Buffer Overflow vs Ochrona Pamięci

Wojciech Balik

1 Czym jest Buffer Overflow?

Niektóre języki programowania, takie jak np. C czy C++, oferują programiście dość dużą kontrolę nad pamięcią. Niestety, jak mówi znane powiedzenie, "*Z wielką władzą wiąże się wielka odpowiedzialność*", co w przypadku kontroli nad pamięcią znaczy że język gwarantuje programiście wiele możliwości strzelenia sobie w stopę. Jednym ze skutków takiego podejścia jest podatność polegająca na przepełnieniu bufora, to znaczy, próby zapisania do niego większej ilości bajtów niż jego wielkość. Prowadzi to do nadpisania danych bezpośrednio za buforem, często na tyle wrażliwych by doszło do zdalnego wykonania kodu.

Klasycznym przykładem tego typy podatności jest użycie funkcji gets, co widać na poniższym kodzie źródłowym:

```
#include <stdio.h>

int main(void)
{
    char buf[32];
    gets(buf);
    puts(buf);

    return 0;
}
```

Funkcja gets w żaden sposób nie kontroluje tego, jak dużo bajtów zapisze do bufora, co prowadzi do tego, że atakujący ma pełną kontrolę nad np. adresem powrotu odłożonym na stosie. Jest to główny powód, dla którego po wydaniu polecenia **man 3 gets** w powłoce Linuxowej, lub nawet przy kompilacji, użytkownik jest ostrzegany, by nie używać tej funkcji.
zakładając najprostszy przypadek, tzn. taki, że mamy pełną wiedzę o wirtualnej przestrzeni adresowej programu, oraz że nie ma podziału na dane i kod, można przeprowadzić atak w następujący sposób:

1. Wpisujemy 36 bajtów paddingu(32 na bufor + 4 na odłożony na stosie rejestr rbp)
2. Adres powrotu nadpisujemy adresem miejsca na stosie, pod którym znaduję się nadpisywany adres powrotu, plus 4(czyli po prostu adresem całej reszty nadpisywanego bufora)
3. resztę pamięci nadpisujemy naszym kodem

Czasami może zdarzyć się tak, że znamy adres naszego bufora, ale tylko w przybliżeniu, tzn. prawdziwy adres jest oddalony o pewną(stosunkowo niewielką) stałą od tego adresu który znamy. Wówczas efektywną techniką jest dodanie do naszego kodu dużego prefiksu składającego

Instytut Informatyki Uniwersytetu Wrocławskiego, 2017–2018
Esej przygotowany w ramach zajęć *Kultura bezpieczeństwa komputerowego*

się z instrukcji NOP(często występujące określenie to **NOP sled**) Wtedy "trafiając" naszym znanym adresem w ten prefiks, mamy pewność, że reszta kodu się wykona, co znacznie zwiększa prawdopodobieństwo trafienia.

2 Zabezpieczenia oraz ich łamanie

Mimo że podatności typu buffer overflow są błędami programistycznymi, dobrze byłoby zapobiegać wykorzystywaniu tego typu błędów. Z tego powodu wprowadzono wiele zabezpieczeń, które w wielu przypadkach udaremniają eksploitację tego typu podatności. Jak można się domyślić, znalazły się także (bardzo ciekawe)sposoby na ich obejście.

2.1 Polityka W^X

W poprzednim przykładzie zakładaliśmy, że nie ma podziału na dane i kod. W praktyce jednak, nie ma żadnego powodu, aby segmenty takie jak stos były wykonywalne. Można też pójść jeszcze dalej i stwierdzić, że żadna sekcja zawierająca dane nie powinna być wykonywalna. Taka polityka nazywa się **Write XOR Execute**, co znaczy tyle, że pamięć nigdy nie powina być jednocześnie zapisywalna i wykonywalna. Wprowadzenie tej idei w życie może być wykonane na poziomie systemu operacyjnego, ale skutkowałoby to spadkiem wydajności[1]. Rozwiązaniem tego problemu było skorzystanie ze wsparcia sprzętowego. Producenci niektórych procesorów(np. tych z rodziny x86), zdecydowali się dodać do każdego wpisu w tablicy stron bit NX(No eXecute), który powoduje, że przy próbie wykonania kodu ze strony z uswawionym bitem NX zostanie wygenerowany wyjątek procesora. Polityka ta została wdrożona stosunkowo niedawno, w systemach Windows oraz Linux została ona zaimplementowana w roku 2004[1].
Takie podejście ma jednak swoje wady. Niektóry technologie, takie jak np. kompilacja JIT, potrzebują możliwości jednoczesnego pisania i wykonywania fragmentów pamięci. Oczywiście, w Linuxie istnieje wywołanie systemowe **mprotect(2)**, pozwalające zmienić prawa dostępu do pamięci, ale to wprowadza narzut czasowy, a przecież celem kompilacji JIT jest zwiększenie wydajności.

