

Bezpieczeństwo systemów komputerowych

Jak pisać poprawne programy?

Marcin Peczarski

Instytut Informatyki Uniwersytetu Warszawskiego

12 stycznia 2011

Na podstawie: David A. Wheeler „Secure Programming for Linux and
Unix HOWTO”

Program, który powinien być bezpieczny

- ▶ Znajduje się na granicy bezpieczeństwa.
- ▶ Pobiera dane ze źródła, które posiada inne prawa dostępu niż sam program.
- ▶ Przykłady:
 - ▶ przeglądarka danych,
 - ▶ serwer,
 - ▶ aplikacja webowa,
 - ▶ skrypt uruchamiany automatycznie przez przeglądarkę,
 - ▶ program typu setuid, setgid,
 - ▶ program uruchamiany z prawami administratora,
 - ▶ jądro systemu operacyjnego.

Uwagi ogólne

- ▶ Bezpieczeństwo i wygoda obsługi nie są niezależne – bezpieczna konfiguracja wymaga poświęcenia więcej czasu.
- ▶ Występuje konflikt między bezpieczeństwem a poziomem abstrakcji:
 - ▶ ze specyfikacji trudno wywnioskować, czy wysokopoziomowe funkcje biblioteczne są bezpieczne;
 - ▶ można samemu zaimplementować potrzebne komponenty;
 - ▶ można zaufać bibliotece.
- ▶ Należy stosować wiele mechanizmów bezpieczeństwa na każdym poziomie oprogramowania.

Dlaczego powstają niebezpieczne programy? (1)

- ▶ Nie ma zajęć poświęconych wyłącznie pisaniu bezpiecznych programów.
- ▶ Większość zajęć z bezpieczeństwa poświęconych jest kryptografii i protokołom sieciowym.
- ▶ Nie ma wielu podręczników z tego tematu.
- ▶ Prawie nikt nie używa formalnych metod weryfikacji.
- ▶ Wiele kodu pisze się w C, który nie sprzyja powstawaniu bezpiecznych programów, a operacje na napisach w bibliotece C są szczególnie narażone na nadużycia.
- ▶ Wiele bibliotek dla innych języków programowania jest napisanych w C.
- ▶ Programiści nie myślą o wielu użytkownikach.

Dlaczego powstają niebezpieczne programy? (2)

- ▶ Programiści są ludźmi, a ludzie są leniwi. Często wybierają uproszczone rozwiązania, a gdy one „zadziałają”, nigdy nie są poprawiane.
- ▶ Niestety wielu programistów nie jest dobrymi programistami.
- ▶ Wielu programistów nie jest specjalistami od bezpieczeństwa, nie myśli jak potencjalny agresor.
- ▶ Teoretyczne modele bezpieczeństwa są okropne.
- ▶ Typowi użytkownicy mają niewielkie pojęcie o bezpieczeństwie i rzadko troszczą się o bezpieczeństwo.
- ▶ Bezpieczeństwo kosztuje: dodatkowe kodowanie, testy.

Open source a bezpieczeństwo (1)

- ▶ Tekst źródłowy jest wystawiony zarówno dla atakujących, jak i dla obrońców.
- ▶ Otwarcie tekstu źródłowego sprzyja poprawie jego jakości.
- ▶ Otwarte oprogramowanie ma większe szanse być bezpieczniejsze, ale samo otwarcie tekstu źródłowego nie zapewnia bezpieczeństwa.
- ▶ Jeden ekspert, czytając tekst źródłowy, znajdzie więcej niż wielu przypadkowych programistów.
- ▶ Opieranie się na sekrecie, który jest zaszyty w kodzie programu i którego nie można zmienić, jest poważnym błędem.
- ▶ Nie ma skutecznego sposobu ochrony przed odczytaniem „sekretnego” kodu (ang. reverse engineering).
- ▶ Samo stwierdzenie, że kod jest otwarty, może uśpić czujność.
- ▶ Wiele projektów open source tworzonych jest przez rozproszone grupy programistów – trudno kontroluje się jakość tak powstającego oprogramowania.

Open source a bezpieczeństwo (2)

- ▶ Otwarcie tekstu źródłowego nie oznacza, że zostanie on zweryfikowany.
- ▶ Analiza łańcuchów tworzonych dla systemów otwartych dostarcza agresorom wiedzy o podobnych problemach w systemach zamkniętych.
- ▶ Wiele ataków to DoS. Łatwiej jest sprawić, aby program przestał działać niż zmienić jego funkcjonalność – do tego nie trzeba znać tekstu źródłowego.
- ▶ Atakujący może poszukiwać w kodzie programu wzorców potencjalnych luk: zarówno w tekście źródłowym, jak i w kodzie maszynowym (dekompilacja).
- ▶ Utaśnianie luk w oprogramowaniu opóźnia ich usuwanie.
- ▶ Otwarte oprogramowanie jest mniej narażone na umieszczenie w nim konia trojańskiego.

