



# Języki programowania

# VS

# Security!

Gynvael Coldwind

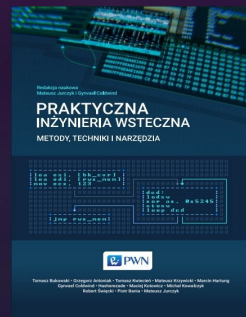
MEGA Sekurak Hacking Party, Kraków, 2019



Jeśli przeglądasz te slajdy, to rzuć okiem na "speakers notes" - jest tam czasem kilka dodatkowych informacji.



# whoami



Tech Lead / Manager @ Google ISE-RIP

*All opinions expressed during this presentation are mine and mine alone, and not those of my barber, my accountant or my employer.*

W poprzednim odcinku

C/C++ VS Security!

Slajdy / Video

<https://gynvael.coldwind.pl/?id=693>

# Jadłospis

- Wstęp i **secure by default**
- **Anty-przykłady** (nie tylko języki programowania)
- Koniec

Podziękowania dla wielu mądrych ludzi z którymi miałem okazję porozmawiać przez lata :)

# Dwa problemy

w bezpieczeństwie aplikacji

# Dwa problemy w bezpieczeństwie aplikacji

Istniejące  
aplikacje

Nowo  
tworzone  
aplikacje

# Nawet najlepsi programiści to tylko ludzie

## 2017-08-14: FAQ: How to find vulnerabilities?

faq

Obligatory FAQ note: Sometimes I get asked questions, e.g. on IRC, via e-mail or during my livestreams. And sometimes I get asked the same question repeatedly. To save myself some time (\*cough\* and be able to give the same answer instead of conflicting ones \*cough\*) I decided to write up selected question and answer pairs in separate blog posts. Please remember that these answers are by no means authoritative - they are limited by my experience, my knowledge and my opinions on things. Do look in the comment section as well - a lot of smart people read my blog and might have a different, and likely better, answer to the same question. If you disagree or just have something to add - by all means, please do comment.

### Q: How does one find vulnerabilities?

A: I'll start by noting that this question is quite high-level - e.g. it doesn't reveal the technology of interest. More importantly, it's not clear whether we're discussing a system vulnerability (i.e. a configuration weakness or a known-but-unpatched bug in an installed service) that one usually looks for during a regular network-wide pentest, or if it's about discovering a previously unknown vulnerability in an application, service, driver / kernel module, operating system, firmware, etc. Given that I'm more into vulnerability research than penetration testing I'll assume it's the latter. And also, the answer will be as high-level as the question, but should give one a general idea.

My personal pet theory is that there are three\* main groups of methods (I'll go in more details below):

\* If I missed anything, please let me know in the comments; as said, it's just a pet theory (or actually a pet hypothesis).

1. Code review (this also includes code that had to be reverse-engineered).
2. Black box (this includes using automated tools like scanners, fuzzers, etc).
3. Documentation research.

All of the above methods have a set of requirements and limitations, and are good at one thing or the other. There is no "best method" that always works - it's more target specific I would say. Usually a combination of the above methods is used during a review of a target anyway.



[Return to dashboard](#) ↑

### Sections

lang: [🇵🇱](#) | [🇬🇧](#)  
RSS: [🇵🇱](#) | [🇬🇧](#)

[About me](#)

[Tools](#)

→ [YT](#) YouTube (EN)

→ [T](#) Twitter

→ [GH](#) GitHub

### Links / Blogs

→ [dragonsector.pl](#)

→ [vexillium.org](#)

**Security/Hacking:**

[j00ru's blog](#)

[lcamtuf's blog](#)

[invisible things \(new\)](#)



# Nawet najlepsi programiści to tylko ludzie

## Q: How does one find vulnerabilities?

...

1. Code review (this also includes code that had to be reverse-engineered).
2. Black box (this includes using automated tools like scanners, fuzzers, etc).
- 3. Documentation research.**

Nawet najlepsi programiści to tylko ludzie

### 3. Documentation research

...

The general idea is to go through the documentation / specification / reference manual and **pinpoint places where the programmer implementing the target might understand something in an incorrect way.**

Nawet najlepsi programiści to tylko ludzie

Programiści popełniają błędy

Nawet najlepsi programiści to tylko ludzie

Programiści popełniają błędy



Dlaczego?

Nawet najlepsi programiści to tylko ludzie

Programiści popełniają błędy



Dlaczego?

- Zapomnieli/nie wiedzieli:
  - że trzeba coś sprawdzić (walidacja).
  - że trzeba coś skonwertować na inną formę (sanitize/escape).
  - że dana operacja może w skrajnych wypadkach mieć efekty uboczne.
- Mieli słabszy dzień / byli zmęczeni / deadline.

Nawet najlepsi programiści to tylko ludzie

Programiści muszą pamiętać /  
wiedzieć o tych detalach.

Tj. **muszą wykonać dodatkową  
pracę**, aby stworzyć bezpieczny kod.

Nawet najlepsi programiści to tylko ludzie

W świecie idealnie kulistych  
kurczaków  
należy wykonać dodatkową pracę,  
aby popełnić błąd.

Czyli **secure by default** w  
oprogramowaniu.

# Kilka losowych efektów

Ekonomia języka:

Krótsza forma jest preferowana względem dłuższej formy.

"I have a request" → "I have an ask"

"I don't know" → "idk"

"That's funny" → "lol xD"

Ekonomia języka w programowaniu:

```
html = secure_escape_mb_string_(user_input)
```

vs

```
html = user_input
```



## Kilka losowych efektów

Pierwsze Prawo Murphiego:

Jeżeli coś może się nie udać - nie uda się na pewno.

W programowaniu:

Jeżeli do wyboru są dwa interfejsy - bezpieczny i niebezpieczny, na pewno zostanie wybrany ten drugi.

(OK, może "na pewno", ale... zasada najślabszego ogniwa)

```
secret = rand()
```

vs

```
secret = csprng()
```

# Kilka losowych efektów

POLA - Principle of Least Astonishment

Zachowanie danego interfejsu nie może być zaskakujące dla użytkownika.

W programowaniu:  
API powinno robić to,  
czego programista się  
spodziewa.

BMP



# Anty-przykłady

(Tj. będę narzekać na losowe rzeczy)

# Operatory porównania

Anty-przykłady

(Tj. będę narzekać na losowe rzeczy)

== w PHP, czyli pseudo-losowe porównanie

== w PHP, czyli pseudo-losowe porównanie

```
"asdf" == "xyz"
```

```
"+1" == "0.1e1"
```

```
"-0e10" == "0"
```

```
"1000" == "0x3e8"
```

```
"123" == "\t\r\n 123"
```

== w PHP, czyli pseudo-losowe porównanie

"asdf" == "xyz"                    false

"+1" == "0.1e1"

"-0e10" == "0"

"1000" == "0x3e8"

"123" == "\t\r\n 123"

== w PHP, czyli pseudo-losowe porównanie

"asdf" == "xyz" false

"+1" == "0.1e1" true

"-0e10" == "0"

"1000" == "0x3e8"

"123" == "\t\r\n 123"



== w PHP, czyli pseudo-losowe porównanie

"asdf" == "xyz" false

"+1" == "0.1e1" true

"-0e10" == "0" true

"1000" == "0x3e8"

"123" == "\t\r\n 123"

== w PHP, czyli pseudo-losowe porównanie

"asdf" == "xyz" false

"+1" == "0.1e1" true

"-0e10" == "0" true

"1000" == "0x3e8" true

"123" == "\t\r\n 123"

== w PHP, czyli pseudo-losowe porównanie

"asdf" == "xyz" false

"+1" == "0.1e1" true

"-0e10" == "0" true

"1000" == "0x3e8" true

"123" == "\t\r\n 123" true

== w PHP i wszystkich ulubiony trick z hashami  
(aka magic hashes)

```
md5( '240610708' ) == md5( 'QNKCDZO' )      true
```

== w PHP i wszystkich ulubiony trick z hashami  
(aka magic hashes)

```
md5( '240610708' ) == md5( 'QNKCDZO' )      true
```

```
md5( '240610708' )
```

```
0e462097431906509019562988736854
```

```
md5( 'QNKCDZO' )
```

```
0e830400451993494058024219903391
```

== w PHP, czyli pseudo-losowe porównanie

"1234512345123451234512345"

==

"1234512345123451234512346"

== w PHP, czyli pseudo-losowe porównanie

```
"1234512345123451234512345"
```

```
==
```

```
false
```

```
"1234512345123451234512346"
```

== w PHP (stare wersje - 5.4.X)

"1234512345123451234512345"

==

true

"1234512345123451234512346"



== w PHP, czyli pseudo-losowe porównanie

0 == "PHP jest najlepsze!"