2.1.1 Atak ret2libc

Wprowadzenie idei niewykonywalnej pamięci skutecznie powstrzymuje atakującego od wykonywania wstrzykniętego kodu. Można jednak zadać sobie pytanie, czy tylko kod umieszczony w pamięci przez atakującego może narobić szkód? Odpowiedź brzmi nie i na tym opiera się atak **ret2libc**.
Żeby zrozumieć na czym on polega, przenieśmy się na chwilę do przykładu z rozdziału pierwszego. Oczywiście nie ma tam żadnego złośliwego kodu, ale trzeba też wziąć pod uwagę jak będzie wyglądała pamięc wirtualna procesu, po uruchomieniu programu. Jest to program napisany w języku c, a więc do pamięci będzie załadowana biblioteka standardowa(czyli inaczej libc), w której znajduje się funkcja **system**, która używając powłoki, wywołuje podane jako argument polecenie. W szczególności, jeśli atakujący zdołałby wywołać tę funkcję z parametem "*/bin/sh*", uzyskałby dostęp do powłoki z prawami użytkownika, do którego należy dany proces, co może skonczyć się bardzo źle jeśli łączymy się z aplikacją przez sieć, a jeszcze gorzej jeśli właścicielem jest root.
Nie porzucając założenia o znajomości całej przestrzenii adresowej procesu,możemy skoczyć do funkcji **system** tak samo jak skoczyliśmy na stos w przykładzie poprzednim, czyli nadpisując adres powrotu i wykorzystując asemblerową instrukcję ret. A co z argumentami? Przecież kontrolujemy stos, więc nie ma problemu abyśmy zaraz za nadpisanym adresem powrotu umieścili argumenty. Jeśli argumenty są przekazywane przez rejestry to sprawa się nieco komplikuje, ale można temu zaradzić stosując ROP, opisane w następnym rozdziale.

2.2 ASLR

Podczas eksploitacji bardzo potrzebna jest znajomość adresów absolutnych. W poprzednich przykładach posługiwaliśmy się adresami stosu oraz segmentu kodu biblioteki standardowej języka C. Z tego powodu, jednym z mechanizmów mających na celu utrudnienie atakującemu wykorzystania błędów bezpieczeństwa jest randomizacja przestrzeni adresowej(ang. Addres space layout randomization). Ta idea wprowadza jednak pewne komplikacje, np. do poprawnego działania, kompilator nie może wygenerować kodu odnoszącego się do adresów absolutnych. Z tym problemem jednak można sobie poradzić generując kod PIC(ang. Position independent code), czyli kod, który posługuje się tylko adresami względnymi, działający niezależnie od tego gdzie zostanie załadowany. Skompilowanie kodu jako PIC nie gwarantuje jednak, że wszystkie segmenty będą ładowane losowo. Mechanizm ASLR jest zaimplementowany w jądrze systemu operacyjnego(w linuxie został wprowadzony w roku 2005). Niestety, w większości systemów(możliwe że nawet we wszystkich) rozkład prawdopodobieństwa nie jest jednostajny, powodem jest np. to, że adres pod którym ładowany jest segment musi być wyrównany do wielkości strony. Na losowość wpływa jeszcze wiele czynników, ale nie będziemy wchodzić w szczegóły. Ludzie przeprowadzili wiele testów pozwalających określić entropię ładowania poszczególnych segmentów, np. jak podaje [2],w przypadku ładowania bibliotek dzielonych, na 32-bitowym systemie linux mamy tylko od 8 do 19 bitów, co znaczy mniej więcej tyle, że dysponując jedynie 32-bitową przestrzenią adresową, ASLR jest praktycznie bezużyteczny, ponieważ atak typu bruteforce to kwestia minut. Wersja 64-bitowa jest nieco lepsza, ponieważ entropia wacha się pomiędzy 28 a 35 bitami(w zależności od implementacji), przez co w przypadku poprzednio podanego przykładu atakujący jest praktycznie bezradny.

