnego pliku.
- **`--target efi-app-x86_64`** – określa docelowy format pliku jako aplikacje UEFI dla architektury 64-bitowej.
- **`--subsystem=10`** – ustawia typ pliku na plik wykonywalny UEFI w nagłówku PE.

2.3.2. Obraz dysku

Jak już mamy plik **main.efi**, który OVMF rozumie i potrafi zinterpretować, to jeszcze musimy stworzyć obraz dysku, z którego nasz emulator odpalimy.

```
if [ ! -f efi.img ]; then
    dd if=/dev/zero of=efi.img bs=512 count=93750
    parted efi.img -s -a minimal mklabel gpt
    parted efi.img -s -a minimal mkpart EFI FAT16 2048s 93716s
    parted efi.img -s -a minimal toggle 1 boot

    dd if=/dev/zero of=/tmp/part.img bs=512 count=91669
    mformat -i /tmp/part.img -h 32 -t 32 -n 64 -c 1

    dd if=/tmp/part.img of=efi.img bs=512 count=91669 seek=2048 conv=notrunc
fi
```

Proces tworzenia obrazu **efi.img** został zaprojektowany tak, aby spełniał wymogi specyfikacji UEFI:

- Początkowo został stworzony pusty obraz dysku o wielkości ok. 46MB ($512\ B * 93750 \approx 46\ MB$).

- Wykorzystano tablicę partycji GPT. Jest ona wymagana przez UEFI i jest przechowywana w pierwszych 33 sektorach dysku (nie licząc Protective MBR) oraz w ostatnich 33 sektorach jako kopia.
- Stworzono pustą partycję FAT16 o nazwie EFI zaczynającą się od sektora 2048 (wyrównanie do 1 MB) i kończącą się na sektorze 93716 (sektory 93717-93749 zajęte są przez kopię nagłówka GPT), to jest miejsce, w którym umieścimy nasz plik **main.efi**.

Kiedy posiadamy już plik **efi.img,** można przejść do etapu umieszczania w tym obrazie naszego programu:

```
1  dd if=efi.img of=/tmp/part.img bs=512 skip=2048 count=91669 status=none
2  mkdir -p EFI/BOOT && cp main.efi EFI/BOOT/BOOTX64.EFI
3  mcopy -o -i /tmp/part.img -s EFI ::
4  dd if=/tmp/part.img of=efi.img bs=512 count=91669 seek=2048 conv=notrunc status=none
```

Najpierw wyodrębniamy z pliku **efi.img** partycję FAT16 do tymczasowego pliku **part.img**. Plik **main.**efi trafia do folderu **/EFI/BOOT/** pod nazwą **BOOTX64.EFI,** jest to ścieżka domyślna, której firmware UEFI poszukuje w celu automatycznego rozruchu. Kopiujemy folder EFI do pliku tymczasowego i nadpisujemy już wcześniej istniejący tam folder (jeżeli istniał) i integrujemy z powrotem partycję z **efi.img**.

Po tym wszystkim posiadamy już obraz dysku z naszym kodem, który jest gotowy do odpalenia.

2.4. Uruchomienie emulatora

Do startu programu również stworzyłem skrypt (**run.sh**), który umożliwia odpalenie kodu zarówno w zwykłym trybie, jak i w trybie debugowania:

```
1   if [ "$1" == "debug" ]; then
2     qemu-system-x86_64 \
3     -cpu qemu64,rdrand=on \
4     -drive if=pflash,format=raw,unit=0,file=$OVMF_PATH,readonly=on \
5     -drive format=raw,file=efi.img \
6     -net none \
7     -serial mon:stdio \
8     -s -S
9   else
10    qemu-system-x86_64 \
11    -cpu qemu64,rdrand=on \
12    -drive if=pflash,format=raw,unit=0,file=$OVMF_PATH,readonly=on \
13    -drive format=raw,file=efi.img \
14    -net none
15  fi
```

Zmienna OVMF_PATH oznacza tutaj ścieżkę do pliku OVMF_CODE.fd, który zawiera obraz firmware'u UEFI. Bez tego pliku QEMU startuje w trybie Legacy BIOS.