1337 == "1337 to fajna liczba"

9223372036854775807

==

"9223372036854775808"

== w PHP, czyli pseudo-losowe porównanie

0 == "PHP jest najlepsze!" true

1337 == "1337 to fajna liczba"

9223372036854775807

==

"9223372036854775808"

== w PHP, czyli pseudo-losowe porównanie

0 == "PHP jest najlepsze!" true

1337 == "1337 to fajna liczba" true

9223372036854775807

==

"9223372036854775808"

== w PHP, czyli pseudo-losowe porównanie

0 == "PHP jest najlepsze!" true

1337 == "1337 to fajna liczba" true

9223372036854775807

== true

"9223372036854775808"

== w PHP, czyli pseudo-losowe porównanie

```
$r = new \PDO("sqlite::memory:");  
$r = $r->query(  
    "SELECT sqlite_version()");  
$r = $r->fetch(\PDO::FETCH_LAZY);  
// PDORow Object  
  
$r == "1"
```

== w PHP, czyli pseudo-losowe porównanie

```
$r = new \PDO("sqlite::memory:");  
$r = $r->query(  
    "SELECT sqlite_version()");  
$r = $r->fetch(\PDO::FETCH_LAZY);  
// PDORow Object
```

```
$r == "1" true
```

# Przechodność w PHP

```
"00" == "0"
```

```
"0" == false
```

```
"00" == false
```

# Przechodność w PHP

```
"00" == "0"      true
```

```
"0" == false
```

```
"00" == false
```



# Przechodność w PHP

```
"00" == "0"      true
```

```
"0" == false    true
```

```
"00" == false
```

# Przechodniość w PHP

```
"00" == "0"      true
```

```
"0" == false    true
```

```
"00" == false   false
```

# Przechodność w PHP

```
NULL == false
```

```
false == "0"
```

```
NULL == "0"
```

# Przechodność w PHP

```
NULL == false    true
```

```
false == "0"
```

```
NULL == "0"
```

# Przechodność w PHP

```
NULL == false    true
```

```
false == "0"    true
```

```
NULL == "0"
```

# Przechodność w PHP

`NULL == false`     `true`

`false == "0"`     `true`

`NULL == "0"`     `false`

# Implicit vs explicit cast w PHP ==

```
1234 == "1234 asdf"
```

```
(string)1234 == "1234 asdf"
```

# Implicit vs explicit cast w PHP ==

```
1234 == "1234 asdf"           true
```

```
(string)1234 == "1234 asdf"
```



# Implicit vs explicit cast w PHP ==

```
1234 == "1234 asdf"      true
```

```
(string)1234 == "1234 asdf"  false
```

# Implicit vs explicit cast w PHP ==

```
[] == "Array"
```

```
(string)[] == "Array"
```

# Implicit vs explicit cast w PHP ==

```
[] == "Array"           false
```

```
(string)[] == "Array"
```

# Implicit vs explicit cast w PHP ==

```
[] == "Array"           false
```

```
(string)[] == "Array"   true
```

# Zrozumieć == w PHP

PHP equal operator == reference table : (2019-01-01)

For more information, tests and examples see this post: <https://gynvael.coldwind.pl/?id=492> (note: this table totally ignores proxy objects)

by Gynvael Coldwind (2019-01-01)

op1 == op2 ↓	NULL	BOOL	LONG	DOUBLE	STRING	ARRAY	OBJECT	RESOURCE
NULL	always equal	op1 is FALSE then equal	equal if op1 is 0	equal if op1 evaluation as false in C	converts NULL to STRING, only same binary strings	equal if array is empty	always not equal	equal if op1 numeric value is 0
BOOL	op1 is FALSE then equal	equal if LONG or LONGDOUBLE is 0	op1 is converted to BOOL, like this: op1 is TRUE or FALSE else TRUE then it's compared with op2	op1 is converted to BOOL, like this: op1 is TRUE or FALSE else TRUE then it's compared with op2	op1 is converted to BOOL, like this: op1 is TRUE or FALSE else TRUE then it's compared with op2	op1 is converted to BOOL, like this: op1 is TRUE or FALSE else TRUE then it's compared with op2	if op1 has cast object, then its cast to BOOL, or fails to cast, its checked against op2 on failure to cast, its checked against op2 on success, its compared as LONG vs LONG	op1 is converted to BOOL, like this: if op1 value is 0, then FALSE, else TRUE then it's compared with op2
LONG	equal if not 0	op1 is converted to BOOL, like this: op1 is TRUE or FALSE else TRUE then it's compared with op2	if not op1 not 0 and then equal	op1 is cast to DOUBLE then compared with op2	op1 is converted to LONG or DOUBLE or to LONGDOUBLE if conversion fails then it's compared - either LONG vs LONG or - op2 is cast to DOUBLE, and its compared DOUBLE vs DOUBLE	always not equal	if op1 has cast object, then its cast to LONG, or fails to cast, its checked on success, its compared as LONG vs LONG	op1 numerical value is used as LONG standard LONG vs LONG comparison follows
DOUBLE	equal if not evaluation as false in C	op1 is converted to BOOL, like this: op1 is TRUE or FALSE else TRUE then it's compared with op2	op1 is cast to DOUBLE then compared with op2	equal if op1 == op2	op1 is converted to LONG or DOUBLE or to LONGDOUBLE if conversion fails then it's compared - either DOUBLE vs DOUBLE or - its cast to DOUBLE, and its compared DOUBLE vs DOUBLE	always not equal	if op1 has cast object, then its cast to LONG, or fails to cast, its checked on success, a standard STRING vs STRING comparison is done	op1 numerical value is used as LONG standard LONG vs LONG comparison follows
STRING	converts NULL to STRING, only same binary strings	op2 is converted to BOOL, like this: if op2 is exactly '0' or empty string, its FALSE else its true then it's compared with op1	op2 is converted to LONG or DOUBLE or to LONGDOUBLE if conversion fails then it's compared - either LONG vs LONG or - its cast to DOUBLE, and its compared DOUBLE vs DOUBLE	op2 is converted to LONG or DOUBLE or to LONGDOUBLE if conversion fails then it's compared - either DOUBLE vs DOUBLE or - its cast to DOUBLE, and its compared DOUBLE vs DOUBLE	tries to convert both strings into LONG or DOUBLE using its numeric string conversion rules - compares LONG vs LONG or compares DOUBLE vs DOUBLE - casts LONG to DOUBLE and compares DOUBLE vs DOUBLE (equal on exact match)	always not equal	if op1 has cast object, op1 is cast to LONG() if op1 has cast object, then its cast to string, or fails to cast, its checked on success, a standard STRING vs STRING comparison is done	op2 is converted to LONG or DOUBLE or to LONGDOUBLE if conversion fails, then comparison is done as LONG vs LONG comparison is done
ARRAY	equal if array is empty	op2 is converted to BOOL, like this: op2 is TRUE or FALSE else TRUE then it's compared with op1	always not equal	always not equal	always not equal	to be equal, keys must match to the byte, but values are deeply compared using ALL THIS again!	if op1 has cast object, then its cast to ARRAY, or fails to cast, its checked on success, its compared as ARRAY vs ARRAY	always not equal
OBJECT	always not equal	if op2 has cast object, then its cast to BOOL, or fails to cast, its checked on success, its compared as LONG vs LONG	if op2 has cast object, then its cast to LONG, or fails to cast, its checked on success, its compared as LONG vs LONG	if op2 has cast object, then its cast to DOUBLE, or fails to cast, its checked on success, its compared as DOUBLE vs DOUBLE	if op2 has cast object, then its cast to STRING, or fails to cast, its checked on success, a standard STRING vs STRING comparison is done	LONG has cast object, op1 has to be cast to LONG, or fails to cast, its checked on success, its compared as LONG vs LONG	if objects have the same compare function - if its the same object, equal - else on PHP 5.4+, by default both objects using cast object to LONG (if op1 has cast object, op2 is converted to LONG())	if op2 has cast object, then its cast to RESOURCE, or fails to cast, its checked on success, its compared as LONG vs LONG
RESOURCE	equal if op2 numeric value is 0	op2 is converted to BOOL, like this: op2 is TRUE or FALSE else TRUE then it's compared with op1	op2 numerical value is used as LONG standard LONG vs LONG comparison follows	op2 numerical value is used as DOUBLE standard LONG vs DOUBLE comparison follows	op1 is converted to LONG or DOUBLE or to LONGDOUBLE if conversion fails, then comparison is done as LONG vs LONG comparison is done	always not equal	if op1 has cast object, op1 is cast to LONG(), op2 is cast to LONG(), and a LONG vs LONG comparison is done	op1 and op2 numerical values are compared as LONG standard LONG vs LONG comparison follows

Źródło: <https://gynvael.coldwind.pl/?id=492>

Rejoice! W JavaScript też mamy == i ===!

