Niezawodność i diagnostyka układów cyfrowych 2

Ochrona Funkcji Systemowych

Spis treści

Wstęp

Wiodące jądra systemów operacyjnych, takie jak FreeBSD, Linux, Windows i XNU (OS X), posiadają elastyczną architekturę składającą się z rdzenia jądra i zewnętrznych modułów. Rdzeń jądra utrzymuje podstawowe elementy, takie jak synchronizacja wątków i zarządzanie pamięcią, oraz funkcje biblioteczne, dostępne dla podsystemów wewnętrznych jądra i modułów jądra. Wartością dodaną są funkcje bliźniacze, które rozszerzają podstawowe elementy do pracy w różnych scenariuszach dzięki drobnym różnicom w semantyce. Na przykład, kmalloc() w jądrze Linuxa ma swoje odpowiedniki: kmalloc_array() i kcalloc(), które zapewniają odpowiednie zarządzanie pamięcią i inicjalizację pamięci. Niestety, często deweloperzy popełniają błędy, wybierając niewłaściwą funkcję bliźniaczą, co prowadzi do usterek i problemów wydajnościowych. Poprawne wykorzystanie funkcji bliźniaczych jest kluczowe dla bezpiecznego i wydajnego kodu.

Postaramy się pomóc społeczności zrozumieć i ograniczyć problem nadużywania funkcji bliźniaczych, raportując nasze wyniki oraz dokonując poprawek w samym systemie. Zidentyfikowaliśmy cztery główne czynniki prowadzące do nadużywania funkcji bliźniaczych oraz określiliśmy odpowiednie ograniczenia programistyczne. Chcemy także odpowiedzieć na pytanie, czy wykrywanie nadużyć za pomocą narzędzi automatycznych jest możliwe.

W związku z tym nasze badanie odpowiada na następujące trzy pytania:

- Jakie czynniki prowadzą do nadużycia funkcji bliźniaczych?

    - Nieodpowiedni kontekst wywołania, który uniemożliwia użycie jednej z funkcji bliźniaczych.
    - Zastępowanie funkcji o gorszej wydajności przez ich bliźniacze, co prowadzi do braku optymalizacji.
    - Brak ulepszeń związanych z bezpieczeństwem w nadużywanych funkcjach.
    - Naruszenie zasad stylu kodowania jądra.
- Jakie ograniczenia programistyczne powinni stosować deweloperzy, aby uniknąć nadużywania funkcji bliźniaczych?
    - Deweloperzy powinni stosować się do określonych zasad, które precyzują, kiedy i jak używać funkcji bliźniaczych w konkretnych przypadkach.
- Czy wykrywanie nadużyć funkcji bliźniaczych za pomocą narzędzi automatycznych jest możliwe?
    - Tak, wykorzystujemy narzędzia analizy statycznej, takie jak Coccinelle, aby wykrywać nadużycia funkcji bliźniaczych. Te narzędzia ostrzegają deweloperów i pomagają identyfikować nieprawidłowości w kodzie.

Szkic

Różnorodność Zastosowań Funkcji Bliźniaczych

W celu uzasadnienia istnienia funkcji bliźniaczych w jądrach systemów operacyjnych, przyjrzymy się rozwojowi rodziny funkcji kmalloc() w jądrze Linux na podstawie dzienników zmian i dyskusji e-mailowych głównych programistów. Początkowo, w wersji Linux 1.0, jedyną opcją alokacji pamięci był kmalloc(). Później, w wersji Linux 2.1.23, wprowadzono mechanizm zarządzania pamięcią slab, co zaowocowało pojawieniem się nowych funkcji, takich jak kmem_cache_alloc(). Wraz z kolejnymi wersjami systemu, wprowadzono także warianty funkcji kmalloc() uwzględniające specyficzne potrzeby, takie jak alokacja pamięci NUMA-aware czy obsługa dużych alokacji pamięci. W sumie, w wersji Linux 5.0, liczba funkcji alokujących pamięć dostępnych w pliku slab.h wzrosła do 23. Wszystkie te funkcje mają nazwy zaczynające się od k*alloc, opcjonalnie z sufiksami.

