

Magazyn programistów i liderów zespołów IT

programista

9/2017 (64)

październik/listopad

Cena 23,90 zł (w tym VAT 5%)

WYDANIE NR:

SPOSOBY NA CRASH

Wokselowy silnik gry

SonarQube

Modernizacja kodu C++

Lemon i re2c

Podpisy kryptograficzne

ISSN 2084-9400

09



9 772084 940701



CYBERCOM
GROUP



Join our team of Makers!
career.cybercom.com

Change
tomorrow
with us

www.cybercom.pl



CYBERCOM
GROUP

Cybercom Poland Sp. z o.o.

ul. Hrubieszowska 2, 01-209 Warszawa / ul. Składowa 35, 90-127 Łódź / ul. Unii Lubelskiej 4c, 85-059 Bydgoszcz

Magiczne 64

Oddajemy w ręce czytelników jubileuszowy 64 numer naszego pisma. Staraliśmy się wypełnić jego łamy wyjątkowo interesującymi artykułami. Ten okrągły pod względem informatycznym numer jest dla nas szczególnie, stąd na jego łamach publikujemy kolejny tekst z cyklu „Wspomnienia zgreda” dotyczący dawnych lat.

Jego treść może w starszych czytelnikach wzbudzić nostalgiczny nastrój, którego uzupełnieniem może być obszerny artykuł dotyczący silnika 3D opracowanego w postaci teoretycznej dawno temu, mającego implementację w kilku znanych współczesnych grach. Mowa o silniku wokselowym, o tworzeniu którego przeczytacie w artykule autorstwa Michała Krawczyka.

Mawia się, że człowiek, który nie robi błędów, zwykle nie robi nic. O 64 dużych sposobach na wywołanie błędu i spowodowanie w ten sposób crasha możecie przeczytać w artykule ołdakowym „64 sposoby na crash”.

Jesteśmy przekonani, że w tym numerze znajdzie się dużo więcej ciekawych treści, a my będziemy gościli w Waszych domach co najmniej do kolejnego okrągłego 128 numeru.

Czego i Wam, i nam pozostaje życzyć i zaprosić do, mam nadzieję, przyjemnej lektury.

Mariusz „mariush” Witkowski

PS. W poprzednim numerze błędnie przypisaliśmy adres emailowy autorowi artykułu „3 najlepsze Python IDE dla Data Scientistów”. Za pomyłkę przepraszamy.

BIBLIOTEKI I NARZĘDZIA

Filtr Blooma – niepozorny strażnik wydajności 4

Damian Stygar

Lemon oraz re2c – duet do analizy gramatyczno-leksykalnej 10

Marek Sawerwain

JĘZYKI PROGRAMOWANIA

Modernizacja kodu w C++ 18

Michał Papierski

PROGRAMOWANIE GIER

Przygoda z wokselowym silnikiem gry. Część 1 24

Michał Krawczyk

INŻYNIERIA OPROGRAMOWANIA

SonarQube – kod mojszy niż twój 40

Marcin Wachulski

64 sposoby na crash 52

Paweł „KrzaQ” Zakrzewski

ZARZĄDZANIE PROJEKTAMI

Niekończąca się praca w toku, czyli zmora Product Ownera 64

Michał Cichy

BEZPIECZEŃSTWO

Praktyczna kryptografia: Hashowanie, podpisy cyfrowe i wyprowadzanie klucza 66

Michał Leszczyński, Jarosław Jedynak

STREFA CTF

Ekoparty CTF 2017 – OldPC 72

Jarosław Górny

WSPOMNIENIA ZGREDA

Jak ciekawość nas napędzała 76

Mariusz „mariush” Witkowski

PLANETA IT

Kurs angielskiego dla programistów. Lekcja 9 80

Łukasz Piwko

Czy programista musi być pasjonatem? 82

Maciej Anisierowicz

KLUB DOBREJ KSIĄŻKI

Game Engine Black Book. Wolfenstein 3D 84

Rafał Kocisz

Zamów prenumeratę magazynu Programista przez formularz na stronie:

<http://programistamag.pl/typy-prenumeraty/>

lub zrealizuj ją na podstawie faktury Pro-forma. W sprawie faktur Pro-forma prosimy kontaktować się z nami drogą mailową:

redakcja@programistamag.pl.

Prenumerata realizowana jest także przez RUCH S.A.

Zamówienia można składać bezpośrednio na stronie: www.prenumerata.ruch.com.pl

Pytania prosimy kierować na adres e-mail: prenumerata@ruch.com.pl

lub kontaktując się telefonicznie z numerem:

801 800 803 lub **22 717 59 59**, godz. 7:00 – 18:00 (koszt połączenia wg taryfy operatora).

Magazyn Programista wydawany jest przez Dom Wydawniczy Anna Adamczyk

Wydawca/Redaktor naczelny: Anna Adamczyk (annaadamczyk@programistamag.pl).

Redaktor prowadzący: Michał Leszczyński (mleszczyński@programistamag.pl).

Korekta: Tomasz Łopuszański. **Kierownik produkcji:** Havok. **DTP:** Havok.

Dział reklamy: reklama@programistamag.pl, tel. +48 663 220 102, tel. +48 604 312 716.

Prenumerata: prenumerata@programistamag.pl.

Współpraca: Michał Bartyzel, Mariusz Sierackiewicz, Dawid Kaliszewski, Marek Sawerwain, Łukasz Mazur, Łukasz Łopuszański, Jacek Matulewski, Sławomir Sobótka, Dawid Borycki, Gynael Coldwind, Bartosz Chrabski, Rafał Kocisz, Michał Sajdak, Michał Bentkowski, Mariusz „mariush” Witkowski, Paweł „KrzaQ” Zakrzewski.

Adres wydawcy: Dereniowa 4/47, 02-776 Warszawa.

Druk: <http://www.moduss.waw.pl/>, Nakład: 4500 egz.

Nota prawna

Redakcja zastrzega sobie prawo do skrótów i opracowań tekstów oraz do zmiany planów wydawniczych, tj. zmian w zapowiadanych tematach artykułów i terminach publikacji, a także nakładzie i objętości czasopisma.

O ile nie zaznaczono inaczej, wszelkie prawa do materiałów i znaków towarowych/firmowych zamieszczanych na łamach magazynu Programista są zastrzeżone. Kopiowanie i rozpowszechnianie ich bez zezwolenia jest zabronione.

Redakcja magazynu Programista nie ponosi odpowiedzialności za szkody bezpośrednie i pośrednie, jak również za inne straty i wydatki poniesione w związku z wykorzystaniem informacji prezentowanych na łamach magazynu Programista.

Filtr Blooma – niepozorny strażnik wydajności

Nikt nie lubi czekać. Miło jest móc dostawać informacje, które nas interesują tak szybko jak to tylko możliwe. W tym artykule opowiem, co sprawia, że filtr Blooma jest tak użyteczny oraz dlaczego firmy między innymi takie jak Google czy Facebook zdecydowały się na jego użycie. Mimo swoich niewątpliwych zalet ma również kaprysy, zdarza się mu czasem oszukiwać. Postaramy się pokazać, jak te błędy można zniwelować.

CZYM JEST FILTR BLOOMA

Filtr Blooma jest probabilistyczną strukturą danych reprezentowaną za pomocą n-bitowej tablicy pozwalającą określić należność elementu do zbioru, gdzie dodawane elementy są reprezentowane przez wartości uzyskane w wyniku działań k niezależnych funkcji haszujących. Kluczowe jest tutaj słowo „probabilistyczna”. Oznacza to, że możemy dostać z jakimś prawdopodobieństwem niezerowym wynik fałszywie pozytywny, natomiast nigdy nie dostaniemy wyniku fałszywie negatywnego. Przedstawiając to jeszcze prościej – możemy dostać odpowiedź, że dany element należy do zadanego zbioru, jednak jest to niepoprawna odpowiedź. Dobrą wiadomością jest to, że jesteśmy w stanie wpływać na wskaźnik prawdopodobieństwa, z jakim dostaniemy nieprawidłowy wynik. Istotną cechą tej struktury jest to, że jest ona wydajna pamięciowo. Sprawia to, że filtry mogą być przetrzymywane bezpośrednio w pamięci.

DOZWOLONE OPERACJE

W wersji Blooma jesteśmy w stanie dodać element do filtra i sprawdzić, czy inny element prawdopodobnie należy do tego zbioru. Musimy jednak pamiętać, że nie mamy możliwości usunięcia dodanego już elementu, wylistowania wszystkich dodanych elementów czy też rozszerzenia struktury.

Listing 1. Operacja dodawania elementu do filtra

```
public void add(byte[] data) {
    int[] hashes = createHashes(data, k);
    for (int hash : hashes) {
        bitSet.set(hash, true);
    }
}
```

Listing 2. Operacja sprawdzenia, czy element znajduje się w filtrze

```
public boolean contains(byte[] data) {
    int[] hashes = createHashes(data, k);
    for (int hash : hashes) {
        if (!bitSet.get(hash)) {
            return false;
        }
    }
    return true;
}
```

PRZYKŁAD DZIAŁANIA

Przyjrzyjmy się teraz, jak wygląda działanie filtra Blooma w praktyce. Załóżmy, że rozmiar n wynosi 16 bitów i zostaną użyte 3 proste funkcje haszujące opisane wzorami:

1. $f(d) = \sum_{i=0}^{k-1} d_i \pmod{n}$
2. $g(d) = \sum_{i=0}^{k-1} \max(d_{(i+1) \bmod k}, d_i) \pmod{n}$
3. $h(d) = \sum_{i=0}^{k-1} (127 - d_i) \pmod{n}$

Listing 3. Implementacja funkcji f(d)

```
public int f(byte[] data) {
    int r = 0;
    for (byte x : data) {
        r += x;
    }
    return r % size;
}
```

Listing 4. Implementacja funkcji g(d)

```
public int g(byte[] data) {
    int r = 0;
    int nextIndex;
    for (int i = 0; i < data.length; i++) {
        nextIndex = (i + 1) % data.length;
        r += Math.max(data[nextIndex], data[i]);
    }
    return Math.abs(r % size);
}
```

Listing 5. Implementacja funkcji h(d)

```
public int h(byte[] data) {
    int r = 0;
    for (byte x : data) {
        r += Byte.MAX_VALUE - x;
    }
    return r % size;
}
```

Chcemy dodać trzy elementy reprezentowane za pomocą tablicy bajtów do naszej struktury.

- » *Magda* = [77, 97, 103, 100, 97]
- » *Jan* = [74, 97, 100]
- » *Piotr* = [80, 105, 111, 116, 114]

Następnie sprawdzimy, czy element *Marek* = [77, 97, 114, 101, 107] może w niej występować.

1. W pierwszym kroku bierzemy słowo „Magda”, wyliczamy wartości trzech funkcji haszujących. Wynikiem tego działania jest

Zintegruj własną aplikację z SMSAPI

Skorzystaj z gotowych bibliotek w językach:

php



C#

 python



Założ konto firmowe z kodem polecenia i odbierz pakiet SMS-ów na przetestowanie naszych usług:

FORMULARZ REJESTRACYJNY:

www.smsapi.pl/rejestracja

KOD POLECENIA:

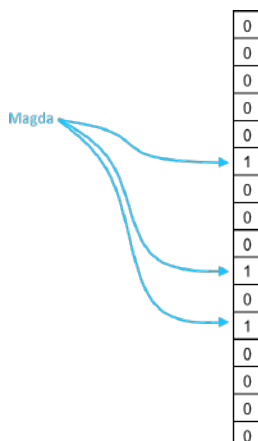
BONUS:

PR56 500 SMS-ÓW

SMSAPI

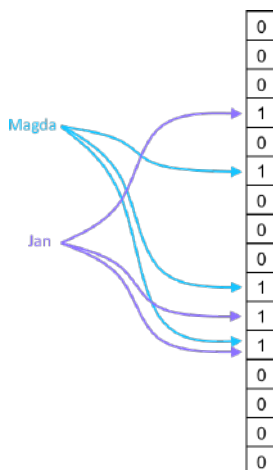


informacja, że musimy „zapalić” bity 5, 9, 11 (w przykładzie indeksujemy od 0).



Rysunek 1. Dodanie elementu „Magda”

- Następnie wykonujemy tę samą operację dla słowa „Jan” i ustawiamy bity 2, 10, 11. Jednak bit 11 został wcześniej „zapalony” przez element „Magda”, pozostaje zatem ustawić tylko bity 2 i 10.



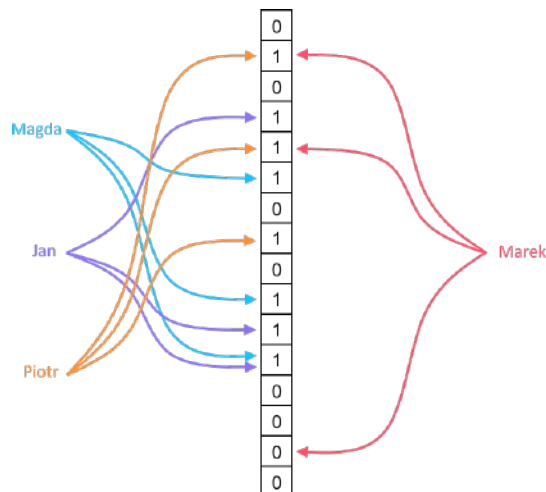
Rysunek 2. Dodanie elementu „Jan”

- Postępujemy identycznie dla ostatniego słowa „Piotr” i dowiadujemy się, że musimy ustawić bity 1, 4, 7.



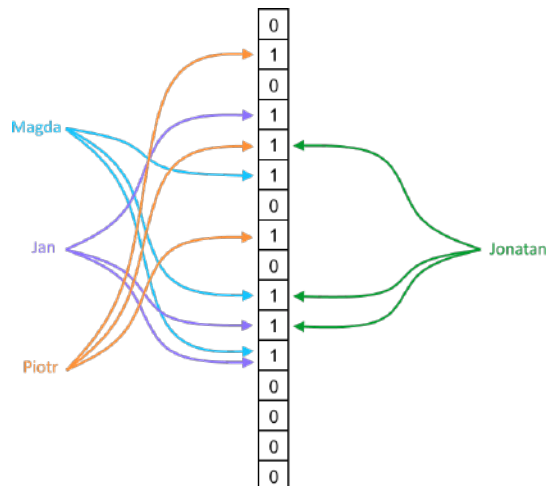
Rysunek 3. Dodanie elementu „Piotr”

- W tym momencie żądane elementy zostały wstawione do naszego filtra. Sprawdźmy zatem, czy znajdziemy w nim słowo „Marek”. Wyliczamy więc, które bity powinny być ustawione, i porównujemy to ze stanem filtra. Wynikiem jest odpowiedź, że element „Marek” z całą pewnością nie należy do naszego zbioru, ponieważ wymagany bit 14 nie został ustawiony.



Rysunek 4. Sprawdzenie obecności elementu „Marek”

Możliwy jest natomiast alternatywny scenariusz (Rysunek 5). Zamiast słowa „Marek” spróbujmy sprawdzić, czy słowo „Jonatan” występuje w filtrze. Wyliczamy, które bity powinny zostać ustawione dla nowego słowa, i otrzymujemy informację, że bit 4, 9 oraz 10. Oznacza to, że słowo „Jonatan” zostało dodane do naszego filtra. W tym przykładzie wiemy jednak, że nie jest to prawdą. Jest to przykład wyniku fałszywie pozytywnego.



Rysunek 5. Przykład wyniku fałszywie pozytywnego

OPTYMALNE PARAMETRY

Teraz, gdy wiemy już, czym jest filtr Bloom’a i jak wygląda jego działanie, pora na trochę matematyki, żeby dowiedzieć się, jak dobrać parametry, aby była to jak najbardziej optymalna dla nas struktura. W pierwszej kolejności określimy rozmiar tablicy, czyli ilość bitów, jaką powinna zawierać. Potrzebna w tym przypadku będzie wiedza na temat tego, ile elementów planujemy umieścić w filtrze, oraz założone prawdopodobieństwo fałszywego wyniku. Należy pa-

miętać, że jeśli będziemy przekraczać założoną ilość elementów, to wskaźnik prawdopodobieństwa będzie się zwiększał. W momencie, gdy mamy te dwie informacje, rozmiar n wyliczamy zgodnie ze wzorem:

$$n = \frac{-m \cdot \ln(p)}{\ln(2)^2}$$

gdzie przez p – oznaczamy prawdopodobieństwo fałszywego wyniku ($0.01 = 1\%$), a m – to oczekiwana ilość elementów.

Skoro mamy już policzony rozmiar, następnym krokiem będzie określenie, ile funkcji haszujących będziemy potrzebować. Korzystamy więc z poniższego wzoru:

$$k = \frac{n}{m} \cdot \ln(2)$$

Na przykład, zamierzamy umieścić w filtrze 2000 elementów i chcemy zachować wskaźnik prawdopodobieństwa na poziomie 0.01%, czyli $m = 2000$, $p = 0.0001$, a więc dostajemy odpowiedź, że rozmiar powinien wynosić $n \approx 38340$ bitów (≈ 4.68 kB) i potrzebujemy $k = 13$ funkcji haszujących. Natomiast gdy przy takiej samej liczbie spodziewanych elementów zwiększymy prawdopodobieństwo do 0.1%, to rozmiar zmniejszy się do 28755 bitów (≈ 3.51 kB), czyli o 25%. Dobór początkowej wartości parametru p jest zależny od konkretnego problemu. To od niego zależy, czy możemy pozwolić sobie na większy rozmiar filtra, czy może jednak istotniejsze jest dla nas prawdopodobieństwo, z jakim możemy otrzymać wynik fałszywie pozytywny.

GENEROWANIE FUNKCJI HASZUJĄCYCH

Ustaliliśmy już, że każdy element musi zostać poddany działaniu k niezależnych funkcji haszujących. Jest to najbardziej złożona część w filtrze Blooma, bowiem określenie „niezależne” rodzi tutaj pewnie problemy. Musimy odpowiedzieć na pytanie, jak poradzić sobie w momencie, gdy z powyższych obliczeń dowiemy się, że potrzebujemy użyć np. 15 takich funkcji? Pora by zająć się generowaniem funkcji haszujących. Drobne zmiany w różnych podejściach mogą znacząco wpływać na końcowy rezultat.

Przed implementacją właściwej części należy zdecydować się, jaką wybrać bazową funkcję haszującą, którą wykorzystamy przy generowaniu innych. Przede wszystkim zależy nam na szybkości, drobnym kosztem gorszego rozkładu w przestrzeni możliwych wartości. To oczywiście zwiększa szanse na ewentualne kolizje. W przykładach zostanie użyta funkcja MurmurHash3. Nie należy ona do specjalnego podzbioru bezpiecznych funkcji haszujących. Jednak za jej użyciem przemawia w naszym przypadku wydajność. Wykorzystanie jej w implementacji znacząco przyspiesza korzystanie z filtra Blooma.

Pierwsza próba

Najbardziej trywialnym podejściem, jakie nasuwa się na myśl, jest użycie różnych wartości seed dla funkcji haszującej. Mogą być to kolejne wartości prostego licznika. Następnie generujemy indeksy za pomocą funkcji haszującej, w tym przypadku musimy to zrobić k razy, gdzie k reprezentuje ilość wartości, jakie musimy wygenerować.

Listing 6. Operacja generowania z różną wartością seed

```
public int[] createHashes(byte[] data, int k) {
    int[] hashes = new int[k];
    int seed = 0, r;

    for (int i = 0; i < k; i++) {
        r = Murmur3.hash32(data, seed);
        hashes[i] = Math.abs(r % size);
        seed++;
    }

    return hashes;
}
```

- » Parametr `data` w Listingu 6 to dowolny element w formie tablicy bajtów, dla którego chcemy uzyskać informację, które bity należy ustawić.
- » Parametr `k` w Listingu 6 określa, ile funkcji haszujących potrzebujemy wygenerować.

Double Hashing

Powyższe rozwiązanie nie jest optymalne i można z powodzeniem zmniejszyć liczbę użyć funkcji haszującej. W podejściu Double Hashing dla każdego wywołania metody `createHashes` używamy tylko dwóch wywołań funkcji haszującej niezależnie od parametru k . Z tych wyników jesteśmy w stanie wygenerować pozostałą dowolną ilość. Podejście to można opisać wzorem:

$$f(i) = a(\delta) + i \cdot b(\delta) \pmod{n}$$

gdzie $a(\delta)$ to wynik pierwszej funkcji haszującej, natomiast $b(\delta)$ jest wynikiem drugiej funkcji haszującej.

Listing 7. Podejście Double Hashing

```
public int[] createHashes(byte[] data, int k) {
    int[] hashes = new int[k];

    int a = Murmur3.hash32(data, 0);
    int b = Murmur3.hash32(data, a);

    hashes[0] = Math.abs(a % size);
    for (int i = 1; i < k; i++) {
        a = (a + b) % size;
        hashes[i] = Math.abs(a);
    }

    return hashes;
}
```

Trzeba natomiast przyrzeć się przypadkowi, gdy $b(\delta) = 0$. W tym przypadku wszystkie wartości dla tego elementu będą miały tę samą wartość wynoszącą $a(\delta)$. Oczywiście ma to bardzo zły wpływ na prawdopodobieństwo, z jakim możemy otrzymać wynik fałszywie pozytywny.

Triple Hashing

Triple Hashing stanowi rozbudowane podejście Double Hashing o jeszcze jedną gotową funkcję haszującą. Można to opisać wzorem:

$$f(i) = a(\delta) + i \cdot b(\delta) + \frac{i(i-1)}{2} c(\delta) \pmod{n}$$

Zabieg ten eliminuje problem z zerową wartością $b(\delta)$.

Listing 8. Podejście Triple Hashing

```
public int[] createHashes(byte[] data, int k) {
    int[] hashes = new int[k];
    int a = Murmur3.hash32(data, 0);
    int b = Murmur3.hash32(data, a);
    int c = Murmur3.hash32(data, b);

    hashes[0] = Math.abs(a % size);
    for (int i = 1; i < k; i++) {
        a = (a + b) % size;
        b = (b + c) % size;
        hashes[i] = Math.abs(a);
    }

    return hashes;
}
```

Enhanced Double Hashing

Podejście Enhanced Double Hashing charakteryzuje się tym, że używamy dwóch funkcji haszujących oraz rezygnujemy z liniowego czynnika, jaki jest obecny w Double Hashing. Rezultatem tego jest to, że na wartość $b(\delta)$ nie są już nałożone restrykcje oraz użyjemy lepszą wydajność niż w Triple Hashing. To podejście zostało opisane wzorem:

$$f(i) = a(\delta) + i \cdot b(\delta) + \frac{i^3 - i}{2} \pmod{n}$$

Listing 9. Podejście Enhanced Double Hashing

```
public int[] createHashes(byte[] data, int k) {
    int[] hashes = new int[k];
    int a = Murmur3.hash32(data, 0);
    int b = Murmur3.hash32(data, a);

    hashes[0] = Math.abs(a % size);
    for (int i = 1; i < k; i++) {
        a = (a + b) % size;
        b = (b + i) % size;
        hashes[i] = Math.abs(a);
    }

    return hashes;
}
```

ZASTOSOWANIE

Filtr Blooma przy odpowiednim wykorzystaniu znacząco pomaga zredukować liczbę kosztownych operacji takich jak np. odczyt/zapis na dysku czy wysłanie mniejszych pakietów danych. Bardzo często nie jesteśmy nawet świadomi, że wydajność narzędzi czy aplikacji, z których korzystamy codziennie, zawdzięczamy właśnie temu elementowi.

Bazy danych

Filtry Blooma znalazły zastosowanie w bazach danych między innymi takich jak Cassandra czy Google BigTable. Na przykład w Cassandra, SSTable składa się dodatkowo z naszego filtra, który przechowywany jest w pamięci. Zapewnia on bardzo szybkie sprawdzenie, czy dany klucz istnieje w SSTable. To pozwala na uniknięcie niepotrzebnego przeglądania bloków danych przechowywanych na dysku, co jest zdecydowanie bardziej kosztowną operacją. Aby zmienić wskaźnik prawdopodobieństwa wyniku fałszywie pozytywnego na danej tabeli, wystarczy użyć polecenia:

```
ALTER TABLE keyspace.table WITH bloom_filter_fp_chance=0.01
```

Aplikacje sieciowe

Google zdecydowało się również na użycie filtrów Blooma w swojej przeglądarce Chrome. Przechowywane są w nich lokalnie informacje o adresach niebezpiecznych stron internetowych. Pomaga to w podjęciu decyzji o bezpieczeństwie danego połączenia, przy okazji unikając wysłania zapytania do serwera.

SQUID Web Proxy Cache używa filtrów w podobny sposób. Proxy wymieniają się filtrami Blooma zawierającymi zawartość *cache* w postaci kombinacji adresów URL oraz metod HTTP.

Media społecznościowe

LinkedIn posiada swoją bibliotekę Cleo napisaną w języku Java, w której została użyta implementacja filtra Blooma. W połączeniu z innymi mechanizmami pomaga w procesie wyszukiwania kontaktów, firm oraz grup. Facebook w swojej implementacji wyszukiwarki znajomych również zdecydował się na jego użycie.

Bibliografia

- ▶ „Bloom Filters in Probabilistic Verification” – Peter C. Dillinger, Panagiotis Manolios
- ▶ „Less Hashing, Same Performance: Building a Better Bloom Filter” – Adam Kirsch, Michael Mitzenmacher
- ▶ „Bigtable: A Distributed Storage System for Structured Data” – Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
- ▶ http://cassandra.apache.org/doc/latest/operating/bloom_filters.html
- ▶ <https://engineering.linkedin.com/open-source/cleo-open-source-technology-behind-linkedin-typeahead-search>
- ▶ „Overlay Networks: Toward Information Networking” – Sasu Tarkoma

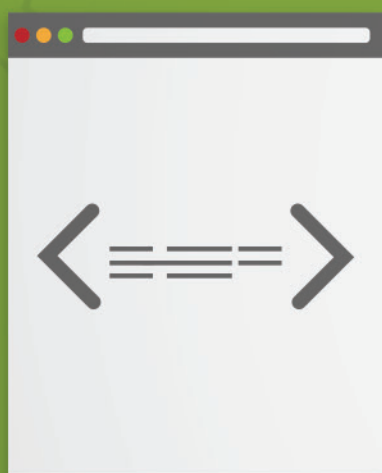


DAMIAN STYGAR

damian.stygar@gmail.com

Obecnie pracuje jako Application Developer w firmie Atos GDC Polska. Student Informatyki na Wydziale Podstawowych Problemów Techniki Politechniki Wrocławskiej. Do jego zainteresowań należy m.in.: bezpieczeństwo aplikacji webowych, kryptologia, reverse engineering.

Najlepsze oferty pracy dla programistów



Lemon oraz re2c – duet do analizy gramatyczno-leksykalnej

Analiza gramatyczno-leksykalna to zadanie powszechnie wykonywane w wielu typach programów. Głównym przykładem programów, gdzie tego typu analiza jest stosowna, są kompilatory, jednak wiele innych programów posiada pliki tekstowe opisujące konfigurację, zatem i tam taka analiza zawartości pliku jest wykonywana.

Istnieją odpowiednie narzędzia wspomagające zadanie analizy, które z racji swojej historii oraz znaczenia stały się elementem kanonu informatyki. Należą do nich `lex` i `yacc`. Pierwszy z nich odpowiada za analizę leksykalną, tj. wyodrębnia np. poszczególne słowa kluczowe z tekstu kodu źródłowego. Natomiast program `yacc` jest odpowiedzialny za analizę gramatyczną, czyli w wielkim skrócie za to, czy poszczególne wyrażenia zostały poprawnie ułożone. Nowe wersje tych programów to `flex` oraz `bison`. Jednakże nie są to jedyne przykłady tego typu narzędzi. Dużą popularność zdobyła także para prezentowana w tym artykule, tj. generator gramatyk `Lemon` oraz pakiet `re2c` wspomagający budowę analizatora leksykalnego. Postaramy się przedstawić, na przykładzie analizy składniowej języka Basic, jak buduje się analizator gramatyki i wcześniej analizator leksykalny, wykorzystując bohaterów naszego artykułu.

ZADANIE GŁÓWNE

Nasze zadanie, jakie opisujemy w tym artykule, to budowa analizatora leksykalnego oraz gramatycznego języka programowania wzorowanego na popularnym w czasach ośmiobitowych komputerów Basicu. Choć zadanie wydaje się łatwe, to wbrew pozorom nie jest trywialne i realizacja interpreterów języka Basic w czasach komputerów z procesorami ośmiobitowymi była naprawdę imponująca. Szczególnie jeśli zwróci się uwagę na fakt, że taki interpreter mieścił się w 16 kilobajtach pamięci np. w przypadku komputera ZX Spectrum 48k.

Nasza wersja języka Basic jest oczywiście tylko pretekstem do poznania narzędzi, które współcześnie wspomagają tego typu zadanie. Przykładowy program, jaki będziemy analizować, został zaprezentowany w Listingu 1.

Listing 1. Przykładowy program w języku Basic

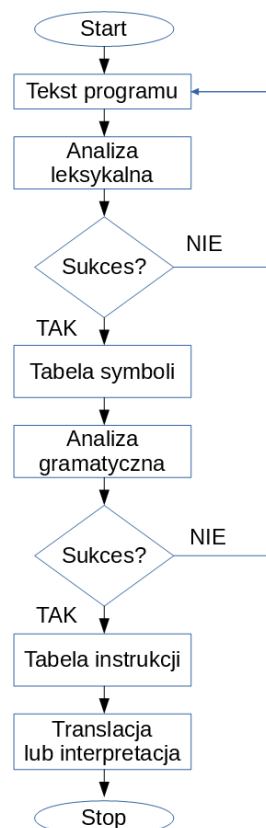
```
5 let i=0\n
10 gosub 100\n\
20 for i=1 to 10\n\
30 print i\n\
40 next i\n\
50 print \"koniec programu\"\n\
60 end\n\
100 print \"+-----+\n\
101 print \"| podprocedura |\"\n\
102 print \"+-----+\n\
110 return\n\";
```

Celowo zostały umieszczone znaki przejścia do nowej linii na końcu każdej linii, ponieważ zakładamy, iż tak będziemy oznaczać koniec instrukcji. Zatem jest to dla nas ważne założenie.

Analiza gramatyczna takiego programu składa się z dwóch głównych etapów zaprezentowanych na Rysunku 1. Pierwszy etap to analiza leksykalna, po której powstaje tablica symboli poszczególnych jednostek językowych w programie. W tablicy zapisujemy, iż mamy np. wartość liczbową, słowo kluczowe, identyfikator i ponownie wartość liczbową. Jednak na tym etapie nie wiadomo, czy odczytana wartość reprezentowana przez liczbę całkowitą jest właściwym numerem linii, bo rozstrzygnięcie tej kwestii zostanie wykonane na poziomie analizy gramatycznej.

W analizie gramatycznej sprawdzamy, czy użytkownik dochował wierności regułom składni naszego języka. Dodatkowo możemy też przygotować tabelę poszczególnych instrukcji, z których zbudowany jest nasz program.

Na Rysunku 1 mamy jeszcze trzeci etap, a mianowicie interpretację lub translację tabeli instrukcji, jaka powstała po analizie gramatycznej. W artykule jednak będziemy zajmować się tylko dwoma pierwszymi etapami.



Rysunek 1. Główne etapy w naszym interpreterze języka Basic

Najlepsze od



PATRONI

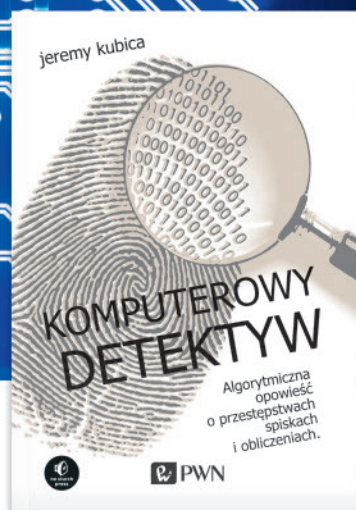
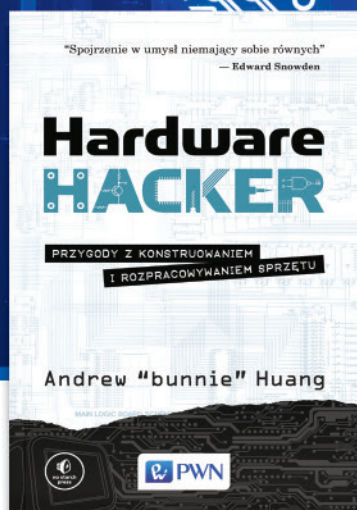


Safety and Security

programista



Polecamy:



KSIAŻKI DOSTĘPNE NA: WWW.KSIEGARNIA.PWN.PL

Odwiedź nas na:
 **IT.PWN.PL**

LICZBY CAŁKOWITE

Pierwszy przykład, jaki podamy, dotyczy narzędzia `re2c`. Kod źródłowy przykładu został w całości przedstawiony w Listingu 1. Dotyczy on analizy ciągów znaków oraz sprawdzania, czy podany ciąg znaków jest poprawnie opisaną liczbą całkowitą w jednym w czterech systemów zapisu: binarnym, ósemkowym, dziesiętnym oraz szesnastkowym. Samodzielne opracowanie analizatora tego typu wyrażen nie jest trudne, ale jak widać z Listingu 1, zastosowanie pakietu `re2c` pozwala uzyskać taki analizator znacznie mniejszym nakładem pracy.

Główne zadanie analizy leksykalnej spoczywa na funkcji `lexer`. Pakiet `re2c` nie narzuca nazwy, zatem możemy funkcję tę nazwać zgodnie ze swoimi zasadami. Liczba argumentów i typ zwracanej wartości leży także w naszej gestii.

Rola argumentów, jakie przyjmuje funkcja `lexer`, jest następująca: w pierwszym `YYCURSOR` podajemy ciąg znaków do analizy. Nazwa tego parametru jest specjalnie dobrana, bowiem pakiet `re2c` stosuje zmienną o takiej samej nazwie do odczytu danych do analizy. Nie będziemy więc musieli dodatkowo określać, który parametr jest odpowiedzialny za dostarczenie danych.

Parametr `output` zgodnie ze swoją nazwą będzie zawierał wyodrębnioną liczbę w jednym z czterech formatów. Istotna jest też zwracana wartość, ponieważ jeśli uda się poprawnie rozpoznać np. liczbę szesnastkową, to zostanie zwrócona wartość `HEX` potwierdzająca ten fakt.

Właściwy fragment kodu, który jest przetwarzany przez pakiet `re2c`, został umieszczony w komentarzu. Pierwsza linia:

```
/*!stags:re2c format = 'const char *@@;';*/
```

jest potrzebna, aby wyciągać potrzebne nam fragmenty tekstu. Będziemy bowiem wczytywać liczby np.:

```
0xAF00 1234 0800
```

W pierwszym przypadku mamy do czynienia z liczbą szesnastkową, choć nie potrzebujemy dwóch pierwszych znaków, to za pomocą mechanizmów tagów będziemy mogli wyciągnąć odpowiedni fragment napisu.

Zanim do tego dojdziemy, w drugim komentarzu określamy na początku m.in. typ danych, na jakim pracujemy, w naszym przykładzie są to znaki:

```
re2c:define:YYCTYPE = char;
```

Następnie określamy za pomocą wyrażen regularnych potrzebne nam wyrażenia. Oznaczamy koniec za pomocą zera:

```
end = "\x00";
```

Jest to jak najbardziej poprawne, ponieważ za argument podajemy ciąg znaków, w języku C zero oznacza koniec ciągu znaków. Następne cztery wyrażenia: `bin`, `oct`, `dec`, `hex` opisują liczby, np. liczbę ósemkową opisano następująco:

```
oct = "0" [0-7]*;
```

Oznacza to, iż na początku musi wystąpić symbol zera, następnie dopuszczamy ciąg znaków zbudowany z ośmiu cyfr, łącznie z ze-

rem i cyfrą siedem. Symbol gwiazdki oznacza, iż liczba powtórzeń może wynosić zero lub więcej. W podobny sposób określamy, jak widać w Listingu 1, postacie dla pozostałych typów liczb. Jeśli jednak chcemy rozpoznać taką liczbę, to należy użyć tzw. akcji. Dla liczb szesnastkowych przedstawia się ona następująco:

```
@c1 hex @c2 end { len = (size_t)(c2-c1); strncpy(output, c1+2, len-2); output[len-2]=0; return HEX; }
```

Początek oznacza, że interesuje nas ciąg `hex`, wyrażenia `@c1` oraz `@c2` oznaczają tagi, czyli wskaźniki do początku napisu `hex` oraz jego końca. Ponieważ są to wskaźniki oraz typ to znak, to możemy zwyczajnie odjąć `c1` od `c2`, aby uzyskać długość ciągu znaków. Instrukcją `strncpy` kopiujemy wyznaczoną liczbę szesnastkową do zmiennej `output`. Dodajemy dwójkę, aby ominąć wspomniany już symbol `0x` oznaczający liczbę szesnastkową.

W naszym przykładzie, oprócz czterech akcji dla liczb, mamy jeszcze akcję o postaci:

```
* { return NOT_A_NUM; }
```

Oznacza ona, iż jeśli nie będzie można dopasować podanego ciągu znaków do jednego z czterech przypadków, to należy wybrać tę akcję, która opuści funkcję `lexer`. Wartością wynikową będzie `NOT_A_NUM`, co oznacza, że podany ciąg nie był liczbą.