Rejoice! W JavaScript też mamy == i ===!

9223372036854775807

==

"9223372036854775808"

W JavaScript wszystko\* jest floatem. BASIC called...

\* Pomijam new Uint8Array etc

Rejoice! W JavaScript też mamy == i ===!

9223372036854775807

==

true

"9223372036854775808"

W JavaScript wszystko\* jest floatem. BASIC called...

\* Pomijam new Uint8Array etc



Rejoice! W JavaScript też mamy == i ===!

```
[] == ""
```

```
[1,3,3,7] == "1,3,3,7"
```

```
[1,2] == [1,2]
```

Rejoice! W JavaScript też mamy == i ===!

`[] == ""` true

`[1,3,3,7] == "1,3,3,7"`

`[1,2] == [1,2]`

Rejoice! W JavaScript też mamy == i ===!

`[] == ""`      `true`

`[1,3,3,7] == "1,3,3,7"`      `true`

`[1,2] == [1,2]`

Rejoice! W JavaScript też mamy == i ===!

```
[] == ""           true  
[1,3,3,7] == "1,3,3,7" true  
[1,2] == [1,2]    false
```

Podsumowując "miękkie porównanie" ==

W zasadzie powinno się zawsze używać ===.

Ekonomia języka:

Można napisać jedno = mniej!!11one

# 0 i null w JavaScript

(podziękowania dla Nasty)

```
0 == null
```

```
0 > null
```

```
0 < null
```

```
0 >= null
```

```
0 <= null
```

# 0 i null w JavaScript

(podziękowania dla Nasty)

```
0 == null           false
```

```
0 > null
```

```
0 < null
```

```
0 >= null
```

```
0 <= null
```

# 0 i null w JavaScript

(podziękowania dla Nasty)

`0 == null`                    `false`

`0 > null`                    `false`

`0 < null`

`0 >= null`

`0 <= null`



# 0 i null w JavaScript

(podziękowania dla Nasty)

`0 == null`                    `false`

`0 > null`                    `false`

`0 < null`                    `false`

`0 >= null`

`0 <= null`

# 0 i null w JavaScript

(podziękowania dla Nasty)

`0 == null`                    `false`

`0 > null`                    `false`

`0 < null`                    `false`

`0 >= null`                  `true`

`0 <= null`

# 0 i null w JavaScript

(podziękowania dla Nasty)

<code>0 == null</code>	<code>false</code>
<code>0 &gt; null</code>	<code>false</code>
<code>0 &lt; null</code>	<code>false</code>
<code>0 &gt;= null</code>	<code>true</code>
<code>0 &lt;= null</code>	<code>true</code>

# Więcej o == i === w JS

Ten wątek na Stack Overflow:

<https://stackoverflow.com/questions/359494/which-equals-operator-vs-should-be-used-in-javascript-comparisons>

I koniecznie:

<https://www.slideshare.net/kkotowicz/biting-into-the-forbidden-fruit-lessons-from-trusting-javascript-crypto>

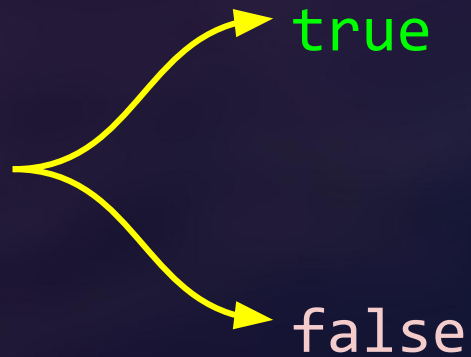
(Krzysztof Kotowicz)

Dobrze, że == w C i C++ są idealne

"ala ma kota" == "ala ma kota"

Dobrze, że == w C i C++ są idealne

Porównanie pointerów

"ala ma kota" == "ala ma kota"  true  
false

Dobrze, że == w C i C++ są idealne

w C trzeba użyć strcmp

```
strcmp("asdf", "asdf")
```

Dobrze, że == w C i C++ są idealne

w C trzeba użyć strcmp

```
strcmp("asdf", "asdf")           0 (false)
```



Dobrze, że == w C i C++ są idealne

w C trzeba użyć strcmp

(ma to niby swoje uzasadnienie, ale...)

```
!strcmp("asdf", "asdf")           1 (true)
```

Dobrze, że == w C i C++ są idealne

w "nowym" C++ można tak  
(ekonomia języka)

"ala ma kota"s == "ala ma kota"s true

Dobrze, że == w C i C++ są idealne

true / false?

```
int main(void) {  
    char x = 0xff;  
  
    if(x == 0xff)  
        puts("true");  
    else  
        puts("false");  
}
```

Źródło: [https://gynvael.coldwind.pl/n/char\\_comparison\\_riddle](https://gynvael.coldwind.pl/n/char_comparison_riddle)

Dobrze, że == w C i C++ są idealne

true / false

```
int main(void) {
```

```
    char x = 0xff;
```

```
    if(x == 0xff)
```

```
        puts("true");
```

```
    else
```

```
        puts("false");
```

```
}
```

char → signed char

x = (int)255;

x → (char)-1

(int)(char)x == (int)255

-1 == 255

Źródło: [https://gynvael.coldwind.pl/n/char\\_comparison\\_riddle](https://gynvael.coldwind.pl/n/char_comparison_riddle)

# Dobrze, że == w C i C++ są idealne

true / false

```
int main(void) {  
    char x = 0xff;
```

char → unsigned char

```
    if(x == 0xff)  
        puts("true");  
    else  
        puts("false");  
}
```

x = (int)255;

x → (char)255

(int)(char)x == (int)255

255 == 255

Źródło: [https://gynvael.coldwind.pl/n/char\\_comparison\\_riddle](https://gynvael.coldwind.pl/n/char_comparison_riddle)

# Protokoły sieciowe

Anty-przykłady

(Tj. będę narzekać na losowe rzeczy)

# TCP i sumy kontrolne



# TCP i sumy kontrolne

**TCP provides reliable, ordered, and error-checked delivery of a stream of octets (bytes) between applications running on hosts communicating via an IP network.**

Źródło: [https://en.wikipedia.org/wiki/Transmission\\_Control\\_Protocol](https://en.wikipedia.org/wiki/Transmission_Control_Protocol)



# TCP i sumy kontrolne

Losowa historia (4p):

Byłem sobie w LV i próbowałem ściągnąć aplikację.

# TCP i sumy kontrolne

Losowa historia (4p):

Byłem sobie w LV i próbowałem ściągnąć aplikację.

- 300 MB
- 1 godzina ściągnięcia
- bardzo duży retransmission rate

# TCP i sumy kontrolne

Losowa historia (4p):

Byłem sobie w LV i próbowałem ściągnąć aplikację.

- 300 MB
- 1 godzina ściągnięcia
- bardzo duży retransmission rate

Dla porównania na domowym serwerze:

- 12 sekund ściągnięcia

# TCP i sumy kontrolne

Losowa historia (4p):

Sprawdzenie podpisu cyfrowego po ściągnięciu aplikacji:

**FAIL**

O nie! ktoś mnie próbuje shaczyć!

