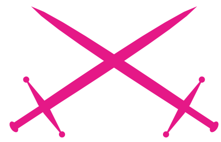


Atak

Niebezpieczne nazwy plików

Michał „Gynvael Coldwind” Składnikiewicz

stopień trudności

Programiści, zarówno początkujący, jak i profesjonaliści, mają tendencje do pokładania przesadnego zaufania w poprawności otrzymywanych z zewnątrz nazw plików. W konsekwencji ich aplikacje mogą stać się wejściem do systemu dla potencjalnego włamywacza.

Niniejszy artykuł ma na celu zaznajomić Czytelnika z najczęstszymi błędami popełnianymi przez programistów podczas obsługi otrzymanych z zewnątrz nazw plików.

Ukrywanie pełnej nazwy pliku

Część aplikacji umożliwiających transfer plików między dwoma użytkownikami (a w szczególności ich odbieranie) przed przystąpieniem do faktycznego transferu danych informuje użytkownika co do nazwy przesyłanego (odbieranego) pliku, aby mógł on zadecydować, czy plik faktycznie powinien zostać odebrany, czy też ma zostać odrzucony (np. z powodów związanych z bezpieczeństwem). Częstym zaniedbaniem w tym miejscu jest dostarczenie przez programistę zbyt wąskiego pola przeznaczonego do wyświetlenia nazwy plików, przez co część nazwy pliku zostanie ukryta przed użytkownikiem. Błąd na pierwszy rzut oka wydaje się być trywialny, jednak jego konsekwencje mogą być poważne. Przykładowo, w roku 2004 błąd tego typu został wykryty przez Bartosza Kwitkowskiego w module przesyłania plików popularnego polskiego komunikatora Gadu-Gadu (*http://securitytracker.com/id?1011037*). Niebezpieczeństwo tego błędu polega na ukryciu przed użytkownikiem prawdziwego rozszerzenia pliku poprzez sformułowanie nazwy w następujący sposób: *obrazek.jpg<DUŻO SPACJI>.exe*. Z uwagi na zbyt wąskie pole wyświetlające nazwę pliku oraz dużą ilość spacji w nazwie, faktyczne rozszerzenie odbieranego pliku nie zmieści się w polu nazwy – a więc nie zostanie wyświetlone (patrz Rysunek 1).

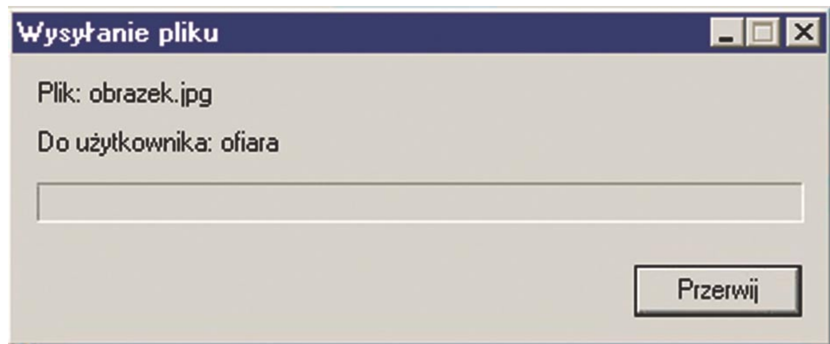
Z artykułu dowiesz się

- o najczęstszych zaniedbaniach w implementacji obsługi nazw plików otrzymanych z zewnątrz,
- jakie są konsekwencje nieodpowiedniej walidacji otrzymanych z zewnątrz danych,
- o trudnościach wynikających z pisania międzyplatformowych aplikacji operujących na plikach.

Co powinieneś wiedzieć

- mieć ogólne pojęcie o popularnych językach programowania oraz najbardziej znanych systemach operacyjnych.

Programista tworzący aplikację powinien zapewnić odpowiednie jej zachowanie w takim wypadku. Przykładowym rozwiązaniem może być użycie pola wieloliniowego lub znacznika informującego użytkownika, że nazwa ma ciąg dalszy – przy czym należy pamiętać, aby użyć dobrze widocznego znacznika. Często używany jest wtedy wielokropek, jednak zdaniem autora trzy piksele tworzące wielokropek mogą zostać przez użytkownika niezauważone.