Listing 2. Analiza liczb całkowitych

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

enum num_t { NOT_A_NUM, BIN, OCT, DEC, HEX };

static enum num_t lexer(const char *YYCURSOR, char *output) {
    const char *YYMARKER, *c1, *c2;
    size_t len;

    /*!stags:re2c format = 'const char *@@;';*/
    /*!re2c
    re2c:define:YYCTYPE = char;
    re2c:yyfill:enable = 0;

    end = "\x00";
    bin = '0b' [01]+;
    oct = "0" [0-7]*;
    dec = [1-9][0-9]*;
    hex = '0x' [0-9a-fA-F]+;

    * { return NOT_A_NUM; }
    @c1 bin @c2 end { len = (size_t)(c2-c1); strncpy(output, c1+2, len-2); output[len-2]=0; return BIN; }
    @c1 oct @c2 end { len = (size_t)(c2-c1); strncpy(output, c1+1, len-1); output[len-1]=0; return OCT; }
    @c1 dec @c2 end { len = (size_t)(c2-c1); strncpy(output, c1, len); output[len]=0; return DEC; }
    @c1 hex @c2 end { len = (size_t)(c2-c1); strncpy(output, c1+2, len-2); output[len-2]=0; return HEX; }
    */
}

int main(int argc, char **argv) {
    char output[127];
    enum num_t status;
    for (int i = 1; i < argc; ++i) {
        status = lexer(argv[i], &output[0]);
        switch (status) {
            case NOT_A_NUM: printf("Błąd, podana wartosc nie jest liczba calkowita.\n"); break;
            case BIN: printf("Podano liczbe binarna: 0b%s.\n", &output[0]); break;
            case OCT: printf("Podano liczbe osemkowa: 0%s.\n", &output[0]); break;
            case DEC: printf("Podano liczbe dziesietna: %s.\n", &output[0]); break;
            case HEX: printf("Podano liczbe szesnastkowa: 0x%s.\n",
```

```

        &output[0] ); break;
    }
}
return 0;
}

```

Zaletą pakietu re2c jest to, iż powstały kod analizujący wyrażenia jest bardzo prosty do użycia. W przykładzie analizujemy argumenty podane przy wywołaniu programu:

```
status = lexer( argv[i], &output[0] );
```

Ciąg znaków w parametrze `argv[i]` zostanie poddany analizie i jeśli będzie to jedna z czterech liczb, to w zmiennej `status` znajdzie się wartość opisująca, jaka to liczba, a jej postać jako ciąg znaków bez symbolu wiodącego zostanie umieszczona w zmiennej `output`.

KOMPILACJA PRZYKŁADU PIERWSZEGO

Po omówieniu naszego pierwszego przykładu możemy przejść do jego kompilacji. Pierwsza czynność, jaką musimy wykonać, to translacja kodu za pomocą re2c:

```
re2c --tags reco_ints.re -o reco.c
```

Używamy w naszym przykładzie tagów, dlatego musimy dodać opcję `--tags`. Przykład z Listingu 1 został umieszczony w pliku `reco_ints.re`, ale nowo utworzony plik określamy za pomocą opcji `-o`.

Powstały plik `reco.c` możemy kompilować wprost za pomocą polecenia `gcc`. Nie są potrzebne dodatkowe dedykowane biblioteki dla re2c, co stanowi dodatkową zaletę tego pakietu:

```
gcc reco.c -o reco
```

Możemy teraz uruchomić przykład:

```
./reco 0xabf0
```

i powinniśmy zobaczyć odpowiedź naszego programu.

LEXER DLA JĘZYKA BASIC

Po pierwszym przykładzie w re2c możemy zbudować kompletny analizator leksykalny dla naszego języka Basic. W Listingu 3 przedstawiono tym razem tylko kod źródłowy funkcji `basic_lexer`, odpowiedzialnej za analizę leksykalną.

Parametr, jaki podajemy, to ciąg znaków z tekstem źródłowym programu w Basicu do analizy. Ponieważ parametr wskazujący na dane wejściowe nie nazywa się `YYCURSOR`, to musimy poinformować o tym pakiet re2c:

```
re2c:define:YYCURSOR = s;
```

by stosować parametr wejściowy w roli parametru `YYCURSOR`. Podobnie jak poprzednio, stosujemy typ znakowy i określamy najważniejsze dla nas wyrażenia regularne. Parametr `end` zawiera informację o znaku końca strumienia danych. Dodatkowo też mamy akcję, która po znalezieniu końca opuszcza funkcję analizującą tekst programu.

Musimy tak zrobić, ponieważ zastosowaliśmy pętlę skonstruowaną za pomocą skoku `goto` do etykiety `lexloop`. Co prawda

Instalacja pakietów Lemon oraz re2c

Jeśli korzystamy z dystrybucji systemu Linux lub Mac OS X, to oba pakiety powinny być dostępne w systemie pakietów używanej przez nas dystrybucji. W przypadku Mac OS X oba pakiety są dostępne poprzez system portów, wystarczy z poziomu konsoli wydać polecenia:

```
sudo port install re2c
sudo port install lemon
```

Jednakże nie ma problemu, by samodzielnie dokonać kompilacji obu pakietów. Jest to wyjątkowo proste w przypadku pakietu Lemon, ponieważ pakiet ten ściągamy jako kod źródłowy w postaci jednego pliku `lemon.c`. Polecenie kompilacji sprowadzi się tylko do bezpośredniego wywołania polecenia `gcc`:

```
gcc lemon.c -o lemon
```

Po krótkiej chwili uzyskamy plik binarny i możemy od razu kompilować nasze gramatyki.

Pakiet re2c jest wyposażony w skrypty z rodziny automake. Zatem jego kompilacja, choć nieco dłuższa, również sprowadza się do wydania kilku poleceń. Po ściągnięciu kodu źródłowego w postaci archiwum np. `re2c-1.0.2.tar.gz` musimy plik rozpakować np.:

```
tar xvpf re2c-1.0.2.tar.gz
```

Następnie wydajemy trzy polecenia przeprowadzające konfigurację, kompilację oraz instalację:

```
./configure --prefix=/opt
make
make install
```

Warto zwrócić uwagę na opcję `--prefix`, gdzie wskazujemy katalog, w którym zostaną umieszczone pliki pakietu re2c.

Jeśli chcemy skorzystać z repozytorium GitHub, to na początek klonujemy kod źródłowy:

```
git clone https://github.com/skvadrik/re2c.git re2c
```

Musimy utworzyć pliki systemu automake:

```
./autogen.sh
```

A następnie wydajemy polecenia konfiguracji, kompilacji oraz instalacji, tak jak w przypadku archiwum z kodem źródłowym.

nie jest to najbardziej fortunne rozwiązanie, to jednak parser musi być szybki i zastosowanie `goto` w naszym programie nie pogorszy w żaden sposób czytelności kodu. Choć możemy wprowadzić pętlę `while` z warunkiem końcowym ustalonym po akcji `end`.

Listing 3. Analizator leksykalny dla języka Basic

```

int basic_lexer(const char *s) {
    int len;
    char tmp[127] = {0};
    const char *mrk, *p1, *p2;

    /*!stags:re2c format = 'const char *@@;';*/
lexloop:
    /*!re2c
    re2c:define:YYCTYPE = char;
    re2c:define:YYCURSOR = s;
    re2c:define:YYMARKER = mrk;
    re2c:yyfill:enable = 0;

    end = "\x00";

    *      { return -1; }
    end { return 0; }

    dec = [0-9][0-9]*;
    char = [^=] \ end;

```

```

str      = "\"\" [^]* \"\"";
whitechar = [ \t];
whitespaces = whitechar*;

id = [a-zA-Z_][a-zA-Z_0-9]*;

whitespaces { goto lexloop; }

@p1 str @p2 { len = (size_t)(p2-p1); strncpy(tmp, p1, len);
tmp[len]=0; insert_token(STR_TOK, &tmp[0]); goto lexloop; }
"let" { insert_token(LET_TOK, NULL); goto lexloop; }
"print" { insert_token(PRINT_TOK, NULL); goto lexloop; }
"if" { insert_token(IF_TOK, NULL); goto lexloop; }
"then" { insert_token(THEN_TOK, NULL); goto lexloop; }
"else" { insert_token(ELSE_TOK, NULL); goto lexloop; }
"for" { insert_token(FOR_TOK, NULL); goto lexloop; }
"to" { insert_token(TO_TOK, NULL); goto lexloop; }
"next" { insert_token(NEXT_TOK, NULL); goto lexloop; }
"goto" { insert_token(GOTO_TOK, NULL); goto lexloop; }
"gosub" { insert_token(GOSUB_TOK, NULL); goto lexloop; }
"return" { insert_token(RETURN_TOK, NULL); goto lexloop; }
"call" { insert_token(CALL_TOK, NULL); goto lexloop; }
"end" { insert_token(END_TOK, NULL); goto lexloop; }
"\n" { insert_token(NEW_LINE_TOK, NULL); goto lexloop; }
"," { insert_token(COMMA_TOK, NULL); goto lexloop; }
";" { insert_token(SEMICOLON_TOK, NULL); goto lexloop; }
"+" { insert_token(PLUS_TOK, NULL); goto lexloop; }
"-" { insert_token(MINUS_TOK, NULL); goto lexloop; }
"&" { insert_token(AMPERSAND_TOK, NULL); goto lexloop; }
"|" { insert_token(PIPE_TOK, NULL); goto lexloop; }
"*" { insert_token(TIMES_TOK, NULL); goto lexloop; }
"/" { insert_token(DIVIDE_TOK, NULL); goto lexloop; }
"%" { insert_token(QUESTION_MARK_TOK, NULL); goto lexloop; }
"(" { insert_token(LEFT_BRACKET_TOK, NULL); goto lexloop; }
")" { insert_token(RIGHT_BRACKET_TOK, NULL); goto lexloop; }
"<" { insert_token(LESS_TOK, NULL); goto lexloop; }
">" { insert_token(BIGGER_TOK, NULL); goto lexloop; }
"=" { insert_token(EQ_TOK, NULL); goto lexloop; }

@p1 id @p2 { len = (size_t)(p2-p1); strncpy(tmp, p1, len);
tmp[len]=0; insert_token(ID_TOK, &tmp[0]); goto lexloop; }
@p1 dec @p2 { len = (size_t)(p2-p1); strncpy(tmp, p1, len);
tmp[len]=0; insert_token(DEcimal_TOK, &tmp[0]); goto lexloop; }
*/
}

```

Akcja oznaczona symbolem gwiazdki podobnie jak w poprzednim przykładzie oznacza zakończenie analizy z błędem, z powodu wystąpienia ciągu znaków, który nie należy do języka, który analizujemy.

Bardzo istotnym elementem jest ciąg znaków ujęty w cudzysłowy. Stosujemy następujące wyrażenie regularne:

```
str      = "\"\" [^]* \"\"";
```

Zastosowaliśmy w tej regule istotne uproszczenie, bowiem nie stosujemy kodów escape, czyli nie możemy w środku ciągu znaków użyć cudzysłowu.

W analizie pomijamy też znaki białe, czyli spację, znaki tabulacji, ale nie przejścia do nowej linii:

```
whitespaces { goto lexloop; }
```

W tym przypadku następuje skok do etykiety, aby od razu przejść do detekcji następnego wyrażenia. Większość reguł, jak widać, jest dość podobna, wyróżniają się tylko trzy z nich. Pierwsza dotyczy ciągu znaków, bowiem musimy za pomocą tagów wyodrębnić jego treść:

```
@p1 str @p2 { len = (size_t)(p2-p1); strncpy(tmp, p1, len);
tmp[len]=0; insert_token(STR_TOK, &tmp[0]); goto lexloop; }
```

Następnie za pomocą funkcji `insert_token` zapisujemy informacje do specjalnej tablicy, którą zaraz omówimy. Dwie pozostałe reguły dotyczą identyfikatora oraz liczby dziesiętnej. I są zbudowane w sposób podobny jak reguła dla ciągu znaków.

Pozostałe reguły dotyczą słów kluczowych oraz znaków specjalnych, jak dodawanie, mnożenie znaków równości. Ogólnie są one zbudowane w następujący sposób:

```
"gosub" { insert_token(GOSUB_TOK, NULL); goto lexloop; }
```

Na początku mamy interesujący nas ciąg znaków, a w akcji podajemy wywołanie funkcji dodającej wykryte słowo kluczowe do tablicy tzw. tokenów reprezentujących nasz program podzielony na poszczególne fragmenty wraz z informacją, czym są owe fragmenty. Czyli mamy wskazanie, że słowo `let` oznacza słowo kluczowe, znak plus to symbol operatora etc.

Tablica dla uproszczenia została zdefiniowana w następujący sposób:

```
t_token arrTok[2048];
int arrTokIdx = 0;
```

Zmienna `arrTokIdx` pełni rolę indeksu dostępu do elementów tablicy. Wywołanie funkcji w następujący sposób:

```
insert_token(ELSE_TOK, NULL);
```

spowoduje dopisanie do tablicy słowa kluczowego `else` oznaczonego przez wartość `ELSE_TOK` i inkrementację indeksu. W naszym programie nie pilnujemy, czy mamy dostateczną ilość miejsca w tablicy. Jeśli więc nasz analizator ma być zastosowany w rzeczywistym środowisku, należy ten element poprawić, zamieniając tablicę na np. listę.

Drugi argument funkcji `insert_token` pozwala nam na podanie wartości np. liczby, czy ciągu znaków, jaki został w trakcie analizy poprawnie wykryty. Pojedynczy element naszej tablicy `arrTok` został określony za pomocą następującej struktury:

```
typedef struct {
    enum tok_type token;
    t_token_value value;
} t_token;
```

Kolejna struktura `t_token_value` zawiera pola odpowiadające typom poszczególnych wartości, jakie możemy wykryć w trakcie analizy. My jednak będziemy stosować tylko ciągi znaków, zatem np. liczby będziemy zapisywać w postaci znakowej i dopiero w trakcie analizy gramatycznej konwertować je na odpowiednie typy.

Analiza tekstu programu pozwala wypełnić tablicę informacjami o słowach kluczowych i wartościach. Natomiast samą analizę wykonamy, wywołując funkcję `basic_lexer`:

```
r = basic_lexer( txtprog );
```

Parametr `txtprog` jest ciągiem znaków naszego programu w Basicu. Jeśli analiza będzie poprawna, to możemy rozpocząć drugi etap, tj. analizę gramatyczną.

GRAMATYKA

W Listingu 4 zawarto wybrane fragmenty gramatyki języka Basic, które pozwalają na pełną analizę gramatyczną programu podanego w Listingu 1. W pierwszych liniach kodu dołączamy pliki nagłówkowe, w tym plik `basiclexer_token.h` zawierający definicje słów kluczowych oraz tablicy jednostek leksykalnych powstałych po analizie pakietem `re2c`.

W tym momencie warto podkreślić, że w naszej analizie gramatycznej będziemy stosować inne oznaczenia poszczególnych słów kluczowych. Dla przykładu podczas analizy leksykalnej LET_TOK oczywiście reprezentuje słowo `let`, ale w analizie gramatycznej będziemy stosować nazwę LET_TOKEN. Nie ma problemu, aby tłumaczyć oznaczenia z jednego typu na drugi, więc ostatecznie nie będzie to aż tak duży problem.

Lemon pozwala na określenie, z jakim typem danych będziemy pracować. W wyniku analizy gramatycznej otrzymamy np. wartość wyrażenia i możemy określić, jaki to będzie typ. My zastosujemy definicję typu `t_token_value`, który będzie zawierać w sobie wszystkie dopuszczalne wartości oraz pole określające, jaki typ aktualnie jest reprezentowany.

Kolejne fragmenty kodu pozwalają określić, w jaki sposób łączą operatory dodania, mnożenia. Podajemy także dodatkowy parametr:

```
%extra_argument { t_ins_table *tinsptr }
```

Podobnie bowiem jak w przypadku analizy leksykalnej, teraz też będziemy wypełniać tabelę, opisując przeznaczenie poszczególnych instrukcji. Na przykład każda linia w Basicu zaczyna się od numeru, więc będziemy sprawdzać, czy użytkownik w swoim programie przestrzega tej reguły. Analiza gramatyczna, jak napisaliśmy we wstępie, to sprawdzenie składni programu, np. czy użytkownik w przypadku pętli `for` zastosował się do składni. Podał nazwę zmiennej, następnie wartość początkową, słowo kluczowe `to` i wartość ograniczającą pętlę. W powstającej tabeli instrukcji będziemy wpisywać wszystkie niezbędne informacje o zakresie podanych wartości oraz użytych zmiennych.

Lemon posiada także sekcję typu:

```
%parse_accept { ... }
```

w której podajemy odpowiedni kod, jaki zostanie wywołany, jeśli analiza gramatyczna przebiegnie poprawnie. Mamy naturalnie sekcje `%syntax_error` oraz `%parse_failure`, które odpowiednio informują nas o błędzie składni oraz niemożności dalszej analizy gramatycznej.

Listing 4. Fragmenty gramatyki naszego języka Basic

```
%include {
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "basiclexer_tokens.h"
}

%token_type {t_token_value}
%default_type {t_token_value}

%type expr {t_token_value}
%type number_line {t_token_value}

%left PLUS_TOKEN MINUS_TOKEN.
%left DIVIDE_TOKEN TIMES_TOKEN.

%extra_argument { t_ins_table *tinsptr }

%parse_accept {
printf("Analiza gramatyczna zakończona poprawnie.\n");
}

%syntax_error {
printf("Błąd składni!\n");
}
```

```
%parse_failure {
printf("Błąd krytyczny. Dalsza praca analizatora nie jest
możliwa!\n");
}

%start_symbol program

program ::= lines. { }

lines ::= line lines. { }

lines ::= line. { }

line ::= number_line(A) instruction NEW_LINE_TOKEN. {
tinsptr[insidx].line=A.value;
insidx++;
}

number_line(A) ::= DECIMAL_TOKEN(B). {
A.value = B.value;
}

instruction ::= LET_TOKEN ID_TOKEN(A) EQ_TOKEN expr(B). {
tinsptr[insidx].ins = LET_INS;
tinsptr[insidx].var1 = A.char_val;
tinsptr[insidx].val1 = B.value;
}

instruction ::= PRINT_TOKEN ID_TOKEN(A). {
tinsptr[insidx].ins=PRINT_ID_INS;
tinsptr[insidx].var1=A.char_val;
}

instruction ::= PRINT_TOKEN STR_TOKEN(A). {
tinsptr[insidx].ins=PRINT_STR_INS;
strcpy(&tinsptr[insidx].valstr[0], &A.valstr[0]);
}

instruction ::= GOSUB_TOKEN DECIMAL_TOKEN(A). {
tinsptr[insidx].ins =GOSUB_INS;
tinsptr[insidx].val1 = A.value;
}

instruction ::= FOR_TOKEN ID_TOKEN(A) EQ_TOKEN DECIMAL_TOKEN(B)
TO_TOKEN DECIMAL_TOKEN(C). {
tinsptr[insidx].ins =FOR_INS;
tinsptr[insidx].var1 = A.char_val;
tinsptr[insidx].val1 = B.value;
tinsptr[insidx].val1 = C.value;
}

instruction ::= NEXT_TOKEN ID_TOKEN(A). {
tinsptr[insidx].ins =NEXT_INS;
tinsptr[insidx].var1 = A.char_val;
}

instruction ::= RETURN_TOKEN. {
tinsptr[insidx].ins = RETURN_INS;
}

instruction ::= END_TOKEN. {
tinsptr[insidx].ins = END_INS;
}

expr(A) ::= expr(B) PLUS_TOKEN expr(C). { A.value = B.value +
C.value; }
expr(A) ::= expr(B) MINUS_TOKEN expr(C). { A.value = B.value +
C.value; }
expr(A) ::= DECIMAL_TOKEN(B). { A.value = B.value; }
```

Właściwy opis gramatyki zaczyna się od linii kodu o postaci:

```
%start_symbol program
```

Wskazujemy w tym miejscu, od której reguły zaczynamy analizę, w naszym przypadku naturalnie rozpoczynamy od symbolu `program`. Powinniśmy teraz opisać regułę gramatyczną, z jakich elementów składa się nasz program. Ponieważ analizujemy program w Basicu, to programy w tym języku składają się z linii:

```
program ::= lines. { }
```

W nawiasach klamrowych możemy umieścić dodatkowy kod, który zostanie wykonany w momencie, gdy analizator Lemon sprawdzi, czy symbol `lines` spełnia wszystkie reguły, jakie podamy.

A symbol `lines` to naturalnie poszczególne linie naszego programu. Stosujemy tu opis rekurencyjny:

```
lines ::= line lines. { }
```

Możemy to rozumieć w ten sposób, że linie naszego programu składają się z jednej linii reprezentowanej przez symbol `line` oraz innych linii (symbol `lines`). Ponieważ jednak musimy zakończyć rekurencję, więc piszemy następującą regułę o postaci:

```
lines ::= line. { }
```

oznaczającą, iż reguła `lines` może składać się też z pojedynczej linii. Natomiast pojedyncza linia została opisana w następujący sposób:

```
line ::= number_line(A) instruction NEW_LINE_TOKEN. {  
    tinsptr[insidx].line=A.value;  
    insidx++;  
}
```

W regule tej mamy zapisane następujące wymagania: każda linia tekstu programu musi zaczynać się numerem linii. Reguła `number_line`, jak spojrzymy na Listing 4, jest opisana przez wartość dziesiętną. Po numerze linii powinna pojawić się instrukcja i znak nowej linii. Jeśli analizator poprawnie zaklasyfikuje nam daną linię kodu, to wykonujemy dwie czynności. Linia kodu:

```
tinsptr[insidx].line=A.value;
```

oznacza, że wpisujemy numer linii odczytany przez analizator gramatyczny, a jest to wyrażenie `A.value`. Przy czym wartość numeru linii odczytujemy we wspomnianej regule `number_line`. Następnie zwiększamy indeks instrukcji, bo wykonanie kodu w regule `line` oznacza, iż linia kodu była poprawnie sformatowana i użyliśmy wszystkie niezbędne informacje.

Pozostałe reguły gramatyczne opisują poszczególne instrukcje, jakie znajdują się w naszym języku. Wyrażenie `let` opisujemy w następujący sposób:

```
instruction ::= LET_TOKEN ID_TOKEN(A) EQ_TOKEN expr(B). {  
    tinsptr[insidx].ins = LET_INS;  
    tinsptr[insidx].var1 = A.char_val;  
    tinsptr[insidx].val1 = B.value;  
}
```

Konstrukcja `let` składa się z następujących części: słowa kluczowego reprezentowanego przez ciąg znaków `let`, następnie występuje nazwa zmiennej `ID_TOKEN(A)`. Interesuje nas nazwa tej zmiennej, dlatego w nawiasach podaliśmy dodatkową zmienną `A`, poprzez którą możemy dostać się do informacji uzyskanych z analizatora leksykalnego (dostarczył on nam informację, iż jest to identyfikator). Język Basic w naszej odmianie stosuje tylko zmienne o literowej nazwie, więc używana litera została zapisana w polu `char_val`. Możemy też odczytać wartość wyrażenia, jaka została uzyskana po analizie za pomocą reguły `expr`. W naszej gramatyce mamy zaimplementowaną obsługę tylko operacji arytmetycznych. Jeśli jednak chcemy dopuścić stosowanie zmiennych w wyrażeniach, to musimy mieć świadomość, iż należy przygotować rozbudowaną implementację obliczenia tych wyrażeń. Nasza tablica instrukcji pozwoli jednak na implementację uproszczonego procesora z obsługą stosu, aby takie wyrażenie obliczać bez większych kłopotów.

W podobny sposób implementujemy obsługę pozostałych instrukcji. Zapis poszczególnych reguł odzwierciedla postać poleceń, choć nie zawsze każda instrukcja jest tak prosta do implementacji. Przykładem może być instrukcja warunkowa `if`. Nie mamy jej w Listingu 4, ale mogłaby ona wyglądać następująco:

```
instruction ::= IF_TOKEN LOGICAL_EXPR THEN_TOKEN instruction .  
{ }
```

Oznaczać to będzie, że po słowie kluczowym `then` będziemy mogli podać tylko jedną instrukcję. Jeśli chcielibyśmy to zmienić, to należałoby wzorem reguły dla linii kodu zbudować rekurencyjną postać reguły dla instrukcji, która pozwalałaby zapisywać poszczególne instrukcje rozdzielone średnikiem albo dwukropkiem wzorem pierwszych wersji języka Basic.

URUCHOMIENIE ANALIZY

W Listingu 5 przedstawiono proces analizy leksykalnej i gramatycznej. Kod pochodzi z pliku `minibasic.c`. W zmiennej `txtprog` powinien znajdować się program z Listingu 1. Następnie uruchamiamy analizę leksykalną za pomocą funkcji `basic_lexer`. Choć tego nie sprawdzamy, to zmienna `r` zawiera status, czy operacja analizy zakończyła się poprawnie. Następnie tworzy się parser analizy gramatycznej:

```
void *pParser = ParseAlloc (malloc);
```

Kasujemy wartość indeksu instrukcji i wypełniamy tablicę instrukcji `ins` zerami. Pętla `for` przegląda zawartość tablicy `arrTok` wy-

reklama



devstyle.pl

ŚWIAT OKIEM PROGRAMISTY

pełnionej informacjami o poszczególnych składowych naszego programu. Sprawdzamy także, czy aktualny symbol to wartość liczbową, ciąg znaków czy też symbol. Jeśli tak jest, to odpowiednio przepisujemy i konwertujemy ciąg znaków do zmiennej `t0` reprezentującej pojedynczy element naszego programu używany w analizie gramatycznej.

Wywołanie parsera następuje symbol po symbolu:

```
Parse( pParser, token_to_lemon(arrTok[i].token), t0, &ins[0]);
```

Funkcja `token_to_lemon` konwertuje wartość opisującą dane słowo kluczowe, tak jak o tym wspominaliśmy na początku akapitu Gramatyka. Zakończenie analizy gramatycznej wykonamy, wywołując następujące trzy linie kodu:

```
t0.value = 0;
t0.type = NONE_TYPE;
Parse( pParser, 0, t0, &ins[0]);
```

Możemy jeszcze usunąć obiekt parsera, ale jeśli analiza gramatyczna zakończyła się poprawnie, to tablica `ins` zawiera opis instrukcji składających się na nasz program. W tym miejscu kończy się nasze zadanie, bowiem wykonaliśmy analizę leksykalną i gramatyczną. Jeśli chcemy wykonać nasz program, to musimy napisać dodatkowy interpreter do tablicy instrukcji lub przetłumaczyć zapisane tam instrukcje na kod maszynowy rzeczywistego lub własnego procesora, a to już jest dodatkowe i równie duże zadanie jak sama analiza. Tymi zagadnieniami zajmiemy się w drugiej części tego artykułu, do lektury którego zapraszamy już w następnym numerze „Programisty”.

Warto w tym momencie podać jeszcze, w jaki sposób kompilujemy nasz kod:

```
lemon basicparser.y
re2c --tags basiclexer.re -o basiclexer.c
```

Pierwsze polecenie buduje kod źródłowy analizatora gramatycznego, drugi naturalnie analizator leksykalny. Natomiast kompilację naszego narzędzia do analizy wykonujemy w następujący sposób:

```
gcc minibasic.c -o minibasic.exe
```

W pliku `minibasic.c` możemy w sposób bezpośredni włączyć kod źródłowy analizatorów:

```
#include "basiclexer_tokens.h"
#define __main_lexer_exclude__
#include "basiclexer.c"
#include "basicparser.h"
#include "basicparser.c"
```

Definicja `__main_lexer_exclude__` nakazuje ominięcie funkcji `main`, gdzie sprawdzamy nasz analizator leksykalny.

Listing 5. Uruchomienie analizy leksykalnej i gramatycznej

```
const char* txtprog = " tekst program";
t_token_value t0;
int i, r=-1;

reset_token_idx();
r = basic_lexer( txtprog );

void *pParser = ParseAlloc (malloc);
insidx=0;
memset(&ins, sizeof(ins), 0);

for(i=0;i<get_token_idx();i++) {
if(arrTok[i].token == DECIMAL_TOK) {
t0.type=INT_TYPE;
t0.value = atoi( &arrTok[i].value.valstr[0] );
}
if(arrTok[i].token == ID_TOK) {
t0.type=CHAR_TYPE;
t0.char_val = arrTok[i].value.valstr[0];
}
if(arrTok[i].token == STR_TOK) {
t0.type=STR_TYPE;
strcpy(&t0.valstr[0], &arrTok[i].value.valstr[0]);
}

Parse( pParser, token_to_lemon(arrTok[i].token), t0,
&ins[0]);
}

t0.value = 0;
t0.type = NONE_TYPE;
Parse (pParser, 0, t0, &ins[0]);

ParseFree(pParser, free);
```

PODSUMOWANIE

Budowa narzędzi do analizy składni i gramatyki staje się znacznie łatwiejszym zadaniem, jeśli wykorzystamy dodatkowe narzędzia takie jak Lemon oraz re2c. Obecnie analizator gramatyczno-leksykalny można zbudować w ciągu jednego dnia roboczego. Choć brakuje w naszym przykładzie opisu np. interpretera, to trzeba podkreślić, że analiza składniowa oraz analiza składni naszego języka Basic jest kompletna. Jak to już zostało zapowiedziane, zajmujemy się tym problemem w drugiej części tego artykułu. Wspomnijmy jeszcze, iż względnie łatwo dodać nowe instrukcje czy wbudowane funkcje do analizatora leksykalnego i gramatyki. Jak zawsze na koniec, zachęcamy, aby to zrobić, bo to bardzo wartościowe zadanie. Istnieją naturalnie także inne rozwiązania, jak np. Gold Parser czy Antlr, które również warto sprawdzić. I na koniec przypominamy, iż wszystkie pliki przykładów omawianych w tym artykule znajdują się na stronach magazynu „Programista”.

W sieci:

- ▶ Strona pakietu Lemon: <https://www.hwaci.com/sw/lemon/>
- ▶ Główna strona pakietu re2c: <http://re2c.org/>
- ▶ Analizator Gold Parser: <http://goldparser.org/>
- ▶ Analizator Antlr: <http://wwwantlr.org/>



MAREK SAWERWAIN

Autor, pracownik naukowy Uniwersytetu Zielonogórskiego, na co dzień zajmuje się teorią kwantowych języków programowania, ale także tworzeniem oprogramowania dla systemów Windows oraz Linux. Zainteresowania: teoria języków programowania oraz dobra literatura.

Modernizacja kodu w C++

We wrześniu tego roku standard C++17 został formalnie zaakceptowany przez komitet ISO, co oznacza, że można uznać go za zamknięty. W najbliższych miesiącach wsparcie dla tego standardu zdecydowanie poprawi się i bez większych problemów będziemy mogli używać go w swoich projektach.

Pomimo tego, że nowy standard jest już na horyzoncie, to jest jeszcze wiele zespołów, które z różnych przyczyn nie mogą podjąć decyzji co do migracji do jego nowszej wersji – czasem nawet do C++11, który zdaje się być już *de facto* domyślną decyzją przy nowych projektach. Powody takich decyzji są różne – wsparcie dla różnych platform z przestarzałymi toolchainami, stary odziedziczony kod (tzw. *legacy code*) lub po prostu brak doświadczenia i pomysłu, jak wdrożyć coś takiego w projekt. W tym artykule przedstawię kilka praktycznych przykładów modernizacji starego kodu, aż do najnowszego standardu. Mam nadzieję, że lektura tego artykułu pomoże w dokonaniu decyzji o przesiadce.

W tej części przedstawię kilka tzw. *low-hanging fruits*, których wdrożenie w istniejącym projekcie jest najłatwiejsze.

TABLICE

Listing 1. Typowa deklaracja tablicy

```
#define DEF_MAXVALUES 1500
int values[DEF_MAXVALUES]
```

Częścią standardu C++11 są tablice o stałym rozmiarze, które mają dokładnie taką samą semantykę, jak ten kawałek kodu powyżej, dodatkowo wzbogacając API o metody znane z innych kontenerów biblioteki standardowej (np. `std::array::data()`). Dużą zaletą tego kontenera jest również naturalne kopiowanie zawartości tablicy przy przekazywaniu jej przez wartość. Nie musimy również pamiętać o naszej stałej `DEF_MAXVALUES`, ponieważ jest ona już częścią tablicy. Po zmianie:

Listing 2. Po zamianie poprzedniego przykładu na jego odpowiednik w bibliotece standardowej

```
std::array<int, 1500> values;
```

Stała `DEF_MAXVALUES` znika z kodu, ponieważ nie jest już potrzebna, a każde odwołanie do tej stałej możemy śmiało zamienić na `values.size()`.

Jeśli jednak mamy kilka tablic o tym samym rozmiarze, to warto zamienić `#define` na `static constexpr`:

Listing 3. Definicje preprocessora zostały zamienione na constexpr

```
static constexpr auto kMaxValues = 1500;
std::array<int, kMaxValues> values1;
std::array<int, kMaxValues> values2;
```

Nowe słowo kluczowe ma tę zaletę względem dyrektyw preprocessora, że jest ewaluowane w czasie kompilacji i typowane.

PUSTE WSKAŹNIKI

`NULL`, `0`, `(void*)NULL`, `return NULL` – kiedy mamy na myśli zwrócenie zera, a może coś jeszcze innego? No właśnie. W C++11 otrzymujemy do dyspozycji `nullptr`, który jest literałem reprezentującym pusty wskaźnik. Do tej pory znaczenie wyrażenia `NULL` było zdefiniowane przez implementację, co mogło prowadzić do różnych błędów.

Listing 4. Która funkcja zostanie wykonana?

```
void foo(char*);
void foo(int);
foo(NULL);
```

Zależnie od tego, czy `NULL` to `0`, czy `(void*)0`, to wykonana zostanie inna funkcja. Docelowo w zmodernizowanym kodzie chcemy używać `nullptr` wszędzie tam, gdzie mamy na myśli pusty wskaźnik (zamiast `NULL`, `0` itp). Unikajmy też sytuacji, w której przekazujemy liczbę (np. `0`) tam, gdzie argumentem funkcji jest wskaźnik.

UNIKAJ NEW/DELETE

Zamiast zarządzać pamięcią po swojemu, używając operatorów `new` i `delete`, skorzystajmy z dobrodziejstw standardu C++11, który dostarcza „inteligentne wskaźniki” i tzw. semantykę przenoszenia. Służą nam do tego klasy `unique_ptr`, `shared_ptr` i funkcja `std::move`.

Pierwszy z nich – `unique_ptr` – jest abstrakcją na wskaźnik o jednym właścicielu. Wynika z tego wprost, że obiektu tego typu nie można skopiować. Jediną możliwością przekazania go w inne miejsce jest wykorzystanie `std::move`. Po zakończeniu czasu życia obiekt `unique_ptr` wywołuje odpowiedni destruktor, który zwalnia zaalokowaną wcześniej pamięć. Dzięki zastosowaniu tej klasy jawnie określamy czas życia zaalokowanego obiektu (jako czas życia `unique_ptr`).

Wyobraźmy sobie uproszczoną klasę `Protocol`, która odbiera jakiś blok danych i tworzy z tego wiadomość, a wiadomość ta jest kolejkowana tak, że w innym miejscu możemy obsługiwać wiadomości w kolejności takiej, jakiej przychodzą:

Listing 5. Kto jest właścicielem obiektu `new_message`? Kto powinien zwolnić pamięć?

```
void Handler::onMessage(Message* message) {
    queue_>push(message);
}
void Decoder::onDataReceived(char* data, int size) {
    Message* new_message = new Message(data, data + size);
    handler_>onMessage(message);
}
```

TTS Company rekomenduje oprogramowanie Microsoft ®



www.OprogramowanieKomputerowe.pl

Microsoft Azure

 **Office 365**

 **OneDrive**

Więcej informacji:  **(22) 272 94 94**  **sales@tts.com.pl**

Sprzedaż



Dystrybucja



Import na zamówienie

TTS Company Sp. z o.o. ul.Domaniewska 44A 02-672 Warszawa e-mail: tts@tts.com.pl www.tts.com.pl

W powyższym przykładzie czas życia obiektu nie jest jasno określony:

- » Czy onDataReceived powinien zwolnić pamięć po wykonaniu funkcji onMessage?
- » Czy onMessage wie, że zostanie obiekt zaalokowany dynamicznie i zwolni pamięć?
- » Co ma zrobić konsument kolejki queue_? Czy powinien zwolnić te obiekty?

Listing 6. W tym przykładzie jest jasność co do tego, kto jest właścicielem pamięci i kto ją zwalnia

```
void Handler::onMessage(std::unique_ptr<Message> message) {
    queue_>push(std::move(message));
}

void Protocol::onDataReceived(char* data, int size) {
    auto new_message = std::make_unique<Message>(data, data + size);
    handler_>onMessage(std::move(message));
}
```

LAMBDA

Kwestia wyrażeń lambda była jedną z największych zmian, która pojawiła się w C++11. Dostajemy możliwość tworzenia anonimowych funkcji, które dodatkowo potrafią przechowywać lub odwoływać się do istniejących zmiennych. Lambda może zawierać w sobie dowolne instrukcje językowe, co sprawia, że kod może być o wiele czytelniejszy. W standardzie C++14 doszła również bardzo ważna zmiana umożliwiająca stosowanie słowa kluczowego auto w argumentach, sprawiając, że lambdy mogą być generyczne. Przykład:

Listing 7. Wyszukiwanie obiektów na liście z użyciem generycznych lambd

```
std::vector<std::unique_ptr<Client>> clients;
auto it = std::find_if(std::cbegin(clients),
    std::cend(clients),
    [](const auto & client) {
        return client->id == 123;
    });
```

Przed wyrażeniami lambda używanie standardowych algorytmów nie było aż tak wygodne jak jest teraz – należało próbować z funktorami, funkcjami i wyrażeniami bind. Docelowo większość z tych konstrukcji można śmiało przepisać na lambdy, w szczególności te ostatnie, co obrazuje poniższy przykład:

Listing 8. Przykładowy kod boost.asio z użyciem wyrażeń bind

```
boost::asio::async_read(
    socket_,
    boost::asio::buffer(read_buffer_),
    boost::bind(&Client::onRead,
        shared_from_this(),
        boost::asio::placeholders::error(),
        boost::asio::placeholders::bytes_
        s_transferred()));
```

W powyższym listingu wyrażenie bind jest z pewnością wygodne i potrzebne. Przedłuża ono czas życia obiektu (dzięki shared_from_this()) i przygotowuje „puste miejsca” na dwa argumenty funkcji. Tracimy niestety na czytelności kodu, a zbudowane pliki obiektowe mogą okazać się duże, z uwagi na dodatkowe typy, które tworzy bind, podczas gdy podobne wyrażenia lambda będą o wiele lżejsze.

Listing 9. Po zamianie wyrażenia bind na lambdę kod staje się czytelniejszy

```
boost::asio::async_read(
    socket_,
    boost::asio::buffer(read_buffer_),
    [sp = shared_from_this()](auto&& error,
        auto&& bytes_transferred) {
        sp->onRead(error, bytes_transferred);
    });
```

ZEGAR

Listing 10. Przykładowy, bardzo stary kod operujący na czasie

```
#include <Windows.h>

DWORD dwGlobalTime = timeGetTime();
DWORD dwActionTime1 = timeGetTime();
DWORD dwActionTime2 = timeGetTime();

void Action1();
void Action2();

void OnTimer() {
    dwGlobalTime = timeGetTime();
    if ((dwTime - dwActionTime1) > 1000 * 60 * 15) {
        Action1();
        dwAction1 = dwGlobalTime;
    }
    if ((dwTime - dwActionTime2) > 400) {
        Action2();
        dwAction2 = dwGlobalTime;
    }
}
```

Każdy może się zgodzić, że przeportowanie tego kawałka kodu na inne platformy powinno być dość trywialne – timeGetTime() można zamienić na inwokację gettimeofday, DWORD można zamienić (lub zdefiniować) jako std::uint32_t lub podobny typ, nagłówki windows.h można użyć tylko pod właściwą platformą. My natomiast zadajmy sobie następujące pytania – czy ten kod jest bezpieczny? Czy w strukturze timeval pierwszy element to sec, czy usec? Czy nikt nie pomylił się w przeliczaniu milisekund? Czy jest jasne, co się stanie po zmianie czasu w systemie? Czy na każdej platformie będzie taki sam efekt?

Do odpowiedzi na te pytania zostanie użyty nagłówek <chrono>, który został wprowadzony wraz ze standardem C++11. Dostarcza on implementację następujących zegarów:

- » steady_clock – zegar, który nigdy się nie cofa,
- » system_clock – zegar systemowy, który może się cofnąć,
- » high_resolution_clock – zegar wysokiej rozdzielczości.

Do mierzenia interwałów w kodzie najlepszym rozwiązaniem jest zegar steady_clock, ponieważ w założeniu ma być odporny na zmiany czasu w systemie. Natomiast system_clock nadaje się najlepiej do przedstawiania czasu użytkownikowi.

Częścią standardu C++11 były również dowolnie zdefiniowane literały użytkownika, jednak dopiero w C++14 mogliśmy uświadczyc praktycznego zastosowania w bibliotece standardowej – np. literał 10s tworzy std::chrono::duration z odpowiednim std::ratio. Do wyboru mamy wbudowane inne operatory: h (godziny), min (minuty), s (sekundy), ms (milisekundy), us (mikrosekundy), ns (nanosekundy).

Listing 11. Ten sam kod, ale używając chrono w połączeniu z literałami czasowymi

```
#include <chrono>

using namespace std::chrono_literals;
```

```

std::chrono::steady_clock::time_point global_time_, action_
time1_, action_time2_;

void Action1();
void Action2();

void OnTimer() {
    global_time_ = std::chrono::steady_clock::now();
    if ((global_time_ - action_time1_) > 15min) {
        Action1();
        action_time1_ = global_time_;
    }
    if ((global_time_ - action_time2_) > 400ms) {
        Action2();
        action_time2_ = global_time_;
    }
}

```

Oczywistą zaletą zmodernizowania tego kodu jest również to, że operujemy teraz na reprezentacji logicznej czasu (różnica dwóch punktów w czasie zwraca nam czas trwania, który jest osobnym typem), a nie na liczbach (oznaczające jednostkę czasu np. mili-sekundy). Dzięki zastosowaniu osobnych typów oraz charakterystycznego dla C++ silnego typowania kompilator nie pozwoli nam wymieszać interwałów z liczbami, dając dodatkową warstwę bezpieczeństwa, a z operatorami formującymi literały kod staje się czytelniejszy i łatwiejszy do zrozumienia.

LICZBY PSEUDOLOSOWE

Dotychczas najczęstszą praktyką na generowanie liczb losowych był zestaw funkcji `rand()` i `srand()`. Pierwsza funkcja służy do wygenerowania liczby z zakresu `[0..RAND_MAX]`, a drugiej używa się do inicjowania generatora liczb pseudolosowych tzw. „ziarnem” (ang. *seed*).

Listing 12. Brzmi znajomo?

