Opis algorytmu exec

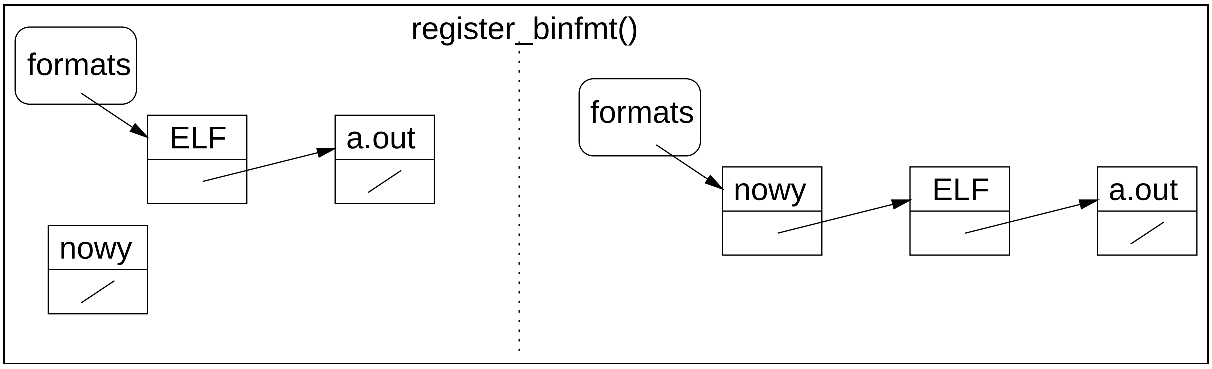
Paweł Misiewicz

08 Grudnia 2000

1 Wprowadzenie.

Zanim zacznę dokładnie omawiać algorytm `exec`, chciałbym zauważyć, że, tak naprawdę, to nie w nim następuje 'załadowanie' programu, tzn. algorytm `exec` wykorzystuje tylko już istniejące w systemie funkcje służące do ładowania plików binarnych, nie implementuje własnych. Głównym zadaniem tego algorytmu jest dokonanie różnego rodzaju asercji, na przykład związanych ze sprawdzeniem uprawnień użytkownika dotyczących ładowanego pliku, oraz przygotowanie parametrów przekazywanych do funkcji bezpośrednio odpowiedzialnych za uruchomienie programu. Każda taka funkcja jest ściśle związana z pewnym formatem pliku (*ELF, a.out*), natomiast każdy taki format musi zostać zarejestrowany w systemie przed załadowniem odpowiadającego mu pliku binarnego.

Rejestracja odbywa się za pomocą funkcji `register_binfmt(struct linux_binfmt *)`, która do systemowej listy zarejestrowanych formatów dołącza nowy, opisany w przekazywanej jako parametr strukturze `struct linux_binfmt` (opisana dalej). Funkcja rejestrująca format jest wywoływana podczas ładowania odpowiadającego formatowi modułu.



Rysunek 1: Rejestracja nowego formatu w systemie

Niestety dokładny opis algorytmów związanych z obsługą plików binarnych wykracza poza ramy tego opracowania, niestety, ponieważ zagadnienia związane z samymi plikami wykonywalnymi wydają się równie interesujące co algorytm `exec`, wykorzystujący je. Niemniej jednak, mam nadzieje, że ten krótki wstęp, pozwoli na lepsze zrozumienie działania algorytmu.

2 Funkcja `execve()`.

2.1 Deklaracja `execve()`.

Deklaracja powyższej funkcji znajduje się w pliku `/include/linux/unistd.h`. Jednak, tak naprawdę, plik ten dołącza jedynie plik nagłówkowy zależny od architektury komputera, zatem dla komputerów PC deklaracja (i zarazem definicja) tej funkcji znajduje się w pliku `/include/asm-i386/unistd.h`.

Funkcja `execve()` jest zadeklarowana w następujący sposób :

```
static inline int execve(const char *file, char **argv, char **envp);
```

Parametry przekazywane do funkcji to :

- `file` - nazwa programu do wykonania (pełna lub względna ścieżka dostępu)
- `argv` - tablica zawierająca argumenty dla programu
- `envp` - tablica zawierająca ustawione zmienne środowiskowe

Oczywiście funkcja ta nie nadaje się do bezpośredniego wykorzystania przez programistów ze względu na słowo kluczowe *inline*, które powoduje wstawianie kodu tej funkcji w miejsca jej wołania. Poza tym widać wyraźnie, że sam sposób przekazywania parametrów nie jest zbyt wygodny, dlatego też właśnie z myślą o programowaniu zostały dodatkowo zdefiniowane specjalne funkcje ułatwiające korzystanie z `execve()`. Są to, między innymi, `execl()`, `execle()`, `execv()`. Ułatwiają one używanie `execve()` właśnie dzięki innemu sposobowi przekazywania parametrów.