W ramach omawiania ewolucji funkcji alokujących pamięć w jądrze Linux, skoncentrujmy się na jednym z problemów, który może pojawić się w praktyce programistycznej. Opisany błąd dotyczy nieprawidłowej inicjalizacji zmiennej xt w kodzie jądra systemu Linux. Głównym problemem jest użycie pola xt->cur jako indeksu tablicy compat_tab bez uprzedniej inicjalizacji zmiennej xt. Aby rozwiązać ten problem, w kodzie została wprowadzona poprawka polegająca na zainicjalizowaniu zmiennej xt za pomocą funkcji kcalloc(), zapewniając tym samym poprawną alokację pamięci dla zmiennej xt. Dzięki temu zapewniono, że pole xt->cur ma poprawną wartość początkową, co pozwala na bezpieczne użycie jako indeksu tablicy compat_tab.

```cpp
#include <iostream>
#include <vector>

using namespace std;
// struktura reprezentująca dane xt_at
struct XtAt
{
    int cur;
    vector<int> compat_tab;
};

int main()
{
    int n = 10;
    int offset = 5;

    // inicjalizacja struktury XtAt
    XtAt *xt = new XtAt;

    // alokacja pamięci dla pola compat_tab przy użyciu kcalloc
    xt->compat_tab.resize(n);

    // użycie pola cur jako indeksu do tablicy compat_tab
    xt->cur = 0;
    xt->compat_tab[xt->cur] = offset;

    cout << "Offset: " << xt->compat_tab[xt->cur] << endl;

    // zwolnienie pamięci zaalokowanej dla xt (na końcu działania programu)
    delete xt;

    return 0;
}
```

rys. 1 - na wycinku przedstawiony jest przykład kodu (symulacja omawianego problemu w C++) rozwiązującego problem związany z nieprawidłową inicjalizacją zmiennej xt w jądrze systemu Linux, gdzie pole xt->cur jest używane jako indeks tablicy compat_tab bez wcześniejszej inicjalizacji zmiennej xt.

Potencjalne Problemy Związane z Funkcjami Bliźniaczymi

W przypadku przedstawionym na rys. 1 występuje błąd w użyciu funkcji kmalloc_array() w systemie Linux. W linii 6 próbuje ona zaalokować tablicę obiektów struct xt_at, ale zapomina o

ich zerowaniu. Niezainicjalizowane xt->cur jest używane w linii 24, co może prowadzić do dostępu do pamięci poza granicami. Poprawka zastępuje kmalloc_array() funkcją kcalloc(), która zeruje wszystkie zaalokowane obiekty.

W tym przypadku deweloperzy muszą wykonać dwie rzeczy. Po pierwsze, kod musi wykryć, czy całkowity rozmiar zaalokowanej tablicy, n*sizeof(struct xt_af), nie przekracza limitu. Po drugie, zaalokowane obiekty powinny być zainicjalizowane wartościami zerowymi. Deweloper dokonuje poprawnego wyboru jedynie w pierwszym przypadku, ponieważ kmalloc_array() nie spełnia drugiego wymagania, podczas gdy jest to spełnione przez kcalloc(). Jednakże ta poprawka nie sugeruje, że kcalloc() jest zawsze najlepszym wyborem. Należy zachęcać do ostrożnego używania kcalloc(), jeśli zaalokowane obiekty są później jawnie inicjalizowane, co tylko marnuje czas procesora. Deweloperzy muszą dokonać precyzyjnego wyboru spośród wszystkich funkcji bliźniaczych.

Przegląd Poprawek do Badania

Rozwój narzędzia do analizy statycznej i poprawki w kodzie jądra:
Te poprawki skupiają się na eliminowaniu błędów i luk w bezpieczeństwie związanych z niewłaściwym użyciem funkcji bliźniaczych w jądrze systemu operacyjnego. Poprzez automatyzację procesu identyfikacji i korekty takich błędów, system staje się bardziej stabilny i bezpieczny, co bezpośrednio wpływa na ochronę krytycznych funkcji systemowych przed potencjalnymi zagrożeniami.