Rysunek 1. *Okno transferu pliku w Gadu-Gadu 6*

Znak nowej linii

Aplikacje serwerowe udostępniające pliki bardzo często zapisują w logach wszystkie nazwy plików żądanych przez użytkowników, wraz z żądaniami, które zakończyły się niepowodzeniem (np. z powodu nieistniejącego zasobu) – co, samo w sobie, jest oczywiście bardzo dobrym pomysłem i jest niezwykle przydatne przy przeprowadzaniu wszelkiego rodzaju analiz i badań, na przykład powłamaniowych. Często w tym celu używa się logów w formacie tekstowym, głównie z uwagi na prostotę zapisu oraz późniejszego odczytu – czy to przez człowieka, czy przez przeznaczone do tego celu skrypty. Problem pojawia się w momencie, gdy programista zaniedba konwersje niektórych znaków specjalnych pojawiających się w nazwie zażądanego zasobu, takich jak znak nowej linii (należy jednak zauważyć, iż atakujący musi mieć możliwość wstawienia znaku nowej linii, który nie zostanie obsłużony w ramach parsowania pakietu – jest to możliwe w protokołach binarnych oraz w protokołach tekstowych udostępniających znaki ucieczki lub alternatywne kodowanie znaków, np. za pomocą ich kodów ASCII). W takim wypadku atakujący może wstawić do logów dodatkowe, nieprawdziwe wpisy, mogące zmylić osoby przeprowadzające analizę powłamaniową lub – w przypadku, gdy system korzysta z analizatora logów – odciąć dostęp do usługi zaufanym hostom. Jako przykład wyobraźmy sobie aplikację udostępniającą pliki, która tworzy logi w postaci:

```
IP_użytkownika: stan(OK lub ERROR) żądany plik
```

Przykładowy wpis w logach mógłby wyglądać tak:

```
192.168.0.12: OK /var/files/cars.db
```

Proszę zauważyć, że atakujący w takim wypadku może wysłać specjalnie spreparowane żądanie, które spowoduje wstawienie dodatkowych wpisów do logów – wystarczy, że użyje znaku nowej linii w nazwie żądanego zasobu. Przykładowo, atakujący mógłby poprosić o plik */var/files/csrs.db\n123.123.123.123: OK /etc/shadow* (gdzie \n to znak nowej linii). Takie żądanie spowodowałoby pojawienie się w logach następującego wpisu:

```
192.168.0.12: ERROR /var/files/csrs.db
123.123.123.123: OK /etc/shadow
```

W konsekwencji wystraszony administrator (lub analizator logów, którego używa) mógłby odciąć dostęp z hosta 123.123.123.123.

W powyższym przykładzie problem wynikający z wystąpienia nieobsłużonego znaku nowej linii wydaje się być stosunkowo niegroźny, jednak nietrudno wyobrazić sobie groźniejszy przypadek, gdy nazwa ostatniego zażądanego przez użytkownika pliku jest zapisywana do tekstowego pliku konfiguracyjnego. W takim wypadku atakujący dostaje możliwość dopisania dowolnej opcji do pliku konfiguracyjnego – przykładowo takiej, która zwiększy jego uprawnienia. Jako przykład weźmy wyimaginowany serwer plików korzystający z tekstowej bazy danych, w której przechowuje informacje o kontach użytkowników, takie jak login, hasło, poziom dostępu oraz ostatnio zażądany plik. Przykład takiej bazy znajduje się na Listingu 1. Gdy użytkownik zażąda jakiegoś pliku, aplikacja serwerowa zapisuje informacje o żądaniu do bazy danych (pole LASTREQUEST), po czym kontynuuje przetwarzanie. W takim wypadku atakujący ma możliwość wysłania żądania zawierającego znak nowej linii, które – nieodpowiednio obsłużone – spowoduje dodanie opcji ACCESSLEVEL=admin do pliku konfiguracyjnego, co po prze-

Listing 1. *Tekstowa baza użytkowników*

```
[User]
LOGIN=admin
PASS=md5:d10a20c8c3b635d4a6f7dcdae96a15d2
ACCESSLEVEL=admin
LASTREQUEST=/var/files/csrs.db
[User]
LOGIN=hax
PASS=md5:d41d8cd98f00b204e9800998ecf8427e
#ACCESSLEVEL=user (default)
LASTREQUEST=/var/files/rules.txt
```

ładowaniu bazy danych spowoduje nadanie użytkownikowi wyższego poziomu dostępu. Żądanie takie może wyglądać następująco:

```
\nACCESSLEVEL=admin
```

Analogiczny błąd występował w Penultima Online – emulatorze serwera Ultimy Online, dotyczył on jednak nie nazwy pliku, a nazwy użytkownika. Za pomocą znaku nowej linii możliwe było nadanie konkretnej postaci w grze uprawnień administratora. Jak widać, błąd ten dotyczy nie tylko nazw plików.