# TCP i sumy kontrolne

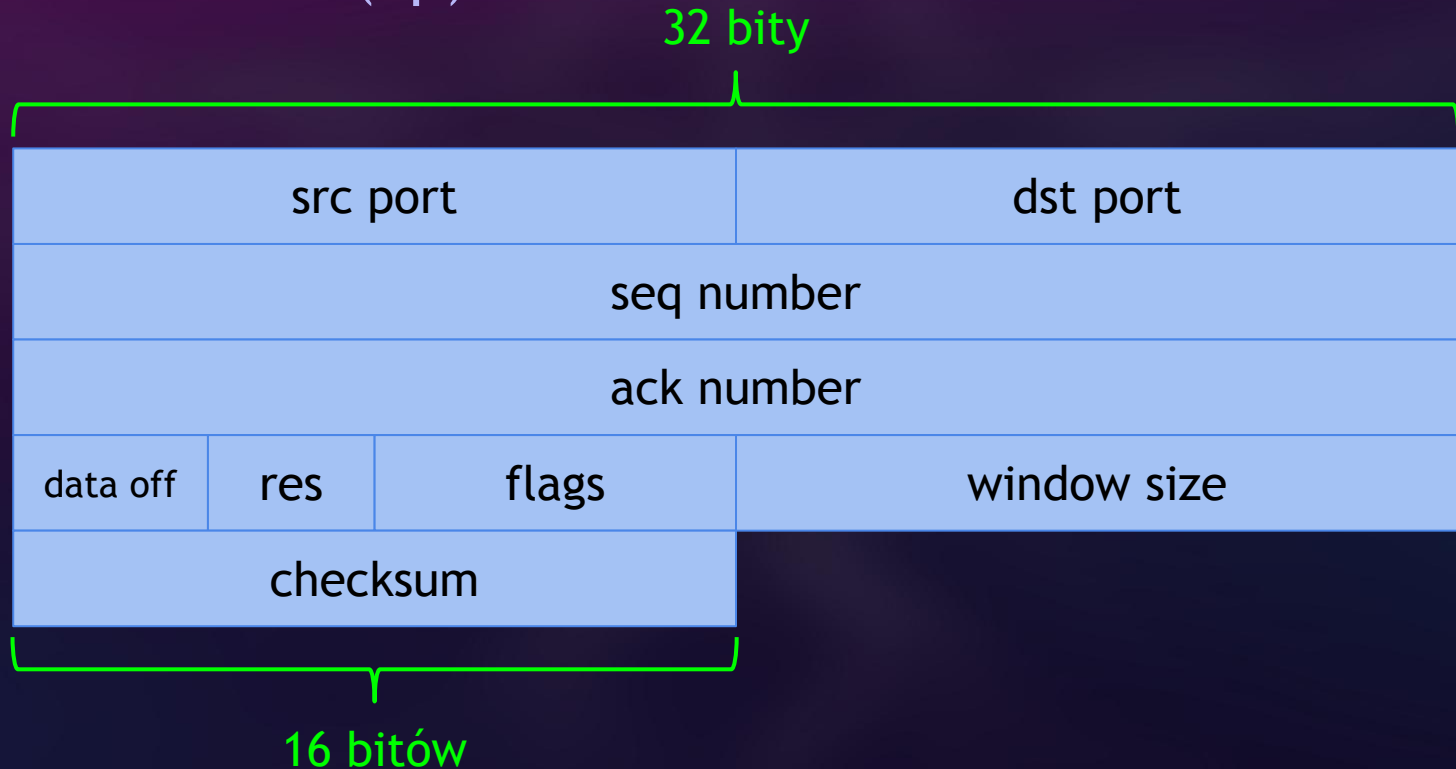
Losowa historia (4p):

Po ściągnięciu na zaufanym serwerze i porównaniu\*  
bloków po 4KB:

- 400 bloków uszkodzonych
- wszystkie w sekcji .rsrc  
("ktoś Ci chciał ikonkę podmienić :)")

# TCP i sumy kontrolne

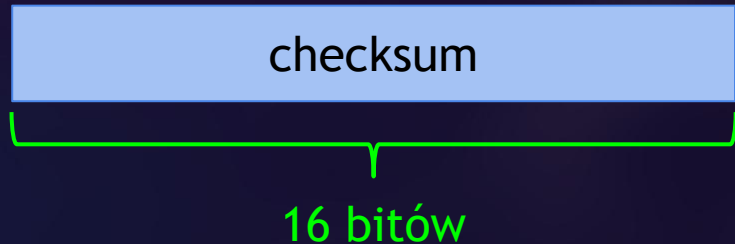
Losowa historia (4p):



# TCP i sumy kontrolne

Losowa historia (4p):

$P = 1 / 65536 = 0.00001526$  ← prawdopodobieństwo kolizji



# TCP i sumy kontrolne

Losowa historia (4p):

$P = 1 / 65536 = 0.00001526$  ← prawdopodobieństwo kolizji

Przy 300MB danych, ściąganych przez 1h przy dużych zakłóceniach, nie ma wyjścia - Kolizje Muszą Się Zdarzyć.



# TCP i sumy kontrolne

Losowa historia (4p):

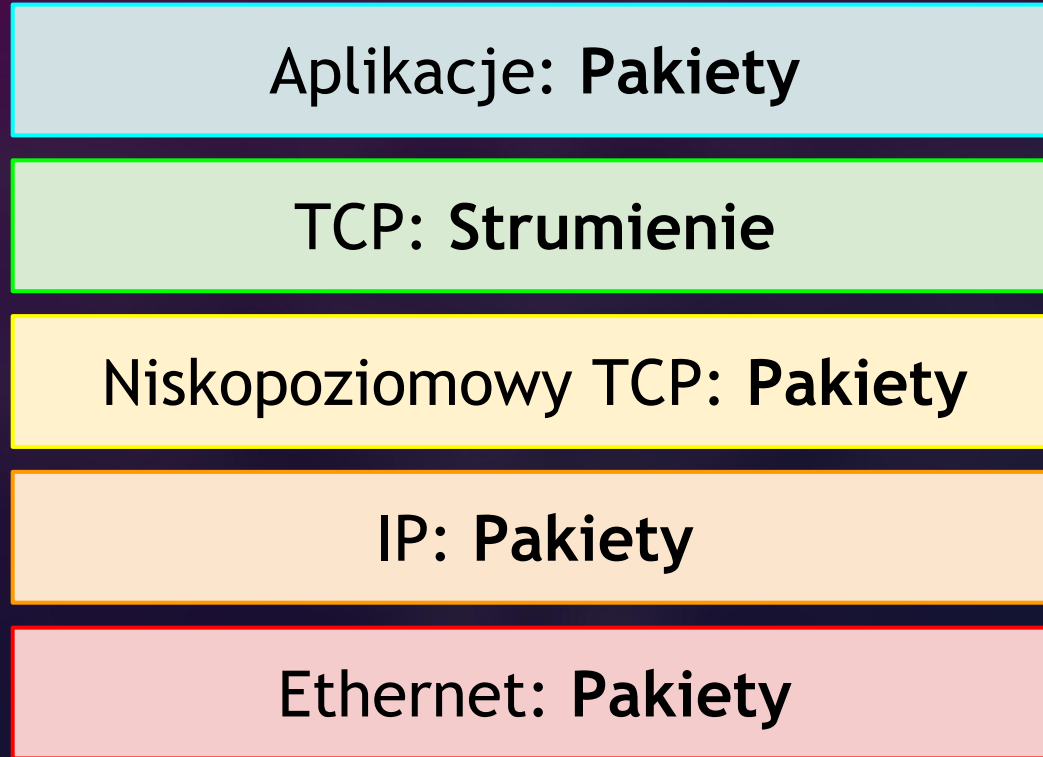
Ostatecznie zastąpiłem uszkodzone bloki poprawnymi.  
(tym razem przesłane po SSH; min. 96-bitów MAC)



A gdyby tak TCP miał jakiś odpowiednio większy checksum...  
Np. 256-bitów? (2019 vs 1974)