```

srand(time(NULL));
for (int i = 1; i <= 10; ++i) {
    int number = rand() % 10;
    cout << number << '\n';
}

```

Liczby zwracane przez `rand()` są z zakresu `[0..RAND_MAX]`, więc jeśli chcemy wylosować wynik rzutu kostką, to najczęściej wywoływana jest funkcja `rand` z modulo `rand() % N`, ograniczając zakres do `[0..N)`. To podejście jest złe z powodu braku możliwości sprawiedliwego przeliczenia tych dwóch zakresów. Wynik losowania zawsze będzie niesprawiedliwy, kiedy `RAND_MAX + 1` nie jest podzielne przez `N`, co sprawia, że mniejsze liczby mogą być częściej losowane niż większe [STL].

Powyższe problemy rozwiązuje standard C++11, który wprowadza nagłówek `<random>` dostarczający implementację generatorów pseudolosowych i rozkładów statystycznych.

Listing 13. Bezpieczny i sprawiedliwy sposób na generowanie dobrej jakości liczb pseudolosowych z rozkładem równomiernym

```

#include <random>
#include <iostream>

int main() {
    std::random_device device;
    std::mt19937 engine{device()};
    std::uniform_int_distribution<> dist(0, 9);
    for (int i = 1; i <= 10; ++i) {
        std::cout << dist(engine) << '\n';
    }
}

```

<code>boost::thread*</code>	<code>std::thread</code>
<code>boost::shared_ptr<T></code>	<code>std::shared_ptr<T></code>
<code>boost::any</code>	<code>std::any</code> (C++17)
<code>boost::optional<T></code>	<code>std::optional<T></code> (C++17)
<code>boost::variant<T1, T2, ...></code>	<code>std::variant<T1, T2, ...></code> ^{**}
<code>boost::array<T, N></code>	<code>std::array<T, N></code>
<code>boost::atomic<T></code>	<code>std::atomic<T></code>
<code>boost::bind</code>	<code>std::bind</code> ^{***}
<code>boost::chrono</code> , <code>boost::date_time</code>	<code>std::chrono</code>
<code>boost::enable_if<bool B, T></code>	<code>std::enable_if<bool B, T></code>
<code>boost::filesystem</code>	<code>std::filesystem</code> (C++17)
<code>Boost.Foreach</code>	<code>foreach()</code>
<code>boost::function<T(Arg...)></code>	<code>std::function<T(Arg...)></code>
<code>boost::hash<T></code>	<code>std::hash<T></code>
<code>boost::move</code>	<code>std::move</code>
<code>boost::random</code>	<code>std::random</code>
<code>boost::regex</code>	<code>std::regex</code>
<code>Boost.StaticAssert</code>	<code>static_assert(...)</code>
<code>boost::tuple<T1, T2, T3...></code>	<code>std::tuple<T1, T2, T3...></code>
<code>Boost.TypeTraits</code>	<code>#include <type_traits></code>
<code>Boost.Unordered</code>	<code>std::unordered_map<K, V></code> , <code>std::unordered_set<T></code>

Tabela 1. Wybrane elementy z biblioteki Boost i ich odpowiedniki w bibliotece standardowej

* Od wersji 1.50.0 kod biblioteki `Boost.Thread` został zmodyfikowany, aby pasował do zachowania biblioteki standardowej.

** `std::variant` w odróżnieniu od `boost::variant` nigdy nie wykona dynamicznej alokacji pamięci.

*** `boost::bind` ma zdefiniowane operatory `!`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&`, `||` dla wyrażeń `bind`, ale warto zastanowić się, czy przy modernizacji nie prościej byłoby zamienić takie wyrażenia na lambdy.

Odpowiednikiem `srand` i `rand` w tym przykładzie jest generator `mersenne twister` inicjowany za pomocą niedeterministycznego źródła danych. Jest to rozwiązane poprzez odczytanie danych ze źródła hardware-owego (np. na Unixach może to być `/dev/urandom`), aczkolwiek standard zezwala implementacjom na użycie generatora liczb pseudolosowych, kiedy owe źródło danych nie jest dostępne na danej platformie. W poprzednim przykładzie zainicjowaliśmy nasz generator liczb pseudolosowych za pomocą czasu – to rodzi dodatkowe problemy:

- » `time()` zwraca `time_t`, który jest 64-bitowym typem, a `srand` przyjmuje typy 32 bitowe.
- » Używamy aktualnego czasu z dokładnością do sekund. Łatwo sobie wyobrazić rozproszoną aplikację, która może uruchamiać się w kilkuset kopiach w tej samej chwili – każde kolejne wywołanie funkcji `rand` zwróci ten sam wynik w każdym procesie¹.

1. Można się dodatkowo zabezpieczyć przed tym problemem i użyć chociażby wyrażenia `time()*getpid()` do zainicjowania ziarna, ale nie zmienia to faktu, iż jego jakość jest nadal niska w porównaniu z wymienionym niedeterministycznym źródłem danych.

Samo sprawiedliwe losowanie liczb polega na wywołaniu rozkładu równomiernego z kolejną wartością serwowaną nam po przez wybrany przez nas generator liczb pseudolosowych.

Pod funkcją `rand` kryje się generator liniowy, który może różnić się parametrami pomiędzy platformami. Jego odpowiednikiem w bibliotece standardowej jest `std::minstd_rand0` lub po prostu `std::default_random_engine`, który jest gwarantowany przez standard do produkowania zwykłych, „codziennych” liczb pseudolosowych.

Boost

Kiedyś standard w projektach C++, z uwagi na ilość bibliotek, które posiada. Z racji bogatego wsparcia różnych środowisk używany był do pierwszych prób modernizacji istniejącego kodu. Aktualnie większość z tych rzeczy, które wprowadza boost, znajduje się prawie w niezmienionej formie w bibliotece standardowej (Tabela 1).

NARZĘDZIA

Żałujemy, że odziedziczyliśmy bardzo starą aplikację, która ma 100 tysięcy linii kodu. Po pierwszej próbie zbudowania w co drugiej linii otrzymujemy ostrzeżenia i błędy od kompilatora.

Z pomocą przychodzą narzędzia, które pomagają nam refaktoryzować kod w sposób automatyczny. Jednym z takich narzędzi jest `clang-tidy`, który operuje na abstrakcyjnym drzewie składni i na podstawie swojej bazy wbudowanych testów wykrywa problemy w kodzie. Integracja jest banalnie prosta, zakładając, że nasz projekt buduje się poprzez CMake.

Pierwszym krokiem w integracji `clang-tidy` w projekcie będzie utworzenie pliku `.clang-tidy` w głównym folderze w repozytorium kodu źródłowego. Na potrzeby przykładu utworzymy domyślny plik konfiguracyjny, który wygeneruje nam `clang-tidy` (lista wszystkich dostarczonych reguł i ich opcji konfiguracyjnych znajduje się tutaj: <http://clang.llvm.org/extra/clang-tidy/checks/list.html>):

Listing 14. Domyślny plik wygenerowany przez wywołanie polecenia `clang-tidy`

```
clang-tidy -dump-config | tee .clang-tidy
---
Checks:          'clang-diagnostic-*,clang-analyzer-*'
WarningsAsErrors: ''
HeaderFilterRegex: ''
AnalyzeTemporaryDtors: false
FormatStyle:     none
CheckOptions:
- key:  google-readability-braces-around-statements.ShortStatementLines
  value: '1'
- key:  google-readability-function-size.StatementThreshold
  value: '800'
```

```
- key:  google-readability-namespace-comments.ShortNamespaceLines
  value: '10'
- key:  google-readability-namespace-comments.SpacesBeforeComments
  value: '2'
- key:  modernize-loop-convert.MaxCopySize
  value: '16'
- key:  modernize-loop-convert.MinConfidence
  value: reasonable
- key:  modernize-loop-convert.NamingStyle
  value: CamelCase
- key:  modernize-pass-by-value.IncludeStyle
  value: llvm
- key:  modernize-replace-auto-ptr.IncludeStyle
  value: llvm
- key:  modernize-use-nullptr.NullMacros
  value: 'NULL'
...
```

Następnie uruchommy CMake z dodatkową opcją `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON`, a po utworzeniu projektu (np. poprzez domyślny generator Makefile) zostanie stworzony również plik `compile_commands.json`, który zawiera listę inwencji kompilatora dla każdego pliku źródłowego włącznie z flagami. Dzięki temu `clang-tidy` wie, skąd wziąć kod źródłowy i jak go przeparsować, zanim uruchomi skonfigurowane reguły.

Uruchomienie tego programu polega na wywołaniu dostarczonego skryptu `run-clang-tidy.py`, który automatyzuje narzędzie `clang-tidy` z właściwymi flagami, bazując na bazie kompilacji i pliku `.clang-tidy`. Jeśli korzystamy z opcji budowania poza kodem źródłowym w CMake, to najlepszym rozwiązaniem byłoby włączenie skryptu `run-clang-tidy.py` z katalogu z kodem źródłowym i dodanie do skryptu flagi `-p ". /build"`, zakładając, że `./build` to katalog budowania. Wtedy mamy pewność, że nasz plik konfiguracyjny zostanie poprawnie załadowany.

Po skonfigurowaniu tego narzędzia spróbujmy zrobić tak, aby te testy były uruchamiane jako część testów wykonywanych po każdym commicie w repozytorium. Niektórzy są zwolennikami przeprowadzania takich testów jeszcze przed dodaniem plików do repozytorium, np. w tzw. *hooku* `pre-commit` w gicie, tak że kod, który nie przechodzi wszystkich skonfigurowanych reguł, nie może być opublikowany.

PODSUMOWANIE

Należy pamiętać, że jakiegokolwiek wielkoskalowe zmiany w kodzie wiążą się z potencjalnym zepsuciem czegoś, co mogło działać. Najpierw zainwestujmy w automatyczne testy, które pozwolą nam powiedzieć, że dany kawałek kodu działa tak samo po zmianach. Przy bardzo rozbudowanych projektach nie próbujmy zmieniać wszystkiego naraz, ponieważ ilość błędów i ostrzeżeń kompilatora może być zbyt duża. Na początek zidentyfikujmy potencjalne problemy, a wsparcie dla nowych kompilatorów dodajmy jako opcję, i tam, gdzie możemy, użyjmy biblioteki standardowej.

Bibliografia

[STL] – Stephan T. Lavavej – `rand()` Considered Harmful: <https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful>

MICHAŁ PAPIERSKI

michal@papierski.net

Programista C++. Absolwent Polsko-Japońskiej Akademii Technik Komputerowych. Aktualnie pracuje nad technologiami automotive w amerykańskiej firmie. Strona autora: <http://michal.papierski.net/>.

SZUKASZ PRACY MARZEŃ?



ZRÓB PIERWSZY KROK!

PRZYJDŹ DO SZKOŁY JĘZYKA ANGIELSKIEGO SPEAK UP

- 🕒 NAUCZYMY CIĘ ANGIELSKIEGO SZYBKO I SKUTECZNIE
- ▶ STAWIAMY NA TWÓJ ROZWÓJ POPRZECZ INNOWACYJNE METODY NAUCZANIA
- 🔗 GWARANTUJEMY ELASTYCZNOŚĆ I INDYWIDUALNE PODEJŚCIE
- 🏆 Z NAMI ZDOBĘDZIESZ CERTYFIKAT JĘZYKOWY

WWW.SPEAK-UP.PL



Przygoda z wokselowym silnikiem gry. Część 1

Na rynku można znaleźć bardzo dużo zaawansowanych silników gier, które pozwalają na szybkie stworzenie gry bez zbytniego zastanawiania się: „jak to działa pod spodem?”. Czasami twórcy gier decydują się na stworzenie własnego silnika, ponieważ te dostępne na rynku nie nadają się do gry, jaką sobie wymarzyli. Często też napisanie własnego silnika pozwala lepiej doszlifować finalny produkt poprzez rezygnację z komponentów, które są dla nas zbędne. Ogólnodostępne silniki posiadają ich wiele, ponieważ muszą objąć swoimi możliwościami jak najwięcej gier, które potencjalni developerzy będą tworzyć za ich pomocą. Jednym z takich przypadków, gdzie warto spróbować napisać własny silnik gry, może być gra opierająca się na woksela. A na pewno będzie to świetny pretekst, by spróbować napisać własny silnik dla celów edukacyjnych.

CZYM ZAJMIEMY SIĘ W ARTYKULE

O tym, jak napisać cały silnik od podstaw, można by napisać całą książkę – nawet jeżeli jest tak prosty jak nasz, dlatego w tym artykule skupimy się na kilku rzeczach, w szczególności tych oscylujących wokół wokseli.

Listingi zawarte w artykule są napisane w C++, podobnie jak cały silnik, dlatego znajomość tego języka może być bardzo przydatna, jednak nie niezbędna do zrozumienia algorytmów i rozwiązań, które zastosowałem.

Przy czytaniu kodu silnika gry przydatna będzie znajomość podstawowych operacji matematycznych stosowanych w grafice 3D – nie będzie tego dużo, jednak niektóre elementy kodu bez tej wiedzy mogą wydać się niezrozumiałe.

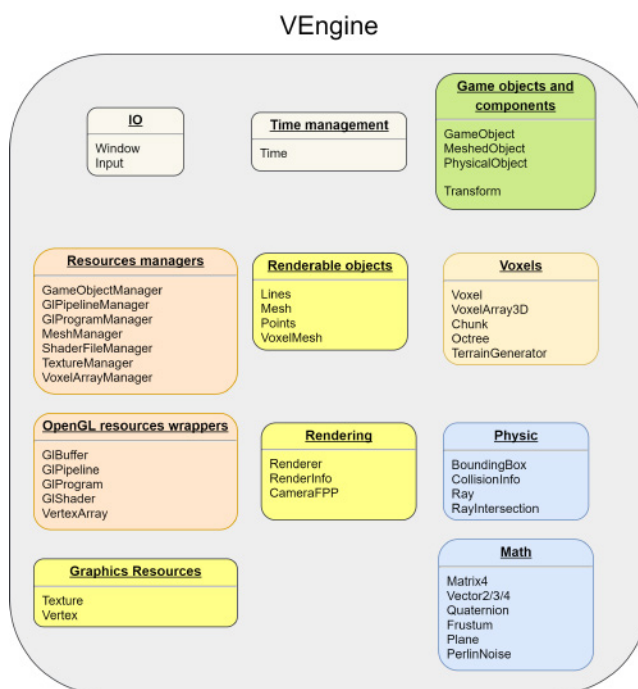
Nie będziemy się zagłębiać we wszystkie zagadnienia związane z grafiką 3D – podejmiemy jedynie najciekawsze tematy, a bardziej podstawową wiedzę postaram się chociaż nieco przybliżyć czytelnikom, którzy swojej przygody z grafiką 3D jeszcze nie zaczęli. Przygotowałem kilka klas pomocniczych, opakowujących API OpenGL, dlatego też, drogi Czytelniku, jego znajomość nie jest wymagana do zrozumienia treści artykułu, jednak pojawiają się pewne nawiązania do samego OpenGLa.

Artykuł będzie podzielony na dwie części, przy czym w pierwszej skupimy się na wyświetlaniu wokseli, w drugiej natomiast na elementach silnika gry oraz prostej fizyce.

Na początku przedstawię komponenty, z jakich składa się mój silnik, następnie zastanowimy się nad woksela – co to takiego jest i jakie są związane z nim problemy, stworzymy optymalną siatkę wierzchołków na podstawie tablicy wokseli, zastanowimy się, jak nałożyć tekstury na nasze woksela, wygenerujemy prosty teren złożony z wokseli, zrobimy sobie bardzo krótki wstęp do potoku graficznego oraz języka GLSL (OpenGL Shading Language), aż w końcu napiszemy proste shadery, które będą potrafiły wyświetlać nasze woksela. Cały kod silnika można znaleźć w repozytorium GitHub, do którego link znajduje się na końcu artykułu. Zachęcam do zapoznania się z nim osobom bardziej zainteresowanym tematem. No to czas start!

BUDOWA SILNIKA

Listingi, które zaprezentuję poniżej, byłyby bardzo ciężkie do analizy w oderwaniu od kontekstu. Dlatego zanim przedstawię jakiś kod, który będzie w większości implementacją pewnych komponentów silnika wokselowego, zobaczmy, z czego on się składa. Roboczo nazwałem go VEngine (od Voxel Engine). Najważniejsze komponenty zostały przedstawione na Rysunku 1.



Rysunek 1. Najważniejsze elementy silnika wokselowego VEngine

Menedżery zasobów

Silnik, na którym zostanie w przyszłości zbudowana gra, musi posiadać bazę danych dla różnych zasobów – czy to tekstur, shaderów,



Największy wybór profesjonalnego oprogramowania w Polsce !

... w ofercie programy ponad 500 producentów ...



www.OprogramowanieKomputerowe.pl



Więcej informacji:

(22) 868 40 42



sales@tts.com.pl

Sprzedaż



Dystrybucja



Import na zamówienie

TTS Company Sp. z o.o.

ul.Domaniewska 44A

02-672 Warszawa

www.tts.com.pl

czy prototypów obiektów służących jako szablon do tworzenia gotowych obiektów w grze. W naszym silniku do zarządzania bazą danych służą klasy znajdujące się w kategorii `Resources managers`. Nie będę się zagłębiał w implementację tego rozwiązania, gdyż sercem tego artykułu są woksele. Ponadto istnieje wiele sposobów implementacji tego rozwiązania (a mój niekoniecznie jest najlepszy), jednak chciałbym przedstawić kilka słów wyjaśnienia, w jaki sposób moje klasy działają.

Menedżery zasobów pozwalają na przechowywanie obiektów dziedziczących po klasie `Resource`. Można odwoływać się do tych obiektów po nazwie lub poprzez specjalny obiekt `Handle<T>`. Jest to po prostu unikalny indeks obiektu wraz z pewną magiczną liczbą, pozwalającą sprawdzić stan danych. Z pomocą tego uchwytu możemy w łatwy sposób sprawdzić, czy odwołujemy się do nieistniejących zasobów (np. do takich, które zostały zwolnione w innym miejscu w kodzie) oraz zlokalizować ich pozycję w menedżerze.

Menedżery pozwalają przechowywać dokładnie jedną instancję danego obiektu w jednym miejscu. Dla przykładu: posiadamy pewną siatkę wierzchołków reprezentującą kilof. Nie musimy powielać tych wierzchołków dla każdego obiektu, który jest kilofem. Możemy przechowywać siatkę w menedżerze, a w obiekcie reprezentującym kilof trzymać jedynie uchwyt. Każdy zasób identyfikowany jest dodatkowo przez nazwę, zatem śledzenie zasobów i ich identyfikacja nie powinna nastręczyć wiele problemów.

Zasobami, które możemy przechowywać w menedżerze, są:

- » `GLPipeline` i `GLProgram` (skompilowane shadery) oraz pliki z kodem shadera.
- » `GameObject` – szablony do instancjonowania obiektów w świecie gry (fabryka tworząca obiekty gry na podstawie wzorca).
- » Tekstury – my będziemy korzystać z jednego atlasu (pojedynczy plik ze wszystkimi teksturami) do tekstuowania wokseli, ale istnieje możliwość rozbudowania silnika o wyświetlanie normalnych modeli 3D – wtedy ten menedżer może przechowywać bardzo dużą ilość tekstur lub różne atlasy, które będą zmieniane dynamicznie.
- » Tablice wokseli `VoxelArray3D` – reprezentują obiekty stworzone z wokseli (np. wokselowy miecz lub kilof albo model przeciwnika).
- » Modele 3D – siatka modelu, złożona z trójkątów.

Obiekty gry

Silnik rozróżnia 3 kategorie obiektów gry:

- » `GameObject` – obiekt logiczny,
- » `MeshedObject` – obiekt, który posiada siatkę do wyświetlenia,
- » `PhysicalObject` – obiekt fizyczny, tj. taki, który może oddziaływać z innymi poprzez kolizje.

Każdy obiekt gry posiada komponent `Transform`, który jest reprezentacją położenia obiektu w świecie gry. Więcej na temat obiektów gry, komponentu `Transform` oraz wykrywania kolizji znajdzie się w drugiej części artykułu.

Biblioteka matematyczna

Silnik posiada własną bibliotekę matematyczną, która dostarcza podstawowe struktury niezbędne do wyświetlania grafiki 3D, takie jak `Matrix4` i `Vector4` oraz pomocniczo `Vector3` i `Vector2`.

Do reprezentacji obrotów wykorzystałem kwaterniony z kilku powodów. Jednym z nich jest to, że kwaterniony są powszechnie wykorzystywane w innych silnikach gier (i nie tylko), a ich imple-

mentacja jest bardzo dobrym treningiem pozwalającym na lepsze zrozumienie tej struktury algebraicznej. Drugim powodem jest uniwersalność w porównaniu do kątów Eulera (yaw, pitch, roll, czyli obrót wzdłuż każdej z osi), ponieważ nie występuje tu zjawisko blokowania przegubu (ang. *gimbal lock*). Ponadto potrzebujemy mniej miejsca do przechowywania obrotu niż w przypadku macierzy (ale więcej niż w przypadku kątów Eulera). Kwaternion składa się z 4 liczb reprezentujących obrót w przestrzeni trójwymiarowej (w, x, y, z), gdzie w reprezentuje obrót, a x, y, z to liczby zespolone reprezentujące oś obrotu. Zatem mamy tylko 4 wartości do przechowywania. Do składania operacji przekształcenia w grafice 3D wykorzystuje się macierze – kwaternion można przekonwertować do macierzy obrotu przy konieczności wygenerowania odpowiedniej macierzy transformacji.

Do obsługi klasy `Quaternion` nie jest wymagana teoretyczna znajomość kwaternionu. Klasa dostarcza pomocniczych funkcji, pozwalających na obrót za pomocą kątów Eulera, a także obrót punktu o dany kąt.

Klasy `Plane` i `Frustum` służą do implementacji metody nazywanej *frustum culling*, czyli przycinania niewidocznych krawędzi. Pozwala to na pominięcie wyświetlania obiektów, które nie znajdują się wewnątrz bryły widoku.

`PerlinNoise` to generator tzw. Szumu Perlina w trzech wymiarach, o czym więcej opowiem w dalszej części artykułu.

Enkapsulacja OpenGL

API OpenGL daje programiście dużą swobodę w działaniu. Jest to duża zaleta, jednak wymaga to opanowania całego API i zwiększa czas niezbędny do programowania.

Tworząc silnik gry, nie potrzebujemy aż takiej uniwersalności – najprawdopodobniej z większości opcji nie będziemy w ogóle korzystać, inna część operacji będzie wykonywana za każdym razem tak samo. Dlatego warto tutaj postawić na wygodę tworzenia kosztem uniwersalności. Pozwoli to na łatwiejszą modyfikację silnika oraz umożliwi projektantom gry szybsze jej tworzenie, gdyż będą mogli się skupić na tworzeniu samej gry zamiast na OpenGLu.

Do osiągnięcia takiego poziomu abstrakcji OpenGL stworzyłem klasę `Renderable`, która na podstawie deskryptora `RenderInfo` steruje shaderami oraz maszyną stanów OpenGL do uzyskania odpowiedniego efektu, opisanego przez deskryptor.

Listing 1. Definicja struktury `RenderInfo` opisującej sposób wyświetlania danego obiektu

```
struct RenderInfo {
    GLuint indicesNumber; // Number of indices

    bool textured;        // Should the object be textured - if false
                          // it will be colored
    unsigned int tex;      // What texture should it use - handle from
                          // textureManager

    bool wired;           // Should the object be drawn wired

    Vector3 color;         // Color of the object if not textured

    GLenum drawType;      // GL_POINTS, GL_TRIANGLES etc.
    float pointSize;      // Size for GL_POINTS
};
```

Sam obiekt `RenderInfo` jest uzupełniany przez klasy dziedziczące po klasie `Renderable`. Klasa ta ma też dodatkowe zadanie – enkapsuluje zasoby OpenGL, które są wymagane do rysowania,

przez co użytkownik musi jedynie wywołać metodę Draw na odpowiednim obiekcie dziedziczącym po Renderable, po wypełnieniu go uprzednio wierzchołkami.

Do określenia, w jaki sposób obiekt ma być rysowany, należy zaimplementować dwie funkcje wirtualne: Draw oraz FillInfo. Przykład takiej implementacji znajduje się w Listingu 2.

Listing 2. Implementacja metod FillInfo oraz Draw dla klasy Mesh dziedziczącej po Renderable

```
void
Mesh::FillInfo(RenderInfo* info)
{
    Renderable::FillInfo(info);
    if (_wireFrame) {
        info->wired = true;
        info->color = _wireColor;
    }
    else {
        info->wired = false;

        if (_textured && _tex != 0 ) {
            info->tex = _tex;
            info->textured = true;
        }
        else {
            info->tex = 0;
            info->textured = false;
        }
    }
    info->drawType = GL_TRIANGLES;
}

void
Mesh::Draw(Renderer* renderer)
{
    RenderInfo info;

    DrawStart(&info);

    renderer->Draw(info, Renderer::STANDARD);

    DrawEnd();
}
```

Implementując funkcję Draw, najpierw musimy wywołać funkcję DrawStart – jest to funkcja klasy Renderable, które zajmuje się przygotowaniem zasobów OpenGL do rysowania. Następnie należy wywołać metodę Draw obiektu Renderer, przekazując jako argument deskryptor oraz wybierając odpowiedni *pipeline* – czyli obiekt reprezentujący potok graficzny, co zostanie dokładniej wytłumaczone w części artykułu traktującej o potoku graficznym. Klasa Renderer posiada kilka rodzajów pipeline'ów, np. `Renderer::STANDARD` dla normalnych wierzchołków, albo `Renderer::VOXEL` dla wierzchołków stworzonych z wokseli. Na końcu, by zwolnić odpowiednie zasoby, należy wywołać funkcję DrawEnd klasy Renderer. Funkcja FillInfo jest wywoływana w DrawStart.

Wejście/wyjście oraz czas

Input, Window oraz Time to klasy będące wrapperami wokół biblioteki GLFW [5], którą wykorzystałem do generycznego dostępu do okien i kontekstu rysowania, oraz wejścia z klawiatury, myszy i licznika czasu.

Window pozwala zarządzać oknem, w którym znajduje się kontekst rysowania dla OpenGL, tworzyć okna o określonej wielkości lub wyświetlać grę w postaci pełnoekranowej. Input dostarcza informacji nt. naciśniętych klawiszy oraz ruchu myszy, a Time pozwala określić przepływ czasu między kolejnymi klatkami. Prawidłowy przepływ czasu jest niezbędny w grze do poprawnego

aplikowania fizyki, ruchu itd., by wyniki były niezależne od czasu renderowania pojedynczej klatki. Gdybyśmy tego nie robili, to, w przypadku bardzo szybkiego komputera, gra działałaby w przyspieszonym tempie, a w przypadku komputera bardzo wolnego gra wydawałaby się spowolniona.

Zasoby wokselse

Nasz silnik wokselsewy musi posiadać kilka struktur dedykowanych zarządzaniu i reprezentacji wokseli. Wszystkie elementy zostaną wytłumaczone dokładnie w dalszej części artykułu (w tym części drugiej), dlatego w tym momencie przedstawię tylko ogólny zarys odpowiednich komponentów.

Klasa Voxel reprezentuje pojedynczy woxel, VoxelArray3D grupę wokseli w trzech wymiarach, a Chunk to specjalny typ VoxelArray3D, który jest zawsze sześcienny i reprezentuje teren w grze.

Octree to drzewo ósemkowe, które ma dwa zadania: organizuje obiekty typu Chunk oraz pomaga w wykrywaniu kolizji.

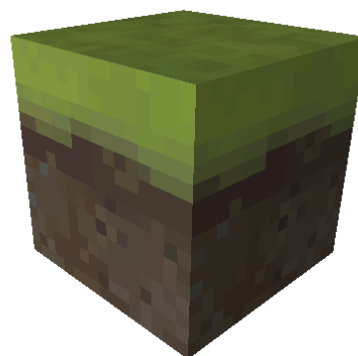
TerrainGenerator to obiekt, który pozwala generować wokselsewy świat za pomocą Szumu Perlina.

WOKSELE

Co to są wokselse?

Zastanówmy się, czym w ogóle są wokselse (ang. *voxel*). Jeśli wiemy, czym jest piksel, to najbardziej intuicyjnym przedstawieniem woksela jest piksel w przestrzeni 3D. Czyli jest to najmniejszy obiekt, „atom” świata trójwymiarowego, który przechowuje jakąś informację. Są one powszechnie używane w medycynie, m.in. przy analizie wyników Rezonansu Magnetycznego Jądrowego, jednak znajdują też zastosowanie w branży gier komputerowych wszędzie tam, gdzie wymagamy od świata dodatkowej responsywności i interaktywności.

Gry komputerowe to pewnego rodzaju symulacja, a każda symulacja przyjmuje pewne założenia i uproszczenia świata – w taki sposób, by model wynikowy był jak najbliższy rzeczywistości (oczywiście przy pewnych założeniach). W naszej symulacji możemy uprościć nieco otaczającą nas rzeczywistość, zakładając, że świat (teren) może składać się jedynie z „atomów”, które są sześciannymi o wielkości 1x1x1. Wymiary te są podawane w jednostkach, których rozmiar może różnić się w zależności od obiektu – czyli np. tworząc teren z wokseli, możemy przyjąć, że jedna jednostka to 1 metr, a tworząc kilof, jedna jednostka może być równa 10 cm. Takie podejście pozwoli pokazać więcej detali na obiekcie.



Rysunek 2. Reprezentacja graficzna pojedynczego woksela typu Voxel::GRASS

Każdy woksel może posiadać pewne dodatkowe abstrakcyjne informacje, jak np. oświetlenie, punkty wytrzymałości, typ, temperaturę itd. Wszystko zależy od tego, co chcemy uzyskać.

W Listingu 3 przedstawiono przykładową definicję woksela, zastosowaną w silniku VEngine

Listing 3. Klasa reprezentująca woksel wykorzystywany w silniku VEngine

```
class Voxel {
public:
    // Sides of the voxels, used for determining atlas textures
    // for each face
    enum Side {
        NORTH = 0,
        SOUTH,
        EAST,
        WEST,
        TOP,
        BOTTOM
    };

    // Types of the voxel
    enum Type {
        NONE = 0,
        DIRT,
        GRASS,
        WOOD,
        STONE,
        LEAFS,
        CLOTH1,
        CLOTH2,
        BODY,
        SKELETONFACE,
        SKELETON,
        NUM_TYPES = SKELETON
    };

    Voxel();
    Voxel(const Voxel& source);
    Voxel(unsigned char type);

    void SetType(unsigned char type);
    unsigned char GetType() const;

    bool IsEmpty() const;
    bool operator==(const Voxel& other) const;
    bool operator!=(const Voxel& other) const;

    // Get x and y coordinates of the face from the atlas texture.
    // It is hardcoded.
    void GetAtlasLocation(Side face, int* x, int* y) const;

    operator unsigned char();

    // Constant information for atlas texturing. How many textures
    // can be stored in each row of the atlas.
    static const int texsPerRow = 16;
    bool IsTransparent() const;

private:
    unsigned char _type;

    // Structure containing information for each face which texture
    // from atlas should it use
    static const struct AtlasCoords {
        int x;
        int y;
    } _offsets[NUM_TYPES * 6]; // And multiply by 6 - for each
    side
};
```

Woksel w naszym silniku będzie przechowywał informacje jedynie o swoim typie, co będzie wpływało na różną reprezentację graficzną danego woksela. Każda ścianka woksela będzie rozróżniana, co pozwoli przypisać osobną teksturę do każdej z nich, jak np. na Ry-sunku 2, gdzie górna ściana ma inną teksturę niż ścianki boczne. Informacja o tym, gdzie szukać tekstury dla każdej ścianki woksela, jest zapisana w programie w tablicy `Voxel::_offsets`, gdzie każdy indeks tablicy odpowiada danemu typowi woksela.

Implementując silnik wokselowy, należy wziąć przede wszystkim pod uwagę pamięć. Dla przykładu weźmy świat złożony z 4096x4096x4096 wokseli. Do przechowania w pamięci takiej struktury będziemy potrzebowali co najmniej 68 719 476 736 bajtów. Co najmniej, ponieważ zakładamy, że woksel przechowuje jedynie informacje o swoim typie i całość znajduje się w ciągłym obszarze pamięci. Jest to 64 GB pamięci, mimo że świat wcale nie jest ogromny. Ilość potrzebnej pamięci będzie rosła wykładniczo wraz ze wzrostem świata w postaci funkcji $y = x^3$, gdzie w przypadku obrazów i pikseli mamy wzrost pamięci równy $y = x^2$.

A co z pozostałymi danymi? Jeśli chcielibyśmy przechowywać dodatkowe 5 bajtów informacji w każdym wokselu, wynik należy zwiększyć pięciokrotnie. Nawet jeśli uda nam się jakimś cudem znaleźć na to miejsce w pamięci, to pozostaje problem wierzchołków. Pojedynczy woksel to 6 ścian, dla każdej ściany należy wygenerować 2 trójkąty (6 wierzchołków lub 4 wierzchołki i 6 indeksów, ale przyjmijmy, dla uproszczenia, że generujemy 6 wierzchołków). Każdy wierzchołek to 3 współrzędne X, Y, Z, z czego każda będzie przedstawiona za pomocą liczby całkowitej lub zmiennoprzecinkowej, czyli 4 bajty... Zatem musimy nanieść poprawkę do naszego wyniku i należy go pomnożyć przez $6 \cdot 6 \cdot 3 \cdot 4 = 432$... Nasze domowe pecety raczej nie dadzą sobie z tym rady.

Część tego kosztu musimy ponieść, można jednak jego część ograniczyć poprzez zmniejszenie rozmiaru świata lub przechowywanie nieużywanej części świata na dysku w postaci skompresowanej (niektóre bezstratne algorytmy kompresji obrazu mogłyby być tutaj przydatne). My zastosujemy pierwszą metodą – przyjmijmy stały rozmiar świata, niezbyt duży, by całość mogła spokojnie zmieścić się w pamięci średniej klasy peceta.

W dalszych rozdziałach przedstawie możliwe sposoby optymalizacji wyświetlania oraz przechowywania naszych wokseli.

Grupowanie wokseli w chunki

Powszechną praktyką jest grupowanie wokseli w zbiory nazywane z języka angielskiego *chunks*. Zyskujemy na tym lepszą organizację danych poprzez podzielenie wielkiego problemu na wiele mniejszych – np. zamiast analizować świat 256x256x256 wokseli, analizujemy 16x16x16 wokseli w obrębie każdego chunka (z 16x16x16 chunkami w świecie).

Założeniem naszego silnika jest możliwość modyfikacji terenu w trybie rzeczywistym oraz interaktywność, dlatego istotny jest rozmiar pojedynczego chunka – jeśli będzie on za duży, to czas generowania nowej siatki dla danego chunka będzie na tyle wysoki, że użytkownik zauważy opóźnienie. Jeśli będzie on natomiast zbyt niski, to nadmierne rozdrobnienie naszego problemu spowoduje dodatkowe narzuty czasowe oraz pamięciowe, a także zmniejszy zysk z algorytmów optymalizacyjnych w obrębie pojedynczego chunka.

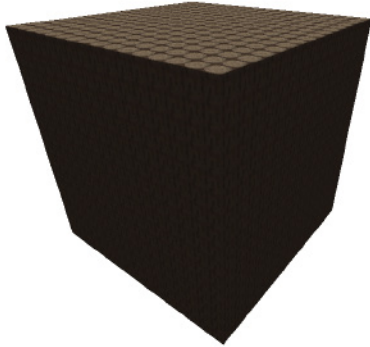
Chunk można zdefiniować albo jako tablicę trójwymiarową, albo w postaci jednowymiarowej tablicy, dla której lokalizacja danego woksela w tablicy będzie obliczana na żądanie – to drugie rozwiązanie może być nieco bardziej przyjazne dla pamięci *cache*.

Dla przypomnienia: w naszym silniku Chunk to specjalizacja klasy `VoxelArray3D`, której rozmiar wzdłuż każdej osi jest taki sam.

Meshing, czyli woksela ukaż się

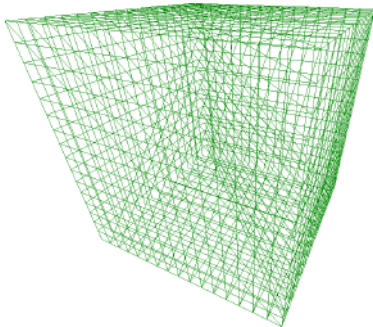
Założmy, że mamy grupę wokseli. Jak już obliczyliśmy wcześniej, potrzebujemy na każdy woksel 8 wierzchołków. Założyliśmy, że analizujemy wokselowy obiekt o wielkości 16x16x16, który jest

w pełni wypełniony. Łącznie mamy 4096 wokseli. Ilość ścian, dla których musimy wygenerować wierzchołki, to $16^3 \cdot 6 = 24\,576$. Ponadto musimy te dane przesłać do karty graficznej, co wiąże się z dodatkowym narzutem czasowym. Rezultat, jaki chcemy otrzymać, znajduje się na Rysunku 3.



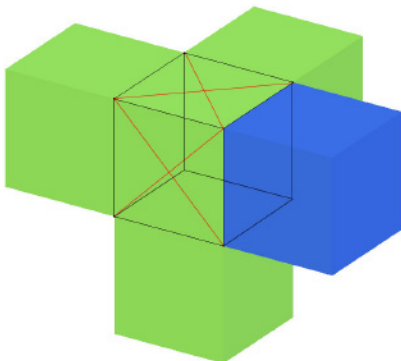
Rysunek 3. Model wygenerowany z grupy wokseli 16x16x16 tego samego typu

Warto w tym miejscu zastanowić się nad optymalizacją generowania modelu. Podejście naiwne, przedstawione na Rysunku 4, jest mało efektywne, ponieważ zostają wyrenderowane ściany, które nigdy nie będą dla danego chunka widoczne.



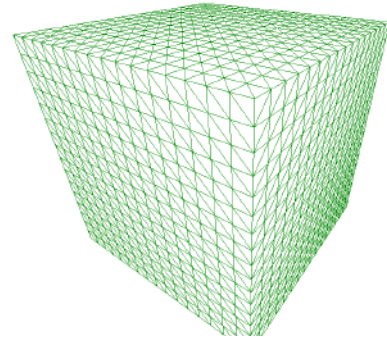
Rysunek 4. Siatka modelu przy podejściu naiwnym – generowaniu wierzchołków dla każdego woksela

Pierwszą zaproponowaną przeze mnie metodą optymalizacji będzie sprawdzenie, czy woxel jest otoczony nieprzezroczystymi woksłami. Możemy pominąć generowanie wierzchołków dla takich ścian jak na Rysunku 5.



Rysunek 5. Reprezentacja metody polegającej na usuwaniu zasłoniętych ścian – przekątnymi zaznaczono ściany, dla których zostaną wyrenderowane wierzchołki

W najlepszym przypadku, gdy nasze woksły tworzą solidny sześcian, zostaną wygenerowane tylko zewnętrzne ściany naszego woksłowego sześcianu, czyli $16^2 \cdot 6 = 1\,536$. Im więcej dziur w środku naszego obiektu, tym wyniki będą gorsze. W najgorszym przypadku, kiedy stworzymy szachownicę w 3D, wygenerujemy wierzchołki dla wszystkich woksli i wynik wcale nie będzie lepszy od podejścia naiwnego, ale jest to dość skrajna sytuacja (12 288 ścian).



Rysunek 6. Siatka modelu przy zastosowaniu optymalizacji polegającej na usuwaniu zasłoniętych ścian woksła

Wróćmy do pierwotnego przypadku, czyli całkowicie wypełnionego woksłowego sześcianu. Można łatwo zauważyć, że na każdej ścianie na Rysunku 6 posiadamy wiele trójkątów, leżących na tej samej płaszczyźnie i sąsiadujących ze sobą. A może by tak połączyć je ze sobą i zamiast 1024 wierzchołków dla jednej ściany woksłowego sześcianu, przechowywać ich jedynie 4? Zysk pamięci, w najbardziej korzystnym dla algorytmu przypadku, byłby bardzo wysoki.

Z pomocą przychodzi nam algorytm Greedy Mesh [4]. Polega on na tym, że analizujemy nasz chunk wzdłuż każdego kierunku $\pm X$, $\pm Y$, $\pm Z$, a następnie, dla każdej płaszczyzny, scalamy ściany woksli tego samego typu. Uwzględniamy przy tym zawartość poprzedniej płaszczyzny, by nie renderować ścian, które są przysłaniane przez nieprzezroczyste woksły. W ten sposób dostajemy połączenie poprzedniego podejścia, wraz ze scalaniem woksli tego samego typu.

Implementacja algorytmu znajduje się w Listingu 4 i jest ona częścią klasy `VoxelArray3D` w naszym silniku.

Listing 4. Implementacja algorytmu Greedy Mesh w silniku VEngine

```
void
VoxelArray3D::GenerateMesh(VoxelMesh* mesh)
{
    assert(IsValid(), "Cannot generate mesh for empty Voxel
    Array.");

    mesh->ClearVertices();

    int total_quads = 0;
    bool front = false;
    for (int i = 0; i < 2; ++i, front = !front) {
        for (int dir = 0; dir < 3; dir++) {

            int u = (dir + 1) % 3;
            int v = (dir + 2) % 3;

            const Voxel** mask = new const Voxel*[_dimension[u] *
            _dimension[v]];
            memset(mask, NULL, _dimension[u] * _dimension[v] *
            sizeof(Voxel *));
            int coords[3] = { 0, 0, 0 };
            // This arrays helps to determine next plane for
            // face comparison
            int next[3] = { 0, 0, 0 };
            next[dir] = 1;
```

```

Voxel::Side face;
// Determine which face to use - it can also be used if
// voxels could be rotated
switch (dir) {
// Axis X
case 0:
    face = front ? Voxel::EAST : Voxel::WEST;
    break;
// Axis Y
case 1:
    face = front ? Voxel::TOP : Voxel::BOTTOM;
    break;
// Axis Z
case 2:
    face = front ? Voxel::NORTH : Voxel::SOUTH;
    break;
}

// For each layer
coords[dir] = -1;
while (coords[dir] < _dimension[dir]) {

    /* Creating mask */
    int n = 0;
    for (coords[v] = 0; coords[v] < _dimension[v];
        coords[v]++) {
        for (coords[u] = 0; coords[u] < _dimension[u];
            coords[u]++) {
            // Get earlier and current layer, we must
            // be careful to do not exceed voxel array
            // size
            const Voxel* earlier = (coords[dir] >= 0) ?
                &Get(coords[0], coords[1], coords[2]) :
                nullptr;
            const Voxel* current = (coords[dir] < _dimension[dir]
                - 1) ?
                &Get(coords[0] + next[0], coords[1] +
                next[1], coords[2] + next[2]) :
                nullptr;

            // If object is air, store it as nullptr
            if (earlier != nullptr && earlier->IsEmpty())
                earlier = nullptr;
            if (current != nullptr && current->IsEmpty())
                current = nullptr;

            // Merge blocks if it is possible
            if (current != nullptr && earlier != nullptr &&
                !earlier->IsTransparent() &&
                !current->IsTransparent())
                mask[n++] = nullptr;
            else
                mask[n++] = front ? earlier : current;
        }
    }

    coords[dir]++;

    // Create layers from mask
    n = 0;
    int w, h;
    for (int i = 0; i < _dimension[v]; i++) {
        for (int j = 0; j < _dimension[u]; j++) {

            if (mask[n] != nullptr) {
                total_quads += 1;
                // Width first, choosing different order can give
                // different result, example: letter T, depending on
                // order it can be meshed with 2 or 3 quads
                for (w = 1; j + w < _dimension[u]; ++w) {
                    int index = n + w;
                    // If faces are different, we cannot merge them

                    if (mask[index] == nullptr ||
                        *mask[index] != *mask[n])
                        break;
                }

                // Height
                bool finish = false;
                for (h = 1; i + h < _dimension[v]; h++) {
                    for (int k = 0; k < w; k++) {
                        int index = n + k + h * _dimension[u];
                        if (mask[index] == nullptr ||

```

```

                            *mask[index] != *mask[n]) {
                                finish = true;
                                break;
                            }
                        }
                    }
                    if (finish) {
                        break;
                    }
                }

                coords[u] = j;
                coords[v] = i;

                int du[3] = { 0, 0, 0 };
                int dv[3] = { 0, 0, 0 };
                du[u] = w;
                dv[v] = h;

                // Create points for the quad
                Vector3 p0 = Vector3(float(coords[0]),
                    float(coords[1]),
                    float(coords[2]));
                Vector3 p1 = Vector3(float(coords[0] + du[0]),
                    float(coords[1] + du[1]),
                    float(coords[2] + du[2]));
                Vector3 p2 = Vector3(float(coords[0] + du[0] +
                    dv[0]),
                    float(coords[1] + du[1] + dv[1]),
                    float(coords[2] + du[2] + dv[2]));
                Vector3 p3 = Vector3(float(coords[0] + dv[0]),
                    float(coords[1] + dv[1]),
                    float(coords[2] + dv[2]));

                // Push quad into the mesh
                InsertQuad(mesh, *mask[n], face, front, w, h,
                    p0, p1, p2, p3);

                // Removing used elements
                for (int l = 0; l < h; ++l) {
                    for (int k = 0; k < w; ++k) {
                        mask[n + k + l * _dimension[u]] = NULL;
                    }
                }

                // Advance
                j += w;
                n += w;
            }
        }
    }