2.2.1 Return Oriented Programming

Wiele kompilatorów generuje kod typu PIC domyślnie, co nie znaczy jednak, że zawsze i wszędzie będziemy spotykać tylko taki kod. Skutkiem tego jest to, że w losowych miejscach będą wszystkie segmenty oprócz kodu naszej aplikacji. Można zadać sobie pytanie, czy można to jakoś wykorzystać, nawet jeśli w kodzie naszej aplikacji nie ma tak krytycznych funkcji jak **system**? Odpowiedź brzmi tak. Używa się do tego bardzo ciekawej i ogólnej metody zwanej **Return Oriented Programming**(w skrócie **ROP**). Należy myśleć o niej jako o sposobie eksploitacji, a nie jak o konkretnym ataku. Załóżmy najpierw, że kontrolujemy adres powrotu z funkcji oraz kawałek stosu za nim. Idea polega na tym, by wykonywać kod po kawałku. Aby tego dokonać, będziemy używać tylko takich kawałków kodu asemblera które spełniają kilka warunków:

1. zawierają kilka instrukcji(np. od dwóch do pięciu)
2. nie używają instrukcji *push*
3. kończą się instrukcją *ret*

Przyjęło się, że tego typu fragmenty kodu nazywa się **gadżetami**. Przykładowo:

```
pop rax
pop rdx
ret
```

Teraz wystarczy adres powrotu nadpisać adresem gadżetu, za adresem powrotu wpisać adres kolejnego gadżetu, być może w międzyczasie wpisując na stos jakieś dane, tak by załadować je do rejestru instrukcją *pop*. W przypadku programów zawierających duża ilość kodu bardzo łatwo będzie znaleźć taki ciąg gadżetów, który umieszcza w rejestrach odpowiednie wartości, a następnie używa instrukcji *int 0x80*, by wykonać wywołanie systemowe **execve** z argumentem "*/bin/sh*".

2.3 Ciasteczka na stosie

Dawniej, w kopalniach węgla problemem było ulatnianie się silnie toksycznego tlenku węgla. Aby zwiększyć bezpieczeństwo, w kopalniach umieszczano kanarki, które dzięki szybkiemu oddychaniu oraz szybkiemu metabolizmowi, w przypadku ulatniania się toksycznego gazu, zdychały bardzo szybko, co dawało górnikom czas na ucieczkę. Na podobny pomysł wpadli projektanci kompilatorów, którzy postanowili, że w ramce stosu, przed adresem powrotu, umieszczą dodatkową wartość wielkości słowa maszynowego, często nazywaną ciasteczkiem bądź kanarkiem. Wartość ta jest całkowicie losowa, a przed powrotem z funkcji sprawdzane jest czy się jakkolwiek zmieniła, jeśli tak, program kończy swoje działanie w sposób tak bezpieczny, jak to tylko możliwe, a jeśli nie, program kontynuuje działanie. Wadą jest to że ogranicza wydajność, ponieważ dokleja do funkcji kod powodujący narzut czasowy. Z tego powodu kompilatory posługują się heurystykami dotyczącymi tego czy warto umieszczać ciasteczko na stosie. Według [3], ciasteczko zostanie umieszczone na stosie gdy funkcja spełnia choć jeden z następujących warunków:

- Operuje na tablicy, która jest większa niż cztery bajty, ma więcej niż dwa elementy i żaden z elementów nie jest wskaźnikiem.
- Zawiera strukturę danych o rozmiarze większym niż 8 bajtów, niezawierającą żadnych wskaźników.
- Rezerwuje pamięć na stosie przy użyciu funkcji *alloca* lub *_malloca*
- Obejmuje dowolną strukturę, lub klasę, która zawiera wymienione typy danych.