# TCP i strumień bajtów

# TCP i strumień bajtów (pseudo net stack)



**BUT  
WHY?!**

# TCP i strumień bajtów

**BUT  
WHY?!**

## TCP: Strumienie

### Wolny internet

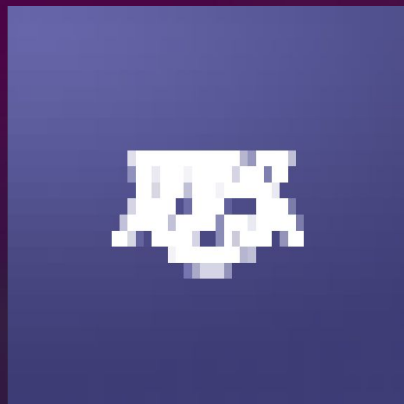
- Progressive JPEG
- Adam7 w PNG

### Play-when-loading

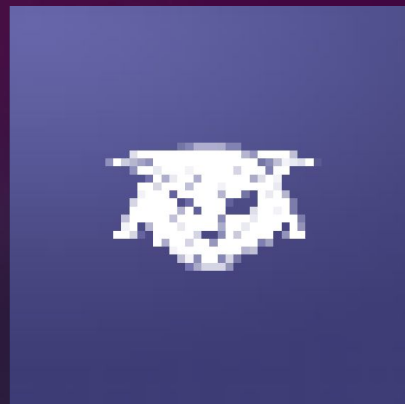
- WAV może? (bo MP3 nie)



Krok 1



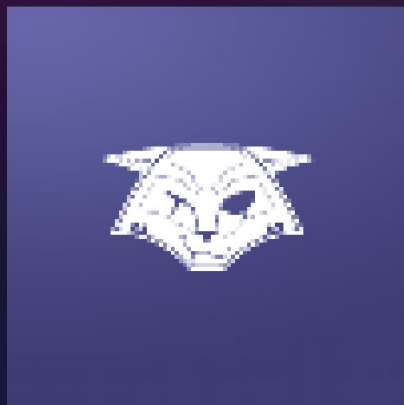
Krok 2



Krok 3



Krok 4



Krok 5



Krok 6



Krok 7

# Pakiety

Co jest wysyłane:

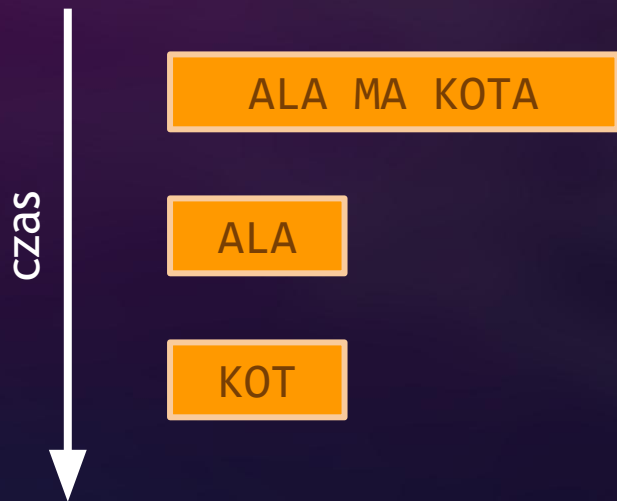


Co aplikacja otrzymuje:

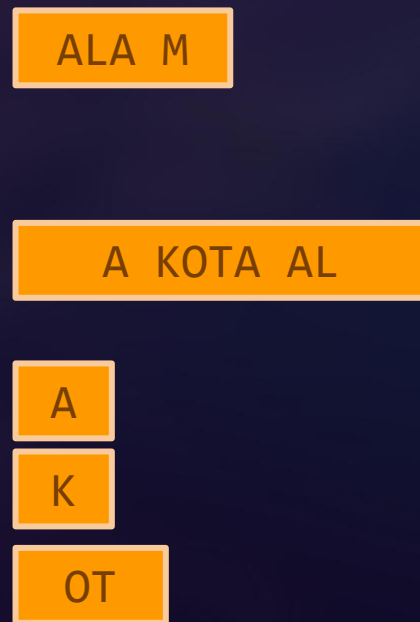


# Strumienie

Co jest wysyłane:



Co aplikacja otrzymuje:



# Strumienie (jeśli mamy pecha podczas testów)

Co jest wysyłane:



Co aplikacja otrzymuje:





# Strumienie (albo)

Co jest wysyłane:



Co aplikacja otrzymuje:

ALA MA KOTA ALA KOT

# Strumienie a fuzzing

Oprócz fuzzowania danych,  
można też fuzzować  
timing / rozbijanie / łączenie "pakietów" TCP

# TCP i strumienie

Zdecydowana większość aplikacji i tak implementuje pakiety na TCP:

- Albo w formie <długość> <dane>
- Albo delimiterów (np. Linia 1\r\nLinia 2\r\n)

Przydałby się taki pakietowy TCP...

(Teraz i tak bez znaczenia, bo **wszyscy poprawnie używają SSL**)

# Ucieczka z Nowego Jorku (kinda)

Anty-przykłady

(Tj. będę narzekać na losowe rzeczy)

# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

# Dopasuj funkcje do kontekstu (PHP)

addslashes

SQL

```
SELECT ... WHERE  
x='input'
```

HTML

```
<tag>input</tag>
```

HTML

```
<tag attrib="input">
```

HTML CSS

```
<style>  
p { color: input }  
</style>
```

HTML JS

```
<script>  
const X='input';  
</script>
```

JSON

```
{ "name": "input" }
```

# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

addslashes

# Dopasuj funkcje do kontekstu (PHP)

`mysql_real_escape_string`

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

*addslashes*



# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

*mysql\_real\_escape\_string*

*addslashes*

# Dopasuj funkcje do kontekstu (PHP)

htmlspecialchars

mysql\_real\_escape\_string

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

addslashes

# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

*htmlentities*

*mysql\_real\_escape\_string*

*addslashes*

# Dopasuj funkcje do kontekstu (PHP)

htmlspecialchars

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

# Dopasuj funkcje do kontekstu (PHP)

`strip_tags`

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

# Dopasuj funkcje do kontekstu (PHP)

addslashes

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>

HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }



# Dopasuj funkcje do kontekstu (PHP)

SQL  
SELECT ... WHERE  
x='input'

HTML  
<tag>input</tag>


HTML  
<tag attrib="input">

HTML CSS  
<style>  
p { color: input }  
</style>

HTML JS  
<script>  
const X='input';  
</script>

JSON  
{ "name": "input" }

# Dopasuj funkcje do kontekstu (PHP)



<p>SQL</p> <pre>SELECT * FROM WHERE x='input'</pre> <p><i>htmlentities</i></p>	<p>HTML</p> <pre>&lt;tag attr="escape_stri"&gt; &lt;/tag&gt;</pre> <p><i>mysql_real_escape_stri</i></p>	<p>HTML</p> <pre>&lt;tag attr="input"&gt;</pre> <p><i>htmlspecialchars</i></p>
<p>HTML CSS</p> <pre>&lt;style&gt; p { color: input } &lt;/style&gt;</pre> <p><i>htmlspecialchars</i></p>	<p>HTML JS</p> <pre>&lt;script&gt; const X='input'; &lt;/script&gt;</pre> <p><i>addslashes</i></p>	<p>JSON</p> <pre>{ "name": "input" }</pre> <p><i>addslashes</i></p>

# Dopasuj funkcje do kontekstu (PHP)

Czy WSZYSCY programiści ZAWSZE to ogarną?

```
SQL  
SELECT WHERE  
htmlentities  
x='input'
```

```
mysql_real_escape_string  
HTML  
<tag>input</tag>
```

```
HTML tags  
<tag attrib="input">
```

```
HTML CSS  
<style>  
color: input }  
</style>
```

```
HTML JS  
<script>  
document.X='input';  
</script>
```

```
JSON  
{ "name": "input" }
```

# Prepared statements vs SQLI (PHP)

Na szczęście mamy prepared statements!

# Prepared statements vs SQLI (PHP)

Na szczęście mamy prepared statements!

**Uwaga: SQL Injection**

```
mysql_query("SELECT * FROM asdf WHERE x='$input'")
```

# Prepared statements vs SQLI (PHP)

Na szczęście mamy prepared statements!

