ARTUR ŻARSKI

Przepełnienie bufora

Stopień trudności

Często tworząc oprogramowanie zastanawiamy się nad jego bezpieczeństwem, ale czy zawsze? Szczególnie wrażliwy na bezpieczeństwo jest sam kod naszej aplikacji, dlatego warto zwrócić uwagę na jedno z największych zagrożeń, jakie czyha na programistę – przepełnienie bufora.

Na łamach hakin9 bardzo często pojawią się tematy związane z bezpieczeństwem kodu i tworzeniem bezpiecznych aplikacji. Każdy z autorów wskazuje wiele miejsc, w których mogą wystąpić problemy mogące spowodować niestabilne działanie systemu, a także stwarzające możliwość włamania do systemu. Jednym z takich zagrożeń i *grzechów* programistycznych jest przepełnienie bufora. Artykuł ma na celu zebranie dostępnej wiedzy na ten temat i przybliżenie tego zagrożenia.

Czym zatem jest przepełnienie bufora i kiedy możemy się z taką sytuacją spotkać? Najkrócej rzecz ujmując, przepełnieniem bufora jest próba zapisania większej ilości elementów niż dopuszcza to rozmiar bufora, z którym pracujemy. Jest to główna przyczyna wielu problemów w ogromnej ilości programów. Programiści bardzo często nie zdają sobie sprawy z tego zagrożenia i nie zwracają uwagi na bufor. Źle deklarowane wartości tablic, nieodpowiednio przekazywane parametry funkcji, nieprawidłowy znak, przepełnienie licznika – to najczęstsze problemy i potencjalne przyczyny umożliwiające włamanie się do naszego programu. Problem ten dotyczy głównie kodu natywnego (C/C++), a także `API` systemu operacyjnego. Przykłady podawane są w `Visual C++ .NET`, wykorzystywany i omawiany będzie kompilator `C++.NET`.

Przyjrzyjmy się następującemu fragmentowi kodu (Listing 1.): W kodzie tym mamy kilka elementów: adres zwrotny, dwie zmienne (jedna typu char, a druga – int), a także wykonujemy funkcję kopiowania ciągu znaków. W przypadku, gdy przekażemy jako parametr prosty wyraz, np. *Test*, wszystko zadziała prawidłowo. Natomiast w przypadku parametru dłuższego niż cztery znaki spowodujemy błąd przepełnienia bufora.

Jednym z najbardziej znanych przypadków wykorzystania przepełnienia bufora jest robak CodeRed. Taki rodzaj przepełnienia bufora jest bardzo często spotykany w aplikacjach opartych o Microsoft Windows, ponieważ bardzo wiele aplikacji używa kodowania znaków w Unicode oraz `ANSI` (co nie jest powodem stwierdzenia, że to wina systemu operacyjnego). `WCHAR` jest znakiem o długości dwóch bajtów, używanym do reprezentowania znaków w Unicode. Liczba znaków, która może być dekodowana przy użyciu `DecodeURLEscapes`, jest określona przez wartość `sizeof wcsAttribute`, która uzyskuje tę wielkość bufora w bajtach, a nie znakach typu `WCHAR`. W rezultacie `sizeof wcsAttribute` ma wartość 400 bajtów, czyli dwa razy więcej niż jest spodziewane. Poniżej fragment kodu, który jest odpowiedzialny za opisaną procedurę (Listing 2.):

W związku z tym może lepiej jest wykonać następującą linię kodu:

```
wwif ( cchAttribute >= sizeof
                    wcsAttribute / sizeof
                    wcsAttrbibute[0])
```

Z ARTYKUŁU DOWIESZ SIĘ

o problemach wynikających z przepełnienia bufora oraz braku jego obsługi,

dodatkowo przedstawione są aspekty jak chronić się przed tego rodzaju problemami.