Implementacja tracerów 'osnoise' i zaawansowane opcje konfiguracji systemu:
Tracery 'osnoise' pozwalają na monitorowanie i analizowanie szumu systemowego, który może wpływać na wydajność i czas odpowiedzi systemu operacyjnego. Poprzez precyzyjne zarządzanie szumem i minimalizację jego wpływu, system jest lepiej przygotowany na obsługę krytycznych zadań w czasie rzeczywistym, co zabezpiecza funkcje systemowe przed niestabilnością i potencjalnymi błędami.

Sprzętowe wsparcie dla funkcji futex:
O ile ten temat został omówiony, sprzętowe wsparcie dla syscalli futex może znacząco poprawić wydajność synchronizacji w przestrzeni użytkownika, co z kolei minimalizuje opóźnienia i zwiększa responsywność systemu. Dzięki temu funkcje systemowe są bardziej odporniejsze na opóźnienia i przeciążenia, co jest szczególnie istotne w systemach czasu rzeczywistego.
Wykorzystanie liczników wydajności sprzętowej:
Metoda ta pozwala na identyfikację i monitorowanie wykonania funkcji w programach, co może być wykorzystane do wczesnego wykrywania nieautoryzowanych lub nieoczekiwanych zmian w działaniu systemu. Jest to kluczowe dla ochrony funkcji systemowych przed złośliwym oprogramowaniem i innymi próbami naruszenia bezpieczeństwa systemu.

Każda z tych poprawek, chociaż może dotyczyć różnych aspektów systemu operacyjnego, przyczynia się do jego ogólnego bezpieczeństwa, stabilności i niezawodności, co jest fundamentalne dla ochrony funkcji systemowych.

Czynniki Niewłaściwego Wykorzystania Funkcji Bliźniaczych

Funkcje bliźniacze, choć podobne, różnią się subtelnymi, ale istotnymi szczegółami, które mogą wpływać na bezpieczeństwo, stabilność oraz wydajność systemu. Oto główne czynniki, które prowadzą do takich nadużyć:

Niewłaściwe zastosowanie w określonych kontekstach: Jednym z najczęstszych błędów jest użycie funkcji bliźniaczej w sytuacji, dla której nie została ona zaprojektowana. Taka sytuacja może wystąpić, gdy programiści, nie będąc w pełni świadomi subtelnych różnic między funkcjami, wybierają te, które wydają się na pierwszy rzut oka odpowiednie, ale w rzeczywistości nie są przystosowane do specyficznych wymagań lub warunków operacyjnych.

Decyzje związane z wydajnością: W niektórych przypadkach wybór między funkcjami bliźniaczymi może być podyktowany dążeniem do optymalizacji wydajności. Takie podejście, choć zrozumiałe, może prowadzić do kompromisów w zakresie bezpieczeństwa lub stabilności systemu, szczególnie gdy wybrana funkcja nie obejmuje niezbędnych mechanizmów ochronnych lub jest mniej odporna na specyficzne błędy i ataki.

Braki w ulepszeniach związanych z bezpieczeństwem: Niektóre funkcje bliźniacze mogą nie zawierać aktualizacji bezpieczeństwa, które są obecne w ich odpowiednikach. Użycie starszej wersji funkcji bez tych kluczowych ulepszeń może nieświadomie otworzyć drzwi dla ataków lub narazić system na nieprzewidziane awarie.

Naruszenie standardów kodowania jądra: Każde środowisko programistyczne, a szczególnie tak skomplikowane jak jądro systemu operacyjnego, wymaga przestrzegania określonych konwencji i standardów kodowania. Niewłaściwe użycie funkcji bliźniaczych, które ignoruje te zasady, może prowadzić do trudności w utrzymaniu kodu, jego aktualizacji oraz w długoterminowej współpracy w ramach zespołu programistów.

Rozumienie i adresowanie tych czynników za pomocą narzędzi do analizy statycznej, takich jak opracowane w ramach omawianego projektu, pozwala na znaczące zwiększenie niezawodności, bezpieczeństwa i wydajności jądra systemu operacyjnego, jednocześnie minimalizując ryzyko błędów wynikających z niewłaściwego wykorzystania funkcji bliźniaczych.