## Uwaga: SQL Injection

```
mysql_query("SELECT * FROM asdf WHERE x='$input'")
```

## Prepared statements!

```
$conn->prepare(  
    "SELECT * FROM asdf WHERE x=:input")  
->bindParam(':input', $input);  
->execute();
```

# Prepared statements vs SQLI (PHP)

Na szczęście mamy prepared statements!

## Uwaga: SQL Injection

```
mysql_query("SELECT * FROM asdf WHERE x='$input'")
```

## Prepared statements!

```
$conn->prepare(  
    "SELECT * FROM asdf WHERE x='$input'")  
->execute();
```

A może by tak...

Idealnie by było, żeby NIE DAŁO SIĘ konkatelować  
kwerendy SQL ze "zwykłymi" stringami.



# A może by tak...

Idealnie by było, żeby NIE DAŁO SIĘ konkatelować kwerendy SQL ze "zwykłymi" stringami.

```
$stmt->select(EVERYTHING, "asdf")  
  ->where(  
    eq->('x', $input)  
  )  
  ->execute();
```

# A może by tak...

Idealnie by było, żeby NIE DAŁO SIĘ konkatelować kwerendy SQL ze "zwykłymi" stringami.

```
$stmt->select(EVERYTHING, "asdf")  
  ->where(  
    eq->('x', $input)  
  )  
->execute();
```

Za długie + tracimy zalety SQL.

# A może by tak...

W idealnym świecie język wie co jest "zwykłym" stringiem,  
a co np. kwerendą SQL.

```
$input = "...";  
$stmt = "SELECT * FROM asdf WHERE x=$input";  
sql_exec($stmt);
```

## A może by tak...

W idealnym świecie język wie co jest "zwykłym" stringiem,  
a co np. kwerendą SQL.

`$input` jest `UtrustedString`

```
$input = "...";  
$stmt = "SELECT * FROM asdf WHERE x=$input";  
sql_exec($stmt);
```



`sql_exec` wymaga `SQLString`

## A może by tak...

W idealnym świecie język wie co jest "zwykłym" stringiem,  
a co np. kwerendą SQL.

`$input` jest `UtrustedString`

```
$input = "...";  
$stmt = "SELECT * FROM asdf WHERE x=$input";  
sql_exec($stmt);
```

literal też musi być  
`SQLString`

`sql_exec` wymaga `SQLString`

# A może by tak...

W idealnym świecie język wie co jest "zwykłym" stringiem,  
a co np. kwerendą SQL.

\$input jest UtrustedString

```
$input = "...";  
$stmt = "SELECT * FROM asdf WHERE x=$input";  
sql_exec($stmt);
```

odpowiedni context-aware  
operator z SQLString jest  
wywołany

sql\_exec wymaga SQLString

literal też musi być  
SQLString

A może by tak...

Kapkę trudne techniczne...  
Nie może być błędów.

```
$input = "...";  
$stmt = "SELECT * FROM asdf WHERE x=$input";  
sql_exec($stmt); // Wymaga SQLString  
echo($stmt); // Wymaga HTMLString
```

What now?

Podobna sprawa z szablonami HTML



# Podobna sprawa z szablonami HTML

W HTMLu już pogodziliśmy się z szablonami  
(Starąłem się dobrać nazwy funkcji zgodnie z metodyką PHP)

```
<html>
  <title>[[[title|escape_tag]]]</title>
  <meta charset='[[[charset|attibSecure]]]'>
  <script>
    const TITLE='[[[title|safejs]]]';
    ...
```

# Podobna sprawa z szablonami HTML

Można lepiej!

```
[[context_aware_auto_escape=on]]
```

```
<html>
```

```
  <title>[[[title]]]</title>
```

```
  <meta charset='[[[charset]]]'>
```

```
  <script>
```

```
    const TITLE='[[[title]]]';
```

```
  ...
```

# Podobna sprawa z szablonami HTML

Idealnie - **secure by default!**

(ale pewnie zadziała tylko dla szablonu HTML)

```
<html>
  <title>[[[title]]]</title>
  <meta charset='[[[charset]]'>
  <script>
    const TITLE='[[[title]]';
  ...
```

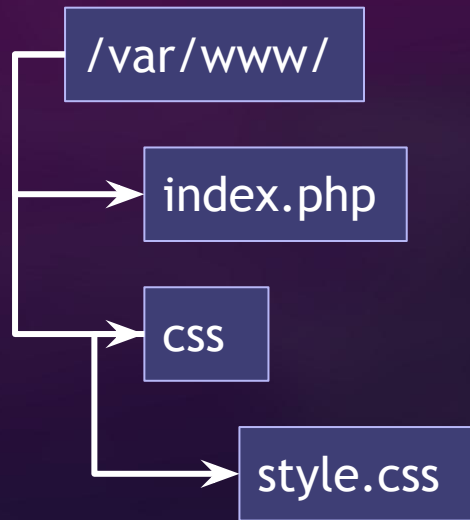
# PHP i mapowanie FS

Anty-przykłady

(Tj. będę narzekać na losowe rzeczy)