Więcej informacji na ten temat można uzyskać na stronach *man*.

2.2 Działanie funkcji `execve()`.

Zanim przejdę do omówienia samego algorytmu chciałbym ogólnie opisać działanie tej funkcji.

Zadaniem `execve()` jest uruchomienie programu wskazywanego przez zmienną `file` (ścieżka dostępu), przy czym `file` musi być albo binarnym plikiem wykonywalnym, albo skryptem zaczynającym się linią (#! *interpretator* [argumenty]). Przy czym w drugim przypadku, *interpretator* musi być nazwą wykonywalnego pliku, który sam skryptem nie jest, a który to zostanie wywołany w celu interpretacji skryptu. Oczywiście po odpowiednich modyfikacjach jądra, czy też po prostu dopisując odpowiedni moduł można dodać analogiczną funkcjonalność na przykład dla klas Javy, to znaczy, żeby programy napisane w Javie 'zachowywały' się jak standardowe pliki binarne.

W przypadku poprawnego wykonania, `execve()` nie zwraca żadnej wartości do programu wywołującego ją, co więcej kod, dane i stos tego programu są zamazywane (niekoniecznie fizycznie) przez te należące do ładowanego programu. Poza tym sygnały czekajace na obsłużenie przez wołający proces są usuwane oraz wszelkie procedury obsługi sygnałów są ustawiane na domyślne.

Niemniej jednak nie wszystkie dane wołającego procesu są niszczone, program uruchomiony, dziedziczy po programie wywołującym identyfikator procesu (PID) oraz wszelkie deskryptory otwartych plików, które nie mają ustawionej opcji zamykania podczas `exec`.

3 Przejście do trybu jądra, wywołanie funkcji sys_execve()

Jak wcześniej zauważyłem w celu uruchomienia programu wywoływana jest funkcja `execve()`, jednak przeglądając jej definicję, nietrudno stwierdzić, że jej główne zadanie to wywołanie odpowiedniego przerwania (SYSCALL_VECTOR) i przekazanie otrzymanych parametrów do procedury obsługi tego przerwania. Otóż parametry są przekazywane w następujących rejestrach :

- EAX - numer funkcji, która ma zostać wywołana (__NR_EXECVE)
- EBX - wskaźnik do nazwy pliku (file)
- ECX - wskaźnik do tablicy wskaźników do parametrów wywołania programu (argv)
- EDX - wskaźnik do tablicy wskaźników do zmiennych środowiskowych programu (argp)

Oczywiście wcześniej, podczas inicjalizacji systemu, musi zostać podana funkcja (`sys_call()`[1]), która będzie wykonywana podczas nadejścia przerwania. Jest to zrealizowane poprzez wywołanie funkcji :

```
set_system_gate(SYSCALL_VECTOR, &system_call);
```

podczas inicjalizacji (tworzenia) bramek w funkcji `trap_init()`[2] .

Należy również zauważyć, że po wywołaniu przerwania, kolejne wywołania funkcji odbywają się w trybie uprzywilejowanym (jądra), czyli z możliwością dostępu do wszelkich zasobów.

Po wejściu do funkcji `sys_call()` następuje seria rutynowych działań - sprawdzenie poprawności rodzaju funkcji przekazanego w EAX, zachowanie rejestrów (makro SAVE_ALL) oraz wywołanie odpowiedniej funkcji (EAX), tzn. następuje skok do miejsca w którym znajduje się początek danej funkcji. W naszym przypadku jest to skok do funkcji `sys_execve()` - adres jej początku znajduje się w tablicy `sys_call_table` pod offsetem EAX ($==$ 11).

4 Funkcja sys_execve().

Funkcja `sys_execve()` jest zdefiniowana w pliku `/arch/i386/kernel/process.c`, jako parametr otrzymuje strukturę zawierającą stan rejestrów procesora `struct pt_regs regs` - zauważmy, że są to te same dane, które funkcja `sys_call()` odłożyła na stos.

Główne zadanie `sys_execve()` polega na przygotowaniu środowiska i danych dla funkcji `do_execve()`.