CO POWINIENEŚ WIEDZIEĆ

znać język C++ oraz podstawy asemblera,

warto również znać podstawy platformy .NET, aby można było skorzystać ze wszystkich omawianych elementów.

bo być może kiedyś nagle wzrośnie rozmiar `WCHAR` albo zmieni się typ – czyli nasz kod w tym przypadku wyglądałby w sposób następujący (Listing 3.):

Innym przykładem jest kolejny dobrze znany robak Blaster, który wykorzystuje mechanizm kopiowania większej ilości bajtów, niż pozwala na to obszar docelowy (Listing 4.):

Architektura x86 i stos

Aby w pełni zrozumieć, jak może dojść do przepełnienia bufora oraz jak działają testy bezpieczeństwa, warto przyjrzeć się budowie stosu. W architekturze x86 stos rośnie w dół, co oznacza, że nowsze dane są zapamiętywane pod adresami niższymi niż elementy wstawione na stos wcześniej. Pojedyncze wywołanie funkcji spowoduje, że jest tworzona nowa ramka stosu. Jego budowa przedstawia się następująco (wg kolejności malejących adresów):

- parametry funkcji,
- adres powrotu funkcji,
- wskaźnik ramki,
- ramka procedur obsługi wyjątków,
- lokalnie zadeklarowane zmienne i bufory,
- rejestry zachowane przez funkcję wywołaną.

Jeśli przyjrzymy się budowie stosu, będziemy mogli stwierdzić, co przepełnienie bufora może nadpisać. Po pierwsze – inne zmienne, które zostały zaalokowane przed buforem, adres powrotu oraz parametry funkcji i wskaźnik ramki. Aby możliwe było przejęcie kontroli nad programem, wystarczy wstawić odpowiednią wartość do ciągu danych, które są ładowane później do rejestru EIP. Najczęściej taką wartością jest adres zwrotny funkcji. Osoby, które wykorzystują tego typu luki działają w taki sposób, że podstawiają własny adres powrotu, który jest *podawany* do programu. Możliwe jest to dzięki prostej funkcji łańcuchowej, kopiującej wartości w stosie z jednego adresu do drugiego. Nie ma w trakcie tego automatycznego sprawdzenia, czy pod adresem docelowym jest wystarczająca ilość miejsca. Haker może wygodnie nadpisać również przyległy adres powrotny.

Listing 1. *Przykładowy kod zagrożony przepełnieniem bufora*

```
void UnSafe (const char* uncheckedData) //adres zwrotny
{
char localVariable[4]; // char[4]
int anotherLocalVariable; //int
strcpy (localVariable, uncheckedData);
}
```

Listing 2. *Fragment kodu wykorzystywany przez robaka o nazwie CodeRed*

```
// cchAttribute jest ilością znaków wprowadzonych przez użytkownika
WCHAR wcsAttribute[200];
if ( cchAttribute >= sizeof wcsAttribute)
    THROW( CException( DB_E_ERRORSINCOMMAND ) );
DecodeURLEscapes( (BYTE *) pszAttribute, cchAttribute, wcsAttribute,webServer.Code
                    Page());
    ...
```

Listing 3. *Poprawiony kod odporny na atak robaka CodeRed*

```
WCHAR wcsAttribute[200];
if ( cchAttribute >= sizeof wcsAttribute / sizeof WCHAR)
    THROW( CException( DB_E_ERRORSINCOMMAND ) );
DecodeURLEscapes( (BYTE *) pszAttribute, cchAttribute, wcsAttribute,webServer.Code
                    Page());
    ...
void DecodeURLEscapes( BYTE * pIn, ULONG & l,
            WCHAR * pOut, ULONG ulCodePage ) {
    WCHAR * p2 = pOut;
    ULONG l2 = l;
    ...
```

Listing 4. *Robak Blaster*

```
size_t cbDest = sizeof(p);

while (--cbDest && *c != '\\')
    *p++ = *c++;
```

Listing 5. *Kod dodany przez użycie przełącznika /GS*

```
sub esp,24h
mov eax,dword ptr [___security_cookie (408040h)]
xor eax,dword ptr [esp+24h]
mov dword ptr [esp+20h],eax
```

Testowanie

Czy istnieje zatem możliwość sprawdzenia i przetestowania kodu na etapie kompilacji – tak, aby zabezpieczyć się przed taką sytuacją? Sprawdzanie, czy w produkcyjnej wersji kodu nie występuje przepełnienie bufora, jest równie ważne. Jednak w takim rozwiązaniu testy muszą mieć dużo mniejszy wpływ na wydajność, niż implementacja testów przeprowadzanych w czasie wykonywania. Dlatego w kompilatorze `Visual C++ .NET` wprowadzono przełącznik `/GS`. Oprócz tego przełącznika przydatne są również dodatkowe biblioteki. Poniższa tabela zawiera ich zestawienie wraz z krótkim opisem. A dokładniej:

- *Run-time Checks (RTC)* – całkowite sprawdzenie sterty danymi niezerowymi, unikając założenia, że sterta jest zawsze pusta. Sprawdzane są granice wszystkich tablic, aby wyłapać nawet jeden bajt przepełniający bufor i znaleźć niezgodne wywołania.
- *PREfast* – narzędzie dla programisty, które pomaga znaleźć błędy trudne do testowania i debugowania przez identyfikację założeń, które mogą być nieprawdziwe. *PREfast* znajduje się w Visual Studio 2005.
- *Source Code Annotation Language (SAL)* – pozwala programiście na

przypisanie wartości do bufora. Kiedy kod jest skompilowany, kompilator zna warunki, które pozwalają na określenie wartości oczekiwanych i aktualnych. Funkcjonalność pozwala programiście na odkrycie błędów, które mogą trudne do znalezienia *ręcznie*.

Application Verifier – to narzędzie pakietu Visual Studio, udostępniające funkcje instrumentacji obecne w systemie operacyjnym Windows. Instrumentacja ta pozwala w czasie działania aplikacji przeprowadzić jej weryfikację w wybranych obszarach, takich, jak przydział pamięci czy użycie sekcji krytycznych i uchwytów. *Application Verifier* wykrywa problemy czasu wykonywania w zakresie alokacji pamięci, wyjścia poza bloki na stercie, użycia pamięci po usunięciu, podwójnego usunięcia i zanieczyszczenia sterty. W zakresie wykorzystania sekcji krytycznych wykrywa działania, które mogą prowadzić do blokowania lub wycieku zasobów. Jeśli chodzi o użycie uchwytów, wykrywa próby powtórnego użycia uchwytów, które już nie są poprawne. Narzędzie to wykorzystuje instrumentację dostępną w systemie operacyjnym, używając jej podczas debugowania wobec danego obrazu wykonywalnego. System operacyjny zmienia warstwę API komunikującą się z aplikacją i przechwytuje

Listing 6. *Funkcja __security_error_handler*

```
void __cdecl __security_error_handler(int code, void *data)
{
 if (user_handler != NULL) {
 __try {
 user_handler(code, data);
 } __except (EXCEPTION_EXECUTE_HANDLER) {}
 } else {
 //...przygotowanie wiadomości outmsg...
 __crtMessageBoxA(
 outmsg,
 "Microsoft Visual C++ Runtime Library",
 MB_OK|MB_ICONHAND|MB_SETFOREGROUND|MB_TASKMODAL);
 }
 _exit(3);
}
```

Listing 7. *Przykład użycia funkcji __set_security_error_handler*

```
void __cdecl sprawdz_blad (int kod_bledu, void * nie_dotyczy)
{
 if (kod_bledu == _SECERR_BUFFER_OVERRUN)
 printf("Wykryto przepełnienie bufora!\n");
}
void main()
{
 _set_security_error_handler(sprawdz_blad);
…
}
```

Listing 8. *Przykład użycia StrCpy*

```
bool obslugaStrCpy(const char* input)
{
    char buf[80];
    if(input == NULL)
    {
        assert(false);
        return false;
    }
    //problem w przypadku braku null na końcu łańcucha
    if(strlen(input) < sizeof(buf))
    {
        //wszystko OK.
        strcpy(buf, input);
    }
    else
    {
        return false;
    }
    //… dalszy kod
    return true;
}
```

Listing 9. *Przykład bezpiecznego Strncpy.*

```
bool obslugaStrncpy(const char* input)
{
    char buf[80];
    if(input == NULL)
    {
        assert(false);
        return false;
    }
    buf[sizeof(buf) - 1] = '\0';
    strncpy(buf, input, sizeof(buf));
    if(buf[sizeof(buf) - 1] != '\0')
    {
        //przepełnienie
        return false;
    }
    //… dalszy kod.
    return true;
}
```

Listing 10. *Przykład użycia funkcji _snprintf.*

```
bool obsluga_snprint(int line, unsigned
                     long err, char*
                     msg)
{
    char buf[132];
    if(msg == NULL)
    {
        assert(false);
        return false;
    }
    if(_snprintf(buf, sizeof(buf)-1,
        "Blad w linii %d = %d - %s\n",
                     line, err, msg)
                     < 0)
    {
        //Przepełnienie - błąd
        return false;
    }
    else
    {
        buf[sizeof(buf)-1] = '\0';
    }
    //.. dalsze działanie programu
    return true;
}
```

wywołania, przekierowując je do warstwy weryfikacji poprawności. W momencie wykrycia nieprawidłowości generowany jest odpowiedni wyjątek, a narzędzie *Application Verifier* dostarcza właściwego kontekstu dla wykrytego błędu.