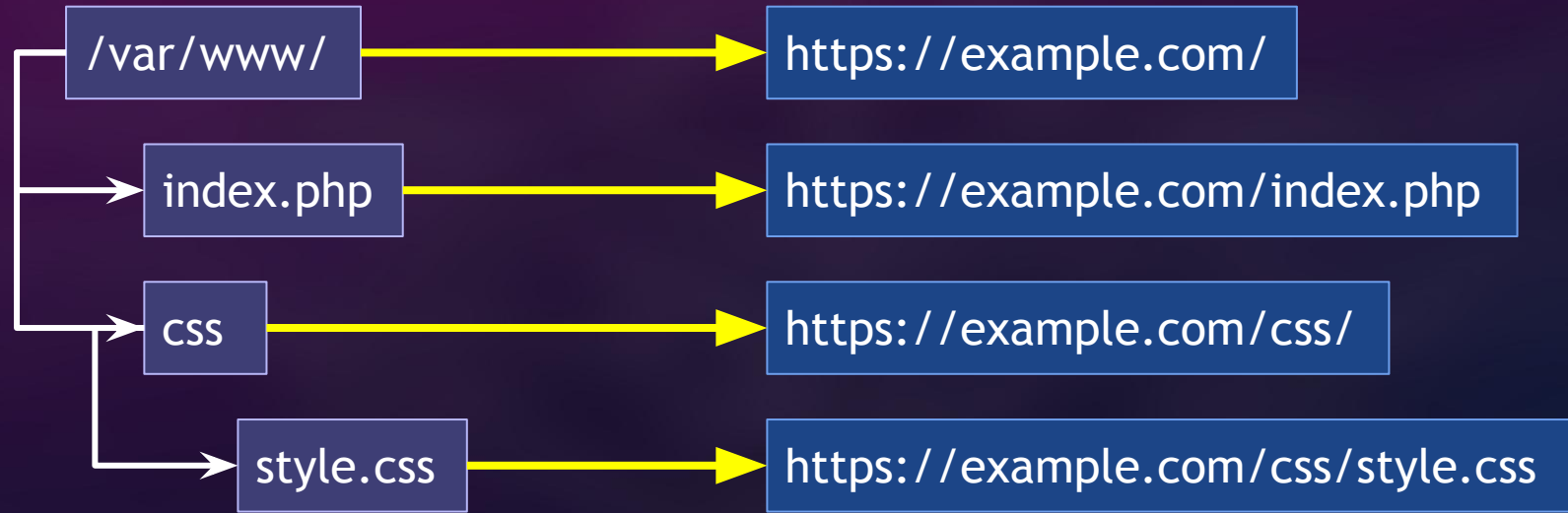
# PHP i mapowanie FS do HTTP

*Bardzo wygodny  
ficzer!*



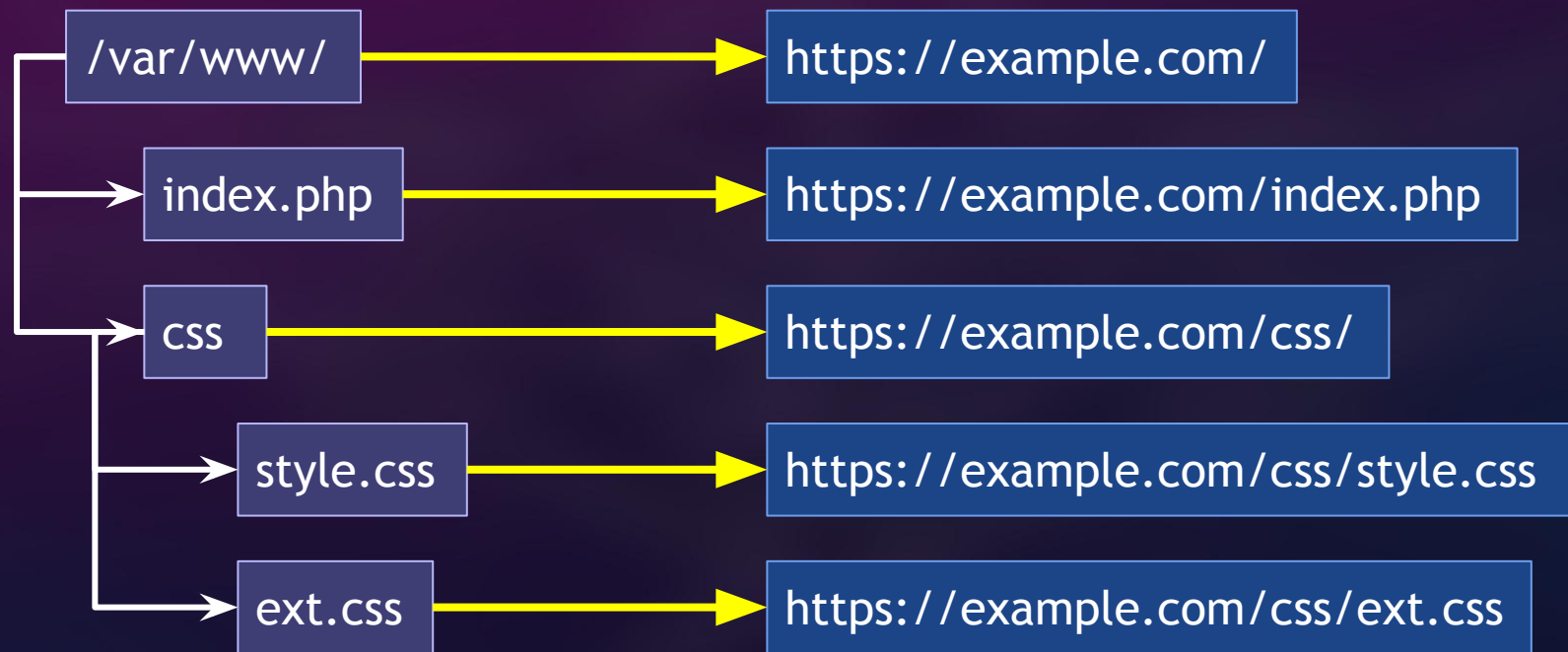
# PHP i mapowanie FS do HTTP

*Bardzo wygodny  
ficzer!*



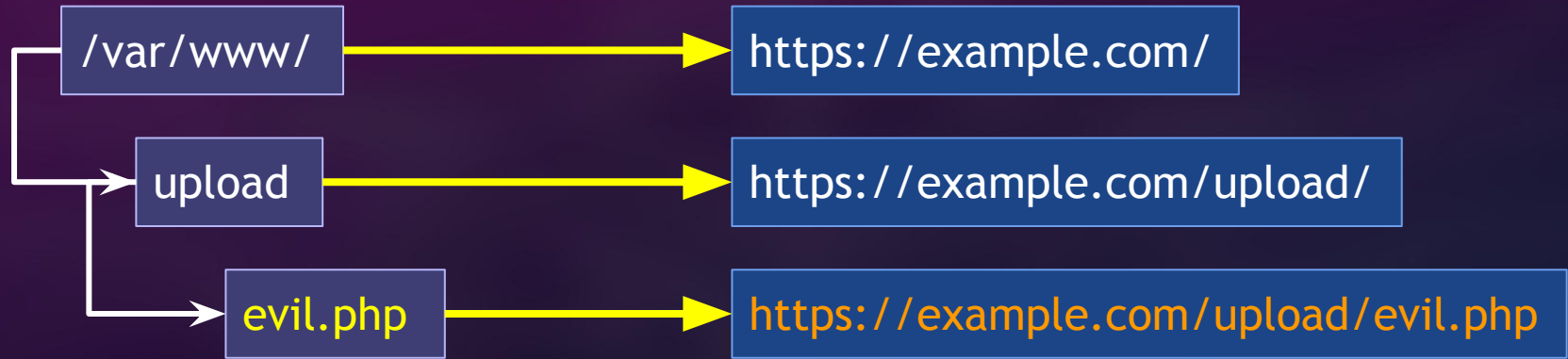
# PHP i mapowanie FS do HTTP

*Bardzo wygodny  
ficzer!*



# PHP i mapowanie FS do HTTP

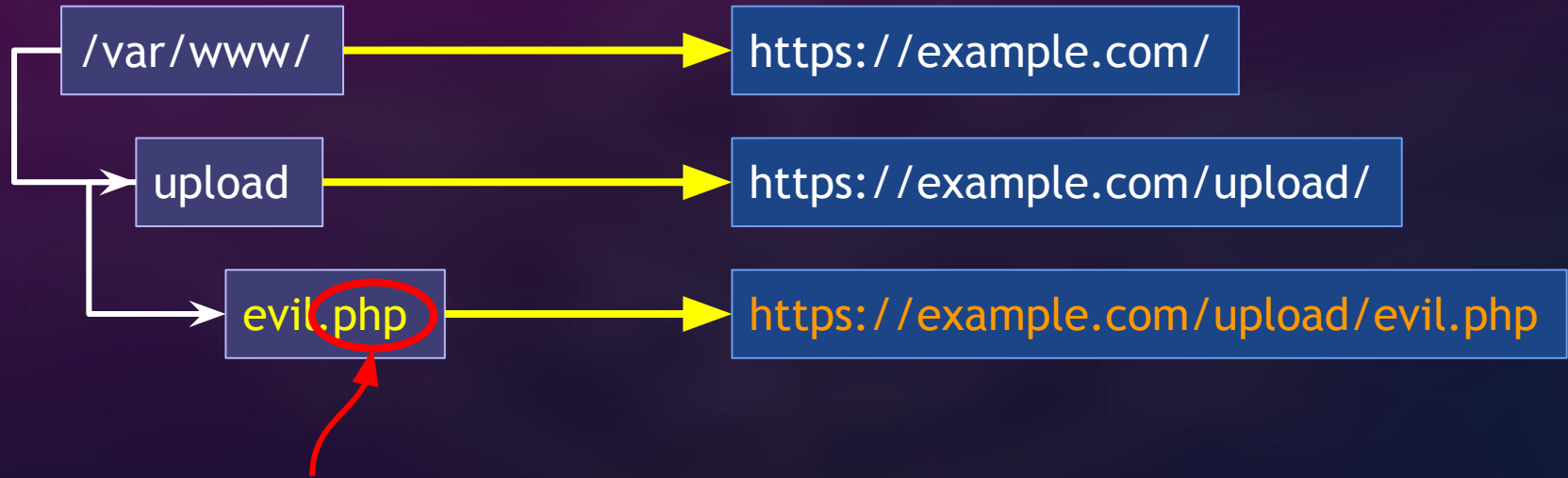
*Bardzo wygodny  
ficzer!*





# PHP i mapowanie FS do HTTP

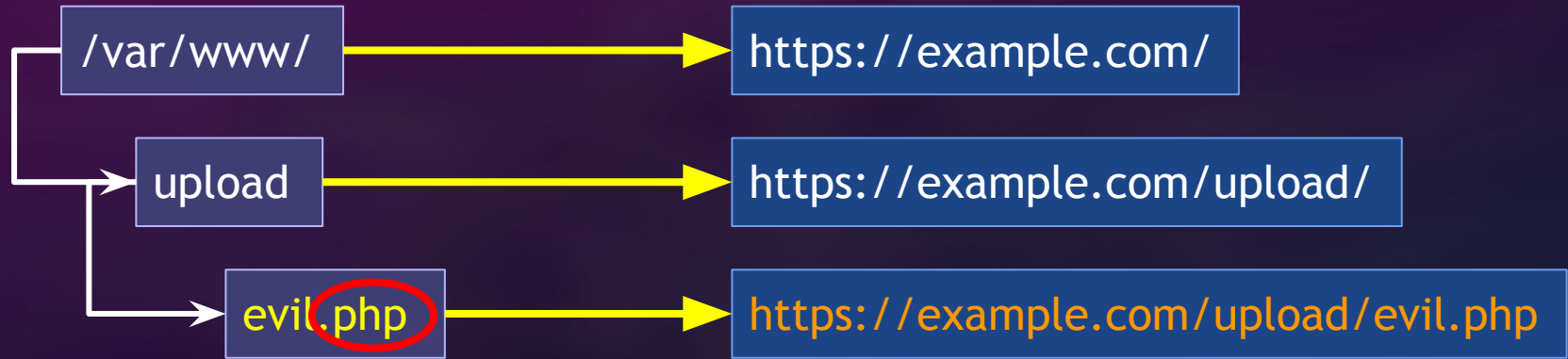
Bardzo wygodny  
ficzer!