Na początku działania blokowany jest dostęp do struktur jądra dla innych procesów, po czym do pamięci jądra kopiowana jest nazwa pliku (razem ze ścieżką dostępu). Takie postępowanie ma na celu przyspieszenie działania funkcji, ponieważ pamięć jądra nie podlega **wymianie** (*swapping*), więc w przypadku *zrzucenia* pamięci wołającego procesu na dysk, nie trzeba czekać na ewentualne ściągnięcie strony do RAM.

Po wykonaniu standardowych asercji związanych z powyższą operacją, tzn. sprawdzeniu czy kopiowanie (oraz alokacja pamięci) się powiodła, wywoływana jest funkcja `do_execve()`.

Ponieważ najważniejsze czynności są wykonywane w `do_execve()`, więc kolejne kroki to : sprawdzenie czy powyższe wywołanie się powiodło, jeśli tak to modyfikowane są flagi procesu. Później standardowo zwalaniana jest pamięć zaalokowana na nazwe pliku, odblokowywane jądro, oraz zwracany wynik `do_execve()` lub w przypadku niepowodzenia którejś z później wywoływanych funkcji - kod błędu.

Jak już wcześniej wspomniałem kolejna ważna funkcja związana z algorytmem `exec` to `do_execve()`, jednak zanim przejdę do jej opisu, proponuję zapoznać się z bardzo ważną z punktu widzenia tej właśnie funkcji strukturą, a mianowicie `struct linux_binprm`[3] .

[1] `/arch/i386/kernel/entry.S`
[2] `/arch/i386/kernel/traps.c`
[3] `/include/linux/binfmts.h`

5 Struktura `struct linux_binprm`.

Struktura ta zawiera następujące pola:

- `char buf[128]` - tu przechowywanych jest 128 pierwszych znaków ładowanego programu.
- `unsigned long page[MAX_ARG_PAGES]` - tablica zawiera adresy stron danych używanych do przekazania do nowego programu zmiennych środowiskowych oraz argumentów. Strony pamięci są alokowane dopiero wtedy, kiedy są potrzebne.
- `unsigned long p` - to jest *wskaźnik* na dane przechowywane w powyższych stronach pamięci. Dane są wstawiane od wyższych adresów do niższych, a `p` wskazuje na początek danych.
- `struct dentry *dentry` - reprezentuje uruchamiany plik.
- `int e_uid, e_gid` - zawierają, odpowiednio, efektywny identyfikator użytkownika oraz identyfikator grupy procesu uruchamiającego. Jeżeli to jest proces typu *su*, to tutaj są nowe wartości.
- `int argc, envc` - reprezentują, odpowiednio, liczbę argumentów i liczbę zmiennych środowiskowych przekazanych do programu.
- `char *filename` - pełna ścieżka dostępu do uruchamianego programu. Wskazuje na napis w przestrzeni adresowej jądra.

Powyżej wypisane pola są w zasadzie najważniejsze z punktu widzenia uruchamianego programu. Pozostałe służą głownie jako pomocnicze podczas tego procesu.

Teraz, mając ogólną wiedzę na temat tej struktury możemy przejść do omawiania funkcji `do_execve()`.

6 Funkcja `do_execve()`.

Funkcja `do_execve()` jest zdefiniowana w pliku `/fs/exec.c`, jako parametry otrzymuje te dane, które otrzymała `execve()` oraz strukturę `struct pt_regs regs`.

W funkcji tej, na początku, deklarowana jest zmienna `struct linux_binprm bprm` w której to będą przekazywane parametry do kolejnych funkcji. Kolejno wypełniane są pola tej struktury. Najpierw ustawiamy `bprm.p` na ostatni element możliwy do zaadresowania za pomocą tablicy `bprm.page`, po czym wypełniamy elementy tej tablicy zerami (NULL). Później pobieramy wskaźnik do struktury `struct dentry` wskazującej na wykonywany plik (nazwę pliku otrzymujemy jako argument `filename`) i po przeprowadzeniu standardowych sprawdzeń ustawiamy na tę wartość zmienną `bprm.dentry` oraz ustawiamy `bprm.filename` na `filename`.

Następnie wywołując funkcję `count()`, zliczającą liczbę argumentów oraz zmiennych środowiskowych, odpowiednio ustawiamy pola `bprm.argc` oraz `bprm.envc`.

Dalsze wypełnianie struktury `struct linux_binprm bprm` odbywa się w funkcji `prepare_binprm()`. Tam też są ustawiane pola `bprm.e_uid` oraz `bprm.e_gid`, jak również te związane z **możliwościami** (*capabilities*) jądra. Jednak najważniejsze czynności funkcji `prepare_binprm()` to sprawdzenie **pozwoleń** (*permissions*) na wykonywanie operacji na i-węźle związanym z uruchamianym programem oraz wczytanie pierwszych 128 bajtów pliku tego programu - realizowane jest to za za pomocą kolejnej pomocniczej funkcji - `read_exec()`.