Niegroźny, nie mniej jednak interesujący, błąd związany z brakiem obsługi znaku nowej linii został znaleziony przez autora w Fileinfo (*http://blog.hispasec.com/lab/advisories/adv_Fileinfo-2_09_multiple_vulnerabilities.txt*), pluginie do Total Commandera autorstwa Francoisa Ganniera. Fileinfo jest pluginem wyświetlającym informacje o strukturze plików wykonywalnych (MZ, PE) oraz obiektowych (COFF). Błąd był umiejscowiony w obsłudze plików PE – autor założył, że nazwa importowanej biblioteki DLL będzie normalną nazwą pliku. Nie przewidział jednak możliwości, że będzie ona zawierała znaki nowej linii, co umożliwia atakującemu dodanie własnych fałszywych informacji o danym pliku PE do raportu generowanego przez Fileinfo (patrz Rysunek 2).

Problem wyboru

Problem wyboru nazwy pliku dotyczy konkretnych przypadków, w których programista dostaje kilka nazw, teoretycznie identycznych (określających jeden plik) i musi dokonać wyboru, której nazwy użyć. Jako że nazwy powinny być – przynajmniej w teorii – identyczne, wybór wydaje się nie mieć znaczenia, ale co w wypadku, gdy atakujący spreparuje podane aplikacji dane i nazwy nie będą takie same?

Tego typu problem pojawia się w przypadku archiwów ZIP, w których nazwa skompresowanego pliku zapisana jest w dwóch miejscach: przed skompresowanymi danymi – w tzw. lokalnym nagłówku pliku (ang. *local file header*) oraz w nagłówku pliku w centralnym katalogu (ang. *central directory*). Programista powinien wybrać jedną nazwę, którą najpierw pokaże użytkownikowi (w celu weryfikacji pliku do rozpakowania lub po prostu wylistowania plików w archiwum), a następnie powinien użyć tej samej nazwy przy tworzeniu rozpakowanego pliku. Okazuje się, że nie zawsze tak się dzieje.

Przykładowo w menadżerze plików Unreal Commander firmy X-Diesel podczas listowania plików archiwum aplikacja pokazuje użytkownikowi nazwy plików pobrane z centralnego katalogu, jednak przy rozpakowywaniu danego pliku aplikacja korzysta z nazwy zawartej w lokalnym nagłówku pliku (*http://blog.hispasec.com/lab/advisories/adv_UnrealCommander_0_92_build_573_Multiple_Vulnerabilities.txt*). W takim wypadku atakujący może ukryć prawdziwą tożsamość pliku nawet przed uważnym użytkownikiem. Należy zauważyć, że zagrożenie wynikające z tego zaniedbania wydaje się być niewielkie, ale w połączeniu z innymi błędami (np. głośnym błędem z kursorami animowanymi .ani – wewnętrzna funkcja systemu Windows była podatna na atak typu *buffer overflow* podczas odczytu danych z pliku z animowanym kursorem, również przy generowaniu podglądu w momencie listowania plików w katalogu za pomocą Windows Explorera lub dowolnego innego menadżera plików wyświetlającego podgląd kursorów animowanych) zagrożenie może wzrosnąć.

Directory traversal

Pozostając przy tematyce archiwów, warto wskazać często występujący błąd, polegający na użyciu nazwy pliku zawartej w archiwum bez sprawdzenia, czy nie zawiera ona znaków specjalnych służących do zapisu ścieżek relatywnych – czyli kropki, slasha oraz backslasha, lub – w wypadku, gdy jest to dopuszczalne – nie poinformowania użytkownika o tym fakcie (np. prezentacja jedynie nazw plików, bez dołączonej do nich ścieżki względnej). Przykładowo w plikach ZIP możliwe jest zapisanie relatywnej ścieżki zawierającej odwołania do katalogu nadrzędnego (czyli /../../../../). W takim wypadku niewystarczająco zabezpieczona aplikacja rozpakowałaby plik do jednego z katalogów nadrzędnych, a nawet w dowolnie wskazane miejsce przez atakującego (w przypadku systemu Windows miejsce ograniczyłoby się do danej partycji). Rozważając konsekwencje takiej możliwości, należy wskazać atak podmiany DLL (ang. *DLL Spoofing*), polegający na utworzeniu w katalogu aplikacji biblioteki DLL o takiej samej nazwie,