Open source a bezpieczeństwo (3)

- ▶ Otwarcie tekstu źródłowego, który był dotychczas zamknięty, powoduje chwilowe zmniejszenie jego bezpieczeństwa – mogą ujawnić się nieznane dotąd luki. Jednak nikt nie zagwarantuje, że luki te nie były znane wcześniej i wykorzystywane przez wąskie grono agresorów.
- ▶ Istnieją programy skanujące tekst źródłowy w poszukiwaniu potencjalnych luk.
- ▶ Otwarcie tekstu źródłowego wcale nie musi zachęcić do jego przeglądania i poprawiania – kto chce pracować za darmo dla jakiejś firmy?
- ▶ Co z tego, że wielu patrzy (na tekst źródłowy), skoro nikt z nich nie wie, jak powinien wyglądać bezpieczny program?
- ▶ Nawet jeśli luki w programie są poprawiane szybko, dystrybucja poprawek do użytkowników odbywa się bardzo wolno i opornie.
- ▶ Każdy może poprawić błąd w programie open source.

Paranoja jest zaletą

- ▶ Pisanie bezpiecznych programów wymaga specjalnego podejścia do błędów.
- ▶ Każdy program zawiera mnóstwo błędów.
- ▶ Jeśli błędy nie są poważne, użytkownik uczy się używać programu tak, aby je omijać – program nie traci wiele ze swej użyteczności.
- ▶ W programach, które muszą być bezpieczne, poszukuje się nawet drobnych, subtelnych, rzadko objawiających się błędów, aby je wykorzystać do przeprowadzenia ataku.

Sprawdzaj dokładnie wszystkie dane wejściowe

- ▶ Napisy
- ▶ Zakresy wartości liczbowych
- ▶ Adresy email
- ▶ Nazwy plików
- ▶ URI
- ▶ Sekwencje znaków mające specjalne znaczenie wewnątrz programu
- ▶ Nazwa programu, dane z linii poleceń
- ▶ Zmienne środowiskowe, szczególnie przekazywane do procesów potomnych
- ▶ Kodowanie znaków

Filtruj przekazywane dane

- ▶ Wiele programów przekazuje dane otrzymane od jednego użytkownika innemu użytkownikowi.
- ▶ Brak weryfikacji może sprzyjać różnego rodzaju atakom skrośnym, np. XSS – corss-site scripting.

Implementuj limity

- ▶ Rozmiar danych wejściowych
- ▶ Czas na wprowadzenie danych wejściowych
- ▶ Czas na odebranie komunikatu
- ▶ Rozmiar alokowanej dynamicznie pamięci

Przepełnienie bufora nadal groźne

```
char b[BUFFER_SIZE];  
sprintf(b, "%*s",  sizeof(b) - 1, "..."); /* źle */  
sprintf(b, "%.s", sizeof(b) - 1, "..."); /* dobrze */
```

- ▶ Niektóre języki (Perl, Python, Java, Ada)chronią przed tym problemem, ale kosztem pogorszenie wydajności. Chcąc poprawić wydajność, wyłącza się ochronę.
- ▶ GCC ma opcje `-fstack-protector` i `-fstack-protector-all`.
- ▶ W C++ `::std::string` daje dostęp do napisu za pomocą metod `c_str` i `data`.
- ▶ Specjalne wersje bibliotek C implementujące „bezpieczne” wersje funkcji typu `strcpy`.
- ▶ Nie ma lepszego rozwiązania niż staranne pisanie programów.

Stosuj dobre praktyki programistyczne (1)

- ▶ Program powinien być uruchamiany z najmniejszymi potrzebnymi mu uprawnieniami.
- ▶ Jeśli potrzebne jest zwiększenie uprawnień, należy ograniczyć ich przyznanie tylko do minimalnego fragmentu programu i na jak najkrótszy czas.
- ▶ Mechanizmy ochronne powinny być proste, łatwe do weryfikacji (ang. KISS – keep it simple, stupid).
- ▶ Mechanizmy bezpieczeństwa powinny być oparte na otwartych, sprawdzonych algorytmach.
- ▶ Serwer musi być odporny na dowolnie złośliwe zachowanie klienta.
- ▶ Domyślne zachowanie powinno odmawiać dostępu usługi.
- ▶ Idealnie dostęp do obiektu powinien zależeć od spełnienia więcej niż jednego warunku.