Po zakończeniu funkcji `prepare_binprm()` sukcesem, kopiowane są do pamięci jądra dane przekazane dla uruchamianego programu, tzn. za pomocą funkcji `copy_strings()` kopiujemy argumenty wywołania uruchamianego programu oraz zmienne środowiskowe.

Jeżeli wszystkie powyższe operacje się powiodą wywoływana jest funkcja `search_binary_handler()` - to ona zajmuje się właściwym uruchomieniem programu.

Po zakończeniu powyższej funkcji przeprowadzane jest zwalnianie zajętych zasobów : struktury `dentry` oraz pamięci, czyli stron, których adresy znajdują się w tablicy `bprm.page[]`.

Przejdźmy zatem do wspomnianej powyżej funkcji `search_binary_handler()`, jednak wcześniej, po krótce, przedstawię strukturę `struct linux_binfmt`, tą dzięki której system może utrzymywać informacje na temat zarejestrowanych formatów plików binarnych.

7 Struktura `struct linux_binfmt`.

Struktura ta jest używana podczas rejestracji formatów plików binarnych. Najważniejsze dla nas pola to :

- `struct linux_binfmt *next` - wskaźnik na następny opis formatu.
- `long *use_count` - wskazuje na liczbę *użytkowników* modułu implementującego poniższą funkcję.
- `int (*load_binary(struct linux_binprm *, struct pt_regs))` - wskaźnik na funkcję (*metodę*) definiowaną przez moduł, służącą do załadowania pliku programu. Może wynosić `NULL`, jeżeli format nie definiuje tej funkcji.

Jak widać nie jest to skomplikowana struktura, niemniej jednak jest niesłychanie użyteczna, ponieważ dzięki takiemu sposobowi implementowania obsługi plików wykonywalnych istnieje możliwość dodefiniowania nowych formatów, bez konieczności rekompilacji jądra. Z drugiej strony, każdy opis formatu, stanowi odrębną, niezależną od reszty kodu całość, co powoduje zmniejszenie ryzyka pojawienia się błędów podczas programowania.

8 Funkcja `search_binary_handler()`.

Funkcja ta jest również zdefiniowana w pliku `/fs/exec.c`, jako parametr otrzymuje dodatkowo (oprócz stanu rejestrów z `sys_call()`) wypełnioną strukturę `struct linux_binprm`.

Głównym zadaniem tej funkcji jest przeszukanie dostępnych (zarejestrowanych) formatów plików binarnych i próba uruchomienia programu na podstawie wczytanego do bufora początku pliku programu. Dokładnie rzecz biorąc, funkcja działa w następujący sposób. Na początku pobierany jest wskaźnik do początku listy zarejestrowanych formatów binarnych - poprzez makro `formats`. Następnie, dla każdego formatu, pobieramy *jego* funkcję służącą do ładowania programu (`load_binary()`), sprawdzamy czy nie jest ona `NULL`-em (niektóre z zarejestrowanych formatów mogą nie obsługiwać ładowania plików binarnych) i wywołujemy. Przekazujemy następujące paramtery : wypełnioną wcześniej strukturę `struct linux_binprm bprm` wraz z wypełnioną strukturą `struct pt_regs regs` - zawierającą wartości rejestrów z momentu wejścia do funkcji obsługi przerwania 0x80 (`sys_call()`). W przypadku powodzenia zwalniamy `dentry` oraz ustawiamy flagę w strukturze procesu mówiącą o tym, że udało się wykonać `exec`. W przeciwnym razie (po sprawdzeniu wszystkich dostępnych formatów) próbujemy załadować moduł (request_module()), który mógłby być odpowiedzialny za ten format i po raz kolejny sprawdzamy wszystkie zarejestrowane formaty.