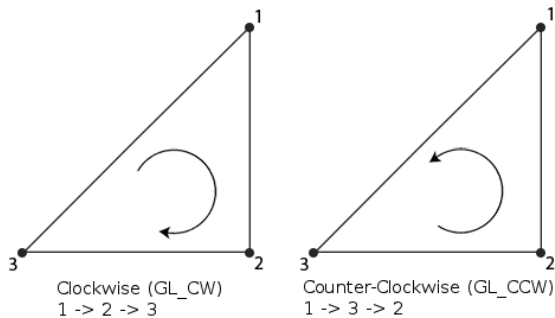
    delete[] mask;
}

// If didn't add any quads, indicate that this voxel array is empty
if (total_quads == 0)
    _empty = true;
}

```

W algorytmie najpierw określamy kierunek, w którym będziemy się poruszali. Każdy ściana woksela posiada swoją orientację, ponieważ zakładamy możliwość nakładania innej tekstury na górę woksela, jego spódzie i dla każdego z boków. Dodatkowo określamy, czy dana ściana jest frontowa, czy nie. Na przykład ściany wschodnie, górne i północne będą ścianami frontowymi, zgodnie z prawoskrętnym układem współrzędnych (układ współrzędnych używany w OpenGL). Określenie ścian frontowych pozwala nam na pominięcie rysowania tylnej części ścian, która byłaby widoczna jedynie z wnętrza woksela (a taka sytuacja nie powinna mieć miejsca ze względu na kolizje). Jest to powszechna praktyka wykorzystywana w grach komputerowych 3D. Wszystko polega na kolejności, w jakiej podajemy wierzchołki trójkąta, dla OpenGL domyślnie przód trójkąta jest wtedy, gdy rysujemy trójkąt przeciw-

nie do ruchu wskazówek zegara, a tył, gdy rysujemy go w kierunku przeciwnym. Obie sytuacje zaprezentowano na Rysunku 7.

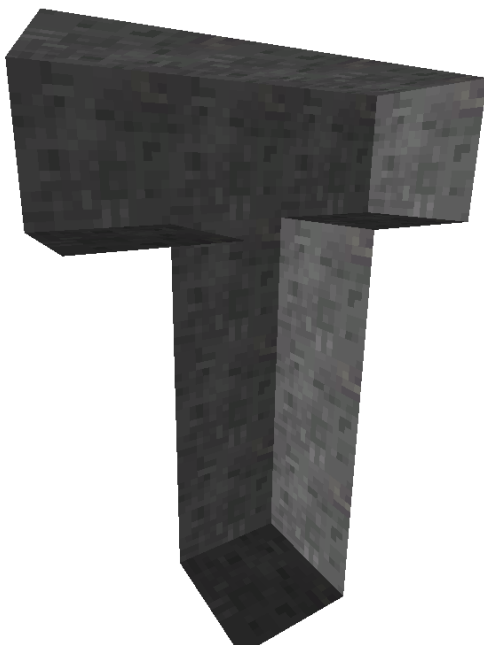


Rysunek 7. Po lewej: trójkąt rysowany zgodnie z ruchem wskazówek zegara (domyślnie tylna ściana); po prawej: trójkąt rysowany przeciwnie do ruchu wskazówek zegara (domyślnie przednia ściana), źródło [3]

Przy wykorzystaniu odpowiedniego API OpenGL (Listing 5) możemy włączyć tę opcję. Nasz silnik pracuje w tym trybie domyślnie. Możemy zweryfikować działanie tej opcji poprzez wejście do środka modelu. Jeśli nie będziemy widzieć jego przeciwnych ścian, to znaczy, że opcja działa poprawnie. Jeśli natomiast widzimy tylko przeciwległe ściany, to oznacza, że podaliśmy wierzchołki trójkąta w złej kolejności. Przykład takiej złej konfiguracji możemy zobaczyć na Rysunku 8.

Listing 5. Włączanie obcinania tylnych ścian w OpenGL

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```



Rysunek 8. Renderowanie litery T złożonej z wokseli wierzchołków podanych w odwrotnej kolejności

Po określeniu kierunku rysowania należy stworzyć maskę, która będzie wskazywać na woksely, które nie są przysłonięte przez inne, nieprzezroczyste. Warstwa poprzednia musi być wzięta z uwzględnieniem kierunku, w którym się poruszamy. Jeśli jest to pierwsza warstwa, to uznajemy, że poprzednia warstwa składa się z samych pustych wokseli.

W tym momencie maska definiuje aktywne woksely na naszej płaszczyźnie i możemy spróbować połączyć kolejne, sąsiadujące woksely ze sobą. Idziemy w jednym kierunku (tutaj nazwanym szerokością w maski), dopóki nie natkniemy się na pusty wokslel lub wokslel innego typu. W tym momencie mamy już pewien prostokąt o szerokości w i wysokości 1. Teraz będziemy rozszerzać nasz prostokąt w górę. Musimy sprawdzić, czy każdy potencjalny rząd wokseli, który chcielibyśmy przyłączyć do naszego istniejącego prostokąta, jest jednolity i tego samego typu. Jeśli tak, to przyłączamy go i idziemy wyżej, jeśli nie, to mamy gotową płaszczyznę.

Generujemy wierzchołki dla niej i konwertujemy czworokąt na dwa trójkąty oraz dodajemy je do naszego modelu mesh.

Jakich danych będziemy potrzebować do naszego wierzchołka? Zastanówmy się, co będzie nam potrzebne, by stworzyć trójkąt.

W pierwszej kolejności będzie nam potrzebna pozycja wierzchołka. Może to być pozycja względna, np. względem środka modelu, ponieważ potem w shaderze można wykonać odpowiednią transformację poprzez pomnożenie współrzędnych wierzchołka przez macierz modelowania, która będzie reprezentować pozycję całego modelu w świecie gry, a nie pojedynczego wierzchołka. Uznaliśmy też, że będziemy nakładać teksturę na nasze woksely, dlatego będziemy potrzebowali przechowywać informacje o miejscu w rastrze tekstury, z którego w danym punkcie ma być pobrana tekstura, zatem nasz wierzchołek będzie musiał przechowywać wartości u i v .

Dla osób słabo zaznajomionych z tematyką grafiki komputerowej kilka słów wyjaśnień na temat tekstur. Każda tekstura 2D posiada dwie współrzędne, którymi można opisać dowolny punkt w teksturze. Są to współrzędne UV, gdzie u to szerokość, a v to wysokość tekstury. Punkt (0, 0) to lewy górny róg tekstury. Czasem współrzędna v może być odwrócona, zależnie od sposobu, w jaki wczytujemy obraz – należy to uwzględnić w shaderze/silniku. Ja wykorzystałem do tego bibliotekę SOIL2, która pozwala wczytać obraz, odwracając oś Y, co umożliwia normalne mapowanie.

Współrzędne UV przyjmują wartości z zakresu [0.0, 1.0], gdzie 0.0 to pierwszy texel (czyli podstawowa jednostka w teksturze, na podobieństwo piksela w obrazie), a 1.0 to ostatni texel. Ponieważ ilość pikseli w rastrze opisującym teksturę jest ograniczona, a zakres współrzędnych UV taki nie jest, istnieją różne sposoby interpolacji wartości texeli pomiędzy nimi, tzw. filtrowanie tekstur. Można zastosować liniową interpolację, co spowoduje, że tekstura w powiększeniu będzie rozmyta, lub wybrać najbliższego sąsiada, gdzie rezultatem będzie wyraźny efekt pikselizacji, co w przypadku mojego silnika będzie lepszym wyborem ze względu na styl artystyczny, jaki zdecydowałem się przyjąć. Ponadto, w przypadku gdy wyjdziemy poza zakres [0.0, 1.0], można przyjąć różne strategie, jaką wartość przyjąć – można powtarzać wartości z zakresu [0.0, 1.0], ignorując to, że jesteśmy poza zakresem (operacja modulo 1.0), można też powtórzyć ostatni texel z ramki tekstury, albo przyjąć, że takie texele przyjmują stałą wartość, np. są czarne lub białe.

Wróćmy do naszego wierzchołka. Zdecydowałem się na użycie w moim silniku atlasu tekstur, co będzie opisane w dalszej części artykułu, zatem musimy także przechowywać współrzędne tekstury w atlasie. Będzie to kolejne pole dla naszego wierzchołka.

Ostatnią niezbędną rzeczą, by nadać scenie głębi, będzie normalna dla danego wierzchołka. Pozwoli ona na nierównomierne oświetlenie trójkątów, w zależności od kierunku normalnej w stosunku do kierunku światła będzie się zmieniać intensywność oświetlenia – jeśli normalna będzie wskazywać w dokładnie przeciwnym kierunku do kierunku światła, nasz wierzchołek będzie

w pełni oświetlony. Jeśli natomiast kierunek ten będzie taki sam jak kierunek światła, wierzchołek nie będzie oświetlony tym światłem w ogóle. Normalne można obliczać na różne sposoby, co przekłada się na różną interpretację obrazu, jednak to temat na inny artykuł. U nas normalne w danym wierzchołku będą zawsze prostopadłe do płaszczyzny, którą opisuje dany wierzchołek.

Skoro już ustaliliśmy, czego będziemy potrzebować, czas na definicję wierzchołka w naszym silniku, która znajduje się w Listingu 6.

Listing 6. Definicja wierzchołka w silniku VEngine

```
struct Vertex
{
    Vector3 position;
    Vector3 normal;
    Vector2 texUV;
    int atlasOffset[2];
    Vector4 color;
};
```

Funkcja generująca wierzchołki znajduje się w Listingu 7. Możemy zauważyć, że generujemy tylko 4 wierzchołki oraz 6 indeksów. Nie musimy powielać wierzchołków, które są reprezentowane przez te same dane – trójkąty można rysować poprzez przekazanie indeksów wierzchołków, które się na niego składają. Kolejność wierzchołków jest określana na podstawie orientacji ściany – jeśli to nie jest jasne, proponuję jeszcze raz spojrzeć na Rysunek 7. Pozycja wierzchołka jest obliczana na podstawie rozmiaru woksela. Normalne muszą być ustalone w zależności od orientacji ściany. Na samym końcu funkcja mapuje tekstury w odpowiedni sposób – proponuję powrócić do tej części listingu po zapoznaniu się z następnym akapitem nt. atlasu tekstur oraz mapowania tekstur – wtedy ta część kodu powinna być dużo bardziej zrozumiała.

Listing 7. Funkcja konwertująca scalone ściany woksela do wierzchołków tworzących trójkąty gotowe do wyrenderowania

```
void
VoxelArray3D::InsertQuad(VoxelMesh* mesh, const Voxel& voxel,
Voxel::Side side,
    bool front, int w, int h, const Vector3& p0,
    const Vector3& p1, const Vector3& p2, const Vector3& p3)
{
    Vertices vertices(4);
    Indices indices(6);
    int indicesOffset = mesh->GetVertices().size();

    // Order used for culling faces
    if (front) {
        indices[0] = indicesOffset + 2;
        indices[1] = indicesOffset + 3;
        indices[2] = indicesOffset;

        indices[3] = indicesOffset;
        indices[4] = indicesOffset + 1;
        indices[5] = indicesOffset + 2;
    }
    else {
        indices[0] = indicesOffset;
        indices[1] = indicesOffset + 3;
        indices[2] = indicesOffset + 2;

        indices[3] = indicesOffset + 2;
        indices[4] = indicesOffset + 1;
        indices[5] = indicesOffset;
    }

    // Calculate position. Model should be in center of the voxel array
    // and add info about voxel size
    vertices[0].position = p0 * _voxelSize - _center;
    vertices[1].position = p1 * _voxelSize - _center;
    vertices[2].position = p2 * _voxelSize - _center;
    vertices[3].position = p3 * _voxelSize - _center;
```

```
// Set normals depending on the side
switch (side) {
case Voxel::NORTH:
    for(int i = 0; i < 4; ++i)
        vertices[i].normal = Vector3::forward;
    break;
case Voxel::SOUTH:
    for (int i = 0; i < 4; ++i)
        vertices[i].normal = Vector3::backward;
    break;
case Voxel::EAST:
    for (int i = 0; i < 4; ++i)
        vertices[i].normal = Vector3::right;
    break;
case Voxel::WEST:
    for (int i = 0; i < 4; ++i)
        vertices[i].normal = Vector3::left;
    break;
case Voxel::TOP:
    for (int i = 0; i < 4; ++i)
        vertices[i].normal = Vector3::up;
    break;
case Voxel::BOTTOM:
    for (int i = 0; i < 4; ++i)
        vertices[i].normal = Vector3::down;
    break;
}
unsigned char type = voxel.GetType();

int u, v;
// Get texture location in the atlas for the given type
voxel.GetAtlasLocation(side, &u, &v);

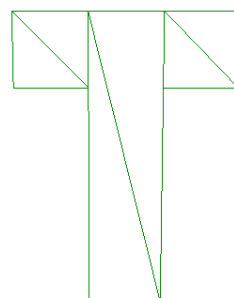
for (int i = 0; i < 4; ++i) {
    vertices[i].atlasOffset[0] = u;
    vertices[i].atlasOffset[1] = v;
}

// For EAST and WEST u v must be setted in different way or they
// will be flipped
bool yAxe = side == Voxel::EAST || side == Voxel::WEST;

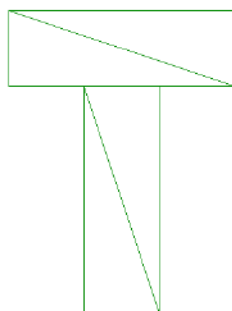
vertices[0].texUV.Set(0.0f, 0.0f);

if (yAxe) {
    vertices[1].texUV.Set(0.0f, float(w));
    vertices[2].texUV.Set(float(w), float(h));
    vertices[3].texUV.Set(float(h), 0.0f);
}
else {
    vertices[1].texUV.Set(float(w), 0.0f);
    vertices[2].texUV.Set(float(w), float(h));
    vertices[3].texUV.Set(0, float(h));
}
mesh->AddVertices(vertices, indices);
}
```

Rezultatem algorytmu nie zawsze będzie najbardziej optymalny kształt. Przykładem może być litera T. W zależności od tego, w jaki sposób rozpoczniemy poszukiwania kolejnych czworokątów (wzdłuż lub wszereż), możemy otrzymać albo najbardziej optymalny podział składający się z 2 czworokątów, albo mniej optymalny, składający się z 3 czworokątów.

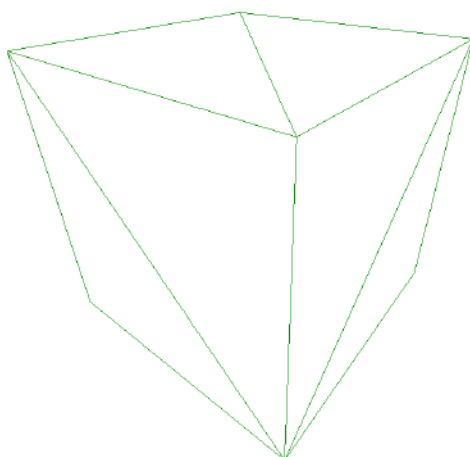


Rysunek 9. Nieoptymalne powierzchnie dla litery T w algorytmie Greedy Mesh



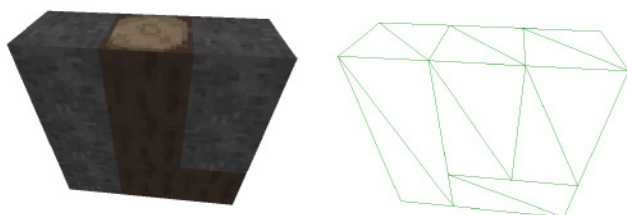
Rysunek 10. Optymalne powierzchnie dla litery T w algorytmie Greedy Mesh

Model dla naszego pierwotnego problemu, czyli wypełnionego sześcianu, zaprezentowano na Rysunku 11. Na wyświetlenie takiego modelu potrzebujemy dokładnie tyle samo wierzchołków co na wyświetlenie pojedynczego woksela!



Rysunek 11. Siatka wypełnionego modelu 16x16x16 przy zastosowaniu algorytmu Greedy Mesh

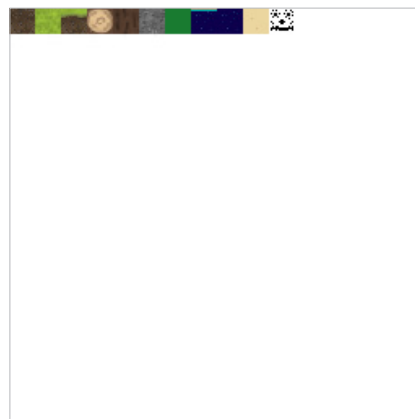
W analizowanym przypadku wszystkie woksela były tego samego typu. Należy mieć na uwadze, że różne typy wokseli będą od siebie odseparowane, co przedstawiono na Rysunku 12, jednak mimo to rezultaty algorytmu są wciąż bardzo zadowalające. W prostym prototypie gry, który stworzyłem na potrzeby prezentacji silnika, przy generowanym proceduralnie terenie o rozmiarze 256x256x256 wokseli (zapełnionym do mniej więcej połowy wysokości), uzyskałem wykorzystanie pamięci rzędu 94 MB. Przy tych samych warunkach testowych, ale zastosowanym algorytmie naiwnym do generowania siatki, program wykorzystywał 425 MB pamięci.



Rysunek 12. Siatka modelu stworzona za pomocą algorytmu Greedy Mesh przy użyciu różnych typów wokseli

Atlas tekstur oraz teksturowanie modelu

W artykule wiele razy pojawiło się odwołanie do atlasu tekstur, który jest używany w silniku. Cóż to jest, ten atlas? Jest to kolejna baza danych, z tą różnicą, że tym razem jest to baza danych tekstur przechowywana w jednym pliku rastrowym.



Rysunek 13. Atlas tekstur użyty w silniku mający tekstury dla wszystkich wokseli – posiada jeszcze bardzo wiele przestrzeni do wypełnienia

Celem, dla którego tak komplikujemy sobie życie, jest optymalizacja. Każdy, kto kiedykolwiek grał w jakąś grę komputerową, która była słabo zoptymalizowana, wie, dlaczego tak ważna jest płynność rozgrywki i warto o nią zabiegać w jak największej ilości miejsc programu. Atlas stosowany jest często w grach komputerowych, gdzie istnieje wiele różnych tekstur, niekoniecznie bardzo wysokiej rozdzielczości, które są często zmieniane – dowiązywanie tekstur do kontekstu OpenGL kosztuje, a posiadając jeden plik ze wszystkimi teksturami, pozbywamy się tego kosztu.

Atlas użyty w naszym silniku ma rozdzielczość 256x256 pikseli, a szerokość i wysokość jednej tekstury to 16 pikseli. Łącznie nasz atlas pozwala przechowywać 256 różnych tekstur. Zastosowanie atlasu do przechowywania tekstur dla woksela ścian ma takie zalety, że możemy wykorzystać całe dostępne miejsce w atlasie – każda tekstura będzie kwadratowa i takiego samego rozmiaru, a wprowadzenie nowego zestawu tekstur wymaga jedynie podmienienia pojedynczego pliku.

Jednak istnieje kilka komplikacji przy implementacji algorytmu Greedy Mesh. Wyobraźmy sobie, że scaliliśmy rząd złożony z pięciu ścian woksela. Jak już wcześniej pisałem, gdy przekroczymy współrzędne u lub v tekstury, można to zignorować i wykonać operację modulo – czyli wynikiem współrzędnej $u = 1.1$ będzie ten sam texel, co dla $u = 0.1$, przy niezmiennym v . Gdy stosujemy atlas, operacja nieco się komplikuje i metodę zapętlania musimy zaimplementować sami, w shaderze, ponieważ $u = 0.0$ to pierwszy texel atlasu, a $u = 1.0$ to jego ostatni texel. Podanie zakresu, np. $u_0 = 0.1$, $u_k = 0.2$, mogłoby zadziałać, gdybyśmy teksturowali tylko jedną ścianę, ale tutaj mamy do czynienia ze scalaniem ścian. Gotowy kod tej operacji znajduje się w Listingu 11, w którym przedstawiono nasz shader fragmentów. Cały kod shadera może nie być jeszcze zrozumiały, jednak postaram się opisać koncepcję, która kryje się za tym mapowaniem.

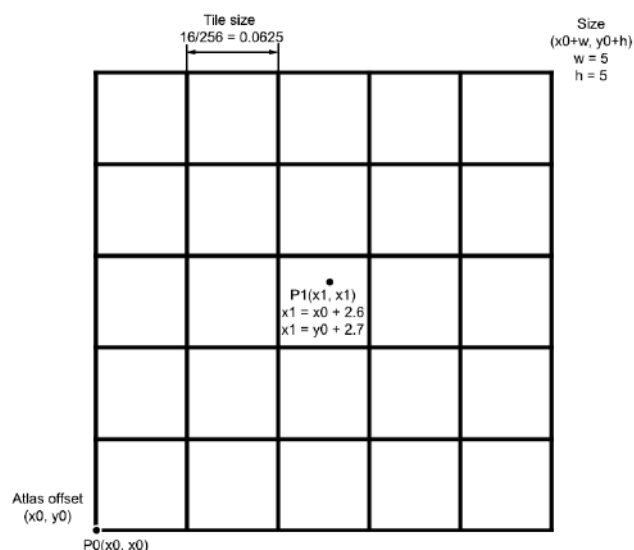
W pierwszej kolejności należy dostarczyć do shadera znormalizowaną szerokość pojedynczej tekstury w atlasie (w moim przypadku $\text{tileSize} = 16\text{px}/256\text{px} = 0.0625$). Będzie ona określać szerokość i wysokość jednego kafelka tekstury we współrzędnych UV. W shaderze mamy dostęp do zmiennych `texCoord` i `texOffset`.

`texCoord` to dwuelementowy wektor, przyjmujący wartości pomiędzy $[0.0, w]$, $[0.0, h]$, który określa wartość współrzędnych UV w przypadku normalnego teksturowania. Jeśli chcemy oteksturować wokselowy prostokąt o szerokości w i wysokości h wokseli, to przy użyciu zwykłej tekstury wierzchołek reprezentujący jego lewy górny róg powinien przyjąć wartość UV = (0, 0), a prawy dolny róg UV = (w, h). Czyli w i h określają, ile razy dana tekstura ma się powtórzyć wzdłuż i w szerz. `texOffset` to indeksy, które są współrzędnymi tekstury w atlasie. Wartości te otrzymujemy na podstawie danych znajdujących się w wierzchołkach, które uzupełniliśmy w Listingu 7. Czyli `texOffset` odpowiada wartości pola `atlasOffset` w strukturze `Vertex`, a `texCoord` odpowiada wartości pola `texUV`. Jak to się dokładnie dzieje, opiszę przy okazji omawiania shaderów.

Posiadamy współrzędne tekstury w atlasie oraz współrzędne, które mogłyby być użyte do mapowania pojedynczej tekstury. Postarajmy się znaleźć punkt P_0 , czyli punkt początkowy kafelka tekstury, który chcemy użyć. Możemy tego dokonać poprzez pomnożenie indeksów `texOffset` reprezentujących pozycję tekstury przez rozmiar kafelka `tileSize`. W ten sposób otrzymamy punkt P_0 w jednostkach UV atlasu. Następnym krokiem będzie wybranie odpowiedniego texela z kafelka, po którym się poruszamy. Skoro `texCoord` służy do mapowania pojedynczej tekstury, odseparujemy ją. Dokonajmy zapętlenia współrzędnych `texCoord` poprzez wykonanie operacji modulo 1.0, a następnie sprowadźmy otrzymaną wartość do przestrzeni UV atlasu – czyli pomnożmy przez `tileSize`. W ten sposób dla UV = (0, 0) otrzymamy punkt P_0 , a dla UV = (1, 1) otrzymamy punkt $P_k = P_0 + \text{tileSize}$, czyli ostatni texel danego kafelka w atlasie. Powyższe dywagacje sprowadzają się do poniższych przekształceń:

- » $\text{resultUV.x} = (\text{texOffset.x} * \text{tileSize}) + \text{mod}(\text{texCoord.x}, 1.0) * \text{tileSize}$,
- » $\text{resultUV.y} = (\text{texOffset.y} * \text{tileSize}) + \text{mod}(\text{texCoord.y}, 1.0) * \text{tileSize}$.

Całość, dla przykładowej scalonej płaszczyzny 5x5 wokseli, została zaprezentowana graficznie na Rysunku 14.



Rysunek 14. Mapowanie tekstur dla powierzchni scalonych algorytmem Greedy Mesh

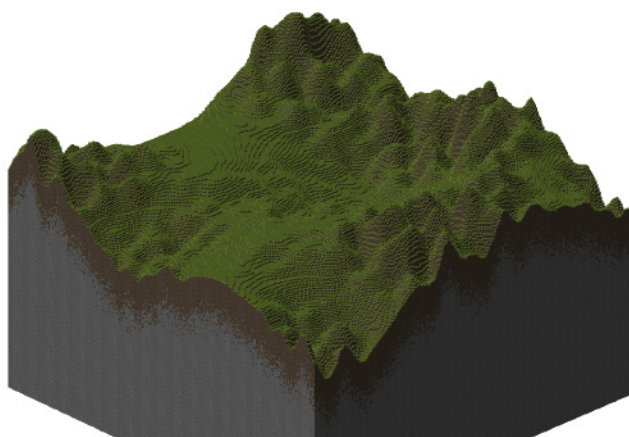
Proceduralnie generowany teren – szum Perlina

Potrąfimy już tworzyć siatkę wokselowych modeli. Teraz stwórzmy jakiś świat.

Jaki powinien być teren? Na pewno nie płaski. Nie powinien być też powtarzalny i monotony. Jeśli jakiś szczęściarz spojrzy teraz przez okno na górski lub pagórkowaty krajobraz zamiast na centrum miasta, zauważy, że w ukształtowaniu terenu widać pewną harmonię, jednak brak mu powtarzalności – ciężko znaleźć dwa identyczne pagórki obok siebie. Dodatkowo teren powinien być spójny i gładko przechodzić od jednej struktury do drugiej. Chcielibyśmy, aby teren mógł być generowany losowo. Taki rodzaj szumu, który przypomina zjawiska występujące w naturze, możemy nazwać szumem organicznym. Położenie wokselu powinno być przypadkowe, ale w skali lokalnej zależne od położenia poprzedniego wokselu.

Idealnym rozwiązaniem będzie użycie Szumu Perlina, używanego bardzo często w branży gier komputerowych do uzyskania właśnie takiego organicznego szumu (np. nierówności powierzchni przy mapowaniu wypukłości czy też generowanie tekstur ognia). Implementację generatora dostarczam wraz z silnikiem i nie będę opisywał jego definicji matematycznej. Bardziej skupię się na tym, co można dzięki niemu uzyskać.

Generator Szumu Perlina tworzy szum w trzech wymiarach. Pozwala to na uzyskanie jaskiń i nieregularności wewnątrz terenu. Ja zdecydowałem się skorzystać z jego dwuwymiarowej postaci (tj. przyjmuję stałą wartość gradientu Y). Nie uzyskam w ten sposób bogatych struktur podziemnych, ale będę w stanie wygenerować różnorodną rzeźbę terenu w zależności od kilku parametrów. Przykład takiego terenu można zaobserwować na Rysunku 15.



Rysunek 15. Przykładowy teren wygenerowany za pomocą dwuwymiarowego Szumu Perlina

Klasa `TerrainGenerator` pozwala sterować generowanym terenem. Jest on tworzony przy użyciu trzech różnych szumów – dwóch o niskich częstotliwościach, pierwszy, *height*, to globalna wysokość terenu, a drugi *rough* zostanie pomnożony przez szum o wysokiej częstotliwości *detail*, reprezentujący detale. Nałożenie tych dwóch szumów da nam wysokość wynikową.

Istnieje kilka parametrów, które dają dodatkową kontrolę nad generowanym terenem:

- » *smoothness* – określa, jak bardzo teren ma być gładki w skali globalnej – zwiększenie tego współczynnika spowoduje, że teren będzie bardziej równinny i będzie sprawiał wrażenie bardziej rozpiętego.
- » *details* – ilość detali lokalnych, czyli lokalnych gór, dołków itd.

- » spread – steruje rozpiętością terenu w górę i dół; jak bardzo stromy ma być cały teren.
- » seaOffset – jest to poziom morza, zmniejszenie tego parametru spowoduje, że cały teren zostanie wygenerowany niżej.
- » rockOffset – jak głęboko pod ziemią mają być generowane skały – dokładna pozycja danej skały jest losowo określana za pomocą rozkładu normalnego.
- » rockSharpness – wyższa wartość tego parametru oznacza wyższą rozbieżność w głębokości, na jakiej będą generowane skały w terenie.

Implementacja algorytmu generującego woksele dla danego chunka znajduje się w Listingu 8.

Listing 8. Proceduralne generowanie wokseli dla danego chunka za pomocą Szumu Perlina

```
bool
TerrainGenerator::GetChunk(Chunk* source)
{
    const int size = Chunk::dimension;
    const Vector3& offset = source->GetOffset();
    int added = 0;

    // First, generate height map for whole chunk
    int heightMap[size][size];
    for (int z = 0; z < size; ++z) {
        for (int x = 0; x < size; ++x) {
            float height = _perlinGenerator.GetNoise(
                (offset.x + x) / _smoothness,
                (offset.z + z) / _smoothness,
                0.2f);
            float rough = _perlinGenerator.GetNoise(
                (offset.x + x) / _smoothness,
                (offset.z + z) / _smoothness,
                0.5f);
            float detail = _perlinGenerator.GetNoise(
                (offset.x + x) / _details,
                (offset.z + z) / _details,
                0.3f);

            height = (height + (rough * detail)) * _spread + _seaOffset;
            heightMap[z][x] = (int)height;
        }
    }

    // Now, for each voxel in the chunk we must check:
    for (int z = 0; z < size; ++z) {
        for (int x = 0; x < size; ++x) {
            int height = heightMap[z][x];

            for (int y = 0; y < size; ++y) {
                int index = x + size * (y + z * size);
                // Generate rock level from normal distribution
                int rocks = (int)_distribution(_generator);

                // Calculate height of the chunk in world coordinates
                int chunkHeight = y + (int)offset.y;

                // And similar like for single voxel: if below air level
                if (chunkHeight < height) {
                    // And if below rock level draw stone
                    if (chunkHeight < height - 1 - rocks)
                        source->SetLocal(x, y, z, Voxel::STONE);
                    // else if at top, draw grass
                    else if (chunkHeight == height - 1)
                        source->SetLocal(x, y, z, Voxel::GRASS);
                    // else just draw dirt
                    else
                        source->SetLocal(x, y, z, Voxel::DIRT);

                    // Indicate that we added one voxel
                    ++added;
                }
                else {
                    source->SetLocal(x, y, z, Voxel::NONE);
                }
            }
        }
    }

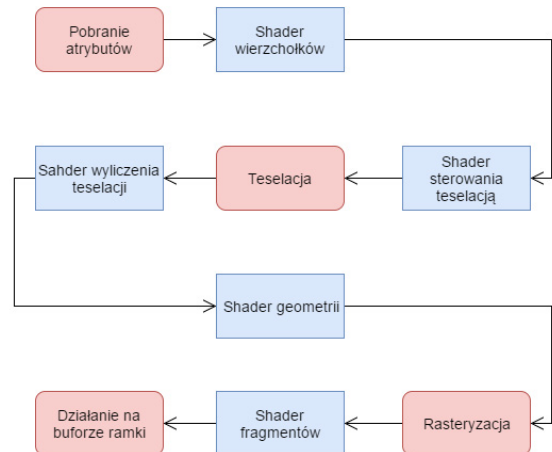
    // If there were no voxels added, we want to indicate that
    return (added != 0);
}
```

POTOK GRAFICZNY I SHADERY

Wzdłuż potoku graficznego w dużym uproszczeniu

By lepiej zrozumieć dalszą część artykułu oraz lepiej zrozumieć tekstuowanie wokseli omówionych wcześniej, warto przyrzeć się, jak wygląda potok graficzny – czyli jaka jest droga od grupy wierzchołków do pikseli widocznych na ekranie.

Potok graficzny najlepiej wyobrazić sobie jako linię produkcyjną, widoczną na Rysunku 16.



Rysunek 16. Uproszczony schemat potoku graficznego – niebieskie kwadraty oznaczają etapy programowalne

Nasze wierzchołki muszą przejść przez każdy z takich bloków. Są to kolejne etapy na drodze do stworzenia sceny 3D. Część z tych bloków jest programowalna i możemy tam wgrać własne programy, zwane shaderami, a część jest wgrana na stałe do karty graficznej. W naszym silniku nie będziemy wykorzystywać etapów teselacji (np. stosowanych do wygładzania terenu) ani shaderów geometrii, zatem ich opis pominię.

Każdy shader przyjmuje dane od poprzedniego etapu (shader wierzchołków przyjmuje dane od etapu odpowiedzialnego za pobranie atrybutów). Shader pozwala na modyfikację danych wejściowych, zanim zostaną one przekazane do następnego etapu (na zasadzie czarnej skrzynki). Te dane można modyfikować na podstawie pewnych globalnych zmiennych (zmiennych typu uniform), którymi można sterować z aplikacji, albo na podstawie wartości, które otrzymaliśmy od etapu poprzedniego.

W shaderach możemy definiować dodatkowe zmienne, które otrzymamy na wejściu, oraz co chcemy dać na wyjściu (prócz shadera fragmentów – tam zawsze musimy uzupełnić predefiniowaną strukturę, ponieważ jest to ostatni programowalny etap). Cały taki potok, z ustalonymi shaderami fragmentów i wierzchołków (minimum, które musimy dostarczyć), tworzy obiekt zwany *pipeline*. Od OpenGL 4.5 możliwe jest wymienianie pojedynczych shaderów w *pipeline*, co sprawia, że wystarczy przechowywać pojedyncze shadery zamiast całych *pipeline*ów, różniących się jedynie pojedynczym shaderem.

Żałujemy, że napisaliśmy nasz pierwszy shader. Shadery kompiluje się w programie, za pomocą API OpenGL (silnik dostarcza wygodnych klas do ładowania i kompilowania shaderów z pliku), i samemu należy sprawdzić komunikaty o błędach kompilacji itd. Skompilowany shader nazywa się *programem*. Programy można przechowywać na dysku, w postaci binarnej, jednak należy pamię-

tać, że nie jest to rozwiązanie przenośne. Z racji tego, że nasze shadery są bardzo proste, kompilacja shaderów za każdym razem, gdy uruchomimy program, nie powinna nastręczać problemów.

W naszym silniku wokselowym będą nas interesować dwa etapy: shader wierzchołków i shader fragmentów. Napišemy proste shadery, które będą wyspecjalizowane w wyświetlaniu wokseli oraz ich teksturuowaniu, a także dodawaniu prostego globalnego światła kierunkowego.

Ekspresowy kurs języka GLSL i pisanie shaderów

Zanim zaczniemy pisać shadery, dowiedzmy się najpierw, w czym się je pisze. Na rzecz OpenGL opracowano język GLSL (OpenGL Shading Language) służący właśnie do tego celu. Zakładam, drogi Czytelniku, że znasz podstawy języka C, ponieważ jest on właśnie na nim oparty. Niemniej wprowadza do niego kilka zmian i usprawnień.

Posiadamy dostęp do typów skalarnych, takich jak:

- » `bool` – wartość logiczna,
- » `float` – 32-bitowa liczba zmiennoprzecinkowa
- » `double` – 64-bitowa liczba zmiennoprzecinkowa,
- » `int` – 32-bitowa wartość całkowita ze znakiem,
- » `unsigned int` – 32-bitowa wartość całkowita bez znaku.

Prócz tego posiadamy dostęp do typów wektorowych, np. `vec3` to trójelementowy wektor przechowujący wartości typu `float`, `ivec2` to dwuelementowy wektor przechowujący wartości typu `int`, `mat4` to macierz 4x4 wartości typu `float`, a `dmat3x2` to macierz 3x2 z wartościami typu `double`. Wektory różnych typów tworzymy, dodając pierwsze litery danych typów jako przedrostki – za wyjątkiem wektorów z wartościami typu `float`.

Do pół wektora możemy odwoływać się jak do tablic lub struktur, czyli równoważne będzie wywołanie:

```
vec4 v(1.0, 2.0, 3.0, 4.0);
float xa = v[0];
float xb = v.x;
```

Ponadto typy wektorowe obsługują sklejanie, czyli wywołanie:

```
vec3 v1 = vec3(1.0, 0.5, 2.0, 3.0);
vec3 v2 = v1.xxx;
vec3 v3 = vec3(v1.x, v1.x, v1.x);
```

dla `v2` i `v3` jest równoważne i spowoduje powstanie nowego wektora, którego współrzędne `x`, `y` oraz `z` będą równe 1.0.

Wektory przeładowują operatory `+`, `-`, `*` oraz `/`. Podobnie jest z typami macierzowymi. Istnieje też możliwość mnożenia macierzy i wektorów (niezbędne do składania transformacji graficznych).

Tablice można definiować w następujące, równoważne sposoby:

```
int iarr1[10];
int[10] iarr2;
```

Tablice posiadają metodę `length()`, która zwraca jej długość:

```
float a[10];
float b[a.length()]; // deklaracja tablicy b o rozmiarze
tablicy a
```

Ponadto można definiować struktury, podobnie jak w języku C:

```
struct simple {
    int a;
    float b;
};
```

W shaderach możemy tworzyć własne funkcje, ale istnieje także wiele wbudowanych funkcji matematycznych, jak np. `mod`, `clamp`, lub operacje macierzowe jak `transpose`, `inverse`, i wektorowe, np. `dot`.

Znamy już podstawy składni oraz podstawowe możliwości języka GLSL. Teraz zajmijmy się samymi shaderami.

Na samym początku powinniśmy zdefiniować wersję języka GLSL, której chcielibyśmy używać:

```
#version 450 core
```

Powyższy zapis oznacza, że używamy języka GLSL w wersji bazowej (`core` – bez rozszerzeń) OpenGL 4.5.

Aby sterować naszymi shaderami z głównego programu, musimy zadeklarować zmienne typu `uniform`:

```
uniform bool wired;
```

Za pomocą API OpenGL możemy zmieniać wartość danej zmiennej, np. poprzez odwołanie się do jej nazwy:

```
GLuint location = glGetUniformLocation(programHandle, "wired");
glProgramUniform1b(programHandle, location, newValue);
```

Każdy z shaderów posiada pewne dane wejściowe i musi uzupełnić na ich podstawie dane wyjściowe. Dla przykładu: shader fragmentów powinien dać na wyjście typ `vec4` określający kolor danego piksela, a shader wierzchołków musi przekazać nową pozycję danego wierzchołka w wektorze `gl_Position`.

Do określania dodatkowych zmiennych, które powinny być przekazywane między etapami, służą słowa kluczowe `in` oraz `out`. Dodatkowo, aby zapewnić, że odpowiednie dane zostaną do siebie dopasowane, istnieje specyfikator `layout(location = X)`:

```
layout(location = 5) out VsOut {
    vec4 color;
    vec3 normal;
} vsOut;
```

Powyższy zapis pozwala nam na przekazanie kilku parametrów w ramach jednego specyfikatora `layout`.

W shaderze, który powinien odbierać dane, powinniśmy zastosować podobny zapis:

```
layout(location = 5) in VsOut {
    vec4 color;
    vec3 normal;
} vsOut;
```

Jeśli lokalizacje specyfikatorów `in/out` nie zostaną dopasowane na różnych etapach potoku, w ramach jednego pipeline'u, to powstanie błąd podczas linkowania programów shaderów.

Shadery na niektórych etapach posiadają specjalne zmienne, np. w shaderze fragmentów możemy sprawdzić ID danego wierzchołka poprzez odwołanie się do zmiennej `gl_VertexID`.

Zadania, które powinien wykonywać nasz shader, definiujemy wewnątrz funkcji:

```
void main() {
    // Zadania...
}
```

Skoro już teorię mamy za sobą, czas działać i przedstawić shadery służące do wyświetlania naszych wokseli!

Shader wierzchołków

Jak już wcześniej pisałem, potok graficzny to linia produkcyjna, gdzie wynik jednego etapu jest przekazywany do drugiego. Etapy mogą być uruchamiane równolegle, czyli w przypadku shadera wierzchołków wierzchołki mogą być przetwarzane na wielu rdzeniach karty graficznej (tzw. rdzeniach cieniujących) równocześnie.

Pozwala to na bardzo szybkie przetwarzanie operacji z racji tego, że karta graficzna posiada bardzo wiele rdzeni cieniujących (bardzo prostych procesorów). Ilość wierzchołków, jakie karta musi przetworzyć, też jest bardzo wysoka.

Aby przesłać wierzchołki do karty graficznej, należy stworzyć specjalny bufor (VBO – *Vertex Buffer Object*), a następnie skopiować do niego odpowiednie dane. By określić kolejność, w jakiej mają być przekazywane wierzchołki (np. do tworzenia linii lub trójkątów – nie musimy kopiować kilka razy tych samych wierzchołków), należy stworzyć kolejny bufor (EBO – *Element Buffer Object*), zawierający indeksy kolejnych wierzchołków. Dodatkowo posiadamy obiekt typu *VertexArray* (VAO – *Vertex Array Object*), który przypisuje danym w buforze VBO odpowiednie pozycje w shaderze wierzchołków – np. pozycja wierzchołka będzie na pozycji 0, jego kolor na pozycji 1 itd. Do VAO należy także dołączyć odpowiedni bufor VBO.

W momencie kiedy prosimy OpenGL o narysowanie przygotowanych wcześniej wierzchołków, odwołuje się on do obiektu VAO, który jest aktywowany i przekazuje dane z dowiązanego do niego obiektu VBO, gdzie kolejność odczytywania kolejnych danych określają indeksy znajdujące się w aktywnie dowiązanym obiekcie typu EBO.

Listing 9. Konfiguracja obiektu VAO, buforów VBO i EBO oraz aktywacja atrybutów

```
void
Renderable::UpdateBuffers()
{
    _vao.Bind();

    _vbo.Bind(GLBuffer::VERTEX);
    _vbo.ReserveMutableSizeData(_vertices.data(), _vertices.size() *
        sizeof(Vertex),
        GLBuffer::DYNAMIC_DRAW);
    ActivateAttributes();
    _vbo.Unbind(GLBuffer::VERTEX);

    _ebo.Bind(GLBuffer::INDICES);
    _ebo.ReserveMutableSizeData(_indices.data(), _indices.size() *
        sizeof(GLuint),
        GLBuffer::DYNAMIC_DRAW);
    _ebo.Unbind(GLBuffer::INDICES);

    _vao.Unbind();

    _changed = false;
}

void
Renderable::DrawStart(RenderInfo* info)
{
    if (_changed) {
        UpdateBuffers();
    }

    FillInfo(info);

    _vao.Bind();
    _ebo.Bind(GLBuffer::INDICES);
}

void
Renderable::DrawEnd()
{
}
```

```
_vao.Unbind();
_ebo.Unbind(GLBuffer::INDICES);
}

void
Renderable::ActivateAttributes()
{
    _vao.ActivateBinded(POSITION, 3, GL_FLOAT, sizeof(Vertex),
        (GLvoid*)0);
    _vao.ActivateBinded(NORMALS, 3, GL_FLOAT, sizeof(Vertex),
        (GLvoid*)offsetof(Vertex, normal));
    _vao.ActivateBinded(TEXTURE, 3, GL_FLOAT, sizeof(Vertex),
        (GLvoid*)offsetof(Vertex, texUV));
    _vao.ActivateBinded(COLOR, 4, GL_FLOAT, sizeof(Vertex),
        (GLvoid*)offsetof(Vertex, color));
}
```

Całość może brzmieć dość zawile, jednak po przyjrzeniu się Listingowi 9 opis powinien być czytelniejszy. Przedstawia on implementację tego mechanizmu przy użyciu odpowiednich funkcji pomocniczych wokół API OpenGL w silniku VEngine – przez co całość powinna być jeszcze bardziej czytelna (operacje na wrapperach odpowiadają operacjom w API OpenGL). Dodatkowo możemy zobaczyć, w jaki sposób są dowiązywane dane do odpowiednich pozycji w shaderze (klasa *Renderer* nie dowiązuje parametru *texOffset*, który jest przeznaczony wyłącznie dla wokseli – dodaje go klasa potomna *VoxelMesh*).

Dane są odczytywane i przekazywane do shadera wierzchołków. Kod naszego shadera wierzchołków dla wokseli znajduje się w Listingu 10.

Listing 10. Shader wierzchołków używany do wyświetlania wokseli

```
#version 450 core

// Input data we receive from Vertex object
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normals;
layout (location = 2) in vec2 texCoord;
layout (location = 3) in vec4 color;
layout (location = 4) in uvec2 texOffset;

// Output data, we should pass to the fragment shader
layout(location = 6) out VsOut {
    vec4 color;
    vec2 texCoord;
    vec2 texOffset;
    vec3 normal;
} vsOut;

// Global matrixes, that allows to stack vertex transformations
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

// The data that MUST be passed to the next stage
// - new position of vertex
out gl_PerVertex
{
    vec4 gl_Position;
};

// Tasks, that our shader is responsible for
void main()
{
    // Stack graphical transformations
    gl_Position = projection * view * model * vec4(position, 1.0);