Stosuj dobre praktyki programistyczne (2)

- ▶ Należy ograniczać współdzielenie obiektów, np. katalogu /tmp.
- ▶ Interfejs użytkownika powinien być intuicyjny, aby nie prowadził do pomyłek.
- ▶ Jeśli program musi awaryjnie zakończyć działanie, powinien zrobić to możliwie bezpiecznie (ang. fail safe).
- ▶ Należy unikać wyścigów (ang. race condition) – nabiera to szczególnego znaczenia w środowiskach wieloprocessorowych.
- ▶ Dane pochodzące z Internetu należy traktować jako niezaufane.
- ▶ Należy przemyśleć typy danych, szczególnie gdy mamy do czynienia z niejawnymi konwersjami.
- ▶ Należy sprawdzać wartości zwracane przez wywołania funkcji systemowych i bibliotecznych.

Ograniczaj informację zwrotną

- ▶ Niezaufanemu użytkownikowi przekazuj tylko minimum niezbędnych informacji.
- ▶ Nie wyświetlaj nieuwierzytelnionemu użytkownikowi szczegółowych informacji o błędach i wersji programu.
- ▶ Nie umieszczaj komentarzy, jeśli nie są potrzebne, np. w skryptach wysyłanych do przeglądarki.

Praktyki zależne od języka programowania (1)

► C

- Zależnie od implementacji języka `char` może być typem ze znakiem lub bez znaku. Wynik rzutowania `char` na `int`, gdy zmienna typu `char` zawiera wartość nie z przedziału `0...127`, może być dodatni lub ujemny, zależnie od implementacji.
- Używaj `enum` zamiast `int` i `#define`.
- Nie ignoruj ostrzeżeń wypisywanych przez kompilator.
`-Wall -Wpointer-arith -Wstrict-prototypes`
`-pedantic -O2`

► C++

- Używaj kontenerów dostarczanych przez bibliotekę standardową (STL).
- Zamiast `new` i `delete` do zarządzania pamięcią używaj `new` i inteligentnych wskaźników dostarczanych przez bibliotekę Boost.

Praktyki zależne od języka programowania (2)

- ▶ Perl, man page perlsec(1)
- ▶ Python,
<http://pychecker.sourceforge.net>
- ▶ Java,
<http://www.oracle.com/technetwork/java/seccodeguide-139067.html>,
<http://www.dwheeler.com/javasec>
- ▶ Skrypty powłoki,
<http://www.dwheeler.com/secure-programs>, rozdz. 10.4
- ▶ Tcl,
<http://www.dwheeler.com/secure-programs>, rozdz. 10.7
- ▶ PHP,
<http://www.dwheeler.com/secure-programs>, rozdz. 10.8

Liczby pseudolosowe

- ▶ W wielu aplikacjach związanych z bezpieczeństwem potrzebne są liczby losowe, trudne do odgadnięcia.
- ▶ Idealnym rozwiązaniem są sprzętowe generatory losowości, bazujące na zliczaniu cząstek promieniowania, szumie termicznym itp.
- ▶ Większość komputerów nie posiada generatora liczb losowych, należy użyć generatora liczb pseudolosowych.
- ▶ Do celów bezpieczeństwa nie należy używać generatorów dostarczanych przez standardowe biblioteki, np. funkcji `rand`.
- ▶ Dobrym generatorem jest `/dev/random`.
- ▶ Jeśli źródło losowości jest słabe, należy jego wyjście przepuścić przez dobrą funkcję mieszającą, np. SHA.

Wrażliwe dane

- ▶ Natychmiast po użyciu wymazuj wrażliwe dane z pamięci.
- ▶ Staraj się przechowywać, jeśli to możliwe, wrażliwe dane na stronach pamięci, które nie podlegają wymianie.
- ▶ Staraj się zablokować zrzut pamięci na dysk po awarii programu (ang. core dump), funkcja `setrlimit`.
- ▶ Niektóre typy danych nie są reinicjowane przed ponownym użyciem (np. String w Javie), co może spowodować wyciek wrażliwych danych.
- ▶ Pamiętaj, że kompilator, optymalizując kod, może usunąć wywołanie zerujące tablicę, jeśli nie jest ona już nigdzie użyta.

Narzędzia do analizy statycznej programów

- ▶ Flawfinder – C/C++, darmowy
- ▶ RATS (Rough Auditing Tool for Security) – C/C++, darmowy
- ▶ ITS4 – C/C++, darmowy do zastosowań niekomercyjnych
- ▶ Splint (Secure Programming Lint) – C, darmowy
- ▶ Cqual – C/C++, darmowy

Narzędzia do analizy poprawności alokacji i dealokacji pamięci

- ▶ Valgrind, darmowy
- ▶ Electric Fence, darmowy
- ▶ Memwatch
- ▶ YAMD (Yet Another Malloc Debugger)
- ▶ Zmienna środowiskowa `MALLOC_CHECK`

Narzędzia do analizy dynamicznej programów

- ▶ BFBTester (Brute Force Binary Tester)
- ▶ fuzz
- ▶ SPIKE
- ▶ Fenris
- ▶ Nessus
- ▶ Metasploit

Języki projektowane z myślą o bezpieczeństwie

- ▶ Cyclone