Myślę, że warto dodać w jaki sposób następuje wybór modułu do załadowania. Otóż, po sprawdzeniu, że pierwsze cztery znaki w pliku nie są (wszystkie) **wypisywalne** (*printable*), tworzymy nazwę modułu wywołując funkcję

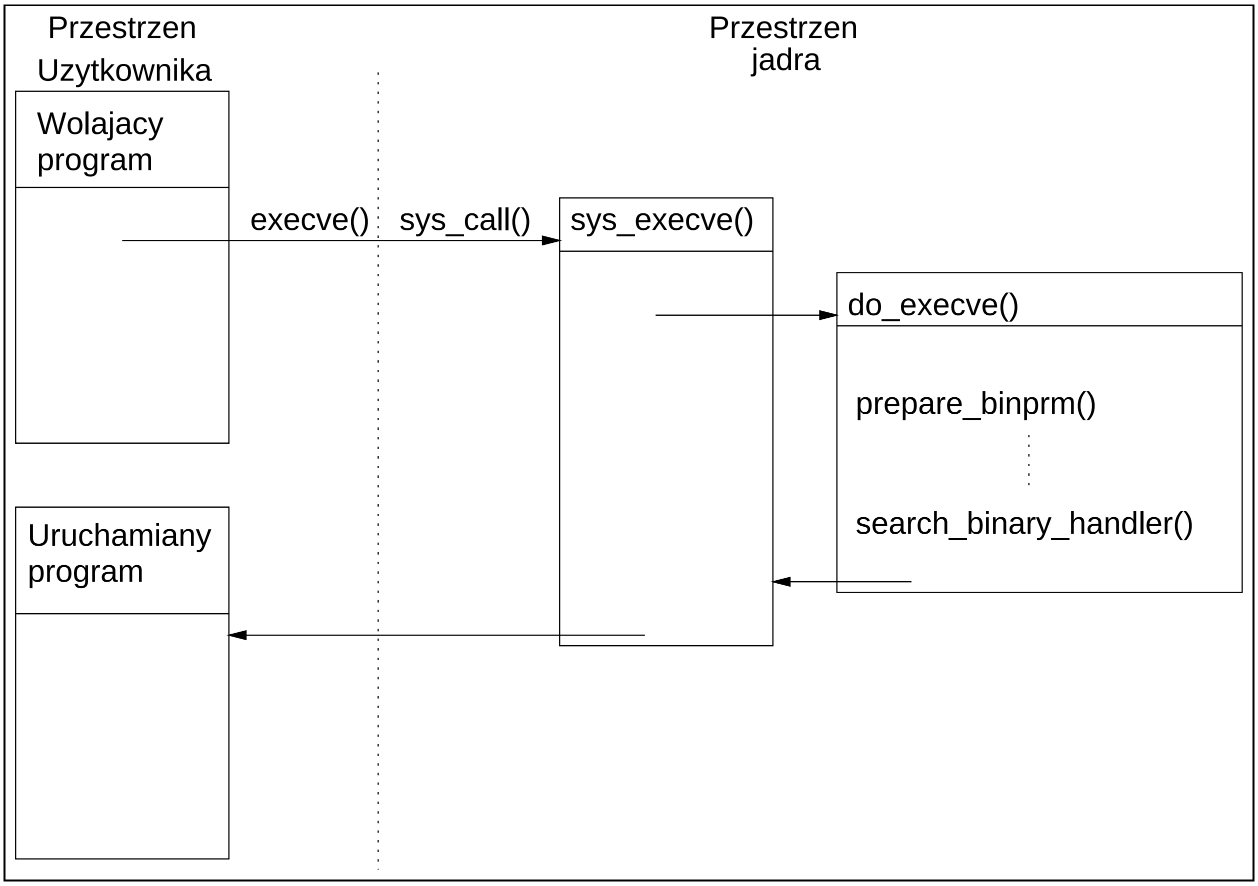
```
sprintf(modname, "binfmt-%04x", *(unsigned short *)(&bprm->buf[2]));
```

Mamy nadzieję, że tak właśnie nazywa się moduł obsługujący dany format, tzn. jeżeli taki moduł istnieje, to tak powinien się nazywać :). Później wywołujemy funkcję `request_module(modname)` żądając wczytania modułu `modname`.

Jeżeli operacja szukania ostatecznie się powiodła, to po wyjściu z kolejnych funkcji do poziomu `sys_call()` oraz po zakończeniu instrukcji jądra, adres kolejnej instrukcji do wykonania zostanie ustawiony na początek nowego (właśnie załadowanego) programu. Jeżeli natomiast nie została znaleziona funkcja obsługująca dany format, to do programu wywołującego `execve()` zostanie zwrócony kod błędu.

9 Podsumowanie - schemat przebiegu sterowania podczas wykonywania funkcji `execve()`

Poniżej przedstawiam schemat przebiegu sterowania podczas wykonywania algorytmu `exec`.



Rysunek 2: Przeplyw sterowania w algorytmie `exec`

Spis treści