    // Pass data to the next stage
    vsOut.color = color;
    vsOut.texCoord = texCoord;
    vsOut.texOffset = texOffset;

    // Transform normals
    vsOut.normal = mat3(transpose(inverse(model))) * normals;
}
```

W pierwszej kolejności definiujemy wartości wejściowe, które otrzymamy z bufora, wraz z ich pozycjami – muszą być one zgodne z pozycjami ustalonymi przez VAO. Następnie deklarujemy wyjście z naszego shadera – są to dane, które będą niezbędne do prawidłowego wyświetlania określania koloru piksela w shaderze fragmentów.

Kolejnym blokiem są zmienne typu `uniform`, reprezentujące nasze macierze. Matematyka używana w grafice 3D to osobny temat, przybliżę jedynie zastosowanie każdej z macierzy.

Macierz projekcji służy do przenoszenia współrzędnych świata do znormalizowanej przestrzeni urządzenia, która jest wymagana przez shadery do poprawnego rysowania wierzchołków (więcej detali na ten temat znajdzie się w drugiej części artykułu, gdy będziemy omawiać metodę *frustum culling*) oraz rodzaj widoku – np. ortogonalny lub perspektywiczny (domyślny tryb).

Macierz widoku określa pozycję kamery (obserwatora) w świecie, a *macierz modelu* określa pozycję w świecie renderowanego obiektu. Każdy wierzchołek posiada swoje współrzędne relatywne do pozycji modelu, zatem złożenie macierzy modelu oraz macierzy widoku da nam macierz pozwalającą określić rzeczywistą pozycję wierzchołka w świecie, a dodatkowe złożenie macierzy projekcji pozwoli na przeniesienie tego wierzchołka do znormalizowanych współrzędnych urządzenia, przy zastosowaniu odpowiedniej metody projekcji.

Macierz projekcji będzie się zmieniać bardzo rzadko, macierz widoku tylko w przypadku zmiany położenia kamery (maksymalnie co klatkę), a macierz modelu w przypadku zmiany renderowanego obiektu (co obiekt). Są one wspólne dla wielu wierzchołków, dlatego są zmiennymi typu `uniform`.

Następny blok `out` jest specjalnym blokiem dla shadera wierzchołków, który trzeba zadeklarować jawnie w przypadku, gdy używamy API OpenGL 4.5 pozwalające na wymianę pojedynczych shaderów w *pipeline*. Określa on pozycję wierzchołka, która musi być przekazana dalej (należy zauważyć, że pozycja nie jest przekazywana w bloku `VsOut`).

Nasz shader wykonuje bardzo mało pracy. Określa nową pozycję wierzchołka na podstawie złożenia odpowiednich macierzy w odpowiedniej kolejności, oblicza nową wartość normalnych na podstawie macierzy modelowania (by uwzględnić rotację i skalę) oraz przekazuje niektóre dane dalej.

Po przeanalizowaniu wszystkich wierzchołków potok przechodzi do kolejnego etapu.

Shader fragmentów

Shader fragmentów to ostatni programowalny etap potoku graficznego. Służy on do określenia koloru dla każdego z pikseli. Będzie wywoływany wiele razy, dlatego należy z rozumą wybierać operacje, które będą tu przeprowadzane. W przypadku tak prostego shadera, jak nasz, nie powinniśmy się zbytnio martwić optymalizacją.

Dane, które otrzymamy z poprzednich etapów (np. współrzędne UV tekstury dla danego piksela), są interpolowane przed wywołaniem shadera fragmentów. Jeśli mamy określić kolor piksela należącego do pewnego trójkąta, to dostaniemy wartości, które będą średnimi ważonymi trzech wierzchołków, gdzie wagi są dobierane na podstawie odległości od każdego z nich – im bliżej danego wierzchołka, tym jego waga jest większa.

Przeanalizujmy kod naszego shadera fragmentów znajdujący się w Listingu 11.

Listing 11. Shader fragmentów używany do wyświetlania wokseli

```
#version 450 core

// Input from the vertex shader
layout(location = 6) in VsOut {
    vec4 color;
    vec2 texCoord;
    vec2 texOffset;
    vec3 normal;
} vsOut;

// Output from the shader
out vec4 color;

// Normalized size of one texture in the atlas
uniform float tileSize;

// Should the output be rendered wireframe?
uniform bool wired;
uniform vec3 wireColor;

// Should the texture be applied?
uniform bool textured;

// Texture sampler
uniform sampler2D tex1;

// Direction of the global light
uniform vec3 globalLightDir;

// Ambient light
uniform vec3 ambientLightColor;
uniform float ambientStrength;

void main()
{
    // Calculate Light value
    vec4 ambient = vec4(ambientStrength * ambientLightColor, 1.0);
    // The higher angle between normal and
    // -(global light direction), the stronger value of the
    // global light is (use the same color as for ambient)
    float diff = max(dot(vsOut.normal, -globalLightDir), 0.0);
    vec4 diffuse = vec4(diff * ambientLightColor, 1.0);

    // Do not apply Light if rendering wireframe (for debug purposes)
    if (wired) {
        color = vec4(wireColor, 1.0f);
    }
    else if (textured) {
        // Calculate real u, v from the values we received from the
        // earlier shaders
        vec2 resultUV;
        resultUV.x = (vsOut.texOffset.x * tileSize) +
            mod(vsOut.texCoord.x, 1.0) * (tileSize);
        resultUV.y = (vsOut.texOffset.y * tileSize) +
            mod(vsOut.texCoord.y, 1.0) * (tileSize);

        color = (ambient + diffuse) * texture(tex1, resultUV);
    }
    else {
        color = (ambient + diffuse) * vsOut.color;
    }
}
```

W pierwszej kolejności musimy się upewnić, że lokalizacja danych wyjściowych z shadera wierzchołków jest taka sama jak lokalizacja danych wejściowych shadera fragmentów. Na podstawie tych danych określimy kolor danego piksela (wartości te są interpolowane między kolejnymi wierzchołkami). Następnie deklarujemy wyjście – kolor piksela w RGBa.

Teraz przejdźmy do zmiennych `uniform`. Pierwsza wartość określa wielkość jednego kafelka tekstury w atlasie tekstur – opisane w akapicie nt. atlasu tekstur i teksturowania. Następnie posiadamy zmienną, która powinna być aktywowana przy włączeniu trybu rysowania *wireframe* w maszynie stanów OpenGL, używanego na potrzeby debugowania. Tryb *wireframe* pozwala renderować szkielek modelu w odpowiednim kolorze i ignorować oświetlenie (przykład takiego rysowania znajduje się na Rysunku 4). Za pomocą kolejnej zmiennej możemy wymusić brak teksturowania – wtedy zostanie użyty kolor (np. przy rysowaniu linii lub punktów). Kolejną

zmienną jest obiekt typu `sampler2D` – jest to specjalny typ obiektu, do którego można dowiązać teksturę. Można posiadać wiele takich obiektów w jednym shaderze. Z racji tego, że korzystamy z atlasu tekstur, potrzebujemy tylko jeden taki obiekt, który przez większość działania programu powinien być niezmienny – chyba że projektant zażyczy sobie zmieniać zestaw tekstur co kilka klatek.

Kolejne parametry służą do określania oświetlenia. `globalLightDir` pozwala określić kierunek światła globalnego, przez co ściany wokseli mogą być różnie oświetlane w zależności od pozycji „Słońca”, a światło środowiskowe `ambient` pozwala określić globalne oświetlenie, równomierne dla wszystkich obiektów. Gdyby nie stosowanie tego typu oświetlenia, obiekty leżące naprzeciwko wektora światła globalnego byłyby całkowicie czarne.

Kod, który wykonuje shader fragmentów, jest nieco bardziej skomplikowany od shadera wierzchołków.

W pierwszej kolejności obliczamy wartość oświetlenia – oświetlenie środowiskowe poprzez zwykłe mnożenie, a światło globalne poprzez zastosowanie iloczynu skalarnego wektora normalnego danego wierzchołka z wektorem przeciwnym do pozycji światła globalnego. Iloczyn skalarny ma taką własność, że im mniejszy kąt między danymi wektorami, tym większa jego wartość. Dlatego musimy użyć wektora przeciwnego do kierunku – normalne wektorów, skierowane zgodnie z kierunkiem źródła światła, nie powinny być w ogóle oświetlone (ponieważ oświetlamy tylną stronę tych ścian, które i tak nie są renderowane). Następnie dodajemy wartości światła kierunkowego i środowiskowego do siebie, co da nam w wyniku oświetlenie danego piksela. Dalej sprawdzamy, jaki posiadamy tryb rysowania, i na podstawie tego określamy jest kolor piksela – albo jako kolor szkieletu, albo nakładając odpowiednią wartość tekstury z atlasu tekstur na podstawie współrzędnych UV (służy do tego funkcja wbudowana `texture`), albo poprzez nałożenie odpowiedniego koloru.

Po zakończeniu tego etapu piksele zostają przesłane do następnego etapu wbudowanego na stałe do karty graficznej i nasz obraz pojawi się na ekranie.

Dodawanie shaderów w silniku VEngine

Aby usprawnić proces dodawania, kompilowania i linkowania shaderów i ich programów, klasa `Renderer` posiada metodę, która wczyta shader z pliku, skompiluje go i dołączy do odpowiedniego pipeline’a, a w przypadku błędu wypisze błąd kompilacji lub linkowania. Przykład znajduje się w Listingu 12.

Listing 12. Wczytanie shadera z pliku w silniku VEngine

```
int error = renderer.AddShader("FragVoxel", "Shaders/Voxel.frag",
    Shader::FRAGMENT, Renderer::VOXEL);
if (error) {
    std::cout << "Adding failed for FragVoxel - Shaders/Voxel.frag"
    << std::endl;
    return error;
}
```



MICHAŁ KRAWCZYK

Programista systemów wbudowanych w krakowskiej firmie Semihalf oraz student Informatyki Stosowanej na Akademii Górniczo-Hutniczej. Pasjonat informatyki, w szczególności jeśli chodzi o systemy wbudowane, grafikę komputerową oraz działanie gier komputerowych „od kuchni”.

W wywołaniu funkcji musimy podać, jakiego typu jest dany shader (np. fragmentów lub wierzchołków) oraz do którego potoku powinien zostać dołączony. Pozwala to na szybkie ładowanie i kompilację shaderów, a w razie błędów umożliwia natychmiastowe otrzymanie logów kompilacji.

CIĄG DALSZY NASTĄPI...

Przebrnęliśmy przez pierwszą część artykułu, przybliżającą część zagadnień grafiki 3D oraz opisującą wokselse, sposoby wyświetlania wokseli, a także napisaliśmy dwa proste shadery.

W drugiej części artykułu przedstawię drzewo ósemkowe jako strukturę organizacji do chunków i optymalizacji wykrywania kolizji, opiszę, jak można wykryć kolizję dwóch obiektów ze sobą oraz z terenem, omówię implementację algorytmu przycinania niewidocznych krawędzi (`frustum culling`), a także implementację mechanizmu modyfikującego teren.

Dodatkowo skupimy się na obiektach w grze, czyli na drzewie list cyklicznych, pozwalającym na hierarchiczną organizację obiektów, oraz przyjrzymy się, jak działa pętla gry.



Rysunek 17. W następnym odcinku...

W sieci:

Kod źródłowy w całości znajduje się pod adresem:

► <https://github.com/gerwand/vengine>

Bibliografia:

- [1] <https://github.com/Gerwand/VEngine/blob/master/Praca-inzynierska/krawczyk-voxel.pdf>
- [2] *OpenGL Księga Eksperta*, Wydanie VII - Graham Sellers, Richard S. Wright Jr., Nicholas Haemel
- [3] https://www.khronos.org/opengl/wiki/File:Winding_order.png
- [4] <https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/>
- [5] <http://www.glfw.org>

SonarQube – kod mojszy niż twojszy

Napisaliśmy kod. Ktoś go zazwyczaj sprawdza. Nietrudno tu o różnicę zdań, który wariant lepszy, czytelniejszy itp. Jak również o wykrycie oczywistych defektów. Dobrze, gdyby pod ręką znajdował się jakiś automat, najlepiej taki, który pozwala na przekrojowe analizy, również w dziedzinie czasu. To nasz nowy kolega – SonarQube. Wykonuje te i kilka innych czynności, a jak się postaramy, to dla każdego pull-requestu. Artykuł pozwala nam go lepiej poznać, a zwłaszcza jak pracuje nad kodem .NET samego Jona Skeeta.

CZY PISZE PAN ZGRABNE I CZYSTE KODY?

Tak mniej więcej brzmiała oferta pracy nadesłana przez firmę rekrutacyjną. Autorka ogłoszenia poradziła się zapewne jednego z programistów, by ten podpowiedział jej kilka haseł mających potencjał pozyskania dobrego kandydata. Całkiem możliwe również, że tamtemu znana była książka *Czysty kod* (ang. *Clean Code*) autorstwa Wujka Boba (Robert C. Martin) [1]. Zamierzony efekt został przez ową panią uzyskany – przykuto uwagę odbiorców. Kilku moich kolegów przy porannej kawie z rozbawieniem wspominało wiadomość, którą i oni otrzymali. Podczas dyskusji doszliśmy do trzech wniosków.

1. Bardzo prawdopodobne, że w tej organizacji znajduje się co najmniej jeden programista, który czyta książki dotyczące swego fachu. Dobrze książki. To już coś. Większość z nas niestety nie czyta [2].
2. „Zgrabny” i „czysty” to przymiotniki wyrażające subiektywną ocenę rzeczy, a więc pojawia się problem uzgodnienia kilku poglądów w zespole/organizacji.
3. „Zgrabny” i „czysty” nie mają jasnych i oczywistych miar, którymi można poddać rzecz ocenie. O ile na przykład kod krótki/długi daje się zmierzyć liczbą linii, o tyle jedna konkretna metoda badania czystości prawdopodobnie nie istnieje. Z pewnością chodzi o jego jakość, ale zaraz pojawia się pytanie o to, które z własności miałyby nań wpływ kluczowy.

Czy zatem niniejszy artykuł należy zakończyć słowami „o gustach się nie dyskutuje”? Czy może jednak istnieje obiektywne podejście godne zaufania?

UZGODNIJ, MIERZ I ZWYCIĘŻAJ

Istnieją standardowe, wypracowane latami metryki jakości [3]. Podobnie jak techniki osiągania konsensusu w zespole [4]. Nie to jednak jest celem artykułu. Większość z nas o nich słyszała, a niektórzy potrafią nawet wymienić kilka. Chciałbym skupić się bardziej na potrzebie mierzenia, którego często nie doceniamy. Zamiast przydługiego akapitu proponuję tych kilka cytatów, które same wystarczą:

- » „Nie możesz kontrolować czegoś, czego nie potrafisz zmierzyć.” [5]
- » „Dla każdego atrybutu projektu istnieje taki sposób jego mierzenia, który jest lepszy niż brak mierzenia.” [6]
- » Ale też: „Co jest mierzone, jest zrobione.” [6]

Jest kwestią drugorzędną, kto ustala reguły/metryki – czy architekt organizacji, czy też autonomiczny zespół. Istotne jest to, że każdego dnia mamy możliwość zajrzenia w kartę zdrowia pacjenta i oce-

nę jego stanu. A jeżeli stosujemy diagnostykę, to mamy całkiem spore szanse na udane leczenie. Zanim będzie za późno i obudzimy się z ręką w... garze spaghetti.

I tutaj na scenę wchodzi narzędzie SonarQube (zwane również Sonarem). Jest to oprogramowanie o otwartym kodzie, aktywnie rozwijane i wspierane przez szwajcarską firmę SonarSource SA [7]. Za cel stawia sobie wykrywanie naruszeń uzgodnionych wcześniej reguł, jak i pomiar własności samego kodu. Działa w trybie klient-serwer, w konsekwencji więc wspomaga spójne zarządzanie kryteriami na poziomie zespołu lub całej organizacji. Dodatkowo w łatwy sposób można rozszerzyć zakres jego kompetencji, aktywując jedną z wielu gotowych wtyczek (np. dedykowanych dla języka czy zbioru metryk) lub własnego autorstwa.

Dobrze, w tym punkcie jesteśmy już skłonni zacząć mierzyć w sposób zorganizowany jakość naszego kodu. Chcemy również zobaczyć, jak robi to za nas SonarQube.

INSTALACJA

Dedykowany serwer w naszej organizacji

Potrzebujemy dwóch składników – serwera i bazy danych. Obsługiwane bazy danych to m.in. MySQL (5.6+), PostgreSQL (8+), MS SQL (2012+), Oracle (11g+). Serwer SonarQube jest licencjonowany w ramach GNU LGPLv3 i działa w środowisku wykonania Oracle JRE 8 lub OpenJDK 8. Z istotnych wymagań sprzętowych warto wymienić pamięć RAM (min. 2 GB) oraz wydajne dyski, które będą miały największy wpływ na szybkość działania Sonara (głównie za sprawą indeksów Elasticsearch). Więcej informacji znajdziemy w przewodniku na stronie twórców [8].

Instalujemy serwer baz danych (w naszym przypadku MySQL 5.7, MariaDb nie jest wspierana!). Następnie pobieramy paczkę ZIP ze strony z dystrybucjami serwera SonarQube [9]. Do wyboru są w zasadzie dwa warianty: najnowsza i LTS¹. Instalacja nie powinna nastręczać trudności (dobrze wiem, co teraz sobie myślisz, drogi Czytelniku, ale istotnie nie ma tutaj wielu kroków)².

1. Tworzymy schemat sonar w naszej bazie danych oraz użytkownika sonar z hasłem sonar.
2. Nadajemy pełne uprawnienia do schematu sonar dla użytkownika sonar.
3. Rozpakowujemy archiwum do wybranego katalogu.
4. Modyfikujemy nasz _sonar/conf/sonar.properties (Listing 1).

1. LTS – ang. *Long Term Support*. Wersja bardziej konserwatywna. Błędy poprawiane są w ramach patchy, tj. bez konieczności aktualizacji do nowszych wersji (tj. innych wersji Major.Minor).

2. Jest też obraz dedykowany Dockerowi dostępny tutaj: https://hub.docker.com/r/_/sonarqube/

Listing 1. Minimum konfiguracji SonarQube

```
# User credentials.
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar
#---- MySQL 5.6 or greater
sonar.jdbc.url=jdbc:mysql://localhost:3306/sonar
?useUnicode=true&characterEncoding=utf8&
rewriteBatchedStatements=true&
useConfigs=maxPerformance&useSSL=false
```

1. Wykonujemy nasz_sonar/bin/windows-x86-32/InstallNTService.bat (Uwaga! Bitowość musi się zgadzać z bitowością JVM na naszym komputerze).
2. Wykonujemy nasz_sonar/bin/windows-x86-32/StartNTService.bat (uwaga jak powyżej).
3. Idziemy do: <http://localhost:9000>.
4. Przedstawiamy się jako admin (hasło: admin) i jesteśmy w panelu zbiorczym SonarQube.

SonarQube jako usługa w chmurze

Firma SonarSource AS prowadzi również usługę typu PaaS o nazwie SonarCloud (<https://about.sonarcloud.io>). Jest ona darmowa dla projektów o kodzie otwartym i pozwala nam bez większego wysiłku skorzystać z dobrodziejstw Sonara.

1. Idziemy do: <https://sonarcloud.io>.
2. Klikamy *Sign in*.
3. Wybieramy autentykację z wykorzystaniem naszego konta na GitHubie (lub Bitbucket); zakładamy, o ile takiego jeszcze nie posiadamy.
4. Znajdujemy się w (jeszcze) pustym panelu projektów.
5. W prawym górnym rogu klikamy na ikonę konta i wybieramy *My Account*.
6. Generujemy token w zakładce *Security* (posłuży zamiast loginu i hasła w komunikacji naszych skryptów analizy z API sonarcloud.io).
7. W zakładce *Organizations* tworzymy naszą organizację (do celów jak wyżej).
8. Mamy wszystko, czego potrzeba, aby dokonać pierwszej analizy projektu.

ANALIZA PROJEKTU GOOGLE CLOUD DLA .NET

Przygotowanie

Jona Skeeta nie trzeba przedstawiać. Jest autorem cenionej książki *C# in Depth* [10], jak również wiedzie prym na StackOverflow [11]. W Google zaś zajmuje się rozwojem bibliotek klienckich .NET do usług chmurowych. Rzućmy zatem Sonara na głęboką wodę – niech zmierzy się z kodem owych bibliotek. W tym celu wykonujemy kilka prostych kroków:

1. Klonujemy repozytorium <https://github.com/GoogleCloud-Platform/google-cloud-dotnet.git>.
2. Upewniamy się, że nasze Visual Studio obsługuje C# 7.
3. Instalujemy za pomocą *chocolatey* [12] pakiet skanera SonarQube, który wstrzyknie do procesu budowy MSBuild elementy analizy, zbierze wyniki i prześle na serwer SonarQube: `cinst msbuild-sonarqube-runner`.
4. Instalujemy narzędzia konsolowe *dotCover* w celu obliczania pokrycia kodu testami jednostkowymi: `cinst dotcover-cli`.

5. Będąc w katalogu głównym projektu, instalujemy narzędzie konsolowe wykonujące testy jednostkowe xunit: `nuget install xunit.runner.console -ExcludeVersion`.

Następnie przygotowujemy skrypt automatyzujący wykonanie analizy SonarQube. W ogólności schemat zawsze jest ten sam:

1. Wykonujemy krok `SonarQube.Scanner.MSBuild begin`, podając parametry bieżącej analizy.
2. Budujemy projekt, ewentualnie wyliczamy pokrycie kodu testami jednostkowymi.
3. Wykonujemy krok `SonarQube.Scanner.MSBuild end`, który zbiera wyniki analizy i przesyła na serwer do dalszej obróbki.

W celu analizy projektów Jona Skeeta musimy wykonać procedurę wielokrotnie, jako że API Google mają tutaj swoje dedykowane solutiony. Całość wykona za nas zautomatyzowany skrypt, który został zaprezentowany poniżej (Listing 2).

Listing 2. Skrypt analizy SonarQube

```
param(
    $version = '',
    $authToken = ''
)

$slns = gci *.sln -r
foreach ($sln in $slns) {
    $sqProj = $sln.Name -replace '\.sln$', ''
    Write-Host "Analysing $sqProj"

    # oznaczamy przyszłe budowanie do analizy SQ
    SonarQube.Scanner.MSBuild begin `
        /key:"$($sqProj -replace '\.sln$', '')" `
        /name:"$($sqProj -replace '\.sln$', '')" `
        /version:"$version" `
        /d:"$($sonar.cs.dotcover.reportsPaths + `
            "dotCover.html")"
    # potrzebne dla sonarcloud.io
    #/d:"sonar.host.url=https://sonarqube.com" `
    #/d:"sonar.organization=nasza_organizacja" `
    #/d:"sonar.login=$authToken"

    nuget restore $sln
    MSBuild $sln /p:Configuration=Release

    # wyznaczenie pokrycia kodu testami jednostk.
    $dll = gci "$($sln.Directory)\**\bin\" + `
        "Release\net452\*.Tests.dll" -r
    $runner = rvpa "$($sln.Directory)\xunit.runner.console.exe" `
        "tools\net452\xunit.console.exe"
    $coverConf = gci "$($sln.Directory)\* `
        "\*.Tests\coverage.xml"
    dotCover analyse $coverConf.FullName `
        /TargetExecutable="$runner" `
        /TargetArguments="$dll" `
        /ReportType=HTML `
        /Output=dotCover.html `

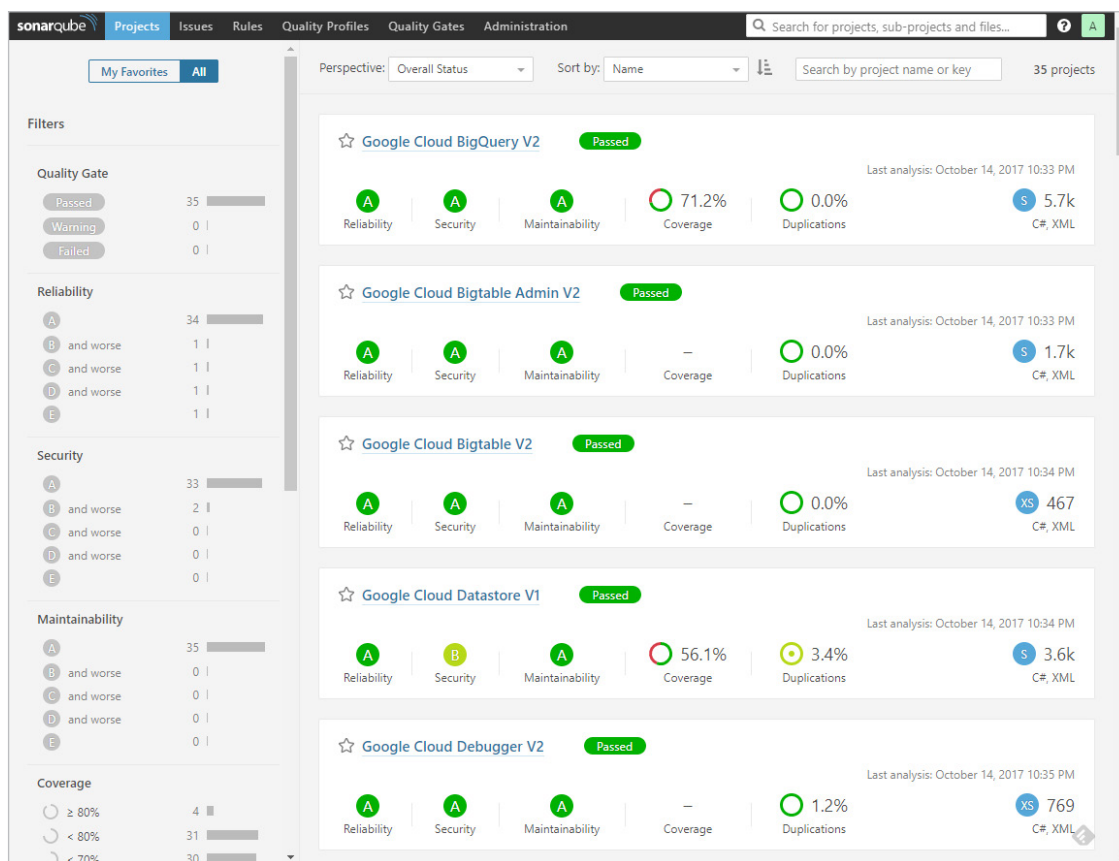
    # zebranie wyników i przesłanie na serwer SQ
    SonarQube.Scanner.MSBuild end