Listing 2. *Prosty skrypt tworzący pliki tekstowe*

```
<?php
// Pobierz argumenty
if(!isset($_GET['name']) || !isset($_GET['data']))
  die('Usage: set name to file name, and data to data');
$fname = $_GET['name'];
$fdata = $_GET['data'];

// Usun slashe jesli trzeba
if(get_magic_quotes_gpc())
{
  $fname = stripslashes($fname);
  $fdata = stripslashes($fdata);
}

// Zapisz
file_put_contents($fname . '.txt', $data);
?>
```

jak dowolna nie-systemowa biblioteka DLL, z której korzysta aplikacja, a która normalnie jest umieszczona w katalogu systemowym (na przykład w *C:\Windows\system32*). Spowodowałoby to, że w momencie uruchomienia aplikacji zostałaby zaimportowana biblioteka z katalogu aplikacji, a nie ta umieszczona w katalogu systemowym – skutkując wykonaniem kodu umieszczonego przez atakującego w bibliotece DLL. Należy zauważyć, że w takim wypadku nie jest konieczna podmiana żadnego pliku, a tym bardziej nie jest konieczne *ręczne* uruchomienie podstawionej biblioteki DLL. Atak ten połączony z opisanym możliwym błędem w implementacji rozpakowywania plików ZIP mógłby wyglądać następująco: w pliku ZIP podana byłaby nazwa pliku w postaci:

```
../../../../../../../../Program Files/
mIRC/wsock32.dll
```

W momencie rozpakowania (przy założeniu, że na danym dysku istnieje katalog *Program Files* oraz katalog *mIRC*) plik *wsock32.dll* zostałby rozpakowany do katalogu *\ProgramFiles\mIRC*, a nie – jak użytkownik by oczekiwał – do katalogu wskazanego przez niego. Dzieje się tak dlatego, iż programista założył, że nazwa nie będzie zawierała ścieżki lub nie będzie to ścieżka, która odwołuje się do nadrzędnych katalogów. W konsekwencji użył więc nazwy pliku – bez uprzedniej korekty – w poleceniu tworzącym plik, na przykład *CreateFile*. Wydawałoby się jednak, że aby przeprowadzić taki atak, należy znać dokładny poziom zagłębienia w strukturę katalogów w celu określenia niezbędnej ilości odwołań do katalogów nadrzędnych (czyli dokładną ilość ../). Jak się okazuje, nie jest to konieczne, ponieważ system Windows nie zgłasza błędu, gdy następuje odwołanie do katalogu nadrzędnego w głównym katalogu dysku, przez co możliwe jest użycie nadmiarowej ilości odwołań do nadrzędnego katalogu w celu upewnienia się iż kontekst faktycznie *dotrze* do głównego katalogu dysku. W momencie, gdy niczego nieświadomy użytkownik uruchomi mIRC'a, uruchomiony zostanie również kod zawarty w podstawionej bibliotece *wsock32.dll*.

Tego typu błąd pojawił się na przykład w wymienionym już wcześniej Unreal Commanderze (w momencie pisania artykułu jest on nadal niepoprawiony) oraz w starszych wersjach Total Commandera.

Aplikacja korzystająca z zewnętrznych nazw plików powinna zwracać szczególną uwagę na znaki umożliwiające *podróżowanie* po drzewie katalogów – czyli slashe, kropki oraz backslashe. Dobrym pomysłem jest konwersja tych znaków na inne (na przykład na *underscore* _) lub usunięcie odwołań do katalogów (np. wykasowanie wszystkich odwołań do nadrzędnych katalogów – czyli ciągów ../ oraz /..) – w tym wypadku należy jednak zwrócić szczególną uwagę na to, czy usunięte zostały faktycznie wszystkie próby odwołania do nadrzędnych katalogów. W wypadku, gdy ciąg byłby sprawdzany tylko raz, i ciąg ../ byłby usuwany tylko w pierwszym sprawdzaniu, możliwe jest stworzenie ciągu który po usunięciu pierwszych ../ ponownie sformułuje ciąg odwołujący się do katalogu nadrzędnego. Przykładowo weźmy poniższy ciąg:

```
ala/.../.../..//kot
```