Przełącznik /GS

Przyjrzyjmy się dokładniej przełącznikowi /GS. Skorzystajmy z informacji zawartej w MSDN (przykład również z MSDN) na jego temat. Przełącznik /GS wstawia pomiędzy bufor a adres powrotu znacznik. Jeśli przepełnienie bufora nadpisze adres powrotu, to nadpisze także znacznik wstawiony pomiędzy adres a bufor. Nowa budowa ramki stosu jest następująca:

- parametry funkcji,
- adres powrotu funkcji,
- wskaźnik ramki,
- znacznik,
- ramka procedur obsługi wyjątków,
- lokalnie zadeklarowane zmienne i bufory,
- rejestry zachowane przez funkcję wywołaną.

Gdy testy bezpieczeństwa są włączone, sposób wykonania funkcji ulega zmianie. Instrukcje, które mają być wykonane zaraz po wywołaniu funkcji, znajdują się w początku funkcji. Weźmy następującą instrukcję:

```
sub esp,20h
```

Instrukcja ta zarezerwuje 32 bajty na wykorzystywane w funkcji zmienne lokalne. Gdy funkcja jest kompilowana z przełącznikiem /GS, zostaną zarezerwowane dodatkowe cztery bajty i dodane zostaną trzy instrukcje (Listing 5.):

Obsługa błędów

Przy przeprowadzaniu testów bezpieczeństwa konieczne jest skorzystanie z biblioteki CRT. Gdy testy wykryją błąd, kontrola przekazywana jest do funkcji

```
__security_error_handler.
```

Funkcja ta wygląda następująco (Listing .6):

Domyślnie aplikacja, w której testy bezpieczeństwa wykryją błąd, wyświetla okno dialogowe informujące o wykryciu przepełnienia bufora. Po zamknięciu okna dialogowego wykonywanie aplikacji zostaje przerwane. Biblioteka CRT daje programistom możliwość zastosowania w przypadku detekcji przepełnienia bufora innej procedury obsługi, która będzie lepiej pasować do aplikacji. Do zainstalowania własnej procedury obsługi błędów można posłużyć się funkcją

```
__set_security_error_handler
```

która przechowuje adres procedury obsługi w zmiennej (Listing 7):

Powyższy fragment kodu spowoduje wypisanie wiadomości na ekranie w momencie wykrycia przepełnienia bufora. Chociaż zdefiniowana powyżej procedura obsługi sama nie kończy wykonywania programu, to po zakończeniu jej wykonywania funkcja

```
__security_error_handler
```

zakończy program, wywołując

`_exit(3)`. Funkcje `__security_error_handler` i `_set_security_error_handler` zdefiniowane są w pliku secfail.c, należącym do plików źródłowych biblioteki CRT.

Zapobieganie przepełnieniom bufora

Aby zapobiegać przepełnieniom bufora, należy zwracać uwagę na funkcje, których używamy w naszych programach.
Najczęstszą przyczyną problemów jest niewłaściwe stosowanie funkcji związanych z obsługą łańcuchów tekstowych. Dlatego też przyjrzymy się im bliżej, aby zobaczyć, jak można zabezpieczyć się przed niepożądanym działaniem.

Funkcja strcpy

Wywołanie funkcji:

```
char *strcpy( char
*strDestination, const char
*strSource );
```

Istnieje bardzo wiele sytuacji, w której działanie funkcji zakończy się z błędem. Najpopularniejsze z nich to między innymi: gdy bufor źródłowy i docelowy są puste, jeżeli bufor źródłowy nie jest zakończony znakiem *null* i największy z problemów – gdy rozmiar ciągu źródłowego jest większy niż bufor docelowy.

Aby bezpiecznie wywołać tę funkcję, należy sprawdzać poszczególne przypadki (Listing 8):

Funkcja strncpy

Wywołanie funkcji:

```
char *strncpy( char *strDest, const
                    char *strSource,
                    size_t count );
```

Jest bezpieczniejsza niż *strcpy*, ale i tutaj nadal mogą pojawiać się problemy, szczególnie w przypadku przekazywania wartości *null* jako ciągu źródłowego lub docelowego. Dodatkowo problemem może być zła wartość licznika. Jedna z różnic pomiędzy tą funkcją a poprzednią to fakt, że jeśli bufor źródłowy nie jest zakończony *null*, to funkcja nie zakończy się z błędem.