Oprócz tego, podczas kompilacji należy podać opcję *-fstack-protector*, lub *-fstack-protector-all* by wymusić używanie ciasteczek we wszystkich funkcjach. Ciasteczek nie da się pokonać w żaden sprytny sposób, jednak to że znajdują się w okolicach kontrolowanego przez użytkownika bufora, czasami daje pewne możliwości by odkryć ich wartość. Jest to możliwe w naszym przykładzie, wystarczy wpisać do bufora wystarczają ilość bajtówm, tak by funkcja puts razem z zawartością bufora, wypisała także ciasteczko. Co prawda nie uda nam się osiągnąć niczego więcej(potrzebna byłaby jeszcze jedna okazja do przepełnienia bufora), ale pokazuj to, że odkrycie wartości ciasteczka wcale nie musi być trudne.
Problemem przy implementacji tej technologii może być źródło entropii. O ile podczas normalnego użytkowania systemu możemy pozyskiwać entropię z różnych źródeł np. z szumu termicznego, to podczas bootowania systemu bedzie nieco trudniej, no bo przecież żeby skorzystać ze wsparcia sprzętowego, system operacyjny musi załadować sterowniki, czyli inaczej mówiąc musi wykonać jakiś kod. No ale skoro brak nam losowości to jak wygenerować losowe ciasteczka? Jak widać determinizm komputerów nie zawsze współgra z bezpieczeństwem, przykładowo, okazało się, że w wielu modułach systemu Windows XP, możliwe było odgadnięcie ciasteczka bez większego problemu[4].

2.4 Shadow Stack

W 2015 roku pojawiła się idea wprowadzenia tzw. Shadow Stack[5], która ma raz na zawsze zakończyć wykorzystywanie błędów związanych z nadpisaniem adresu powrotu, co zazwyczaj jest spowodowane przepełnieniem bufora. Polega ona na tym by przy wywoływaniu funkcji, oprócz umieszczania adresu powrotu na zwykłym stosie, wrzucić go także na specjalny stos, kontrolowany przez system operacyjny oraz hardware, a przy powrocie sprawdzać, czy adresy na obu stosach są sobie równe, jeśli nie, rzucany jest wyjątek. Niestety, wprowadza to kolejny narzut czasowy, w niektórych przypadkach dość duży. Mimo to, w czerwcu roku 2016, Intel wydał ponad 100-stronnicowy dokument zapowiadający wprowadzenie tej technologii w życie[6]. Niektórzy[7] nazywają tę technologię "*RIP ROP*", sugerując że to koniec eksploitacji tego typu.

Jak będzie w rzeczywistości? Czy powstaną nowe techniki które zastąpią ROP? Czas pokaże, jest jescze za wcześnie by wydawać wyroki.

2.5 Buffer Overflow na stercie

Dotychczas mówiąc o przepełnieniu bufora zakładaliśmy, że wszystko dzieje się na stosie. A co jeśli bufor znajduje się na stercie? Jest lepiej, gorzej? Na pewno jest źle. By dobrze przedstawić wszystkie konsekwencje, które za tym stoją wymagana jest wiedza o tym, jak działa alokator pamięci, czyli np. funkcja **malloc** z języka C. Implementacji tej funkcji jest kilka, przykładowo, Linux używa algorytmu **ptmalloc**, który jest bardzo skomplikowany[8], zawierający ponad 6 tysięcy linii kodu w języku C dlatego nie jest możliwe jego szczegółowe omówienie.