Kluczowe parametry skryptu:

- **`-cpu qemu64,rdrand=on`** – definiuje model emulowanego procesora. Flaga rdrand=on jest kluczowa dla logiki gry, ponieważ udostępnia mechanizm generujący liczby losowe.
- **`-drive if=pflash,format=raw,unit=0,file=$OVMF_PATH,readonly=on`** – wczytuje firmware UEFI do wirtualnej pamięci flash procesora.
- **`-drive format=raw,file=efi.img`** – podłącza wytworzony wcześniej obraz dysku jako główny napęd. QEMU automatycznie wykrywa w nim i uruchamia plik **BOOTX64.EFI**.

- **-s** i **-S** – są wykorzystywane w trybie **debug** skryptu i oznaczają kolejno, że umożliwiona jest opcja podłączenia zewnętrznego debugera GDB za pomocą portu 1234, oraz że wirtualny procesor zatrzyma się na starcie i będzie czekał na komunikat **continue** od debugera.

Pomyślne wykonanie skryptu skutkuje otwarciem nowego okna emulatora QEMU. Firmware lokalizuje partycję ESP na obrazie efi.img i przekazuje sterowanie do aplikacji, co pozwala na rozpoczęcie rozgrywki.

3. Implementacja

3.1. Wykorzystane protokoły

W architekturze UEFI komunikacja ze sprzętem i usługami systemowymi odbywa się poprzez protokoły. Są to zestawy funkcji zdefiniowane przez unikalne identyfikatory (GUID), które aplikacja lokalizuje w pamięci za pomocą tablicy **BootServices**. Oto kluczowe protokoły, które wykorzystano w projekcie:

- **EFI_GRAPHICS_OUTPUT_PROTOCOL (GOP)** – jest to najważniejszy protokół z punktu widzenia warstwy wizualnej, ponieważ daje nam bezpośredni dostęp do Frame Buffer, który kontroluje kolory pikseli na ekranie. Wykorzystałem ten protokół m.in. w funkcji **putPixel**, która kolorowała dany piksel na podany kolor. Funkcję tę wykorzystywałem potem np. do **drawRect**, która od razu rysowała całe prostokąty o zadanych wymiarach, co się przydawało do rysowania zarówno węża, jak i planszy.
- E**FI_RNG_PROTOCOL (Random Number Generator)** – protokół ten zapewnia dostęp do generatora liczb losowych. Wykorzystywany jest do losowania pozycji „pokarmu” dla węża.
- **EFI_SIMPLE_FILE_SYSTEM_PROTOCOL / EFI_FILE_PROTOCOL** – oba te protokoły zostały wykorzystane do stworzenia Hall of Fame, którego zawartość jest zapisywana w pliku, co sprawia, że przy ponownym odpaleniu kodu nie stracimy zapisanego wyniku. Pierwszy z nich pełni rolę punktu wejścia do partycji systemowej, a drugi służy do bezpośredniej manipulacji obiektami na dysku.
- **EFI_SIMPLE_TEXT_INPUT_PROTOCOL / EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL** – podstawowe protokoły do obsługi wczytywania, jak i wypisywania tekstu. Wykorzystano je do stworzenia menu (wyświetlanie tekstu i sterowanie menu klawiszami), jak i sterowania wężem.
- **EFI_LOADED_IMAGE_PROTOCOL** – protokół ten dostarcza informacji o samej uruchomionej aplikacji. Wykorzystywany jedynie w mechanizmie debugowania, aby poznać adres bazowy, pod którym aplikacja została załadowana do pamięci.