Przykład bezpiecznego użycia przedstawia (Listing 9).

Funkcja sprintf

Wywołanie funkcji:

```
int _snprintf( char *buffer, size_t
      count, const char *format [,
      argument] ... );
```

Jest to jedna z bezpieczniejszych funkcji. Jedyna rzecz, na którą należy zwrócić uwagę to bufor docelowy jest zakończony znakiem null.

Przykład użycia funkcji przedstawia (Listing 10).

Funkcja _snprintf

Wywołanie funkcji:

```
int _snprintf( char *buffer,
    size_t count, const
    char *format [, argument] ... );
```

Jest to jedna z bezpieczniejszych funkcji. Jedyna rzecz, na którą należy zwrócić uwagę, to sprawdzenie, czy bufor docelowy jest zakończony znakiem *null*. Przykład użycia funkcji (Listing 9.):

Kod zarządzany

Opisana sytuacja zmienia się w przypadku kodu zarządzanego. Posiada on swego rodzaju automat zabezpieczający przed:

- przepełnieniem bufora,
- wyjściem poza zakres tablicy,
- nieprawidłowym kodem wykonywalnym,
- *zewnętrzną* modyfikacją pliku wykonywalnego.

Są też oczywiście rzeczy, przed którymi nie zabezpiecza kod zarządzany. Jest to między innymi:

- marnowanie pamięci
- błędy na styku wejście użytkownika – kod
- podejrzenie ukrytych informacji.

Podsumowanie

Przepełnienie bufora to poważny problem. Każdy programista powinien mieć świadomość tego rodzaju zagrożenia. Zanim zacznie tworzyć kod, powinien wziąć pod uwagę podobne problemy i dużo wcześniej przemyśleć *architekturę* kodu. Z drugiej strony, programista powinien mieć pomoc ze strony narzędzi programistycznych – tak, aby to one mogły sprawdzić i rozwiązać takie problemy (przynajmniej częściowo). Po analizie tekstu możemy zadać następujące pytanie *jak samemu wyrobić sobie nawyk nie popełniania takich błędów, jak przepełnienie bufora*?. O ile nigdy nie doświadczymy na własnej skórze problemu z włamaniem i nie stracimy przez to jakichś istotnych danych, to pewnie trudno nam będzie o tym pamiętać. A na poważnie – warto zrobić sobie listę wszystkich kroków, które musimy wykonać, i sprawdzić, czy wśród nich jest sprawdzenie możliwości wystąpienia przepełnienia bufora.

Tabela 1. *Przełącznik kompilatora /GS oraz przydatne bibliotekiPrzełącznik kompilatora /GS oraz przydatne biblioteki*

Przełącznik/biblioteka	Opis
/GS: Przepełnienie stosu	Skuteczność 100%
/GS: Przekierowanie wskaźnika, zmiana wartości EBP	Skuteczność niska (chyba, że trafi na adres zwrotny)
RTC: stos niezerowy	Pomoc przy śledzeniu
PREfast: analizator kodu	Pożyteczny; dodatkowe ostrzeżenia
SAL:dodatkowe adnotacje (_deref, _ecount …)	Duża skuteczność; wykrycie „losowych" błędów; pomoc w analizie
Application Verifier	Dobry pomocnik testera
ACGP: rozrzucenie kodu w segmencie	Duża skuteczność; wykrycie „losowych" błędów
Biblioteki standardowe	Funkcje mają wbudowany safeguard + SAL + …

W Sieci

- *http://www.sans.org/rr/whitepapers/securecode/386.php* – Inside the buffer overflow attack: mechanism, method & prevention,
- *http://pl.wikipedia.org/wiki/Przepełnienie_bufora* – Wikipedia,
- *http://support.microsoft.com/kb/325483* – Compiler Security Checks: The /GS Compiler Switch, *Writing Secure Code* , Michael Howard, Microsoft Press 2002.

Artur Żarski
Jest pracownikiem firmy Microsoft. Na co dzień zajmuje sięm.in. Tworzeniem rozwiązań w oparciu o SQL Server w różnych aspektach – bazy relacyjne, usługi integracyjne, usługi analityczne. Jest certyfikowanym administratorem baz danych (MCDBA).
Kontakt z autorem: arturz@microsoft.comh