    Write-Host "Analysing $sqProj completed!"
}
```

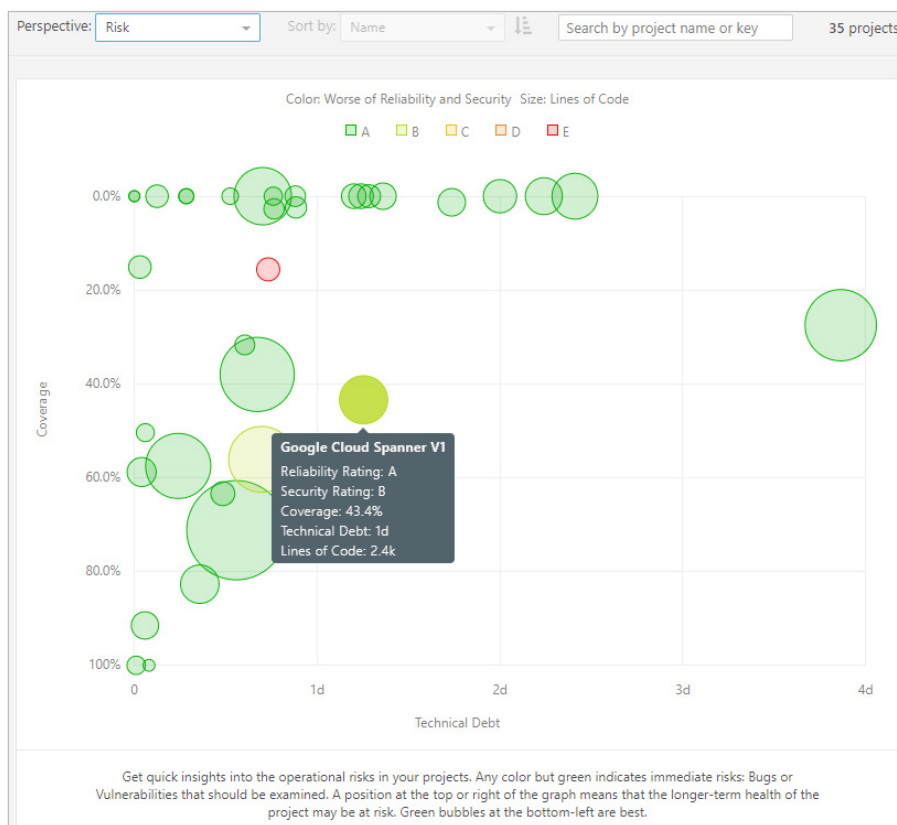
Analizę umieściłem na serwerze lokalnym, jak również opublikowałem ją w chmurze SonarCloud.io [13]. Prezentowane w niniejszym artykule rysunki mogą się różnić od tych, które będą dostępne na SonarCloud.io. Wynika to m.in. z innych wersji SonarQube na tych środowiskach, jak również innych zestawów reguł branych pod uwagę.

Jonie Skeet – mamy Cię!

Nic z tego. Kto się łudził, ten będzie rozczarowany. Niemal wszystkie projekty błyszczą się soczystą zielenią (Rysunek 1). Kolor ten oznacza tu oczywiście stan pożądany – projekty są w dobrej kondycji.



Rysunek 1. Widok ogólny projektów w SonarQube



Rysunek 2. Ryzyko w projektach liczone % pokrycia kodu i ilością długu tech

Przeanalizowano 35 rozwiązań Visual Studio (1 rozwiązanie VS = 1 projekt SonarQube), wszystkie wypadły znakomicie. Są to niewielkie projekty (klasyfikowane jako rozmiar XS lub S, tj. do kilku tysięcy linii kodu), część z nich nie miała nawet projektów z testami jednostkowymi.

Z lewej strony panelu mamy do dyspozycji intuicyjny filtr (wg rodzajów metryk i innych własności). Projekty możemy nawet dodawać do ulubionych, by w codziennej pracy skupiać się na wąskim wycinku naszych bezpośrednich zainteresowań. Dostępne jest także etykietowanie (tagi), dzięki którym możemy tworzyć dowolne zgrupowania projektów. Na górze widać pole sortowania – idealne do wzajemnych rywalizacji między zespołami w firmie. Na szczególną uwagę zasługuje jednak kilka predefiniowanych wykresów (perspektyw), które omówię poniżej.

Pierwsza z nich, *Risk* (Rysunek 2), uwypukla ryzyko związane ze stanem aktualnym (kolor koła) i przyszłym (położenie koła). Im kolor bardziej czerwony, tym większy kłopot obecnie. A im dalej na północny-wschód, tym rokowania na przyszłość są gorsze. Skalę wyznacza rozmiar koła, a więc im większe, tym więcej uwagi powinniśmy mu poświęcić (o ile niesie ry-

zyko obecnie lub w przyszłości). Cenna wizualizacja, nieprawdą?

Kolejny widok (Rysunek 3) dotyczy wizualizacji długu technicznego w zbiorze projektów. Głównie chodzi o wykrycie projektów-odstępów, którym nie po drodze z innymi, zmierzającymi w ustalonym kierunku. O jakie kierunki tutaj chodzi? Po pierwsze, istnienie zależności liniowej świadczy prawdopodobnie o ignorowaniu jednej lub kilku reguł przez projekty tworzące linię. Taką sytuację mamy w projektach Jona. Nic dziwnego, nie umawialiśmy się z nim przecież na żaden zestaw metryk, którym mielibyśmy poddawać go próbie („fair enough” – pewnie by rzekł). Następnie należałoby wyznaczyć pewien kierunek, którym podążają projekty wraz z rosnącą złożonością. Im bardziej logarytmiczna (lub stała!), tym lepiej. Zależność wykładnicza sugeruje, że albo nie skonfigurowaliśmy właściwie reguł SonarQube, albo mamy tam istotnie do czynienia ze słabym procesem rozwoju oprogramowania. Na koniec o projektach-odstępach. Jeżeli zauważyliśmy, że jakiś osobnik odbiega od reszty rozpędanego stada, wówczas dobrze jest przyjrzeć się mu wnikliwie. Czy wykazuje on może istotnie mniejszą/większą liczbę błędów na testach (lub produkcji), krótszy/dłuższy czas dostarczania nowych funkcjonalności itp. Jeżeli odpowiedź brzmi tak, to być może warto wziąć pod rozwagę szkolenie wewnątrzfirmowe.

Ostatni³ widok (Rysunek 4) dotyczy pokrycia kodu testami jednostkowymi. Jest oczywiste, że nikt nie chciałby skończyć ze swoim projektem w prawym górnym rogu (duża złożoność, brak testów) takiego zestawienia. Jest on niejako narożnikiem, gdzie biją, a szczególnie mocno wszelkie regresyjne sierpowe. Wykres ten dużą pomoc niesie przy nowych projektach, rozwijanych dynamicznie. Możemy wówczas zbadać ich relację złożoności i pokrycia testami wobec trwających dłużej projektów, o których posiadamy znacznie rozleglejszą wiedzę. A następnie wykorzystać ich doświadczenie, aby uniknąć cudzych błędów.

Szczegóły projektu

Na Rysunku 5 przedstawiono status pojedynczego projektu. Wymienię teraz składowe tego podsumowania, a ważniejsze z nich rozwinę w dalszej części artykułu. U samej jego góry znajduje się zwięzła postać wyniku ostatniej analizy: *jakość ak-*

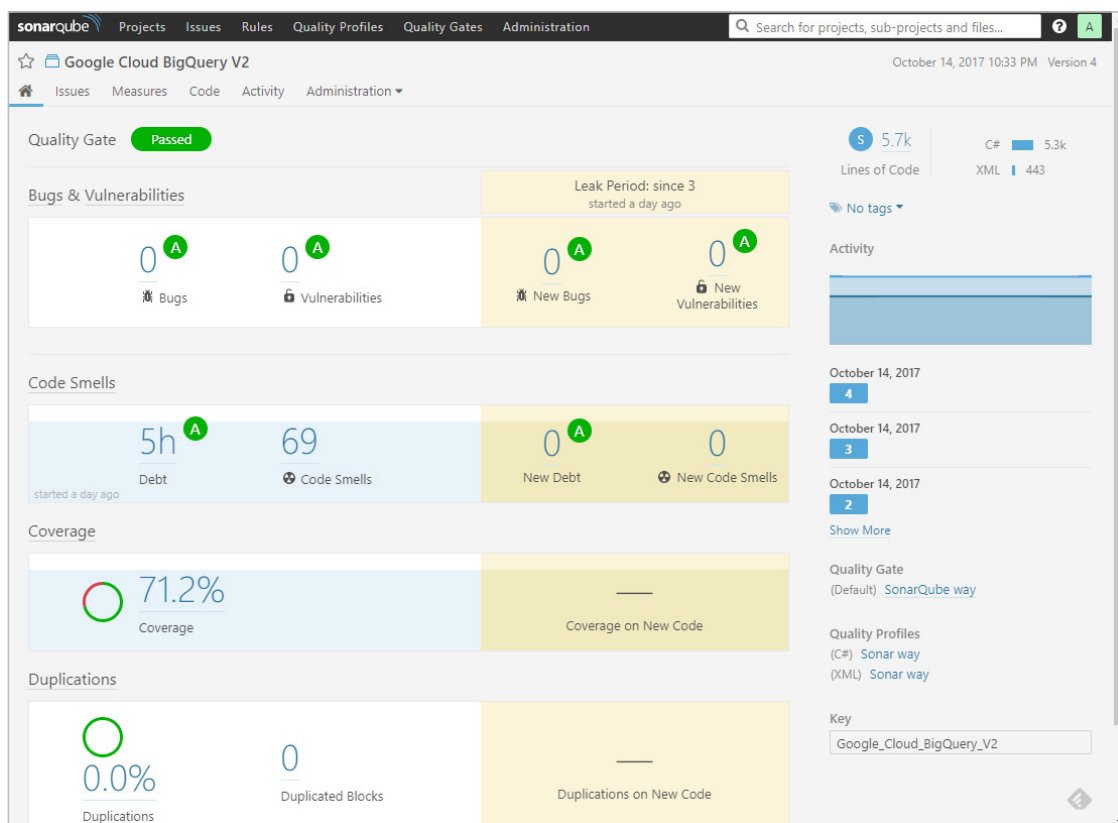


Rysunek 3. Dług techniczny i rozmiar projektu składają się na łatwość utrzymania

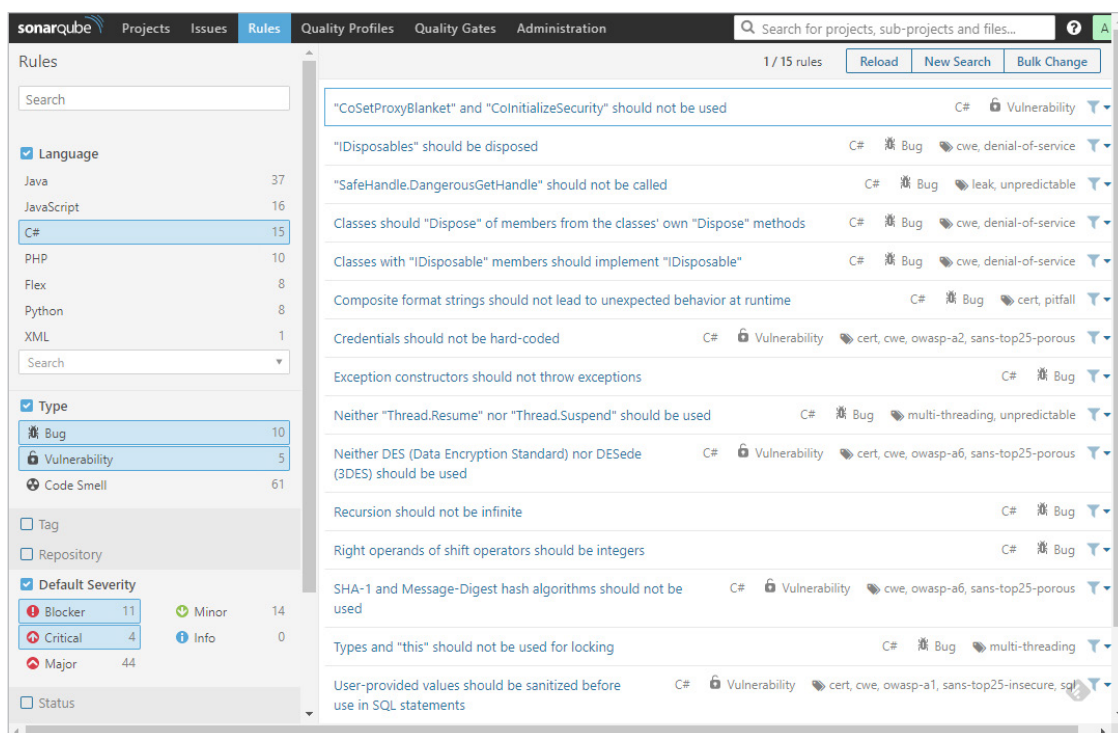


Rysunek 4. Złożoność projektu a stopień jego pokrycia testami jednostkowymi

3. Ostatni opisany przeze mnie. Perspektyw oferowanych przez SonarQube mamy więcej.



Rysunek 5. Widok szczegółowy projektu

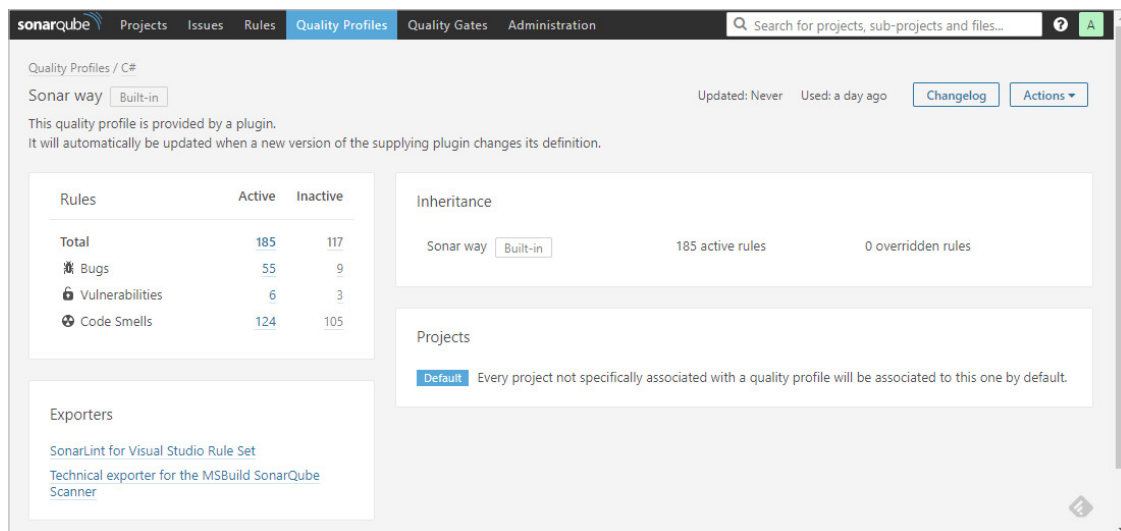


Rysunek 6. Repozytorium reguł

ceptowna lub jakość nie do przyjęcia. W dziedzinie jakości oprogramowania pełni podobną funkcję jak czerwony/zielony stan procesu budowania artefaktów projektu. Poniżej mamy cztery sekcje: defekty, smród w kodzie⁴, pokrycie kodu, duplikaty. Każda z nich przedstawia najważniejsze z metryk, które dotyczą danego

obszaru. Stosowana jest tutaj kolorowa klasyfikacja A-F, która pozwala szybko zorientować się w miejscach szczególnie problematycznych. Prawą kolumnę każdej z sekcji wypełnia analogiczna informacja, ale uwzględniająca tzw. leak period, czyli analizę jedynie tej zmiany, która zaszła od poprzednio wykonanej analizy. Jeżeli wykonujemy Sonarem pomiary każdej zmiany w repozytorium, bardzo łatwo możemy wówczas ustalić, kto i kiedy przyczynił się

4. Długo zastanawiałem się, jak przetłumaczyć ang. code smells. Uznałem, że w czasach, kiedy owy czysty kod to rzadkość, trzeba nazywać rzeczy po imieniu.



Rysunek 7. Profile jakości – przykład C#

do poprawy/pogorszenia stanu rzeczy. Szerszy horyzont historycznych zmian znajdziemy już w szczegółowych zakładkach metryk (namiastką tego w tym widoku są wykresy rysowane w tłach sekcji). Na prawo od metryk znajdują się:

1. Krótka charakterystyka projektu w liczbach, czyli jego rozmiar (etykieta XS-XL oraz liczba linii) z podziałem na linie napisane w każdym z zastosowanych języków programowania.
2. Wykres warstwowy pokrytych linii i pozostałych do pokrycia przez testy jednostkowe.
3. Kalendarium analiz.
4. Konfiguracja projektu, czyli nazwy profili i bram, które są stosowane w analizie projektu.
5. Klucz projektu, czyli jego unikatowy identyfikator. Istotne jest, by w wykonywanych analizach podawać poprawny identyfikator istniejącego projektu, ponieważ w innym przypadku stworzony zostanie nowy, nieposiadający historii.

Reguły, profile, bramy

Podstawą wyciszenia rodzin metryk takich jak niezawodność, bezpieczeństwo, utrzymanie są tzw. problemy (lub zgłoszenia; ang. *issues*). W raportach Sonara są one zawsze bezpośrednią konsekwencją naruszeń wcześniej ustalonych reguł. Każda z nich jest przypisana do owej rodziny metryk i języka programowania, a linia kodu, która jej nie spełnia, jest klasyfikowana do dalszej obróbki. Na Rysunku 6 pokazano 15 reguł o największej wadze (ang. *severity*) przynależących do języka C# i dostępnych w Sonarze. Baza reguł (wraz z dobrą wyszukiwarką) znajduje się w zakładce *Rules* w górnym menu. Kolejną zaletą jest tutaj możliwość edycji własności reguł (np. wspomniana wyżej waga czy etykiety, dodanie rozszerzonego opisu itp.) Klikając w wiersz zawierający jedną z nich, przechodzimy do widoku szczegółowego, a w nim:

- » rodzaj reguły: defekt, luka bezpieczeństwa czy smród w kodzie,
- » waga (blokujący, krytyczny itp.),
- » etykiety,
- » data obowiązywania,
- » język programowania, którego dotyczy,
- » czas potrzebny do naprawienia naruszenia reguły,
- » szczegółowy opis, uzasadnienie, konsekwencje naruszenia,
- » przykład kodu zgodnego,
- » przykład kodu niezgodnego, wyjątki od reguły,

- » referencje do standardów branżowych (np. OWASP),
- » profile, w których reguła jest aktywna (więcej o profilach poniżej),
- » zestawienie projektów, które lekceważą regułę najbardziej.

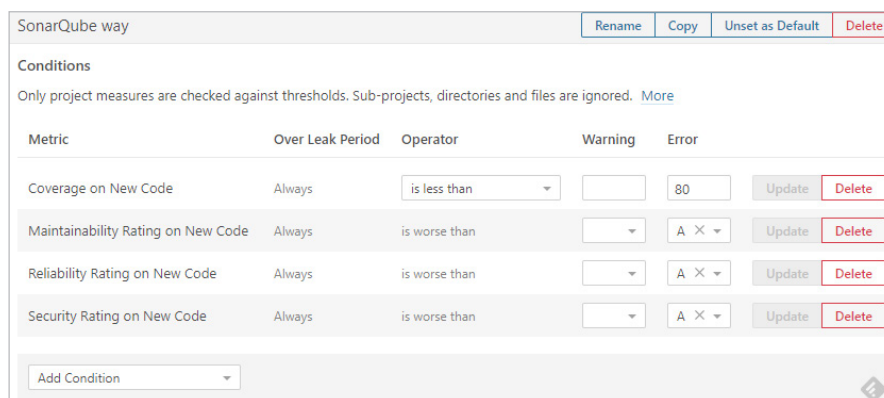
Każda reguła może zostać przypisana do profilu. Te są dostępne w zakładce *Quality Profiles* i, najprościej rzecz ujmując, stanowią nazwane zbiory reguł danego języka programowania. Przykład domyślnego profilu języka C# pokazano na Rysunku 7. Każdy projekt ma przypisane profile, wg których wykonana będzie następna analiza. Kluczowe znaczenie odgrywa tutaj problem właściwego dobrania reguł, które utworzą efektywny profil analiz. Wracamy tym samym do tzw. „problemu uzgadniania”, który został poruszony na początku artykułu. Ponadto pokrewne zagadnienie stanowi odpowiedź na następujące pytanie – jeden profil na organizację, czy kilka (i jakie)? Wyobrażam sobie teraz krzywiznę uśmiechu architektów i co bardziej autorytarnych liderów zespołów. Niezależnie od przyjętych rozwiązań SonarQube znakomicie ułatwia ewoluowanie profilu w pożądanym kierunku (zaostrowanie lub rozluźnianie kryteriów). Do tego celu służą akcje dostępne w prawym górnym rogu panelu profilu: audyt zmian, zbiorcze porównanie i kopiowanie. Wypróbowałam i sprawdzoną przeze mnie techniką jest rozpoczęcie z profilem domyślnym i wyłączanie co bardziej restrykcyjnych reguł, gdy ludzie zaczynają narzekać⁵.

Ostatnim tematem są bramy jakości, tzn. nazwane zbiory kryteriów decydujących o akceptacji lub odrzuceniu kodu (najczęściej decyzja osoby weryfikującej lub automatu w zaawansowanych konfiguracjach CI). Z założenia mają chronić kod, nad którym pracuje wiele osób, przed niezauważoną degradacją jakości. Mając Sonara na pokładzie, zazwyczaj jest tak, że zawstydzony autor szybko poprawia defekty i oszczędza się czas na zbędnych dyskusjach w zespole. Na Rysunku 8 przedstawiono przykład domyślnej bramy jakości w SonarQube. Jak poprzednio, tak i tutaj fundamentalne znaczenie ma możliwość dopasowania do potrzeb, a jest w czym wybierać – Sonar wprost obfituje w metryki jakości.

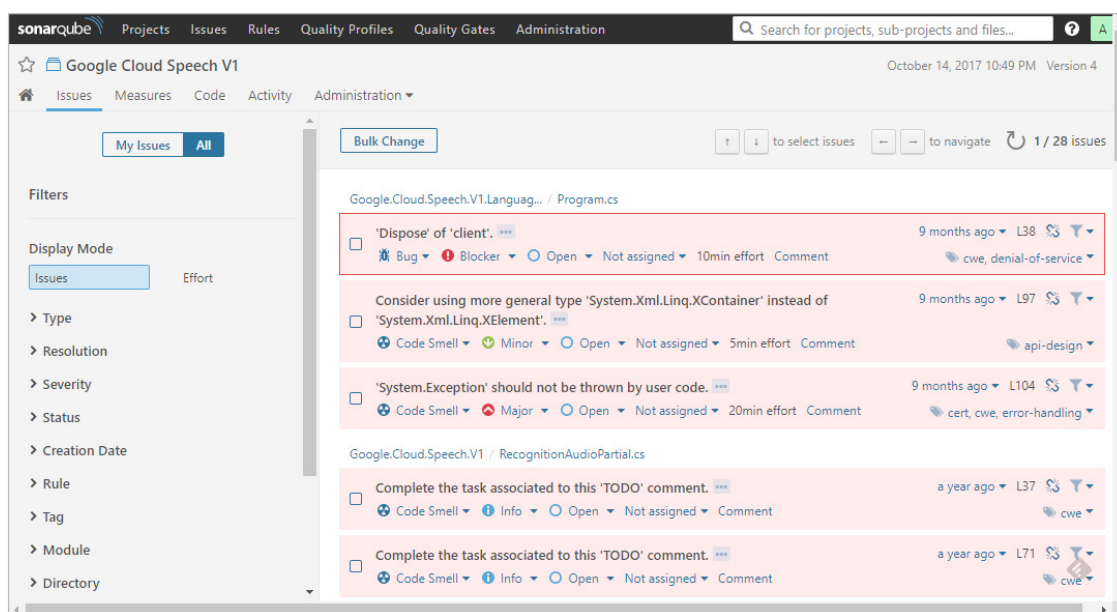
Kod, na którym można polegać

Podstawowym kryterium jakości kodu według twórców SonarQube jest jego poprawność, tj. brak ewidentnych symptomów

5. Podając oczywiście szereg mocnych argumentów.



Rysunek 8. Brama jakości – strzeże dobrego kodu



Rysunek 9. Widok problemów dla wybranego projektu

błędne działanie. Znajdując się w widoku szczegółowym projektu, widzimy na górze zakładkę *Issues*. Klikając w nią, przechodzimy do listy błędów, luk bezpieczeństwa oraz „smrodów w kodzie” (Rysunek 9). Nie zaskakuje nas sekcja filtrów z lewej strony (spójne podejście w całym Sonarze). Ciekawym rozwiązaniem jest możliwość przypisania każdego z problemów do osoby, a gdy nasza integracja jest dostatecznie wypolerowana, to Sonar zrobi to za nas automatycznie (posiłkując się informacją z systemu kontroli wersji). Stąd główny filtr – *My Issues*. Z prawej strony ekranu widnieje lista występujących w projekcie problemów. Przejdźmy do najpoważniejszego z nich – błędu blokującego (Rysunek 10).

Ukazał się widok z podglądem na kod źródłowy. Z lewej strony – nawigacja, z prawej zaś – problem w kontekście kodu, którego dotyczy. Przekonujemy się, że na niewielkiej przestrzeni twórcom Sonara udało się umieścić całkiem sporo istotnych informacji. Linie kodu mają przypisanych autorów (z systemu kontroli wersji), kod jest pokolorowany, w prostokątnych ramkach mamy garść własności zgłaszanego problemu. I tak, w kolejności:

[górze]

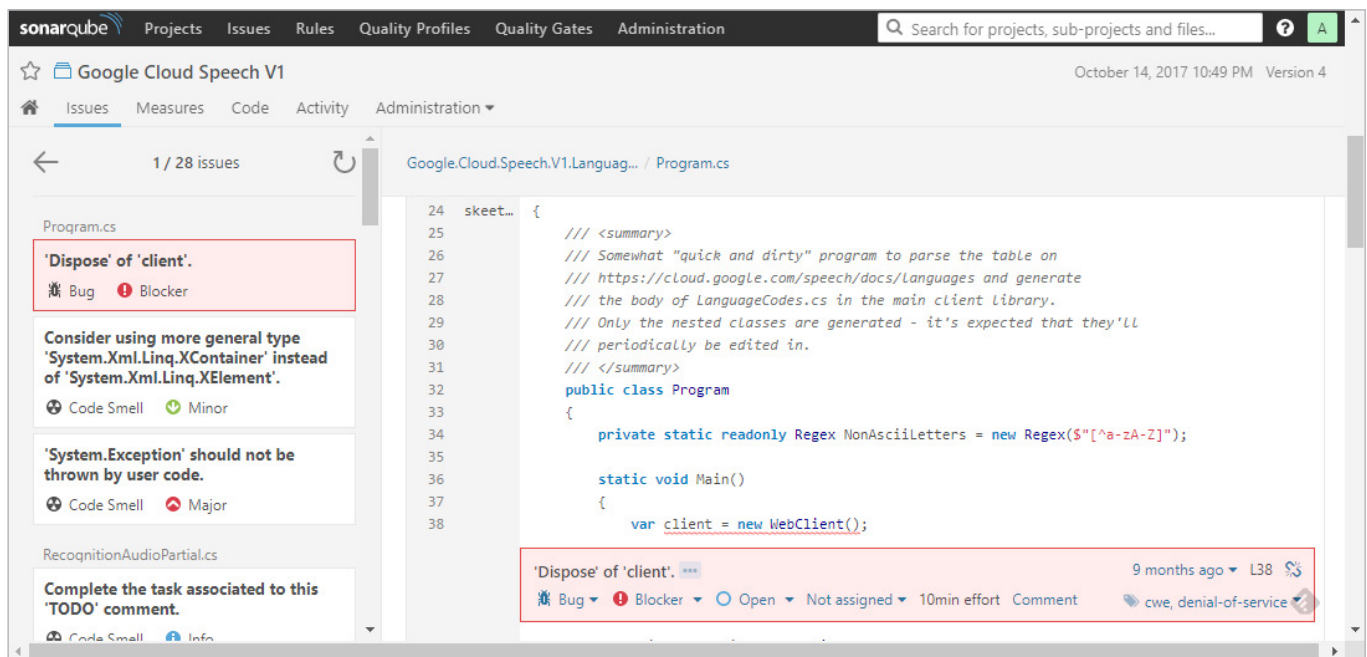
- » tytuł,
- » trzy kropki to link do szczegółowego opisu reguły w omówionym wyżej repozytorium reguł,

- » data zmiany linii kodu (po kliknięciu – data i autor),
- » numer linii,
- » permalink do repozytorium reguł.

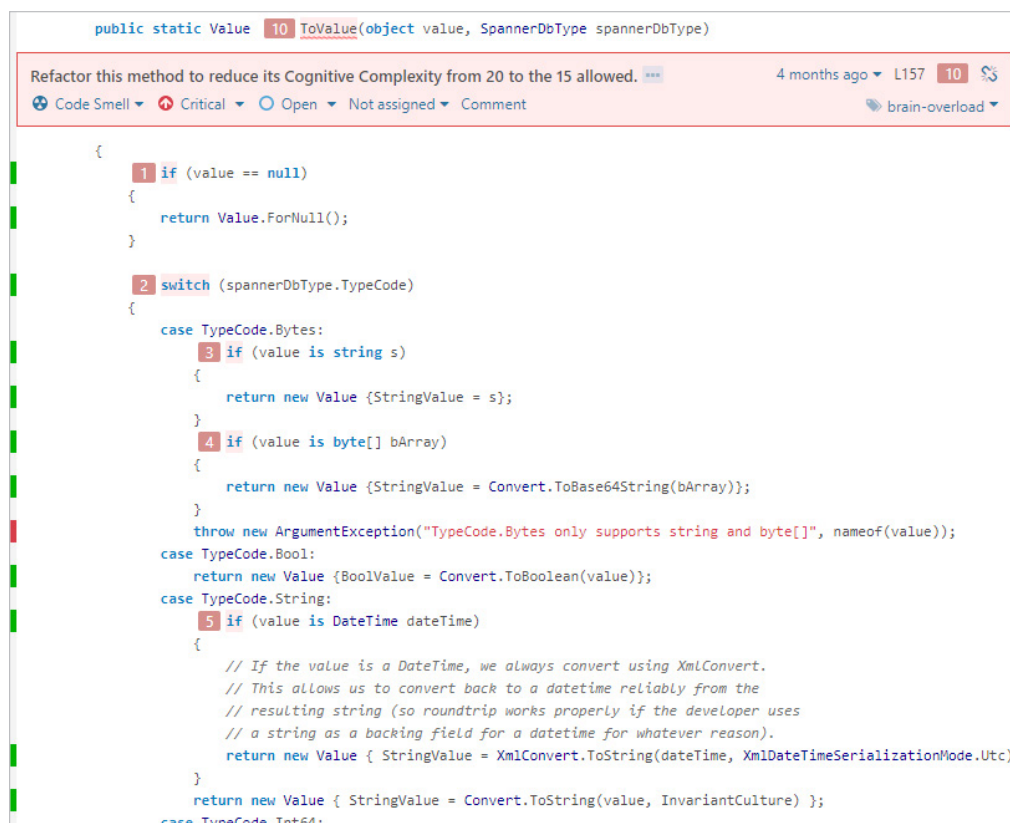
[dół]

- » rodzaj problemu (błąd itd.),
- » waga,
- » status naprawy: otwarty, potwierdzony, rozwiązany jako naprawiony/fałszywy/nieistotny,
- » przypisanie do osoby, która zajmuje się rozwiązywaniem problemu,
- » szacowany czas naprawy,
- » możliwość dodania komentarza, dyskusji nad rozwiązaniem problemu,
- » etykiety.

Warto zauważyć, że część pól jest tutaj edytowalna – twórcy Sonara dużą wagę przyłożyli do minimalizacji liczby kroków potrzebnych do osiągnięcia celu. Jeżeli zaś chodzi o kod Jona Skeeta, to cóż – sam przyznał w linii 26, że ów program był pisany na kolanie.



Rysunek 10. Błąd blokujący – jedyny taki w projekcie Jona Skeeta



Rysunek 11. Przykład naruszenia złożoności poznawczej

Bezpieczeństwo w czasach pośpiechu

Podobnie jak błędy kodu, problemy mogą dotyczyć luk bezpieczeństwa. Sonar jest tu szczególnie pomocny, ponieważ od przeciętnego programisty nie wymaga się obecnie posiadania nadzwyczajnych kompetencji z dziedziny bezpieczeństwa. Mówiąc krótko – nietrudno o przeoczenie, które może mieć fatalne konsekwencje dla całej organizacji. Przyjrzyjmy się zatem kilku ciekawszym regułom (Tabela 1). Przy tworzeniu reguł SonarSource kieruje się wytycznymi standardów branżowych takich jak MITRE, CWE, CERT, OWASP, SANS.

Kod	Nazwa z opisem
S2278	Ani DES, ani 3DES nie powinien być używany
S2070	MD-* i SHA-1 nie powinny być używane
S3649	Wartości dostarczone przez użytkownika powinny być zweryfikowane przed zastosowaniem w SQL
S2386	Pola mutowalne nie powinny być publiczne i statyczne jednocześnie

Tabela 1. Niektóre reguły traktujące o bezpieczeństwie

Dług techniczny i złożoność – ekonomia a filozofia

Warto poświęcić trochę czasu, by zrozumieć naturę długu technicznego. Dobre wprowadzenie podaje M. Fowler [14]. Kiedy mamy to już za sobą, chcemy sprawdzić, jak pomaga nam w zarządzaniu długiem technicznym SonarQube. Mamy dwa ujęcia: całkowity czas potrzebny do zniwelowania tzw. smrodów kodu (ang. *code smells*) i ich odsetek w danym fragmencie. Im mniejsza wartość – tym lepiej dla nas. Warto przywołać kilka ciekawszych reguł definiujących owy smród w kodzie (Tabela 2).

Kod	Nazwa z opisem
S3216	ConfigureAwait(false) powinien zostać użyty na Tasku
S1163	Wyjątki nie powinny być rzucane z bloku finally
S2696	Składowe instancje obiektów nie powinny pisać do składowych statycznych

Tabela 2. Niektóre ze smrodów kodu

Ponadto na uwagę zasługuje specjalna reguła traktująca o poznawczej złożoności metod (ang. *cognitive complexity*). Różni się ona od klasycznej złożoności cyklomatycznej McCabe [15]. Metodę jej wyznaczania zobrazowano na Rysunku 11.

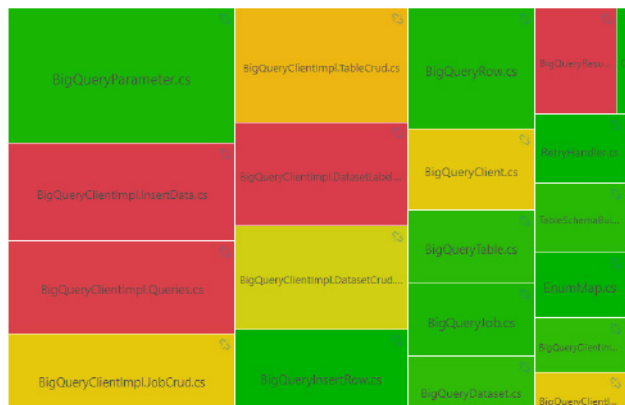
Jeżeli już mowa o złożoności, to Sonar udostępnia bardzo prostą statystykę dowolnego wycinka kodu (Rysunek 12). Według mnie jest ona niedoceniana, zwłaszcza w analizie kodu logiki biznesowej, gdzie krótkie i dobrze nazwane metody są na wagę złota.

Complexity / Function	1.6
Complexity / File	14.6
Complexity / Class	15.5
Cognitive Complexity	406

Rysunek 12. Prosta statystyka złożoności projektu

Pokrycie kodu – mityczne 100%

Podczas gdy w Internecie toczą się dyskusje [16], czy i ile można poświęcić, aby uzyskać 100% pokrycia kodu⁶, my skupmy się na tym, co oferuje Sonar, by podejrzeć, które fragmenty mają małe pokrycie. Na Rysunku 13 przedstawiono kolorową mapę pokrycia kodu. Im bardziej czerwono, tym większe wołanie o testy jednostkowe. Możemy szybko wywnioskować, którym fragmentom należy się więcej uwagi w przyszłości. Dostępne są również: widok listy sortowanej wg rosnącego pokrycia, a także wykres względem osi czasu.



Rysunek 13. Pokrycie kodu a jego złożoność

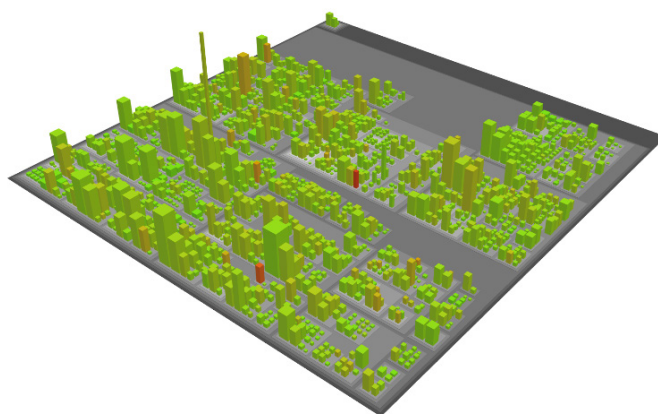
6. Każda linijka kodu ma przynajmniej jeden test jednostkowy, który ją wykonuje.

PRZEGLĄD ROZSZERZEŃ

Architektura SonarQube została tak pomyślana, by w stosunkowo łatwy sposób dało się go rozszerzać za pomocą wtyczek. W Tabeli 3 przedstawiono kilka ciekawszych reprezentantów wraz z ich umiejętnościami.

Nazwa	Funkcjonalność
3D Code Metrics	Przedstawia kod źródłowy jako budynki 3D w rozwijającym się mieście (Rysunek 14)
Azure AD Authentication	SSO z wykorzystaniem Azure AD
CSS / SCSS / Less	Analiza CSS i Less
GitHub	Komentarze do pull-requestów na GitHub
GitHub Auth	SSO z wykorzystaniem kont GitHub
GitLab (Auth)	Odpowiednik wtyczki do GitHub
LDAP	SSO z wykorzystaniem LDAP
SVG Badges	Etykiety zwięzłe podsumowujące stan projektu lub jego własności, którymi możemy pochwalić się na samej górze pliku README w repozytorium
SonarTS	Analiza TypeScript
inne	Analiza dla innych języków programowania lub inne zestawy standardowych reguł

Tabela 3. Wtyczki do SonarQube



Rysunek 14. Przykład wtyczki do SonarQube

Od czasu wydania przez Microsoft biblioteki kompilacji Roslyn programista ma możliwość w stosunkowo łatwy sposób pisania własnych analizatorów [17]. Przepis, jak to zrobić, wykracza jednak poza ramy artykułu. Warto nadmienić jedynie, że twórcy wtyczki C# do SonarQube przygotowali dedykowany projekt [18], by tworzenie własnych rozszerzeń było jak najprostsze.

Ciekawą opcją są wtyczki pozwalające na automatyczne komentarze do pull-requestów (GitHub) czy merge-requestów (GitLab). Wykorzystują one specjalny tryb pracy Sonara: preview. Działa to w ten sposób, że wyniki analizy nie odkładają się w historii, ale służą utworzeniu komentarzy do zmian, które ktoś zamierza scalić z główną gałęzią projektu. W przypadku, gdy brama jakości zapala się na czerwono (brak akceptacji), dany krok CI/CD kończy się niepowodzeniem i blokuje możliwość scalenia.

SONARLINT – BY NIE CZEKAĆ

Poznaliśmy już przekrój funkcjonalności SonarQube, ale w dalszym ciągu możemy zadawać sobie pytanie: jak często powinienem wykonywać analizę w trakcie codziennego pisania kodu? Tu z kolei

na scenę wchodzi młodszy kuzyn Sonara – SonarLint. Pochodzi z rodziny tzw. linterów kodu, czyli udogodnień wykrywających podejrzaną (w ten lub inny sposób problematyczne itp.) użycia danego języka. Za jego rozwój odpowiada ta sama organizacja – SonarSource. Jest on dostępny na większości wiodących środowisk developerskich (Eclipse, IntelliJ IDEA, VS, VS Code, Atom), jak również języków programowania. W przypadku Visual Studio ściągamy go z galerii rozszerzeń [19] i w zasadzie jest gotowy do użycia.

Uważny czytelnik z pewnością dostrzeże kolejny mankament – skąd SonarLint będzie wiedział, jak dokonuje się analiza danego projektu. Czy nie pominie naruszenia jakiejś reguły, którą SonarQube później ochoczo wykryje? A jeżeli miałby dla każdego projektu analizować wszystkie dostępne reguły – czy nie utoniemy w szumie ostrzeżeń i błędów? W tym właśnie celu SonarLint został wyposażony w mechanizm synchronizacji z serwerem SonarQube. W prosty sposób możemy połączyć nasz lokalny projekt z jego odpowiednikiem na serwerze (Rysunek 15). Dzięki temu mamy pełną zgodność obu konfiguracji, które są aktualizowane na bieżąco. Dodatkowo niezbędne analizatory Roslyn zostają dołączone w ramach pakietów NuGet. W konsekwencji nasze autorskie zbiory reguł również zostaną wzięte pod uwagę przez SonarLint (Rysunek 16).

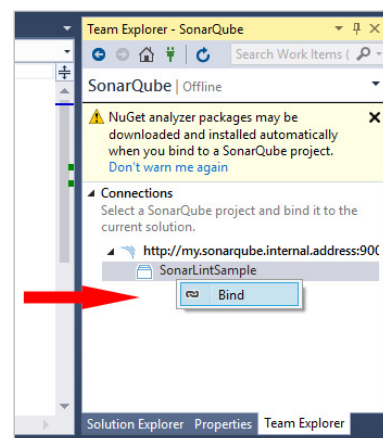
Ciekawostką jest, że SonarLint dla Visual Studio potrafi analizować języki JavaScript oraz rodzinę C/C++. Odnosił już także kilka sukcesów, analizując projekty o otwartym kodzie i znajdując w nich banalne błędy (Rysunek 17).

JEŚLI NIE SONAR, TO CO?

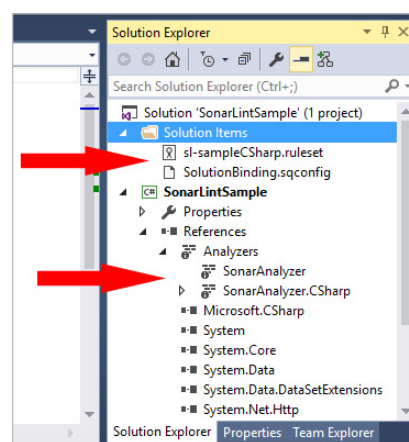
Przeczuwam, że do tego momentu kilku czytelników postanowiło spróbować SonarQube w swoim projekcie. Nie ukrywam, że niniejszy artykuł (podobnie jak jego autor) jest bardzo entuzjastycznie nastawiony do opisywanego produktu. Trzeba jednak powiedzieć o jego wadach i rozwiązaniach alternatywnych.

Głównym kosztem jest oczywiście czas, który trzeba poświęcić na utrzymanie usługi, zarządzanie nią i konfigurację projektów. Wdrożenie ogólnofirmowe może napotykać trudności natury politycznej (porównania, rywalizacja, transparentność), jak również stwarzać wyzwania dla jasnej komunikacji i utrzymania spójności. Koszt ten jednak szybko się spłaca, zwłaszcza wtedy gdy dbamy o zdrowy stan naszych projektów. Możemy dostarczać więcej funkcjonalności, czując na koniec dnia satysfakcję z jakości naszej pracy.

Teraz kilka słów o alternatywach. Jak można się domyślać, dostępnych jest ich wiele. Przeglądając ofertę integracji na GitHubie



Rysunek 15. Synchronizacja SonarLint z serwerem SonarQube



Rysunek 16. SonarLint w trybie synchronizacji: plik reguł *.ruleset i analizatory Roslyn

[20], widzimy szeroki wachlarz do wyboru. CodeCov, Codacy, Code Climate – to tylko kilka z nich. Zakres funkcjonalności w zasadzie pokrywa się z ofertą SonarQube (pokrycie kodu, dług techniczny, wsparcie podczas weryfikacji zmian itp.); część z nich może być lepiej zintegrowana z serwisem GitHub. Charakteryzują się one również tym, że mają następujące zalety: przechowują historię, pozwalają na ciekawe analizy, ułatwiają jednolite zarządzanie i raportowanie. Dużym ich atutem jest dobra (zazwyczaj) integracja z potokami CI/CD. Inną rodziną wsparcia programisty są narzędzia desktopowe, rozszerzenia IDE (np. IntelliJ IDEA, ReSharper, dotCover czy NDepend). Tych jest jeszcze więcej niż reprezentantów poprzedniej grupy. Na pewno znakomicie wspierają codzienną pracę

reklama

Szkolenie dla Ciebie lub Twojego zespołu

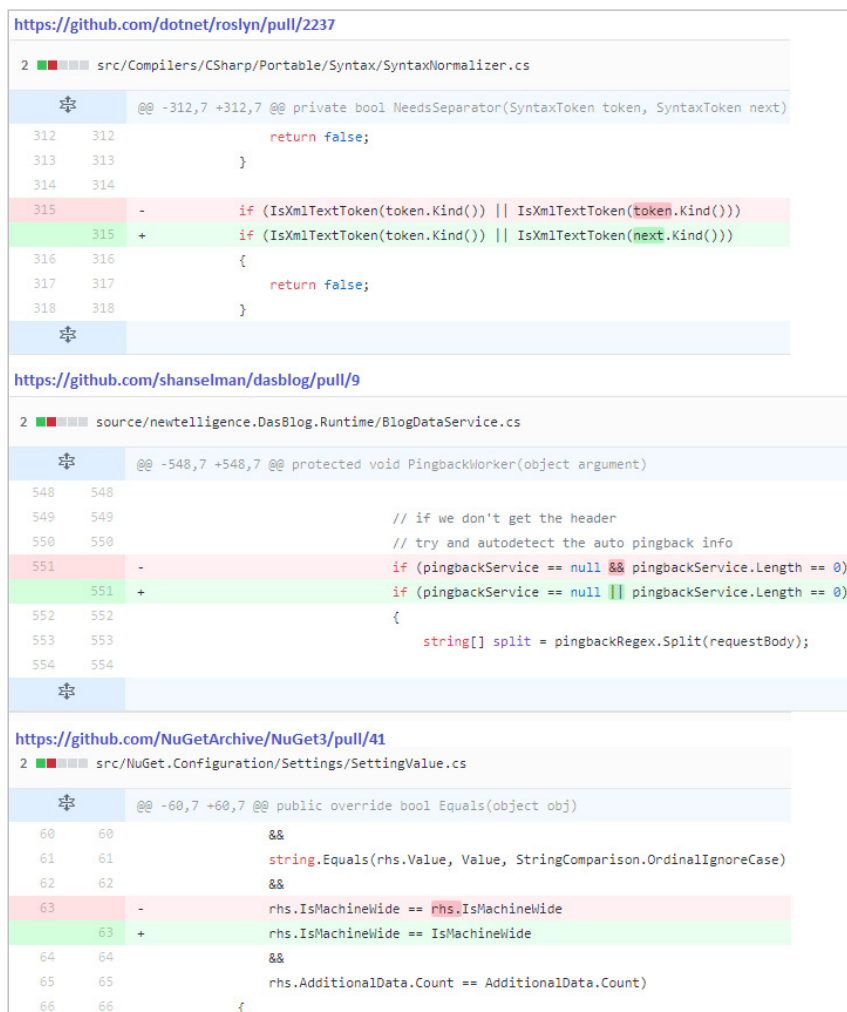
Analiza kodu za pomocą SonarQube

Skorzystaj z 10% zniżki na wszystkie szkolenie otwarte z autorskiej oferty Sages ważnej przy zamówieniach złożonych do końca grudnia 2017 r. Hasło: PROGRAMISTAMAG

 zapewniony sprzęt	 2 dni	 16h	 zdalnie lub stacjonarnie
 programiści	 trenerzy praktycy	 różne lokalizacje	 projekty indywidualne

przez statyczną analizę kodu, liczne refaktoryzacje, skróty, generację według szablonów, analizę złożoności i pokrycia testami. Należy powiedzieć, że o ile w znacznym stopniu ich zakres pokrywa się z oprogramowaniem typu klient-serwer, to jednak ich przeznaczenie jest inne. Ważnym aspektem jest tutaj możliwość współdzielenia pliku konfiguracji narzędzia w zespole lub organizacji, co daje efekt spójnego podejścia do zarządzania jakością. Szczegółowy ich opis czy też analiza porównawcza wykraczają znacznie poza ramy niniejszego artykułu.

Na koniec należy poświęcić kilka słów jeszcze jednemu, dość popularnemu podejściu – ignorancji. Doskonale sprawdza się ono w początkowych fazach projektu, nie nastrocza kłopotów z konfiguracją, pilnowaniem reguł, analizą metryk. Ponadto nie blokuje potoku CI/CD przed umieszczeniem kodu na środowisku testowym/produkcyjnym. Najważniejszą zaletą tego podejścia jest jednak wsparcie w projektach dojrzałych, dużych – pozwala programiście spać spokojnie, bo przecież ten kod już taki jest, zawsze taki był. Ani mojszy, ani twojszy, ani nasz – ichni po prostu. A skoro płacą...



Rysunek 17. Wkład SonarLint w rozwój oprogramowania open source

Źródła

- [1] <https://helion.pl/ksiazki/czykov.htm> – „Czysty kod” autorstwa Roberta C. Martina (Wujka Boba).
- [2] <https://blog.codinghorror.com/programmers-dont-read-books-but-you-should/> – Jeff Atwood o tym, dlaczego nie czytamy fachowych książek.
- [3] C. Jones, Applied Software Measurement: Assuring Productivity and Quality, Nowy Jork: McGraw-Hill, 1997.
- [4] T. DeMarco i T. Lister, Peopleware: Productive Projects and Teams, Addison-Wesley Professional, 2013.
- [5] T. DeMarco, Controlling Software Projects: Management, Measurement, and Estimates, Prentice Hall, 1986.
- [6] S. McConnell, Code Complete: A Practical Handbook of Software Construction, Microsoft Press, 2004.
- [7] <https://www.sonarsource.com/> – firma odpowiadająca za rozwój SonarQube.
- [8] <https://docs.sonarqube.org/display/SONAR/Documentation> – dokumentacja Sonara; zawiera przewodnik instalacji i konfiguracji.
- [9] <https://www.sonarqube.org/downloads/> – dystrybuje Sonara do pobrania.
- [10] J. Skeet, C# in Depth, Manning Publications, 2013.
- [11] <https://stackoverflow.com/users?tab=Reputation&filter=all> – Jon Skeet na StackOverflow.
- [12] <https://chocolatey.org/packages/msbuild-sonarqube-runner> – skaner SonarQube dla MSBuild jako pakiet chocolatey.
- [13] <https://sonarcloud.io/organizations/wachulski-github/projects> – analiza projektów Google w chmurze SonarCloud.io.
- [14] <https://martinfowler.com/tags/technical%20debt.html> – M. Fowler o długu technicznym.
- [15] <https://www.sonarsource.com/docs/CognitiveComplexity.pdf> – złożoność poznawcza proponowana przez SonarSource SA.
- [16] <https://twitter.com/unclebobmartin/status/55966620509667328> Wujek Bob o 100% pokrycia kodu.
- [17] <https://msdn.microsoft.com/en-us/magazine/dn879356.aspx> – jak napisać własny analizator Roslyn.
- [18] <https://github.com/SonarSource/sonarqube-roslyn-sdk> – SDK do stworzenia własnej wtyczki do Sonara.
- [19] <https://marketplace.visualstudio.com/items?itemName=SonarSource.SonarLintforVisualStudio2017> – SonarLint do ściągnięcia.
- [20] <https://github.com/marketplace> – oferta integracji na GitHub, sekcja Code quality.



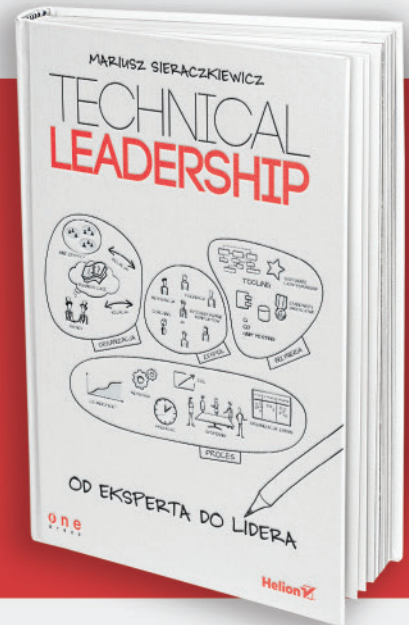
MARCIN WACHULSKI

Przede wszystkim mąż i ojciec. W tej odrobinie czasu, która mu pozostaje, stara się być lepszym programistą-inżynierem. Obecnie ramię w ramię z biblioteką Roslyn zgłębia tajniki kompilacji C#. Poznaje również świat mikrouślug. Raczej konserwatywny, tj. woli poczytać klasykę programistyczną niż przewijać tweety.

WARSZAWA / 08-10.01.2018

TECHNICAL LEADERSHIP™ ROLA LIDERA TECHNICZNEGO

1. Rola lidera technicznego
2. Motywacja własna i innych
3. Ludzie
4. Zespół
5. Kompetencje lidera

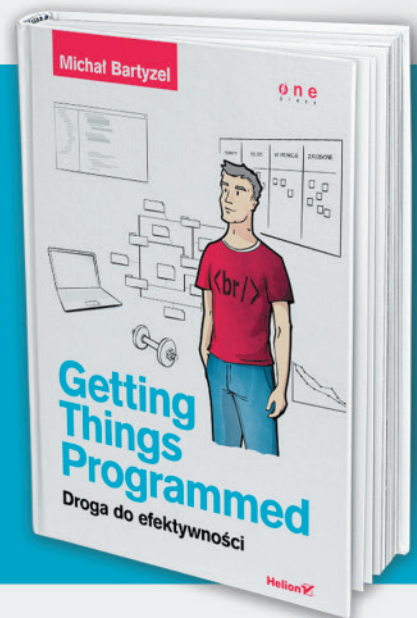


ŁÓDŹ / 07-08.12.2017

GETTING THINGS PROGRAMMED DROGA DO EFEKTYWNOŚCI

1. Oprogramowanie „na czas”
2. Opracowywanie zadań
3. Planowanie pracy programisty
4. Wykonywanie zadań programistycznych
5. Szacowanie zadań programistycznych

www.getting-things-programmed-edycja2.evenea.pl



POZOSTAŁE SZKOLENIA OTWARTE :

Zbieranie wymagań i współpraca z klientem	Łódź	13-15.11.2017	2100,00 PLN
Techniki pracy z kodem	Warszawa	29.11-1.12.2017	2100,00 PLN
Kanban w 1 dzień	Warszawa	08.12.2017	1300,00 PLN
Zespoły rozproszone - techniki skutecznej pracy zdalnej	Warszawa	14-15.12.2017	1800,00 PLN
Wzorce projektowe i refaktoryzacja do wzorców	Łódź	18-20.12.2017	2100,00 PLN
Nowoczesne architektury aplikacji	Warszawa	21-23.02.2018	2100,00 PLN

CENY NETTO

64 sposoby na crash

Czy zastanawialiście się kiedyś, jak bardzo kruche są podstawy, na których budujemy oprogramowanie, i jak niewiele trzeba, aby zachęcić system operacyjny do nieplanowanego zamknięcia procesu? W tym artykule przedstawione zostaną okrągłe 64 wybrane metody na uzyskanie takiego rezultatu.

Ze względu na umiejętności i zainteresowania autora w dużej mierze będą to mniej lub bardziej praktyczne przykłady użycia niezdefiniowanego zachowania (ang. *undefined behavior*), w większości inspirowane na błędach napotkanych w rzeczywistym kodzie. W znaczącej części opisywanych przypadków można uzyskać zdecydowanie ciekawsze efekty niż crash, chociaż nie jest to przedmiotem tego artykułu.

Należy również zauważyć, że z powodu błędów zawartych w kodzie niektóre crashe mogą wymagać bardzo specyficznych warunków, aby mogły wystąpić: poczynając od wersji kompilatora, flag kompilacji i wersji użytych bibliotek, kontynuując przez system operacyjny i architekturę procesora, a kończąc na konkretnym sprzęcie i warunkach zewnętrznych.

1. STACK OVERFLOW – WYWOŁANIA FUNKCJI

Stos (ang. *stack*) – przestrzeń w pamięci służąca do przetrzymywania podstawowych informacji o stanie wykonania danego wątku. Przechowywane są w niej zmienne lokalne oraz parametry i adresy powrotu funkcji¹. Więcej informacji: [1].

Chociaż wykorzystanie ciągłego stosu o stałej wielkości nie jest wymagane ani przez standardy C, ani C++, to autor nie słyszał o żadnej architekturze, która rozwiązywałaby ten problem w inny sposób. W związku z tym możliwe jest zużycie całej dostępnej dla stosu pamięci. Przykładową funkcję realizującą mnożenie poprzez rekurencyjne dodawanie wraz z przykładem jej użycia przedstawiono w Listingu 1.

Listing 1. Stack overflow – wywołania funkcji

```
int multiply(int a, int b)
{
    return b == 0 ? 0 : a + multiply(a, b - 1);
}

int main()
{
    return multiply(4, 262144);
}
```

2. STACK OVERFLOW – ZA DUŻY BUFOR NA STOSIE

Zużycie stosu danego wątku może zostać zdominowane przez tablice lokalne, a nie tylko ramki funkcji. Przedstawiono to na przykładzie w Listingu 2.

Listing 2. Stack overflow – za duży bufor na stosie

```
int main()
{
    char buf[1024 * 65536];
    memset(buf, 42, sizeof(buf));
}
```

3. BUFFER OVERFLOW

Przepełnienie bufora, czyli zapis poza przydzielonymi mu ramami pamięci, to jeden z najczęstszych błędów programów pisanych w C lub w innych językach, ale z mentalnością C. Jeśli w okolicy takiego bufora znajdują się inne obiekty, mogą one zostać częściowo lub całkowicie nadpisane, powodując błędne działanie programu. W Listingu 3 przedstawiono typowe nadpisanie bufora buf, nadpisując jednocześnie wartości wewnątrz obiektu str, przez co jego destruktor nie może wykonać się pomyślnie.

Listing 3. Buffer overflow