3.2. Pętla gry i obsługa zdarzeń

Bardzo ważnym aspektem tej gry jest fakt, że występuje tutaj cykliczne ruszanie wężem na ekranie oraz reakcja na przyciski klawiszami przez użytkownika. Te mechanizmy są możliwe dzięki funkcjom:

- **CreateEvent oraz SetTimer** – funkcje te służą do zainicjowania cyklicznego zdarzenia czasowego, które wyznacza rytm ruchu węża. Funkcja **SetTimer** została również wykorzystana do zwiększania prędkości poruszania węża z każdym zjedzonym pokarmem.

- **WaitForEvent** – stanowi centralny punkt pętli głównej gry. Funkcja ta przyjmuje tablicę zdarzeń (w tym przypadku zdarzenie timera oraz zdarzenie wejścia klawiatury **WaitForKey**) i przechodzi w stan oczekiwania, blokując wykonywanie dalszych instrukcji do momentu wystąpienia jednego z nich. W momencie, gdy jedno z tych wydarzeń się wykona, to funkcja ta zwróci indeks tego wydarzenia w tablicy zdarzeń. Jeżeli użytkownik nacisnął klawisz, to program obsłuży ten klawisz, jeżeli nastąpiło „tyknięcie" zegara, to program przesunie na ekranie węża. Funkcja ta jest również wykorzystywana w **getKey**, która jest używana przy obsłudze menu, gdzie jedynym eventem, jaki obsługujemy, są przyciski klawiszy.

3.3. Niskopoziomowe zarządzanie pamięcią i zasobami

W tradycyjnym programowaniu w C korzysta się z funkcji takich jak **malloc** czy **free** do zaimplementowania takiej struktury. W środowisku UEFI te funkcje nie są dostępne, ale na szczęście istnieją ich zamienniki: **AllocatePool** oraz **FreePool**.

Aby umożliwić dynamiczne rośnięcie węża bez konieczności definiowana sztywnego limitu jego długości, zaimplementowano strukturę **Vector**:

```
1   struct Vector{
2       struct Pair* data;
3       int capacity;
4       int size;
5   };
6
7   struct Snake{
8       struct Vector segments;
9       struct Pair direction;
10      struct Pair previousDirection;
11      UINT32 color;
12  };
```

Struktura **Vector** posiada takie pola jak:

- **data** – wskaźnik do tablicy struktur **Pair** (współrzędne segmentów).
- **size** – aktualna liczba segmentów węża.
- **capacity** – aktualnie zarezerwowany pamięci.

Dodawanie nowego segmentu odbywa się za pomocą funkcji **push_back**, która jest napisana według algorytmu:

1. Jeśli $\mathbf{size} < \mathbf{capacity}$, nowy segment jest dopisywany do istniejącej tablicy.
2. Jeżeli zabrałko miejsca, następuje wywołanie **AllocatePool** dla nowego obszaru o dwukrotnie większym rozmiarze $(capacity * 2)$.
3. Zawartość starej tablicy jest kopiowana do nowej lokalizacji.
4. Oryginalny, niewystarczający już obszar pamięci jest zwracany do systemu za pomocą **FreePool**.
5. Wskaźnik data zostaje ustawiony na nowo przydzielony adres.

4. Debugowanie

Debugowanie nie jest łatwym zadaniem w środowisku bez systemu operacyjnego i z działającym ASLR, który sprawia, że nasz program jest zawsze ładowany w inne miejsce w pamięci. Trzeba zastosować jakiś mechanizm, który pozwoli znaleźć adres bazowy naszego programu i przekazać debugerowi. Można do tego wykorzystać tzw. **magic marker**.

Na samym początku programu znajdują się takie linijki kodu:

```
1  EFI_LOADED_IMAGE_PROTOCOL *loaded_image;
2  EFI_GUID LoadedImageProtocolGUID = EFI_LOADED_IMAGE_PROTOCOL_GUID;
3  uefi_call_wrapper(SystemTable->BootServices->HandleProtocol, 3,
4                    ImageHandle, &LoadedImageProtocolGUID, (void **)&loaded_image);
5  volatile uint64_t *marker_ptr = (uint64_t *)0x10000;
6  volatile uint64_t *image_base_ptr = (uint64_t *)0x10008;
7  *image_base_ptr = (uint64_t)loaded_image->ImageBase;
8  *marker_ptr = 0xDEADBEEF;
```

UEFI używa innej konwencji wywołań funkcji niż kompilator GCC, sprawia to, że potrzebujemy **uefi_call_wrapper**, który odpowiada za poprawną translację argumentów między nimi.