Funkcja usunęłaby kolejne wystąpienia ciągu ../, co spowodowałoby powstanie poniższego ciągu:

```
ala/../kot
```

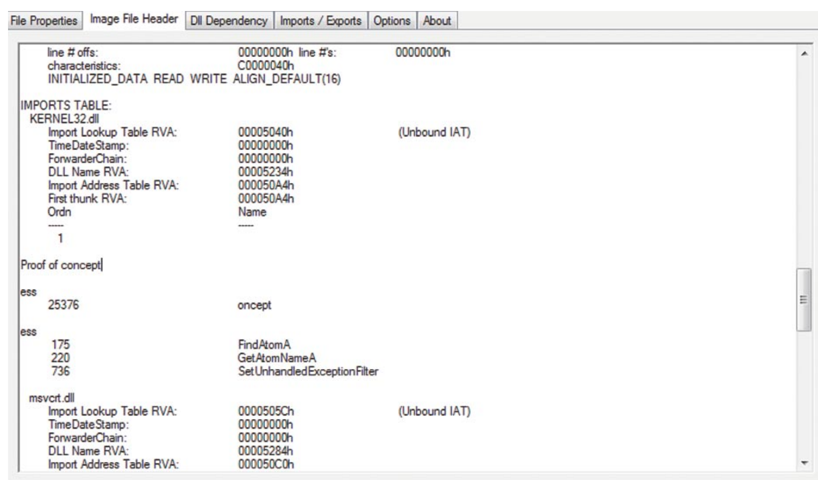
Funkcja usuwająca powinna więc poszukiwać odwołań do katalogów nadrzędnych, dopóki na pewno nie zostaną usunięte wszystkie ciągi ../.

Remote directory traversal

Błąd typu directory traversal można wykorzystać również zdalnie, na przykład w przypadku aplikacji klienckich protokołu FTP, które pokładają zbyt duże zaufanie w poprawności otrzymanych od serwera FTP nazw plików. Tego typu błąd występował w menadżerze plików Total Commander, do wersji 7.01 włącznie (*http://blog.hispasec.com/lab/advisories/adv_TotalCommander_7_01_Remote_Traversal.txt*).

Total Commander po połączeniu z serwerem FTP może wysłać do niego jedną z komend takich jak: USER, PASS, SYST, FEAT, PWD, TYPE, PASV oraz LIST. Odpowiedzią serwera na ostatnią komendę jest lista plików obecnych w danym katalogu. Format listy zależy od serwera FTP, między innymi format UNIX Type: L8 wygląda identycznie, jak wynik polecenia *ls -l*, na przykład:

```
-rwxr-xr-x 1 ftp ftp 4096 Aug 1 02:28
                    plik
```



Rysunek 2. *Okno raportu Fileinfo z wstawionym napisem Proof of concept*

Aplikacja spodziewa się, że ostatnie pole będzie faktycznie nazwą pliku w danym katalogu i nie będzie zawierać ścieżki względnej ani bezwzględnej. Możliwe jest zatem stworzenie serwera FTP, który w liście plików będzie wysyłać plik z backslashami oraz kropkami w nazwie, na przykład:

```
-rwxr-xr-x 1 ftp ftp 4096 Aug 1 02:
28 ..\..\..\..\..\Program Files\mIRC\
wsock32.dll
```

Okazuje się, iż w takim wypadku Total Commander w liście plików w danym katalogu FTP wyświetli jedynie samą nazwę pliku – czyli *wsock32.dll*, bez pełnej ścieżki katalogów. Jednak w momencie, gdy użytkownik zdecyduje się ściągnąć plik na dysk lokalny, Total Commander użyje pełnej znanej sobie „nazwy” pliku, łącznie ze ścieżką katalogów i odwołaniami do katalogów nadrzędnych. W przypadku, gdy użytkownik ściąga z serwera więcej niż jeden plik, na przykład kilka plików lub katalogów wraz z zawartością, Total Commander nie informuje w żaden sposób użytkownika o pliku skopiowanym do innej lokacji. Jak widać, błąd jest dość groźny – w końcu nietrudno przekonać użytkownika, aby ściągnął kilka katalogów z serwera FTP.

Luka ta została poprawiona w wersji 7.02 (trzeba przyznać, iż autor Total Commandera dba o swoją aplikację – łatkę wypuścił kilka dni po otrzymaniu informacji o luce).