```
struct foo
{
    char buf[42];
    std::string str;
};

int main()
{
    auto f = std::make_unique<foo>();
    memset(f->buf, 0, 64);
}
```

4. STACK-BASED BUFFER OVERFLOW

Najpopularniejsza wariacja powyższego – nadpisanie bufora znajdującego się na stosie. Może skutkować zmianą adresu powrotu funkcji i, jeśli jest użyty, kanarka stosu (ang. *stack canary*).

Listing 4. Stack buffer overflow

```
int main()
{
    char buf[42];
    memset(buf, 0, 64);
}
```

5. NULL POINTER DEREFERENCE

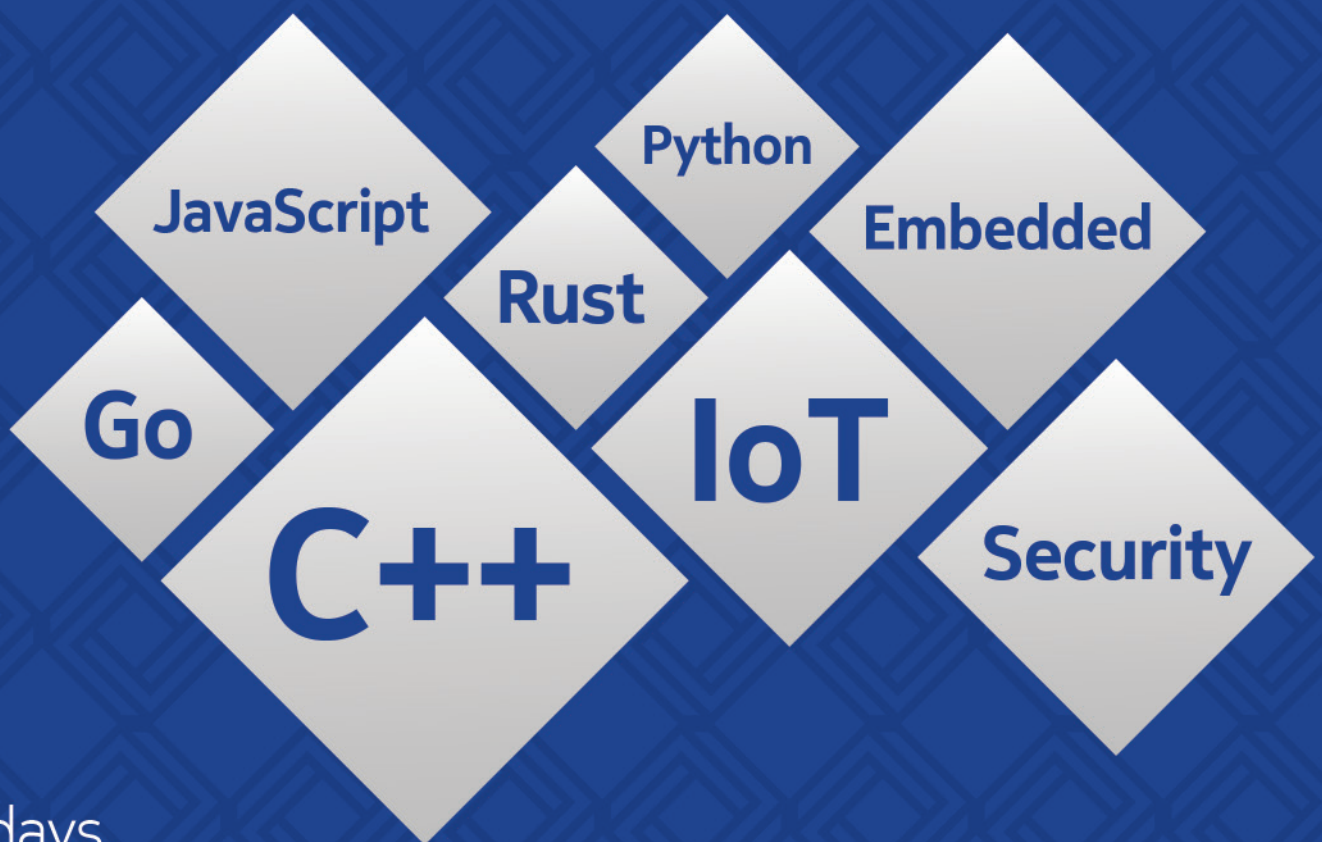
Dereferencja pustego wskaźnika zdarza się nawet w językach, które wskaźników nie mają. W C i C++, technicznie rzecz biorąc, wskaźnik może wskazywać na adres zero [2], ale musi wtedy różnić się od nullptr oraz NULL. W Listingu 5 znajduje się przykładowa dereferencja pustego wskaźnika.

¹. Nie chodzi tutaj o stos jako o strukturę danych, choć powiązanie semantyczne istnieje niebezpośrednio.



code::dive 2017

14-15 November 2017



2 days,
4 stages,
33 speakers,
40 lectures,
1500 participants

- you have to be there too!



codedive.pl

powered by **NOKIA**

Listing 5. Null pointer dereference

```
int main()
{
    std::string* ptr{};
    std::cout << *ptr;
}
```

6. NULL REFERENCE

Ponieważ referencje są implementowane jako wskaźniki, tylko bez ich arytmetyki, uzyskanie wyjścia z procesu za pomocą null referencji jest bardzo zbliżone.

Listing 6. Null reference

```
int main()
{
    std::string& ref= *(std::string*)nullptr;
    std::cout << ref;
}
```

7. WISZĄCY WSKAŹNIK

Tak zwane „wiszące wskaźniki” (ang. *dangling pointers*) to wskaźniki do obiektów, których czas życia już się skończył. W Listingu 7 przedstawiono użycie wskaźnika na obiekt, którego zasoby zostały już zwolnione, a czas życia zakończony.

Listing 7. Dangling pointer

```
std::string* dangle()
{
    std::string str{"foo"};
    return &str;
}

int main()
{
    auto str = dangle();
    std::cout << *str;
}
```

8. WISZĄCA REFERENCJA

Analogicznie do przykładów z pustym wskaźnikiem oraz referencją, tzw. „wisząca referencja” (ang. *dangling reference*) to referencja do obiektu, którego istnienie zostało już zakończone. Autor uważa, że o ile w przykładzie z Listingu 7 użyty kompilator bardzo wyraźnie ostrzegął o niebezpieczeństwie, to już w przykładzie z Listingu 8 tak nie było.

Listing 8. Wisząca referencja

```
std::string& dangle()
{
    std::string str{"foo"};
    return str;
}

int main()
{
    auto str = dangle();
    std::cout << str;
}
```

9. DOUBLE FREE

Różne implementacje stosują różne rozwiązania dla problemu alokatorów. Często są to bardzo skomplikowane mechanizmy rozdzielające alokacje pod względem ich wielkości, aktualnego stanu systemu, dostępnych *freelist* [F], przewidywanego sposobu użycia pamięci przez program, a także pod paroma względami, które wychodzą poza pojęcie autora tego artykułu.

Z tego powodu funkcja `free`, odpowiedzialna w idiomatycznym C za zwalnianie zasobów pamięciowych, nie przewiduje nadmiarowej weryfikacji, czy przekazany wskaźnik jest wskaźnikiem na początek zaalokowanej pamięci i czy jest zaalokowany w ogóle. W Listingu 9 przedstawiono kod skutecznie wywołujący crash na systemie autora².

Listing 9. Double free

```
int main()
{
    void* arr[1024];
    for(int i = 0; i < 1024; i++) {
        arr[i] = malloc(1024);
        free(arr[i]);
    }
    for(int i = 0; i < 1024; i++) {
        free(arr[i]);
    }
}
```

10. DOUBLE DELETE

Problem jest podobny do tego opisanego w sekcji poświęconej `free`, z tym że `delete` ma do dyspozycji trochę więcej danych: typ (a zatem i wielkość) obiektu, który usuwa. Z drugiej strony, o ile nie jest to obiekt trywialnie kopiowalny (ang. *trivially copyable*) [3], to musi zostać wywołany jego destruktor. W Listingu 10 przedstawiono wywołanie błędu.

Listing 10. Double delete

```
int main()
{
    std::string* arr[1024];
    for(int i = 0; i < 1024; i++) {
        arr[i] = new std::string;
        delete arr[i];
    }
    for(int i = 0; i < 1024; i++) {
        delete arr[i];
    }
}
```

11. FREE – OFFSET

Funkcja `free` musi otrzymać dokładnie tę samą wartość, która została zwrócona przez `malloc`/`calloc`/`realloc` (lub funkcję wywołującą jedną z nich). Dlatego też niedopuszczalne jest przekazanie zinkrementowanego wskaźnika, tak jak to się dzieje w Listingu 11.

Listing 11. Niepoprawny wskaźnik przekazany do free

```
int main()
{
    int* arr = (int*)malloc(sizeof(int) * 64);
    for(int i = 0; i < 42; i++) {
```

2. Arch Linux x86_64, zbudowany przez gcc z parametrami `-std=c++1z -g -fPIC`.

```

    *arr = i;
    arr++;
}
free(arr);
}

```

12. DELETE[] – OFFSET

Podobnie jak to jest w przypadku funkcji `free`, `delete[]` również musi otrzymać wskaźnik na początek zaalokowanej tablicy. W praktyce w większości kompilatorów `delete[]` trzyma wartość oznaczającą długość zaalokowanej tablicy bezpośrednio przed jej pierwszym elementem – i tam właśnie implementacja będzie jej szukać. Przykład został przedstawiony w Listingu 12.

Listing 12. Delete na przesuniętym wskaźniku

```

int main()
{
    int* arr = new int[64];
    for(int i = 0; i < 42; i++) {
        *arr = i;
        arr++;
    }
    delete[] arr;
}

```

13. DELETE VS DELETE[]

Mieszanie tych dwóch operatorów to dość typowy błąd nowicjuszy. Pierwszy służy do zwalniania pojedynczych obiektów, a drugi do dynamicznych tablic. Ich pomylenie ze sobą zawsze skutkuje tzw. UB (niezdefiniowane zachowanie, ang. *undefined behavior*), chociaż tylko w jednym kierunku można ją łatwo przekuć na crash aplikacji. Jest to bardzo prawdopodobne w sytuacji, gdy do `delete[]` zostanie podany wskaźnik na jeden obiekt. Zostało to zobrazowane w Listingu 13.

Listing 13. Delete[] użyte zamiast delete

```

int main()
{
    delete[] new std::string;
}

```

14. DELETE[] ZAMIAST FREE

Tak jak zostało to już wcześniej opisane, w przypadku typów, które nie są trywialnie kopiowalne, `delete[]` wywołuje też destruktor. Ponadto sprawdzanie wielkości zaalokowanej pamięci może być realizowane w inny sposób niż robi to `free`. W Listingu 14 przedstawiono wykorzystanie takiego błędu.

Listing 14. Delete[] użyte zamiast free

```

int main()
{
    auto* arr = (std::string*)malloc(sizeof(int) * 1024);
    delete[] arr;
}

```

15. FREE ZAMIAST DELETE

Przypadek prawie dokładnie odwrotny do przypadku 14. W kodzie przedstawionym w Listingu 15 destruktor nie zostanie wywołany,

co spowoduje, że utworzony wątek będzie pisał po pamięci bufora po jego czasie życia.

Listing 15. Free użyte zamiast delete

```

struct Thread
{
    Thread(std::function<void()> f) : t{f} {}

    ~Thread(){ t.join(); }
    std::thread t;
};

int main()
{
    char buf[1024];
    auto t = new Thread([&buf]{
        auto b = chrono::steady_clock::now();
        int i = 0;
        while(chrono::steady_clock::now() - b < 500ms) {
            std::fill_n(buf, 1024, i % 10);
        }
    });
    free(t);
}

```

16. USE AFTER FREE

Użycie obiektu po jego czasie życia oraz po zwolnieniu go do systemu to dość częsty błąd w programach pisanych w C oraz napisanych w C++, ale w stylu C, co przekłada się na pominięcie dobrodziejstw RAI, a w szczególności Rule of Zero [4]. W Listingu 16 zawarto przykładowy kod wykazujący tę podatność.

Ciekawostka: błąd tej klasy występował w grze *Sim City*, ale nie powodował żadnych objawów, gdy gra była uruchomiona pod DOSem. Podczas jej testów na systemie Windows 95 okazało się, że gra nieustannie się crashuje. Błąd został odnaleziony przez programistów Microsoftu, a aby mu zapobiec, zaimplementowano funkcję, dzięki której system wykrywa, czy ta konkretna gra jest włączona, i przełącza alokator pamięci w specjalny tryb zezwalający na używanie pamięci po jej zwolnieniu [5].

Listing 16. Użycie pamięci po jej zwolnieniu

```

int main()
{
    char* arr[1024];
    for(int i = 0; i < 1024; i++) {
        arr[i] = (char*)malloc(1024);
        memset(arr[i], 0, 1024);
    }
    for(int i = 0; i < 1024; i++) {
        free(arr[i]);
        memset(arr[i], ' ', 64);
        printf("%s", arr[i]);
    }
}

```

17. USE AFTER DELETE

W tym przypadku jest to również użycie obiektu po czasie jego życia. Problem przedstawiono w Listingu 17.

Listing 17. Użycie pamięci po jej zwolnieniu

```

int main()
{
    auto str = new std::string("foo");
    delete str;
    cout << *str;
}

```

18. CZYSTY MALLOC DLA OBIEKTÓW C++

Obiekty z nietrywialnym konstruktorem wymagają jego uruchomienia do poprawnej inicjalizacji. malloc dostarcza tylko zaalokowaną pamięć. Można tę trudność ominąć, używając placement new [6], ale na potrzeby przykładu przedstawionego w Listingu 18 zostało to pominięte.

Listing 18. Użycie niezainicjalizowanej zmiennej

```
int main()
{
    auto str = (std::string*)malloc(sizeof(std::string));
    *str = "foo";
    std::cout << *str;
}
```

19. CAST NA ZŁY TYPI

Standardy C i C++ zezwalają na rzutowanie pomiędzy różnymi typami tylko w wyraźnie określonych warunkach. Nie jest tak tylko dlatego, że ich twórcy chcą utrudnić życie programistom, ale dlatego, że ich bezpośrednie reprezentacje bitowe są niekompatybilne. Przedstawione jest to w Listingu 19 na przykładzie rzutowania `std::vector` na `std::list`.

Listing 19. Niepoprawne rzutowanie

```
int main()
{
    std::vector<int> foo{1, 2, 3};
    std::list<int>& as_vec = *(std::list<int>*)&foo;
    for(int el : as_vec) {
        std::cout << el << std::endl;
    }
}
```

20. REINTERPRET_CAST PRZY WIEŁODZIEDZICZENIU

Wśród wielu programistów pokutuje przeświadczenie, że `reinterpret_cast` jest bardziej ogólną wersją pozostałych rzutowań wprowadzonych przez C++. Jest to przeświadczenie błędne, ponieważ rzutowania te różnią się nie tylko możliwościami, ale również semantyką w pozornie identycznych zastosowaniach. W Listingu 20 przedstawiono taką różnicę – `reinterpret_cast` zachowa bezpośrednią wartość wskaźnika na klasę bazową, gdzie `static_cast` przesunąłby ją o odpowiedni offset, aby wskazywał na rzeczywisty obiekt klasy `Derived`.

Listing 20. Niepoprawne rzutowanie z klasy bazowej

```
struct Base1
{
    void print() { std::cout << str; }
    std::string str;
};

struct Base2
{
    uintptr_t ptr;
};

struct Derived : Base1, Base2
{
    Derived():
        Base1{"foo"},
        Base2{42}
    {}
};
```

```
void print(Base2* ptr)
{
    reinterpret_cast<Derived*>(ptr)->print();
}

int main()
{
    Derived d;
    print(&d);
}
```

21. REINTERPRET_CAST PRZY KLASACH POLIMORFICZNYCH

Ten przykład jest bardzo zbliżony do poprzedniego. Istotą rzeczy jest fakt, że klasa `Base` nie jest polimorficzna, a `Derived` już tak. Wszystkie znane autorowi implementacje realizują polimorfizm przez technikę *vtable*, czyli tablicę wskaźników na klasy bazowe i funkcje wirtualne. Wskaźnik na *vtable* jest pierwszym elementem klasy – nawet przed klasami bazowymi. Dlatego też ponownie `reinterpret_cast`, bezmyślnie zmieniając typ bez niezbędnej kalkulacji faktycznego adresu bazowego, w rezultacie da błędny wynik. Efektem uruchomienia kodu z Listingu 21 jest, zgodnie z oczekiwaniami, crash.

Listing 21. Niepoprawne rzutowanie z klasy bazowej

```
struct Base
{
    std::string foo;
};

struct Derived : Base
{
    virtual void print() { std::cout << foo; }
};

void print(Base* ptr) {
    reinterpret_cast<Derived*>(ptr)->print();
}

int main()
{
    Derived d;
    print(&d);
}
```

22. C-CAST NA NIEKOMPLETNY TYPI

W C++ rzutowanie w stylu C jest realizowane przez sekwencję prób czterech rzutowań w stylu C++ w kolejności narastającej pod względem „niebezpieczeństwa” (`static_cast`, `static_cast + const_cast`, `reinterpret_cast`, `reinterpret_cast + const_cast`). Gdy jedno się nie uda, podejmowane jest kolejne z listy.

Oznacza to, że użycie go w przykładach 20 i 21 zapobiegłoby crashowi, chociaż wciąż nie mogłoby być traktowane jako rozwiązanie poprawne. Nie stanie się tak jednak w Listingu 22, ponieważ w momencie definicji funkcji `from_base2` klasa `Derived` nie jest jeszcze zdefiniowana, a zatem `static_cast` się nie powiedzie. Crasha można uniknąć, przenosząc definicję klasy `Derived` nad definicję `from_base2`.

Listing 22. Niepoprawne rzutowanie na niekompletny typ

```
struct Base1
{
    void print() { std::cout << str; }
    std::string str;
};
```

```

};

struct Base2
{
    uintptr_t ptr;
};

struct Derived;

Derived* from_base2(Base2* ptr)
{
    return (Derived*)(ptr);
}

struct Derived : Base1, Base2
{
    Derived():
        Base1{"foo"},
        Base2{42}
    {}
};

int main()
{
    Derived d;
    from_base2(&d)->print();
}

```

23. NIEZALIGNOWANY DOSTĘP

Wiele architektur, w tym x86-64³, oczekuje, że obiekty będą umieszczone w pamięci pod adresami spełniającymi odpowiednie warunki, co najczęściej oznacza bycie wielokrotnością odpowiedniej potęgi liczby dwa. W Listingu 23 przedstawiono taki dostęp, jednak pozwala on na uzyskanie crasha tylko na niektórych architekturach.

Listing 23. Niezalignowany dostęp do danych

```

int main()
{
    char* buf = (char*)malloc(5);
    *(int*)(buf+1) = 42;
    return *(int*)(buf+1);
}

```

24. TYPE PUNNING

Type punning to technika obchodzenia ograniczeń nałożonych przez system typów języka. Według reguł kompilator nie musi śledzić zmian danych dokonanych przez wskaźnik innego typu (z pewnymi wyjątkami, np. klasy w jednym drzewie dziedziczenia lub `char*` i jego warianty). Na przykładzie kodu z Listingu 24: kompilator może uznać, że zapis do `*s` nie zmodyfikuje `val` (ponieważ mają niekompatybilne typy), więc może uznać `!*u` za wyrażenie stałe o wartości `true`.

Listing 24. Type punning

```

void pun(int64_t* s, uint64_t* u)
{
    *u = 0;
    *s = 1;

    if(!*u) {
        throw std::runtime_error("bye");
    }
}

int main()
{
    uint64_t val{};
    pun((int64_t*)&val, &val);
}

```

3. Chociażby dla instrukcji wektorowych, np. `movdq`, którą mogą wygenerować kompilatory optymalizujące.

25. NIEZŁAPANY WYJĄTEK

Rzucenie wyjątku, który nie zostanie nigdzie złapany, oznacza natychmiastowe zamknięcie programu. Należy zauważyć, że program zamyka się sam (wywołując `std::terminate()`), więc nie podpada to pod wąską definicję słowa *crash*. Przykład można znaleźć w Listingu 25.

Listing 25. Niezłapany wyjątek

```

int main()
{
    throw std::runtime_error("bye");
}

```

26. DWA WYJĄTKI JEDNOCZEŚNIE

Rzucenie wyjątku w momencie, gdy aktywny jest inny wyjątek, również powoduje natychmiastowe wywołanie `std::terminate()`. Jest to główny powód, dla którego złą praktyką w C++ jest rzucanie wyjątków w destruktorach – destruktory obiektów o automatycznym czasie życia wywoływane są automatycznie przy wyjściu z zakresu, również w przypadku rzucenia przez kod wyjątku. W Listingu 26 naświetlono ten problem.

Listing 26. Dwa aktywne wyjątki jednocześnie

```

struct DestructorThrower
{
    ~DestructorThrower()
    {
        throw std::runtime_error("bye");
    }
};

int main()
{
    try {
        DestructorThrower dt;
        throw std::runtime_error("bye");
    } catch (std::exception const&) {}
}

```

27. INTEGER OVERFLOW

Kompilatory optymalizujące podchodzą do UB w bardzo prosty sposób: jest to błąd, więc zakładają, że w poprawnym kodzie nigdy się nie wydarzy, a ścieżki wywołań, które do niego by prowadziły, odrzucają jako niemożliwe.

W kodzie z Listingu 27 pętla warunkowa staje się pętlą nieskończoną, ponieważ warunek jest uznany za zawsze prawdziwy. Może to być zaskakujące dla wielu osób, które mogą się spodziewać, że przy osiągnięciu wartości maksymalnej licznik „przekreśli się” do `INT_MIN` lub do zera, tak jak by to się stało dla typu bez znaku.

Listing 27. Integer overflow

```

int main()
{
    for(int i = INT_MAX-10; i > 0; ++i) {
        new char[65536];
    }
}

```

28. DZIELENIE PRZESZ ZERO

Dzielenie przez zero jest zakazane przez standard. Autor słyszał również, że jest to ogólnie dość kłopotliwa sprawa.

Listing 28. Dzielenie przez zero

```
int main()
{
    return 1/0;
}
```

29. NIEPOPRAWNE DZIELENIE U2

Jeśli wynik dzielenia nie mieści się w zakresie typu wynikowego, na niektórych architekturach rzucany jest wyjątek. Jest tak w przypadku dzielenia wartości minimalnej liczby całkowitej ze znakiem przez -1 w systemach reprezentujących je za pomocą kodu U2.

Listing 29. Niepoprawne dzielenie U2

```
volatile int val;

int main()
{
    val = -1;
    cout << INT_MIN/val;
}
```

30. PRZESŁANIANIE NAZW

Przesłaniając nazwy, programista może omyłkowo użyć nie tego obiektu, którego chciał. Przykład przedstawiono w Listingu 30.

Listing 30. Przesłanianie nazw

```
struct foo
{
    char const* bar;
    foo(char const* bar) {
        bar = bar;
    }
};

int main()
{
    foo f{"foo"};
    std::cout << f.bar;
}
```

31. MODYFIKACJA LITERAŁÓW TEKSTOWYCH

Wszystkie standardy C i C++ zabraniają modyfikacji literałów tekstowych, a ich typ to `char const[ ]`. Pomimo tego, ze względu na kompatybilność z kodem napisanym przed jakąkolwiek standaryzacją, C i C++98 zezwalają na przypisanie takiej tablicy do mutowalnego wskaźnika na `char`. Od C++11 jest to nielegalny zapis [9].

Listing 31. Modyfikacja literałów tekstowych

```
int main()
{
    char* evil = "foo";
    strcpy(evil, "bar");
}
```

32. MODYFIKACJA STAŁYCH

Próba modyfikacji stałych, które zwykle zapisane są w segmencie `.rodata`, również skończy się niepowodzeniem.

Listing 32. Modyfikacja stałych

```
const int arr[] = {1,2,3,4,5};

int main()
{
    memset(const_cast<int*>(arr), 0, sizeof(arr));
}
```

33. WYKONYWANIE DANYCH

W zamierzczłych czasach procesory traktowały pamięć jednakowo i to od użytkownika zależało, czy jest ona danymi, czy kodem. Jedną z pierwszych technik zapobiegania wykorzystaniu powstałych z tego powodu luk bezpieczeństwa (podane przez użytkownika odpowiednio spreparowane dane mogły być wykonane jako kod programu) było podzielenie pamięci procesu na bloki o różnych uprawnieniach. Z reguły blok danych nie może zostać wykonany, więc przykład z Listingu 33 spowoduje nagłe zakończenie procesu ze względów bezpieczeństwa.

Listing 33. Próba wykonania danych jako kodu

```
int main()
{
    char ops[] = "\xc3";
    using function_ptr = void(*)();
    auto ptr = (function_ptr)(ops);
    ptr();
}
```

34. SYNCHRONIZACJA SLEEPEM

Popularną, choć błędną, techniką synchronizacji jest opóźnienie wykonania jednego z wątków o jakiś interwał. Najczęściej jest on dobrany w taki sposób, aby umożliwić innym wątkom dokonanie niezbędnych inicjalizacji i ustawień obiektów. Niemniej jednak, jest to rozwiązanie błędne na wszystkich systemach z planistą, jak również co najmniej wątpliwe na systemach czasu rzeczywistego. W Listing 34 przedstawiono program wykorzystujący funkcję `sleep` do synchronizacji.

Listing 34. Synchronizacja sleepem

```
int main()
{
    std::string out;

    std::thread t1{[&]{
        while(true) {
            std::this_thread::sleep_for(2ms);
            out.push_back('A');
        }
    }};

    std::this_thread::sleep_for(1ms);

    std::thread t2{[&]{
        while(true) {
            std::this_thread::sleep_for(2ms);
            out.erase(out.begin());
        }
    }};

    t1.join();
    t2.join();
}
```

35. URWANE ODCZYT/ZAPISY

Kolejną popularną i błędną praktyką w C i C++ jest używanie zmiennych kwalifikowanych jako `volatile` do synchronizacji międzywątkowej. Kwalifikator ten ma zupełnie inne znaczenie: wyłącza jedynie optymalizacje przy dostępie do zmiennej, nie wpływając zupełnie na synchronizację tego dostępu. Prawidłowym rozwiązaniem problemu synchronizacji jest użycie biblioteki kompletnie ukrywającej go przed programistą, a jeśli jest to nie- możliwe – wykorzystanie typów `std::atomic` lub `std::mutex`.

W Listingu 35 przedstawiono kod zapisujący w dwóch wątkach dwie różne liczby do 64-bitowej zmiennej: `0000 0000 0000 000116` oraz `0000 0001 0000 000016`. Jak widać, mają one tylko jeden „zapalony” bit, odpowiednio pierwszy i trzydziesty trzeci. Na systemach 32-bitowych⁴ 64-bitowy zapis realizowany jest jako dwa 32-bitowe zapisy, co w przypadku tych liczb przekłada się na zapisy `0000 000116` oraz `0000 000016`, lub to samo w odwrotnej kolejności.

Ponieważ nie odbywa się żadna synchronizacja, przez krótką chwilę `val` przybiera wartość różną od obu przypisywanych liczb, może to być `0000 0001 0000 000116` lub `0000 0000 0000 000016`. Dlatego też faktycznie możliwe jest rzucenie wyjątku „Impossible value”.

Listing 35. Urwane zapisy/odczyty do 64-bitowej zmiennej

```
volatile uint64_t val = 1;

int main()
{
    thread t{[&]{
        for(int i = 0; i < 1000000; ++i) {
            val = 0x100000000u;
            if(val == 0x100000001u){
                throw std::runtime_error("Impossible value!");
            }
        }
    }};

    for(int i = 0; i < 1000000; ++i) {
        val = 0x000000001u;
        if(val == 0x100000001u){
            throw std::runtime_error("Impossible value!");
        }
    }

    t.join();
}
```

36. UŻYCIE UPRZYWILEJOWANEJ INSTRUKCJI

Niektóre instrukcje mogą być wykonane wyłącznie w kontekście *ring0 - kernel mode* [A] – najwyższego poziomu uprzywilejowania w systemie, posiadającego bezpośredni dostęp do sprzętu, przeznaczonego dla jądra systemu. Jedną z takich instrukcji na x86 jest `hlt`. W Listingu 36 przedstawiono kod programu, który wywołany w kontekście użytkownika (z reguły *ring3 - user mode*) zostanie zabity przez system.

Listing 36. Użycie uprzywilejowanej instrukcji `hlt`

```
int main()
{
    asm("hlt;");
}
```

4. Ten trik z bardzo dużym prawdopodobieństwem będzie działał wyłącznie dla 32-bitowego kodu, choć na 64-bitowej architekturze można spróbować analogicznej techniki z niestandardowym 128-bitowym typem danych.

37. UŻYCIE NIEZDEFINIOWANEJ INSTRUKCJI

Niezdefiniowane instrukcje powinny zatrzymać wywołanie programu. Przykładem tego może być instrukcja `ud2` (Listing 37), która jest zdefiniowana jako niezdefiniowana [B].

Listing 37. Użycie niezdefiniowanej instrukcji `ud2`

```
int main()
{
    asm("ud2;");
}
```

38. NIEPOPRAWNY TYP PODANY DO PRINTF

Funkcja `printf` nie rozpoznaje typów podanych argumentów i ślepo wierzy programiście, że wszystko podał poprawnie. Jeśli tak nie jest, tak jak to pokazano w Listingu 38, można spodziewać się crasha.

Listing 38. Niepoprawny typ podany do `printf`

```
int main()
{
    printf("%s", 42);
}
```

39. NIEPOPRAWNA LICZBA ARGUMENTÓW PODANA DO PRINTF

Funkcja `printf` nie wie, ile dostała argumentów, kieruje się w tym zakresie ciągiem formatującym przekazany przez programistę. Jeśli jest on niezgodny z prawdą, tak jak jest to pokazane w przykładzie z Listingu 39, crash jest prawie pewny.

Listing 39. Niepoprawna liczba argumentów podana do `printf`

```
int main()
{
    printf("%d %s %s %s", 42);
}
```

40. PRINTF Z %N (VC++)

Jedną z mniej znanych funkcji funkcji `printf` jest zapisanie zapisanej dotychczas liczby znaków do zmiennej za pomocą formatu `%n`. Jest to sytuacja wyjątkowa, ponieważ jest to jedyny przypadek, w którym funkcje z rodziny `printf` traktują argument z formatu jako wyjściowy.

Z powodu małej przydatności faktycznej – jedyna znana autorowi aplikacja takiego działania to pisanie zaciemnionego złośliwego kodu – kompilator Visual C++ domyślnie ma ją wyłączoną, a kod z Listingu 40 powoduje crash.

Listing 40. `Printf` z formatem `%n`

```
int main()
{
    int n;
    printf("%n", &n);
    return 0;
}
```

41. PRZEKAZANIE WARTOŚCI ZAMIAST ADRESU DO SCANF

Podobnie jak funkcja `printf`, `scanf` nie jest w stanie rozpoznać typów argumentów i ślepo wierzy ciągowi formatującemu. Aby zapisać wartość do obiektu, potrzebny jest jego adres, a jeśli zostanie podane coś innego, to zostanie potraktowane jak adres. W przypadku z Listingu 41 wartość `n (0)` zostanie potraktowana jak adres, czyli `scanf` zapisze wczytaną liczbę pod adres `0` zamiast pod adres `n`. Jest to częsty błąd popełniany przez nowicjuszy.

Listing 41. Przekazanie wartości zamiast adresu do `scanf`

```
int main()
{
    int n = 0;
    scanf("%d", n);
}
```

42. MIESZANIE LITERAŁÓW ÓSEMKOWYCH Z DZIESIĘTNymi

Literały ósemkowe tworzy się, zaczynając liczbę od zera, po którym następują wyłącznie cyfry z zakresu $[0, 7]^5$. Wobec tego, o ile 64_{10} , to 064_8 , czyli 52_{10} . Podając niecelowo błędną wartość w programie, nie jest trudno o błąd, co przedstawiono w Listingu 42.

Listing 42. Mieszanie literałów ósemkowych z dziesiętnymi

```
int main()
{
    char buf[064];
    memset(buf, 0, 64);
}
```

43. MEMSET NA KLASIE POLIMORFICZNEJ

Częstą praktyką w C jest użycie funkcji `memset` do inicjalizacji całej struktury lub tablicy. W C++ jest to akceptowalne wyłącznie w przypadku typów *Plain Old Data* [C]. Przedstawiona w Listingu 43 klasa `bar` zawiera niejawni element – wskaźnik na `vtable` (a przynajmniej jest tak we wszystkich popularnych implementacjach), który również zostaje nadpisany przez `memset`. Dlatego też wywołanie funkcji `get_n` powoduje crash.

Listing 43. `Memset` na klasie polimorficznej

```
struct foo
{
    virtual ~foo(){}
    virtual int get_n()=0;
};

struct bar : foo
{
    int get_n() override {
        return n;
    }
    int n;
};

int main()
{
    auto b = std::make_unique<bar>();
    memset(b.get(), 42, sizeof(*b));
    return b->get_n();
}
```

44. PURE VIRTUAL CALL – KONSTRUKTOR

Funkcje czysto wirtualne, czyli takie, które są zadeklarowane jako `=0` i których definicja nie została podana, nie mogą zostać wywołane – służą wyłącznie za wymóg dla klas dziedziczących, jako namiastka interfejsów z innych języków. Podczas tworzenia obiektów tworzone są najpierw klasy bazowe, a dopiero potem dziedziczące. Z tego powodu podczas tworzenia klasy bazowej, `vtable` kieruje na implementacje (lub ich brak) funkcji wirtualnych tejże klasy. W przypadku `foo` z Listingu 44 oznacza to, że konstruktor `foo` spróbuje wywołać funkcję `twice_n`, a co za tym idzie `get_n` na obiekcie `foo`, który tej funkcji nie definiuje. Oczywistym efektem takiego działania jest crash.

Listing 44. Pure virtual call w konstruktorze

```
struct foo
{
    foo() { twice_n(); }
    int twice_n() { return get_n() * 2; }
    virtual ~foo(){}
    virtual int get_n()=0;
};

struct bar : foo
{
    int get_n() override {
        return n;
    }
    int n;
};

int main()
{
    bar b;
}
```

45. PURE VIRTUAL CALL – DESTRUKTOR

Podobnie jak w przypadku 44, wywołanie funkcji czysto wirtualnej z destruktora obiektu jest równie łatwe, co przedstawiono w Listingu 45.

Listing 45. Pure virtual call w destruktorze

```
struct foo
{
    foo(){}
    int twice_n() { return get_n() * 2; }
    virtual ~foo() { twice_n(); }
    virtual int get_n()=0;
};

struct bar : foo
{
    int get_n() override {
        return n;
    }
    int n;
};

int main()
{
    bar b;
}
```

46. STATIC W NAGŁÓWKACH JAKO ZMIENNE GLOBALNE

Słowo kluczowe `static` ma w C i C++ wiele znaczeń. Jednym z nich, istotnym w tym przykładzie, jest zmiana linkowania zmiennej na wewnętrzne (ang. *internal*), lokalne dla danej jednostki translacji

5. Ciekawostką jest fakt, że sama liczba `0`, z perspektywy gramatyki języków C i C++, też jest literałem ósemkowym – chociaż cyfry od `1` do `9` są już literałami dziesiętnymi.

(ang. *Translation Unit*, w uproszczeniu plik `.c/.cpp`). Oznacza to, że można zdefiniować `static int foo`; w kilku plikach z kodem i w każdym będzie to inna zmienna. W kodzie z Listingu 46.3 oznacza to, że wskaźnik na literał tekstowy jest przypisany do zupełnie innej zmiennej, niż ta, z której korzysta funkcja `print_x`.

Listing 46.1 `foo.hpp`

```
void print_x();
static char const* x;
```

Listing 46.2 `foo.cpp`

```
void print_x() {
    std::cout << x;
}
```

Listing 46.3 `main.cpp`

```
int main()
{
    x = "foo";
    print_x();
}
```

47. WYKORZYSTANIE NIEZAINICJALIZOWANYCH ZMIENNYCH

Niestatyczne i nieglobalne obiekty w C i C++ nie podlegają zero-inicjalizacji, co oznacza, że jeśli nie są klasami z domyślnym konstruktorem, ich wartości są niezdeterminowane i nie wolno ich odczytać. W praktyce oznacza to, że mają wartości wyglądające na losowe – takie, jakie wcześniej były w zajmowanym przez nie miejscu w pamięci. Z tego korzysta przykład z Listingu 47, aby wywołać crash.

Listing 47. Odczyt niezainicjalizowanej zmiennej

```
int main()
{
    char* x;
    std::cout << x;
}
```

48. WYCIEKI PAMIĘCI

Program potrzebuje pamięci, aby działać. Jeśli stale ją alokuje, ale jej nie zwalnia, to prędzej czy później się ona skończy i niemożliwym będzie dalsza alokacja, np. na potrzeby stosu kolejnego wątku wykonania – tak jak jest to pokazane w Listingu 48.

Listing 48. Wyciek pamięci

```
int main()
{
    while(malloc(0x100000));

    std::thread t([](){});
    t.join();
}
```

49. WYCIEKI ZASOBÓW

Nie tylko wycieki pamięci są niebezpieczne, choć jest to najpopularniejszy zasób, który jest gubiony przez programy. Za przykład można wziąć wątki, których utworzyć również można ograniczoną liczbę, tak jak to pokazano w Listingu 49.

Listing 49. Wyciek zasobu

```
int main()
{
    std::mutex m;
    std::lock_guard<std::mutex> l{m};

    while(true) {
        std::thread{
            [&m]{ m.lock(); }
        }.detach();
        std::this_thread::sleep_for(1ms);
    }
}
```

50. UŻYCIE NIEWAŻNYCH ITERATORÓW

Różne kontenery z biblioteki standardowej C++ unieważniają iteratory przy różnych operacjach, co jest podyktowane ich zróżnicowaną budową. Na przykładzie `std::list` iterator do usuniętego elementu nie może być dalej wykorzystywany, nawet aby za jego pomocą dotrzeć do następnego elementu listy. Przykład problemu znajduje się w Listingu 50.

Listing 50. Użycie nieważnych iteratorów

```
int main()
{
    std::list<int> c;
    for(int i = 0; i < 64; i++)
        c.push_back(i);

    for(auto it = c.begin(); it != c.end(); ++it)
        c.erase(it);
}
```

51. ITERACJA MIĘDZY ITERATORAMI RÓŻNYCH KONTENERÓW

Idiomatyczne dla C++ pary iteratorów w żaden sposób nie gwarantują, że odnoszą się do tego samego obiektu (ani nawet, że typy kontenerów, których są iteratorami, są takie same!). Obowiązek zapewnienia tego spoczywa na programiście, a jeśli się tak nie stanie, to może nastąpić crash, tak jak w przykładzie z Listingu 51.

Listing 51. Iteracja pomiędzy różnymi kontenerami

```
int main()
{
    std::list<std::string> c1, c2;
    for(int i = 0; i < 64; i++) {
        c1.push_back(std::to_string(i));
        c2.push_back(std::to_string(i));
    }

    for(auto it = c1.begin(); it != c2.end(); ++it)
        std::cout << *it;
}
```

52. NIEPOPRAWNIE ZASTOSOWANY IDIOM ERASE-REMOVE

Drobna pomyłka, która pojawia się przy wywołaniu metody `erase` kontenera, może skończyć się crashem. W przykładzie z Listingu 52 pokazane jest takie wywołanie – funkcja `remove` nie znajdzie żadnego elementu do usunięcia, więc zwróci iterator na koniec wektora `v`, a funkcja `erase` spróbuje go usunąć. Programista miał tutaj zamiar użyć innego przeładowania `erase`, przyjmującego parę iteratorów [D].

Listing 52. Błędne użycie funkcji erase w powiązaniu z std::remove

```
int main()
{
    std::vector<int> v{1,2,3,4,5};
    v.erase(std::remove(v.begin(), v.end(), 42));
}
```

53. DEREFERENCJA „PUSTEGO” ITERATORA

Nie każdy iterator wskazuje na jakąś wartość, np. te zwrócone przez funkcję `std::end()` nie mogą być poddawane dereferencji.

Listing 53. Dereferencja pustego iteratora

```
int main()
{
    std::vector<int> v;
    std::cout << *std::end(v);
}
```

54. WYJĄTKI ZMIENNOPRZECINKOWE

Po włączeniu wyjątków zmiennoprzecinkowych za pomocą funkcji `feenableexcept` operacje, które normalnie zwróciłyby NaN, rzucają wyjątek, zabijając program. Przykład w Listingu 54.

Listing 54. Wyjątki zmiennoprzecinkowe

```
int main()
{
    feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW);
    sqrt(-1.0);
}
```

55. POLEGANIE NA KOLEJNOŚCI INICJALIZACJI ZMIENNYCH GLOBALNYCH

Zmienne globalne wewnątrz jednej jednostki translacji inicjalizowane są w kolejności definicji, ale kolejność inicjalizacji samych jednostek translacji nie jest w żaden sposób ustandaryzowana. Dlatego też poniższy przykład może crashować lub nie, w zależności od szczęścia.

Listing 55.1 foo.hpp

```
extern char const* foo;
extern std::string bar;
```

Listing 55.2 foo.cpp

```
std::string bar = foo;
```

Listing 55.3 main.cpp

```
char const* foo = time(nullptr) % 2 ? "foo" : "foo!";

int main()
{
    std::cout << bar;
}
```

56. RZUCENIE WYJĄTKU Z FUNKCJI OZNACZONEJ NOEXCEPT

Funkcja oznaczona jako `noexcept` nie powinna rzucać wyjątków. Jeśli jakiś wyjdzie z jej zakresu, standard C++ gwarantuje wywołanie `std::terminate`.

Listing 56. Funkcja noexcept rzucająca wyjątek

```
void foo() noexcept
{
    throw std::runtime_error("bye");
}

int main()
{
    try {
        foo();
    } catch (...) {}
}
```

57. UŻYĆCIE OBIEKTU PROXY PO CZASIE ŻYCIA JEGO WŁAŚCICIELA

Niektóre klasy, jak np. te reprezentujące działania na macierzach, oferują leniwy dostęp do swoich danych, ponieważ ich obliczenie jest kosztowne, a rzadko cała zawartość jest potrzebna. Dostęp taki realizowany jest przez obiekty proxy, czyli takie, które *ad-hoc* obliczają oczekiwaną wartość dopiero w momencie przypisania ich do obiektu wynikowego. Inną klasą stosującą podobną technikę jest `std::vector<bool>`, który przechowuje dane w formie uniemożliwiającej przekazanie do nich wskaźnika – nie można przecież zaadresować konkretnego bitu.

Niestety, takie obiekty często nie mogą być lub nie są ze względów wydajnościowych ściśle powiązane z kontenerem źródłowym, co wykorzystuje kod z Listingu 57. W tym przykładzie obiekt proxy odnosi się do 43 bitu wektora `v`. Przypisanie dowolnej wartości do proxy po tym, jak `v` został wyczyszczony, powoduje crash.

Listing 57. Użycie obiektu proxy po czasie życia jego właściciela