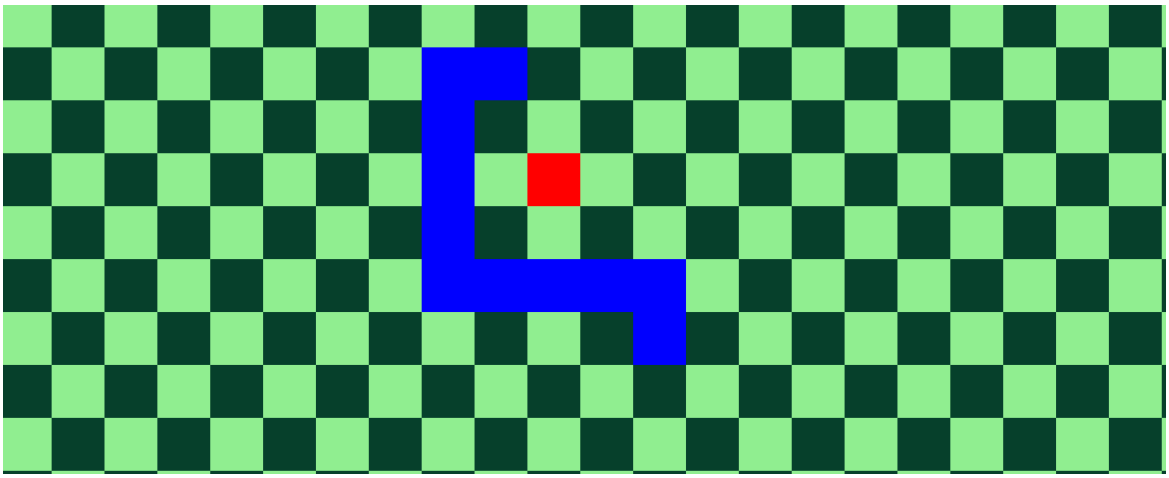
Za pomocą protokołu **EFI_LOADED_IMAGE_PROTOCOL** sprawdzane, jaki jest adres bazowy programu, a następnie jest wstawiany pod stały adres **0x10008**. Za to tuż obok pod **0x10000** ustawiana jest wartość **0xDEADBEEF**. Ta wartość stanowi nasz marker i jest znakiem dla debugera, że już program się załadował i może pobrać faktyczny adres bazowy i rozpocząć debugowanie programu.

Przy takim podejściu, aby rozpocząć, debugowanie należy odpalić program z flagą **debug** oraz w osobnym terminalu wpisać komendy, które obserwują dany adres, aż pojawi się tam **magic marker**:

```
1  gdb main.efi.debug
2  target remote localhost:1234
3  watch *(unsigned long long*)0x10000 == 0xDEADBEEF
4  continue
5  set $base = *(unsigned long long*)0x10008
6  add-symbol-file main.efi.debug -o $base
7  layout split
```

Po wykonaniu tych komend można korzystać ze standardowych poleceń GDB do debugowania.

5. Prezentacja wyników (warstwa wizualna)



Ekran aktywnej rozgrywki

Menu główne gry

Hall of Fame

6. Podsumowanie i wnioski

Projekt zakończył się sukcesem, a wszystkie założone cele zostały osiągnięte. Aplikacja Snake poprawnie inicjuje grafikę, obsługuje wejście użytkownika oraz przechowuje wyniki rozgrywek na dysku. Projekt udowodnił, że UEFI to w rzeczywistości zaawansowane środowisko operacyjne, które oferuje własne usługi zarządzania pamięcią i zasobami, eliminując potrzebę jądra systemu w prostych zastosowaniach.