Poison NULL byte

Kolejny błąd, już nie tak oczywisty jak poprzednie, dotyczy konwersji metody przechowywania znaków z binarnie bezpiecznej (ang. *binary safe*) na metodę stosującą terminator (na przykład ASCIIZ – ASCII zakończone bajtem zerowym, ang. *ASCII Zero-terminated*). Większość języków skryptowych i opartych na maszynach wirtualnych, takich jak Perl, PHP, Python, Java, itd., przechowuje łańcuchy znaków stosując strukturę składającą się z pola zawierającego liczbę bajtów (znaków) w łańcuchu oraz tablicy bajtów (znaków) – danych łańcucha. Natomiast API popularnych systemów operacyjnych, takich jak Windows czy systemów opartych o jądro Linuksa, korzystają ze stringów będących po prostu tablicą znaków zakończoną terminatorem (w przypadku w/w systemów jest to znak o kodzie binarnym równym zero). W związku z tym, gdy implementacja poleceń języka wymaga odwołania do API systemowych, zachodzi potrzeba konwersji metody zapisu łańcucha znaków. Problem pojawia się w momencie, gdy łańcuch zawierał znak o kodzie równym zero, który jest całkowicie legalnym znakiem dla metody używanej wewnątrz języka, jednak w przypadku łańcuchów z terminatorem zostanie zinterpretowany jako znak końca łańcucha. Korzystając z tej metody atakujący – używając bajtu zerowego – może przedwcześnie zakończyć łańcuch, czyli zapobiec konkatenacji kolejnych łańcuchów (w rzeczywistości będą one „doklejane”, jednak funkcje API będą ignorować wszystko, co następuje po bajcie zerowym). Weźmy pod uwagę przykładowy skrypt znajdujący się na Listingu 2.

Celem programisty piszącego ten skrypt było stworzenie prostego interfejsu pozwalającego łatwo generować pliki tekstowe z rozszerzeniem *.txt*, w dowolnej lokacji na dysku. Atakujący mógłby jednak użyć bajtu zerowego, by utworzyć plik o dowolnym rozszerzeniu, wykonując na przykład zapytanie:

```
http://serwer/skrypt.php?name=evil
.php%00&data=evil _ script _ goes _
here
```

W tym wypadku API systemowe otrzymałoby polecenie utworzenia pliku o nazwie *evil.php\0.txt* (gdzie \0 to bajt zerowy), natomiast – jako że bajt zerowy to w tym wypadku terminator – utworzony zostałby plik *evil.php*, a reszta nazwy zostałaby zignorowana. Warto wspomnieć iż termin *trujący bajt zerowy* (ang. *poison NULL byte*) został pierwszy raz użyty w roku 1998 przez Olafa Kircha na liście Bugtraq. Na błąd typu poison NULL byte podatne było między innymi phpBB (błąd wykryty przez hakera o pseudonimie ShAnKaR w roku 2004).

Shell injection

Niekiedy programiści uważają za stosowne przekazać otrzymaną z zewnątrz nazwę pliku do innego (zewnętrznego) skryptu bądź programu za pomocą polecenia typu system(), popen() lub analogicznych. W takich wypadkach wyjątkowo ważne jest zadbanie o to, by otrzymana nazwa pliku nie zawierała żadnego separatora poleceń – znaku lub ciągu znaków, który umożliwiłby atakującemu uruchomienie dodatkowego polecenia. W przypadku systemów opartych o jądro Linuksa uruchomienie dodatkowego polecenia można uzyskać w jeden z następujących sposobów (Listing 3). Dodatkowo możliwe jest przekierowanie standardowych strumieni za pomocą < (stdin), > (stdout) oraz 2> (stderr). Przykładowy skrypt podatny na tego typu atak wygląda następująco:

```
<?php system(„file” . $_GET['name']); ?>
```

W zamierzeniu programisty skrypt miał wypisywać informację korzystając z polecenia *file* o typie wskazanego pliku. Atakujący mógłby jednak wykonać zapytanie, które spowodowałoby uruchomienie dodatkowego polecenia, na przykład:

```
http://serwer/info.php?name=;cat%20.ht
passwd
```

W takim wypadku wykonane zostałoby polecenie *system* z parametrem *file ;cat .htpasswd*.