3 Buffer Overflow wiecznie żywy

Mimo wyżej wymienionych zabezpieczeń podatności typu buffer overflow nadal występują i nadal są wykorzystywane. Nie jest to coś, co można zbagatelizować, ponieważ skutki tego typu podatności są katastrofalne, często prowadzi to do zdalnego wykonania kodu, dlatego w bazie CVE zazwyczaj dostają one stopień blisko 9. Jak często występują takie podatności? Można się o tym samemu przekonać wchodząc na stronę *https://www.cvedetails.com/*, wybierając najwyższy stopień zagrożenia(9-10), wciskając **CTRL+F** oraz wpisując "*buffer overflow*". W moim przypadku, tzn. w dniu 13.11.2017 o godzinie 00:57 widać 5 różnych podatności, najstarsza z nich pochodzi z przed trzech miesięcy, a najmłodsza z przed piętnastu dni. Jak to możliwe? Myślę, że powodów jest kilka.
Jednym z nich jest złożoność oprogramowania. Skoro w kilkulinijkowym programie można zostawić furtkę dla atakującego, to ile takich furtek można zostawić w oprogramowaniu zawierającego miliony linii kodu? Z praktycznego punktu widzenia, nie jest możliwe, aby tego typu oprogramowanie nie zawierało dziur, i nie chodzi tutaj o głupie pomyłki lub błędy wynikające z niewiedzy. Nad jądrem Linuxa pracowali wybitni programiści, mimo to, od początku jego istnienia na jaw wychodzą różne, mniej lub bardziej poważne błędy. Można wyobrazić sobie sytuację gdzie dwóch programistów pisze dwie różne funkcje, takie, że jedna zależy od drugiej, oraz każda z nich korzysta z bufora pochodzącego od użytkownika. Co, jeśli pierwszy programista założy, że sprawdzenie wielkości bufora powinno odbywać się wcześniej lub później, więc nie zaimplementuje tego, natomiast drugi programista założy, że wielkość bufora została sprawdzona przez programistę numer jeden? Obaj będą korzystać z potencjalnie niebezpiecznych danych. Zarządzanie pamięcią w dużych projektach nie jest rzeczą prostą, ponieważ tworzy wiele niepotrzebnych zależności, oraz zmusza programistów do skupiania się na niskopoziomowych detalach.
Innym czynnikiem wpływającym na powstawanie błędów jest ludzka ignorancja. Wielu ludzi nie interesuje bezpieczeństwo, no bo co złego może się stać, jeśli nie użyje się funkcji kontrolującej ilość kopiowanych bajtów do bufora? "*Skoro sprawdziłem, że działa to działa, i nie ma się czym przejmować*". Niestety ludzie którzy nigdy niczego umyślnie nie zepsuli często nie doceniają zagrożeń czychających na każdym kroku przy używaniu języków niskopoziomowych. Ciężko coś na to poradzić, no bo chyba nikt nie wierzy w to, że da się nauczyć hordę Hindusów czy innych klepaczy kodu tego jak działa pamięć wirtualna, lub jak wygląda wykonywanie kodu na poziomie asemblera. Z tego powodu odchodzi się od języków typu C, na rzecz języków które nie dają programiście tak dużych możliwości. Nie wszystko jednak można napisać w Javie, w szczególności nie można napisać maszyny wirtualnej Javy, która także nie może być napisana w innym języku wysokopoziomowym ze względów wydajnościowych.
Ogólnie rzecz biorąc, niektórzy próbują się zabezpieczać, inni próbują uciekać, i mimo że widać tego efekty, to zarówno buffer overflow jak i podobne, niskopoziomowe błędy mają się dobrze i dalej robią wiele szkód.

Literatura

[1] Argento Daniele, Boschi Patrizio, Del Basso Luca, *A hardware-enforced BOF protection* http://www.index-of.es/EBooks/NX-bit.pdf

[2] Hector Marco-Gisbert, Ismael Ripoll *Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems* https://www.blackhat.com/docs/asia-16/materials/asia-16-Marco-Gisbert-Exploiting-Linux-And-PaX-ASLRS-Weaknesses-On-32-And-64-Bit-Syste pdf

[3] Tomasz Kwiecień *Praktyczna Inżynieria Wsteczna - Rozdział 3*

[4] Matthew "j00ru"Jurczyk, Gynvael Coldwind *Windows Kernel-mode GS Cookies subverted* http://vexillium.org/dl.php?/Windows_Kernel-mode_GS_Cookies_subverted.pdf

[5] Thurston H.Y. Dang, Petros Maniatis, David Wagner *The Performance Cost of Shadow Stacks and Stack Canaries* https://people.eecs.berkeley.edu/~daw/papers/shadow-asiaccs15.pdf

[6] Intel *Control-flow Enforcement Technology Preview* https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[7] Chris Williams *RIP ROP: Intel's cunning plot to kill stack-hopping exploits at CPU level* https://www.theregister.co.uk/2016/06/10/intel_control_flow_enforcement/

[8] Dhaval Kapil *Heap Exploitation* https://heap-exploitation.dhavalkapil.com/