Można wyfiltrować .php!

# PHP i mapowanie FS do HTTP

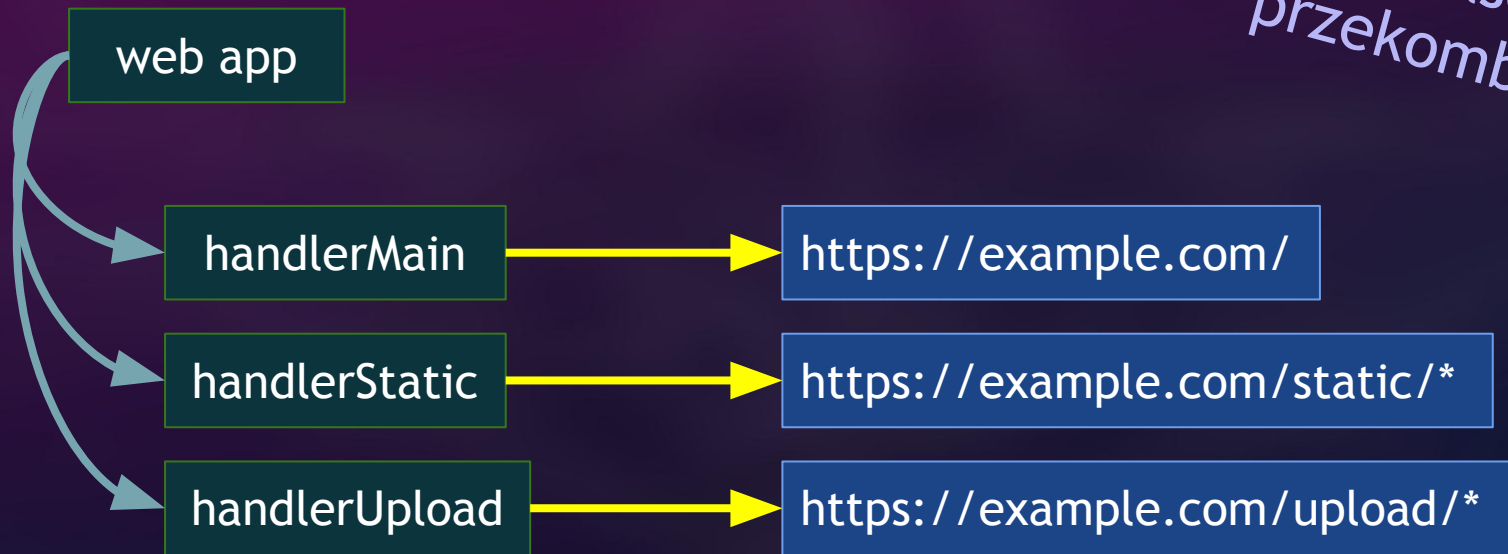
Bardzo wygodny  
ficzer!



```
(Arch, all versions) php$: .php  
(CentOS with PHP 5.4) php$: .php  
(Fedora with PHP 7.0, 7.1) php$: .php  
(Fedora with PHP 7.2) (php|phar)$: .php, .phar  
(Ubuntu/Debian with PHP 7.1, 7.2) ph(ar|p|tml): .phar, .php, .phtml  
(Ubuntu/Debian with PHP 7.0) ph(p[3457]?|t|tml): .php, .php3, .php4, .php5, .php7, .pht, .phtml  
(Debian with PHP 5.6) ph(p[345]?|t|tml): .php, .php3, .php4, .php5, .pht, .phtml  
(Ubuntu with PHP 5.5) ph(p[345]?|t|tml): .php, .php3, .php4, .php5, .pht, .phtml  
(Debian with PHP 5.4) ph(p[345]?|t|tml): .php, .php3, .php4, .php5, .pht, .phtml  
(Ubuntu with PHP 5.3) ph(p3?|tml): .php, .php3, .phtml  
(Ubuntu with PHP 5.): .phtm (source)
```

.htaccess ;)

# Frameworki / inne języki umieją bez FS



*też czasem można przekombinować ;)*

(btw, zuploadowane pliki i tak powinny być serwowane z innej domeny)

# Inne

Anty-przykłady

(Tj. będę narzekać na losowe rzeczy)

# Losowa lista rzeczy

- Nazwy funkcji vs spójność
- Kolejność argumentów (ln, fgets vs fwrite)
- Wyniki zwracane zgodnie z przewidywaniami
- Możliwość obsługi błędów (atoi)
- Składnia spójna między językami (default value, ^)
- Wbudowane thread-safety (na ile się da)

# Podsumowanie

Podsumowanie

Podsumowanie

Podsumowanie

## Podsumowanie

Jeśli uczestniczą Państwo w tworzeniu języków, bibliotek, frameworków, etc, zachęcam do zwrócenia szczególnej uwagi na ich design vs koncept secure by default.

(a bug hunterom / testerom / QA przypominam o możliwości poszukania w dokumentacji tych problematycznych miejsc)

Warto też rzucić okiem na...

<https://wiki.theory.org/index.php/YourLanguageSucks>

## YourLanguageSucks

### Contents [hide]

- 1 JavaScript sucks because
  - 1.1 Poor Design
  - 1.2 Type System
  - 1.3 Bad Features
  - 1.4 Missing Features
  - 1.5 DOM
- 2 Lua sucks because
- 3 PHP sucks because
  - 3.1 Fixed in currently supported versions
- 4 Perl 5 sucks because
- 5 Python sucks because
  - 5.1 Fixed in Python 3
- 6 Ruby sucks because
- 7 Flex/ActionScript sucks because
- 8 Scripting languages suck because
- 9 C sucks because
- 10 C++ sucks because
- 11 .NET sucks because



W sumie można by też poprawić tutoriale...

Dzisiaj nauczymy się tworzyć strony internetowe przy użyciu PHP!

Zacznijmy od najważniejszego:

```
<?php  
include "page/" . $_GET['page'];
```

# KONIEC



Czy są jakieś proste pytania?

E-mail: [gynvael@coldwind.pl](mailto:gynvael@coldwind.pl) Twitter: [@gynvael](https://twitter.com/@gynvael) YT: [GynvaelColdwind](https://www.youtube.com/channel/UCGynvaelColdwind) / [GynvaelEN](https://www.youtube.com/channel/UCGynvaelEN)  
Blog: <http://gynvael.coldwind.pl/> (Soon also: <http://gynvael.live>)

# Dodatkowe slajdy

Niedokończone lub niewykorzystane

Rozdzielczość w funkcjach mierzących czas

`clock()` w `glibc/Linux`

`CLOCKS_PER_SEC` → 1 000 000

Czyli jednostki to mikrosekundy (1e-6 sec)!

# Rozdzielczość w funkcjach mierzących czas

## clock() w glibc/Linux

CLOCKS\_PER\_SEC  $\rightarrow$  1 000 000

Czyli jednostki to mikrosekundy (1e-6 sec)!

```
a = clock();  
usleep(1 /*microsec*/);  
b = clock();  
b - a == ?
```