Listing 3. *Wykorzystanie separatorów do nieautoryzowanych poleceń*

```
; polecenie
| polecenie
&& polecenie
|| polecenie
`polecenie`
$(polecenie)
```

Programista korzystający z zewnętrznych poleceń powinien zwrócić szczególną uwagę na otrzymywane z zewnątrz dane, w szczególności powinien z nich usunąć wszelkie znaki umożliwiające wykonanie dodatkowych poleceń – takie jak `; | & ` $ ( )` (przykładowo język PHP udostępnia w tym celu funkcje *escapeshellarg* oraz *escapeshellcmd*).

Inne

Oprócz powyższych, często popełnianych zaniedbań, programista powinien oczywiście również uważać na standardowe błędy typu przepełnienie bufora czy *format bug*. Błędy te były opisywane już wielokrotnie, więc zainteresowany Czytelnik nie powinien mieć trudności ze znalezieniem dokładniejszych informacji na ich temat.

Różnice w systemach plików

Na koniec warto wspomnieć o kilku trywialnych różnicach między systemami plików używanych w systemach z rodziny Windows oraz POSIX-owych (czyli stosujących systemy plików rozróżniające wielkość liter), które mogą utrudnić życie programiście tworzącemu międzyplatformową aplikację.

Systemy POSIX-owe w większości przypadków korzystają z systemów plików, które jednoznacznie określają nazwę pliku – czyli (z pominięciem linków symbolicznych) jeden plik ma tylko i wyłącznie jedną nazwę, i wyłącznie po tej nazwie można się do niego odwołać (chodzi o wrażliwość na wielkość liter itp.). Nie jest tak w przypadku systemów z rodziny Windows. Jako przykład weźmy plik *ala.txt* – można się do niego odwołać używając nazwy *ala.txt*, *ALA.TXT*, *aLa.TxT*, lub *ala.TXT* – system ten, nie dość że jest niewrażliwy na wielkość liter, to jeszcze ignoruje wszystkie kropki występujące na końcu nazwy pliku. Dodatkowy wariant odwołania do pliku pojawia się w wypadku, gdy plik ma nazwę, której nie można zapisać w formacie 8.3 (8 znaków nazwy, 3 znaki rozszerzenia) – na przykład *alamakota.txt*. Do pliku odwołać się można wtedy korzystając ze skróconej wersji jego długiej nazwy, czyli na przykład *alamak~1.txt*. Należy zauważyć, że z tego powodu archiwum zawierające pliki *ala.txt*, *aLa.TxT*, oraz *ALA.TXT* zostanie rozpakowane w zupełnie inny sposób w systemie POSIX-owym oraz systemie z rodziny Windows.

Kolejnym problemem mogą być zastrzeżone (już w czasach MS-DOS) nazwy plików, takie jak *con*, *nul* czy *com1 – com9*. Pliki o takich nazwach mogą bez problemu istnieć w systemach POSIX-owych, natomiast w systemach z rodziny Windows takie pliki nie mogą zostać utworzone (przynajmniej w standardowy sposób).

Jeszcze inny problem mogą sprawić alternatywne strumienie danych w systemie plików NTFS, do których odwołuje się za pomocą dwukropka. Przykładowo plik *ala:kot* mógłby być normalnym plikiem w systemie POSIX-owym, natomiast w przypadku systemu korzystającego z NTFS *ala:kot* oznacza strumień *kot* w pliku *ala*.

Ostatnią bardzo widoczną różnicą jest zapis ścieżek katalogów – w przypadku systemów POSIX-owych separatorem katalogów jest slash, natomiast w przypadku systemów z rodziny Windows może to być zarówno slash, jak i backslash. Tak więc plik *ala\kot* mógłby być normalnym plikiem w systemie POSIX-owym, ale w systemie z rodziny Windows oznaczałby on plik *kot* w katalogu *ala*.

Podsumowanie

Mam nadzieję, iż niniejszy artykuł pomoże programistom ustrzec się błędów związanych z obsługą nazw plików otrzymanych z zewnątrz, a testerom wskaże miejsca, w których potencjalny błąd mógłby zaistnieć.

Chciałbym w tym miejscu podziękować Arashi Coldwind za korektę oraz hakerowi o pseudonimie *j00ru* za wspólnie prowadzone badania. ●

O autorze

Michał Składnikiewicz, inżynier informatyki, ma wieloletnie doświadczenie jako programista oraz *reverse engineer.* Obecnie jest koordynatorem działu analiz w międzynarodowej firmie specjalizującej się w bezpieczeństwie komputerowym.

Kontakt z autorem: *gynvael@coldwind.pl*