```
int main()
{
    std::vector<bool> v{};
    std::fill_n(v.begin(), 64, true);
    auto proxy = v[42];
    v.clear();
    proxy = false;
}
```

58. LD_PRELOAD Z BŁĘDNĄ FUNKCJĄ

Ustawienie zmiennej środowiskowej `LD_PRELOAD` na linuksach skutkuje tym, że wskazany przez tę ścieżkę plik zostanie załadowany przed wszystkimi innymi – z biblioteką standardową włącznie. Dzięki temu uruchomienie programu w sposób pokazany w Listingu 58 skutkuje crashem – o ile plik `bad_malloc.so` zostanie odpowiednio spreparowany (np. będzie eksportował crashującą funkcję `malloc`).

Listing 58. Załadowanie złośliwej implementacji malloc

```
> LD_PRELOAD=./bad_malloc.so ./a.out
```

59. MALLOC HOOK

`__malloc_hook` [E] to narzędzie służące do modyfikacji zachowania funkcji `malloc`. W Listingu 59 przedstawiono jego niepoprawne użycie, co skutkuje crashem.

Listing 59. Malloc hook

```
extern "C" int __malloc_hook=1;

int main()
{
    malloc(1);
}
```

60. NADPISANIE KODU MASZYNOWEGO PROCESU

Nadpisanie kodu maszynowego procesu jest skuteczną metodą na uzyskanie crasha. W przypadku z Listingu 60 użyty został do tego wirtualny plik `/proc/self/mem`.

Listing 60. Nadpisanie kodu maszynowego procesu

```
int tab[1024];

int main()
{
    FILE *f = fopen("/proc/self/mem", "wb");
    fseek(f, (size_t)main, SEEK_SET);
    fwrite(tab, 1, 1024, f);
    fclose(f);
}
```

61. NADPISANIE KODU MASZYNOWEGO PROCESU (SPOSÓB #2)

Innym wariantem jest alokacja nowej pamięci w miejscu funkcji `main`. Przykład z Listingu 61 pokazuje, jak to zrobić z `mmap`.

Listing 61. Nadpisanie kodu maszynowego procesu

```
int main()
{
    mmap(
        (void*)((size_t)main&~0xffffULL),
        4096,
        PROT_EXEC|PROT_READ,
        MAP_ANON|MAP_PRIVATE|MAP_FIXED,
        -1,
        0
    );
}
```

62. FORK BOMB

Nie jest to bezpośredni błąd w kodzie, jednak jest w stanie sprawić, że cały system przestanie odpowiadać, i jedynym sensownym rozwiązaniem będzie restart maszyny. W Listingu 62 przedstawiono przykładową implementację – jest to definicja funkcji w bashu, która rekursywnie będzie dwukrotnie (raz w tle) wywoływać samą siebie, efektywnie mnożąc się i doprowadzając niezabezpieczony system

do nieużywalności. Można się przed tym bronić poprzez nałożenie na użytkowników limitu jednocześnie uruchomionych procesów.

Listing 62. Fork bomb

```
:(){ :|: & };;:
```

63. PROMIENIOWANIE KOSMICZNE

Wbrew pozorom nie jest to humorystyczna część wpisu. Wraz z postępującą miniaturyzacją układów potrzebne są coraz mniejsze nakłady mocy, aby tranzystor „przeskoczył” z jednego stanu w drugi. Prawdopodobieństwo zaistnienia błędu jest zależne od okresu czasu, przez jaki program jest uruchomiony, oraz od ilości pamięci przez niego używanej. Między innymi z tego powodu powstają pamięci RAM typu ECC dla zastosowań w dziedzinach, gdzie błędy obliczeniowe są niedopuszczalne.

64. HALT AND CATCH FIRE

Halt and Catch Fire [7] (pol. *zatrzymaj się i zapal się*) – humorystyczna nazwa dla instrukcji powodującej kompletne zatrzymanie pracy procesora. Druga część nazwy odnosi się do tego, że procesor wykonujący taką instrukcję z reguły zamknięty jest w nieskończonej pętli, powodując przegrzanie i, w efekcie, zapłon.

Instrukcja ta z reguły nie występuje pod tym mnemonikiem, ponieważ jej istnienie w takim wariantcie jest błędem twórców procesora. W Listingu 64 przedstawiono tę instrukcję umieszczoną jako *easter egg* w zbiorze instrukcji procesora TIS-100 z gry TIS-100 [8].

Listing 64. Halt and Catch Fire

```
hcf;
```

Chciałbym serdecznie podziękować następującym osobom, bez których nie udało mi się dotrzeć do magicznej liczby 64 crashy. Są to: Gynvael Coldwind, Michał Kluczek, pingwindyktor oraz satirev (kolejność alfabetyczna, ignorując wielkość znaków).

Bibliografia:

- [1]: Tomasz Kwiecień, *Praktyczne Inżynieria Wsteczna*, s. 104-105, Warszawa, Polskie Wydawnictwo Naukowe, 2016 r., ISBN 978-83-01-18951-8
- [2]: <https://stackoverflow.com/a/2761494/2456565>
- [3]: <http://en.cppreference.com/w/cpp/concept/TriviallyCopyable>
- [4]: Paweł „KrzaQ” Zakrzewski, Piękny kod, *Programista* 5/2017, s. 16-21
- [5]: Raymond Chen, *The Best Software Writing I: Selected and introduced by Joel Spolsky*, s. 47, Apress, ISBN 1-59059-500-9.
- [6]: <https://goo.gl/Yj2X6A>
- [7]: https://en.wikipedia.org/wiki/Halt_and_Catch_Fire
- [8]: <https://alandesmet.github.io/TIS-100-Hackers-Guide/assembly.html>
- [9]: <https://goo.gl/78oE9C>
- [A]: https://en.wikipedia.org/wiki/Protection_ring
- [B]: http://x86.renejeschke.de/html/file_module_x86_id_318.html
- [C]: <http://en.cppreference.com/w/cpp/concept/PODType>
- [D]: <https://dev.krzaq.cc/post/an-erase-remove-idiom-gotcha/>
- [E]: <https://goo.gl/tr7uFb>
- [F]: https://en.wikipedia.org/wiki/Free_list

PAWEŁ "KRZAQ" ZAKRZEWSKI

<https://dev.krzaq.cc>

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Obecnie pracuje jako starszy programista w polskiej firmie. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.

Niekończąca się praca w toku, czyli zmora Product Ownera



Praca rozpoczęta i niezakończona w ustalonym terminie to coś, co często zdarza się przy pracy z niejednym produktem. Często winą obarczamy zespoły. Jednak powodów porażki możemy doszukiwać się także w działaniach PO. W dalszej części artykułu przedstawię Ci kilka przyczyn niezakończonych pracy oraz kilka rozwiązań, które pomogą lepiej sobie z nimi radzić.

CO SPRAWIA, ŻE PRACA NIE JEST KOŃCZONA NA CZAS?

Powodów wiecznej pracy w toku jest prawdopodobnie tyle, ilu ludzi pracuje w projekcie, a może jest ich nawet więcej. Poniżej postaram się przedstawić Ci te, z którymi spotykam się na co dzień.

Zależności zewnętrzne – jest to coś, na co zespół nie ma wpływu, a skoro nie masz na coś wpływu, to jak możesz dobrze to zaplanować? Tym bardziej jak możesz określić, kiedy skończysz zadanie?

Mało atrakcyjne zadania – to także coś, co nie pomaga w codziennej pracy. Jak byś się czuł, jeśli musiałbyś kolejny raz zrobić coś, co według Ciebie nie ma żadnej wartości?

„Nic się nie stało” – jeśli Twój zespół uważa, że nieskończenie pracy w terminie to „nic wielkiego” i nie niesie za sobą żadnych negatywnych konsekwencji, to dlaczego miałby ją kończyć na czas?

Niespodziewane „wrzutki” – jeśli dołożysz zespołowi pracy w trakcie sprintu, albo zmienisz priorytety dzień przed planowaniem, nie możesz się dziwić, że inne zadanie nie zostało skończone.

Niejasno komunikowane priorytety – mało przejrzysty Product Backlog sprawia, że ciężko jest się skupić na właściwym zadaniu i przygotować do sprintu.

Mało klarowne, skomplikowane zadania – jeśli nie wiesz, co dokładnie masz zrobić, ciężko jest Ci stwierdzić, czy skończyłeś.

Przyczyn niekończącej pracy jest z pewnością o wiele więcej. Zastanów się chwilę, czy odnajdujesz w swoim zespole którąś z wymienionych przyczyn? Mam nadzieję, że w dalszej części artykułu znajdziesz wskazówkę, jak poradzić sobie z tym problemem.

CO MOŻESZ ZROBIĆ?

Rozwiązania ściśle zależą od przyczyny problemu, pamiętaj zatem, że w każdym przypadku warto podejść do sprawy indywidualnie. Poniżej przedstawię kilka rad, które prawdopodobnie pozwolą Ci zredukować ilość pracy w toku.

Jasny i przejrzysty cel sprintu to coś, co pozwoli zespołowi trzymać się odpowiedniego kierunku prac i unikać pobocznych zadań, które często kuszą developerów. Ustal go razem z zespołem na planowaniu sprintu. Skoro masz już cel ustalony z zespołem, zapytaj ich na koniec planowania, w jaki sposób będą w stanie go osiągnąć? Jakie widzą wyzwania? Nie chodzi o to, żebyś prowadził wnikliwą analizę, co zamierzają zrobić. Po prostu pokaż, że interesujesz się tym, co robią, i że jest to ważne.

Pokaż, że niedokończona praca to problem. Jakie są jej konsekwencje? Niezadowolenie klienta, przesunięte terminy, straty dla firmy. Jest ich pewnie dużo więcej. Jedne są poważne, inne pewnie błahe, skup się zatem na tych pierwszych. Pamiętaj jednak, że wytwarzanie oprogramowania jest pracą skomplikowaną i nie zawsze developerzy są w stanie dokładnie wycenić, ile pracy będzie potrzebne na zrobienie danego zadania. Zbyt duża presja też może zaszkodzić. Twój klient także powinien mieć tego świadomość.

Zadbaj o to, żeby zespół, z którym pracujesz, wiedział, jak ocenisz jego pracę. Jesteś z niego zadowolony, czy nie? Dobrze przekazany i konstruktywny feedback potrafi zdziałać więcej niż nie jedna wyszukana technika. Rozmawiając z zespołem, przekazując informację zwrotną, nie zawsze musisz być miły, zawsze jednak powinieneś okazywać szacunek.

Zainteresuj się pracą niedokończoną. Nie prowadź śledztwa, ale zadaj pytanie: „Co się stało?”, „Czy dało się tego uniknąć?”, „Co zrobiliście, żeby się udało?”, „Jak ja mogłem Wam pomóc?”. Uwidocznij tę pracę, zwizualizuj, policz stratę. Pokażesz w ten sposób, że zależy Ci na tym, aby dostarczać pracę na czas, okażesz wsparcie i pokażesz, że jest to ważne.

Ustal z zespołem jasne reguły „gry”. Nie chodzi o to, żebyście spisali umowę na 30 stron, której już na zawsze będziecie się trzymać. Krótki „kontrakt” spisany markerem na flipcharcie, który będziecie budować i zmieniać np. na retrospektywie, może być ciekawym rozwiązaniem.

Zbuduj autorytet w zespole. Poznaj dobrze produkt, który razem budujecie. Poznaj także swój zespół, dowiedz się, jakie ma kompetencje i jak je wykorzystać. Znajomość technologii, w której pracujecie, i rynku produktu także będzie pomocna w zbudowaniu swojej pozycji. Bądź konsekwentny, pamiętaj, że to Ty jesteś osobą decyzyjną i masz ostatni głos co do kolejności wykonywania pracy.

Pokaż swojemu zespołowi, że masz dla niego czas i jego zdanie się liczy. Będzie Cię chętniej pytał, kiedy pojawią się wątpliwości. Zapytaj, czy ustalone priorytety są wykonalne, dowiedz się wcześniej, czy zadania nie są zablokowane, albo zbyt kosztowne, żeby implementować je w tym sprincie.

Jeśli widzisz, że zespół kolejny raz nie kończy tego samego zadania na czas i bierze kolejne zadania na sprint, porozmawiaj z nim

o tym. Zastanówcie się wspólnie, co jest przyczyną nieskończonej pracy i czy zespół powinien brać kolejne zadania, skoro nie udaje mu się skończyć tego, co zaczął.

Aktywnie uczestnicz w retrospektywie zespołu. Jest to idealna okazja do zastanowienia się, jak poprawić pracę w następnym sprincie.

Zapytaj o radę Scrum Mastera. Prawdopodobnie będzie wiedział, gdzie leży problem. Jeśli nie, to na pewno pomoże Ci rozwiązać ten problem.

PODSUMOWANIE

Wytwarzanie oprogramowania jest pracą skomplikowaną i często ciężko jest oszacować, ile czasu zajmą dane zadania. Dlatego tego unikamy. Przewidywalność w Scrum opiera się na stałej długości sprintu i estymowaniu przez zespół ilości pracy, którą są w stanie w nim wykonać. Warto mieć to na uwadze, podejmując jakiegokolwiek działania. Dlatego pamiętaj, żeby do każdego problemu starać się podejść indywidualnie. W pierwszej kolejności zastanów się, czy Ty możesz zrobić coś lepiej. Poobserwuj swój zespół, a przede wszystkim zastanówcie się razem, czy możecie lepiej współpracować.

Grafika autorstwa Małgorzaty Lamorskiej



MICHAŁ CICHY

Absolwent Elektroniki i Telekomunikacji oraz Zarządzania Projektami na Politechnice Wrocławskiej. Pasjonat metodologii zwinnych i pracy z ludźmi. Obecnie Scrum Master w Nokia Wrocław.

reklama



@PROGRAMISTAMAG



Praktyczna kryptografia: Hashowanie, podpisy cyfrowe i wyprowadzanie klucza

W dzisiejszych czasach bezpieczeństwo każdej istniejącej aplikacji internetowej silnie polega na skrótach kryptograficznych. Co więcej, znalezienie praktycznej metody ich odwracania wywołałoby globalny kryzys w całej informatyce. To argument, który sprawia, że warto się z nimi zapoznać. W tym artykule pokażemy, jak poprawnie używać hashy oraz jak skonstruowane są algorytmy HMAC i PBKDF2.

WPROWADZENIE: HASHOWANIE

Artykuł ten kierowany jest głównie do osób, które posiadają podstawowe doświadczenie z używaniem hashy kryptograficznych. W tym wprowadzeniu skupimy się na krótkim przypomnieniu podstawowych właściwości funkcji skrótu (ang. *hash*).

Hash kryptograficzny jest jednokierunkową funkcją jednego argumentu o następującej sygnaturze: `bytearray hash(bytearray input)`, gdzie wejście jest jakimś ciągiem bajtów o arbitralnej długości, a wyjście – ciągiem bajtów o stałej długości (np. 128 bitów w przypadku MD5). Funkcje te są deterministyczne (zawsze zwracają to samo dla tego samego wejścia) i cechują się tzw. efektem lawinowym – zmiana zaledwie jednego bitu na wejściu powoduje diametralną zmianę wartości całego hashu:

```
>>> hashlib.sha256('bezpieczenstwo').hexdigest()
'8c133246c0c91fc9df1f7f25e475d1e2626362cf0492df8015a79b847da196e4'

# dwukrotne obliczenie hasha tej samej wartości daje ten sam
wynik
>>> hashlib.sha256('bezpieczenstwo').hexdigest()
'8c133246c0c91fc9df1f7f25e475d1e2626362cf0492df8015a79b847da196e4'

# ale zmiana chociaż jednego znaku kompletnie go modyfikuje
>>> hashlib.sha256('bezpieczenstwa').hexdigest()
'd8ca11a4a877464d99218e84b28cd84371901a433edb93d8fac50e565ab6e122'
```

Co najważniejsze, możliwe jest szybkie obliczenie wartości funkcji $hash(x)$, natomiast funkcja $hash^{-1}(x)$ (wyliczenie oryginalnej wiadomości z hashu) jest zbyt kosztowna obliczeniowo, aby zrealizować ją w praktyce. Mówi się również, że hash jest funkcją pseudo-

losową (PRF), albowiem w przypadku idealnej funkcji wyniki nie powinny być możliwe do odróżnienia od czystej losowości. Dodatkowo funkcje skrótu powinny uniemożliwiać wygenerowanie tzw. kolizji, czyli dwóch różnych wiadomości, dla których funkcja skrótu daje tę samą wartość hashu.

To tyle, jeśli chodzi o przypomnienie tematu funkcji skrótu, przejdźmy teraz do omówienia popularnej abstrakcji nad nimi, czyli podpisów elektronicznych.

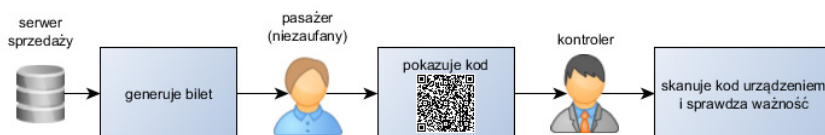
CZĘŚĆ I: PODPISY ELEKTRONICZNE

Protokoły oparte na podpisach

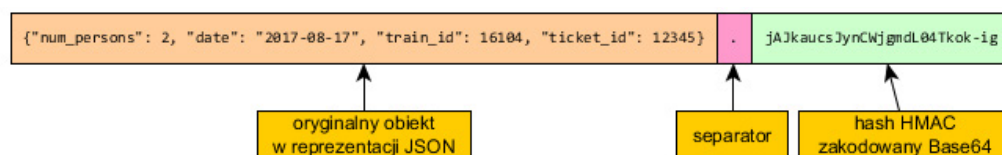
...w dzisiejszych czasach wykorzystujemy je niemal wszędzie. Jeden z intuicyjnych przykładów mogą stanowić chociażby elektroniczne bilety na pociąg (Rysunek 1). Jak to działa?

Załóżmy, że kod generowany dla użytkownika zawiera wszystkie kluczowe informacje, takie jak data i godzina odjazdu, stacja początkowa, końcowa, liczba osób itp. Dlaczego systemu nie da się oszukać? Prawdopodobnie zaprojektowane rozwiązanie tego typu wykorzystuje podpisy cyfrowe, najczęściej oparte albo na stosunkowo prostej technice „współdzielonego sekretu” (ang. *shared secret*), albo na kryptografii asymetrycznej¹. Pierwsza, mniej skomplikowana metoda wymaga tego samego klucza do wystawiania i do weryfikowania podpisów, natomiast druga posługuje się dwoma osobnymi kluczami służącymi do przeprowadzania tych czynności.

1. Nieprawidłowo zaprojektowane rozwiązanie tego typu wykorzystuje szyfrowanie. W poprzednim numerze „Programisty” w artykule „Praktyczna kryptografia: Szyfry blokowe” pokazywaliśmy, jak można łamać systemy tego typu na wszelkie możliwe sposoby.



Rysunek 1. Schemat działania biletów elektronicznych



Rysunek 2. Wynik działania funkcji `Serializer.dumps`

W tej części skupimy się na podpisach ze „współdzielonym sekretem”, które są stosunkowo proste i szybkie w użyciu (o ile wiemy, jak to zrobić dobrze!), jednocześnie zapewniając satysfakcjonujący poziom bezpieczeństwa w wielu zastosowaniach.

Jak zastosować taki podpis w naszej aplikacji? Python posiada do tego godny pochwały moduł `itsdangerous`:

Listing 1. Wykorzystanie `itsdangerous`² do podpisywania obiektu biletu (po stronie serwera wystawiającego bilet)

```
from itsdangerous import Serializer
s = Serializer('secret-key-123')
s.dumps({'ticket_id': 12345, 'train_id': 16104, 'date': '2017-08-17', 'num_persons': 2})
# wynik: '{"num_persons": 2, "date": "2017-08-17", "train_id": 16104, "ticket_id": 12345}.jAJkaucsJynCWjgmdL04Tkok-ig'
```

Co się stało? Utworzyliśmy obiekt `Serializer`, przekazując mu sekretny klucz `secret-key-123`. Następnie przekazaliśmy obiekt biletu do metody `dumps`, która skonwertowała go na JSON, a następnie wynikowy ciąg znaków podpisała. Podpis został dołączony na końcu tekstowej reprezentacji obiektu (Rysunek 2).

Następnym krokiem byłoby wygenerowanie kodu QR zawierającego podpisany bilet i przesłanie go do użytkownika, ale tę część intencjonalnie pominiemy (istnieją do tego odpowiednie biblioteki).

Czytnik biletów, po odczytaniu tekstu znajdującego się w kodzie QR, powinien wykonać następujące czynności:

Listing 2. Wykorzystanie `itsdangerous` do odczytania obiektu biletu wraz z weryfikacją poprawności podpisu

```
>>> from itsdangerous import Serializer
>>> # wartosc odczytana z kodu QR
>>> ticket = '{"num_persons": 2, "date": "2017-08-17", "train_id": 16104, "ticket_id": 12345}.jAJkaucsJynCWjgmdL04Tkok-ig'
>>> s = Serializer('secret-key-123')
>>> s.loads(ticket)
{'num_persons': 2, 'train_id': 16104, 'ticket_id': 12345, 'date': '2017-08-17'}
```

Co dokładnie zrobiła metoda `loads`? Ciąg znaków przekazany do niej został rozbity na dwa pod-ciągi (message oraz signature), odpowiednio: od początku do ostatniej kropki w ciągu oraz od ostatniej kropki do końca. Następnie wykonane zostało sprawdzenie, czy signature jest prawidłowym podpisem ciągu message dla klucza `secret-key-123`. Okazało się, że tak, w związku z tym pod-ciąg message został zdeserializowany do postaci obiektu.

Jeżeli w międzyczasie użytkownik dokonał nieautoryzowanej modyfikacji swojego biletu, otrzymamy wyjątek:

Listing 3. Nieautoryzowana modyfikacja użytkownika

```
>>> ticket = '{"num_persons": 7, "date": "2017-08-20", "train_id": 16104, "ticket_id": 12345}.jAJkaucsJynCWjgmdL04Tkok-ig'
>>> s.loads(ticket)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/itsdangerous.py", line 582, in loads
    return self.load_payload(self.make_signer(salt).unsign(s))
  File "/usr/local/lib/python3.5/dist-packages/itsdangerous.py", line 374, in unsign
    payload=value)
itsdangerous.BadSignature: Signature
b'jAJkaucsJynCWjgmdL04Tkok-ig' does not match
```

2. Warto podkreślić, że istnieje społecznościowy port tego modułu do PHP: <https://github.com/mat-tbasta/itsdangerous-php> oraz innych technologii.

Typowym przypadkiem wykorzystania podpisów cyfrowych jest sytuacja, kiedy potrzebujemy przekazać informację z zaufanego punktu A do zaufanego punktu B (tutaj: z serwera biletów do czytnika), ale poprzez niezaufany punkt C (czyli przez potencjalnie złośliwego użytkownika). Olbrzymią zaletą jest tutaj fakt, że weryfikacja poprawności podpisu może nastąpić offline – mobilny skaner biletów nie musi łączyć się z centralą, aby sprawdzić, czy bilet jest ważny.

W analogiczny sposób można zaimplementować kody rabatowe, kwity magazynowe, znaczki pocztowe i wiele innych rozwiązań.

Wiemy już, jak działają podpisy, widzieliśmy, jak działa biblioteka do podpisywania... pora zaimplementować to samodzielnie (np. w naszym nowym rewolucyjnym frameworku).

Hash to nie uwierzytelnianie, czyli dlaczego MD5(secret || data) to błąd

Posłużmy się teraz nieco prostszym przykładem: serwer A wystawia token upoważniający do zakupu jabłek z rabatem 10% i wysyła go klientowi na maila w postaci linku do serwisu hostowanego przez serwer B. Klient może kliknąć i skorzystać z rabatu. Serwer B zna współdzielony sekret użyty do wygenerowania tokenu, więc bez łączenia się z serwerem A weryfikuje poprawność tokenu znajdującego się w adresie URL i przyznaje rabat. Ruch użytkowników jest tak duży, że takie rozwiązanie jest znacznie bardziej korzystne niż utrzymywanie aktywnego połączenia pomiędzy serwerami A i B.

Jak stworzyć najbardziej intuicyjną i zarazem błędną implementację takiego rozwiązania?

Listing 4. Błędna implementacja podpisu cyfrowego za pomocą hasha (PHP)

```
$secret = 'pH2ef40hUdec3eSPdrujAy4gdec3eSPdrudec3eSPdru';
$promotion = 'name=rabat_na_jablka&amount=10'; // rabat 10%
$token = md5($secret . ':' . $promotion);

echo('Link promocyjny: http://serwer_b/rabat/' . $promotion . '/' . $token);
```

Sekret o takiej długości jest praktycznie niemożliwy do złamania, używamy funkcji hashującej, gdzie jest problem?

Pomijając wykorzystanie przestarzałego algorytmu MD5 (nigdy nie róbmy tego w nowych projektach!), jest to ewidentny przykład błędnego zrozumienia i błędnego wykorzystania prymitywu kryptograficznego, jakim jest funkcja hashująca. Czym w ogóle jest funkcja hashująca? Odwołajmy się do definicji.

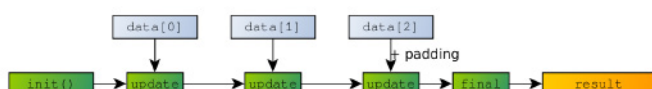
Kryptograficzna funkcja hashująca musi:

- » powodować, że próba wygenerowania wiadomości mającej wybraną wartość hasha jest niepraktyczna (tzw. *pre-image resistance*);
- » powodować, że próba znalezienia dwóch różnych wiadomości z tą samą wartością hasha jest niepraktyczna (tzw. *collision resistance*);
- » być szybka do policzenia dla dowolnej zadanej wiadomości (z powodów praktycznych; wyłączając niektóre przypadki, gdzie kosztowność obliczania hasha jest pożądana, np. hashowanie hasel).

Funkcja skrótu sprawdza się świetnie, jeżeli naszą intencją jest sprawdzenie integralności pliku (np. pobraliśmy go od innej osoby, która wcześniej wysłała nam SMSem hash SHA256). Jeżeli po pobraniu pliku policzymy hash i będzie on pasował do tego, który

otrzymaliśmy, to możemy być niemal pewni, że plik został pobrany prawidłowo, w całości i nie został w trakcie przez kogokolwiek zmodyfikowany. Wiele osób uważa, że w takim razie można wykorzystać je również bezpośrednio do podpisywania wiadomości, ale podejście to jest błędne – jak się zaraz przekonamy.

Funkcje hashujące MD5, SHA-1 i SHA-2 zbudowane są przy użyciu konstrukcji Merklego–Damgård, co sprawia, że wewnętrznie operują one na N-bajtowych blokach. Wykonywanie takiej funkcji hashującej zawsze rozpoczyna się od pewnego stanu początkowego, który jest ustalony na stałe w implementacji, a początkowo hashowanie dotyczy jedynie pierwszych N bajtów, a w wyniku tego zmienia się wewnętrzny stan funkcji hashującej. Dalej oddzielnie konsumowane i hashowane jest kolejne N bajtów wejścia, tym razem już z nowym stanem utworzonym przez poprzednią operację, i tak aż do momentu przetworzenia całej wiadomości.



Rysunek 3. Uproszczony schemat działania konstrukcji Merklea–Damgåarda

Co jednak najważniejsze, wynikowy hash eksponuje cały stan końcowy, który został wyliczony w wyniku przetworzenia wiadomości. Nic nie powstrzymuje nas przed załadowaniem tego stanu z powrotem i doliczeniem dodatkowego bloku do istniejącego hashu.

Powróćmy do wspomnianego przykładu z rabatem. Zamiast rabatu 10% zażyczymy sobie rabat 1000%. Nie ma konieczności łamania hasha ani sekretu. Możemy dodać kolejny blok do wiadomości i wyliczyć kolejny stan, w jakim znajdzie się funkcja hashująca.

Wykorzystajmy poprzedni przykład:

```
>>> hashlib.md5('pH2ef40hUdec3eSPdrujAy4gdec3eSPdrudec3eSPdru' +
                 ':name=ratat_na_jablka&amount=10').hexdigest()
'8b990fd727e79353a7fad6395fa7c4'
```

W tym przypadku znany użytkownikowi podpis to '8b990fd-f27e79353a7fadf639e5fa7c4', zna też treść podpisanej wiadomości, nie zna natomiast sekretu. Atak automatyzuje narzędzie hashpump:

[illegible]

Druga linia wyniku oznacza naszą nową wiadomość, a pierwsza podpis dla niej. Łatwo sprawdzić, czy atak byłby skuteczny:

[illegible]

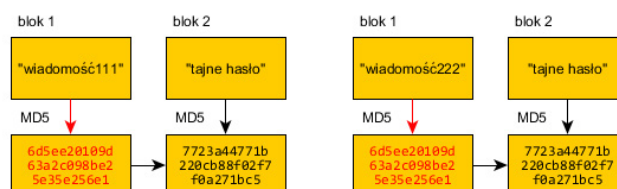
Faktycznie, podpis się zgadza. W ten sposób dokleiliśmy parametr amount drugi raz do wiadomości. Z uwagi na specyfikę mechanizmów parsowania parametrów w serwerach webowych wzięta pod uwagę będzie druga wartość i amount wyniesie 100.

Trudniejsze do złamania, wciąż niepoprawne:
MD5(message || secret)

W tym miejscu czas na pytanie: czy przedstawienie kolejnością sekretu oraz wiadomości rozwiąże problem? Mamy na myśli użycie: `hash(message || secret)`³. Niestety, nie. Uważny czytelnik po lekturze poprzednich akapitów zauważy, że konstrukcja Merklego-Damgård umożliwia wystąpienie kolizji nie tyle w obrębie hasha jako całości, co w poszczególnych blokach hashowanej wiadomości.

Znając wiadomość $message$ oraz wartość $hash(message)$, jeżeli znajdziemy wiadomość $message'$, taką że $message \neq message'$ oraz $hash(message) = hash(message' \parallel secret)$, to implikuje, że $hash(message \parallel secret) = hash(message' \parallel secret)$.

Inaczej mówiąc, jeśli atakujący zna jakąś kolizję w funkcji hashującej – czyli zna dwie wiadomości A i B , dla których wartość hasha jest taka sama – może „namówić” serwer do podpisania wiadomości A i będzie od razu znał poprawny podpis dla wiadomości B . Nie brzmi to równie źle jak problem z poprzedniego podpunktu.



Rysunek 4. Demonstracja kolizji na poziomie pierwszego bloku

Jak poważny jest to atak? Po pierwsze, osłabia siłę podpisu z 2^n (pełny atak siłowy na hash) do $2^{n/2}$ (atak urodzinowy w celu wygenerowania kolizji). Po drugie, każda znaleziona kolizja może zostać wykorzystana przeciwko nam. Na przykład firma Google niedawno opublikowała pierwszą znaną kolizję na algorytm SHA-1 – oznacza to, że teraz każdy może wykorzystać tę kolizję do przeprowadzenia ataku na taki mechanizm podpisywania.

Istnieje również bardziej praktyczny atak. Jeżeli znamy całą wiadomość oprócz sekretu, a jej długość jest wielokrotnością rozmiaru bloku, możliwe jest obliczenie wewnętrznego stanu, który zostanie osiągnięty po zhashowaniu wiadomości, ale przed zhashowaniem sekretu. Problem redukuje się wtedy do brute-force'owania funkcji MD5' (secret) z innym stanem początkowym, a to oznacza, że taki hash można łamać bez żadnego dodatkowego narzutu. Osiem kart Nvidia GTX 1080 jest w stanie łamać MD5 z szybkością 200 miliardów hashy na sekundę, co w praktyce oznacza, że ośmioznakowy sekret składający się z małych liter i cyfr da się na tym sprzecie złamać w niecałe 6 sekund!

CTF

Czytelników zachęcamy do samodzielnego spróbowania swoich sił z łamaniem się do aplikacji zawierającej podobne błędy: <https://var.tailcall.net/hmac>

Prawidłowe rozwiązanie: HMAC

Czas na przedstawienie prawidłowego rozwiązania. Do podpisywania dedykowane są tzw. Message Authentication Codes (MAC),

3. Podobny błąd w implementacji popełnili niektórzy polscy operatorzy serwisów z płatnościami online. Błędów tych nie da się łatwo wykorzystać wyłącznie przez szczęśliwy zbieg okoliczności (protokoły wymuszają dodatkowe sprawdzenia danych, więc „złośliwe” operacje zostaną prędko czy później wykryte i odrzucone).

które gwarantują prawidłowe uwierzytelnienie. Istnieje również podzbiór MACów skonstruowanych na bazie funkcji skrótu, czyli HMAC. Są to algorytmy zaprojektowane z myślą o podpisywaniu, które mają wyraźnie sprecyzowane osobne argumenty wiadomości oraz sekretu (hasła).

Operatory

Operatory i funkcje pomocnicze:

- ▶ $a \oplus b$ – wynik xora bitowego a i b ;
- ▶ $a \parallel b$ – złączenie ze sobą ciągów a i b (długość wynikowego ciągu jest sumą długości a i b);
- ▶ $\text{int32BigEndian}(\text{int})$ – liczba zapisana w postaci `uint32_t`, poczynając od najbardziej znaczącego bajtu.

Przyjrzyjmy się wewnętrznej konstrukcji HMAC, która nie jest skomplikowana:

$$\text{HMAC}(\text{hash}, \text{message}, \text{secret}) = \text{hash}((\text{secret} \oplus \text{opad}) \parallel \text{hash}((\text{secret} \oplus \text{ipad}) \parallel \text{message}))$$

$\text{opad} = 0x5c5c5c \dots 5c5c$

$\text{ipad} = 0x363636 \dots 3636$

...gdzie *opad* i *ipad* oznaczają kolejno *outer* i *internal padding* i są to stałe o długości takiej, jak długość bloku używanego hashu. Stałe są dobrane w taki sposób, aby dwie wersje klucza: $\text{secret} \oplus \text{opad}$ oraz $\text{secret} \oplus \text{ipad}$ miały jak najmniej wspólnych bitów. Z punktu widzenia HMAC ważne jest, aby dwie wersje klucza różniły się między sobą przynajmniej jednym bitem.

„Zagnieżdżona” konstrukcja HMACa (hash z hashu) oraz manipulacje z kluczem dodatkowo zabezpieczają używany schemat hashowania przed wspomnianymi wcześniej atakami związanymi z kolizjami. Inaczej mówiąc: używanie tego schematu istotnie łagodzi konsekwencje, które mogą wystąpić w wyniku znalezienia słabości w funkcji hashującej znajdującej się pod spodem.

Prawidłowe konstruowanie payloadu do HMACa

Do tego momentu omawialiśmy metody prawidłowego podpisywania jakiegoś abstrakcyjnego ciągu *message*, który zgodnie z definicją musi być niczym innym jak po prostu ciągiem bajtów. To zadowala nas od strony teoretycznej, ale w niektórych zastosowaniach wyjątkowo rzadko będziemy w posiadaniu bufora, który można by natychmiastowo podpisać.

Wróćmy do poprzedniego przykładu z biletami kolejowymi, gdzie naszą intencją jest podpisanie typu złożonego (obiektu, słownika). Programiści często są zmieszani faktem, że znana im literatura nie wspomina, jak prawidłowo przygotować „wsad” do funkcji HMAC, bowiem skupia się ona wyłącznie na wewnętrznej budowie funkcji.

Nieprawidłowe przygotowanie ciągu wejściowego stanowi świetny sposób na wytworzenie nowych luk bezpieczeństwa, które umożliwią ominięcie całego mechanizmu podpisywania. Przekonała się o tym firma MasterCard⁴ 5, kiedy indonezyjski badacz Yohanes Nugroho opracował sposób zmiany kwoty transakcji na 0 złotych, który nie powodował unieważnienia podpisu. Jak to możliwe, przecież używamy superbezpiecznego schematu podpisywania?

Taka sytuacja jest możliwa za każdym razem, kiedy wykorzystamy niejednoznaczny sposób kodowania danych. Uproszczony przykład podobnego ataku:

„Poprzez HTTP POST należy przekazać parametry: «numer_konta», «numer_przedmiotu», «ilosc_przedmiotu», «kwota». Oprócz tego należy dołączyć parametr «podpis», wyliczony zgodnie ze wzorem:

$$\text{HMAC}(\text{sekret}, \text{str}(\text{numer_konta}) + \text{str}(\text{numer_przedmiotu}) + \text{str}(\text{ilosc_przedmiotu}) + \text{str}(\text{kwota}))$$

... gdzie `str()` oznacza konwersję wartości na ASCII string, a operator plus oznacza złączenie ze sobą dwóch stringów”.

Po wykonaniu tej procedury tworzymy mniej więcej takie żądanie:

```
numer_konta=07290109873664746444176233&numer_
przedmiotu=123&ilosc_przedmiotu=1&kwota=10&podpis=cf23df2207d99
a74fbe169e3eba035e633b65d94
```

... gdzie obliczenie podpisu wygląda tak:

$$\text{HMAC}(\text{sekret}, \text{b"07290109873664746444176233123110"}) = \text{cf23df2207d99a74fbe169e3eba035e633b65d94}$$

... ale nic nie stoi na przeszkodzie, aby użytkownik posunął się do następującej manipulacji:

```
numer_konta=07290109873664746444176233&numer_
przedmiotu=123&ilosc_przedmiotu=11&kwota=0&podpis=cf23df2207d99
a74fbe169e3eba035e633b65d94
```

4. <http://tinyhack.com/2017/09/05/mastercard-internet-gateway-service-hashing-design-flaw/>

5. Jeden z autorów tego artykułu, Michał Leszczyński, w lutym 2015 roku znalazł podobny błąd w implementacji bramki płatności firmy Blue Media; błąd został poprawiony w ciągu kilku godzin.

reklama

Szkolenie dla Ciebie lub Twojego zespołu

Kryptografia na platformie Java w praktyce

Skorzystaj z 10% zniżki na wszystkie szkolenie otwarte z autorskiej oferty Sages ważnej przy zamówieniach złożonych do końca grudnia 2017 r. Hasło: PROGRAMISTAMAG

 zapewniony sprzęt	 3 dni	 24h	 zdalnie lub stacjonarnie
 programiści	 trenerzy praktycy	 różne lokalizacje	 projekty indywidualne

Tzn. przesunięcia „jedynki” z pola kwota do pola ilosc_przedmiotu. W związku z tym zamiast 1 przedmiotu za 10 złotych otrzymujemy 11 przedmiotów za 0 złotych. Obliczenie podpisu zgodnie z podaną procedurą:

```
HMAC(sekret, b"07290109873664746444176233123110") =
cf23df2207d99a74fbc169e3eba035e633b65d94
```

... daje dokładnie ten sam wynik, ponieważ procedura niejednoznacznie koduje oba żądania w ten sam sposób. Wyciągnąć z tego można następującą konkluzję:

Patrząc jedynie na ciąg bajtów przekazany do funkcji HMAC, powinniśmy być w stanie jednoznacznie odtworzyć oryginalny obiekt, z którego został on wygenerowany. Co prawda nie jest to niezbędne, ale „odwracalność” jest bardziej praktycznym pojęciem niż „jednoznaczność”. Prawidłowo zmodyfikowany protokół będzie wyglądał tak:

„Poprzez HTTP POST należy przekazać dwa parametry:

1. «zapytanie» (obiekt JSON zawierający wszystkie pola opisujące transakcję: «numer_konta», «numer_przedmiotu», «ilosc_przedmiotu», «kwota»),
2. «podpis», czyli HMAC obliczony z parametru «zapytanie» reprezentowanego bajtowo jako UTF-8”.

Oczywiście nie jest to jedyne poprawne rozwiązanie problemu kodowania obiektów, ale warto je rozważyć – wykorzystujemy tutaj stosunkowo prosty i sprawdzony format JSON, który posiada właściwości wystarczające do zapewnienia bezpieczeństwa podpisu.

Wnioski

Najważniejszy z nich brzmi: „hashowanie to nie uwierzytelnianie”. Funkcja hashująca jest jedynie prymitywem kryptograficznym, a zapewniane przez nią bezpieczeństwo ściśle zależy od kontekstu jej użycia. Do podpisywania danych zawsze wykorzystujemy dobrze zbadane (powszechne) odmiany algorytmów (H)MAC.

CZĘŚĆ II: HASŁO UŻYTKOWNIKA A KLUCZ KRYPTOGRAFICZNY

Drugą kwestią, która zostanie poruszona w tym artykule, jest często pomijana różnica pomiędzy hasłem i kluczem. Ponownie posłużymy się tutaj hashami, podpisami, a nawet szyfrowaniem symetrycznym. Pora na kolejny, ilustracyjny przykład intuicyjnego użycia kryptografii:

Projektujemy prosty protokół do bezprzewodowego przesyłania danych. W naszym prostym WLANie potrzebujemy szyfrowania symetrycznego oraz uwierzytelniania. Założmy następujący sposób działania: użytkownicy znają współdzielone hasło w postaci ASCII, które jednocześnie jest kluczem szyfrowania dla AES256 (jeżeli hasło ma mniej niż 32 znaki, to jest dopełniane zerami do 32 bajtów). Pakiety są numerowane, aby zapobiec atakom powtórzenia (jeżeli któraś ze stron komunikacji otrzyma dwa razy pakiet z tym samym numerem, to go po prostu ignoruje). Każdy pakiet wysłany w eter jest szyfrowany, a wynikowy szyfrogram dodatkowo jest podpisywany za pomocą HMAC (identycznym kluczem jak ten służący do szyfrowania).

Podsumujemy: pakiety są zaszyfrowane i uwierzytelnione. Korzystamy z silnych i sprawdzonych algorytmów. Gdzie jest problem?

1. Każdy bajt może przyjąć jedną z 256 wartości...

...ale typowy użytkownik będzie wykorzystywał jedynie 36 z nich, a konkretniej kody ASCII odpowiedzialne za małe litery oraz cyfry. Niektóre białe znaki, takie jak `\x00` czy `\x15`, są trudne do wprowadzenia i praktycznie nikt nie zdecyduje się na ich wykorzystanie.

2. Długość typowego hasła wynosi około 8 znaków

Atakujący może założyć, że pozostałe 24 znaków to zera. Powoduje to, że do złamania mamy wyłącznie 8 z 32 znaków hasła.

3. Ataki statystyczne i słownikowe

Dalsza redukcja przestrzeni jest możliwa dzięki zastosowaniu wspomnianych technik. Bardzo pesymistycznie założymy, że dzięki temu do sprawdzenia jest dwa razy mniej kombinacji haseł.

Dokonajmy więc obliczenia wynikowej przestrzeni haseł, która ma rozmiar:

$$\frac{36^8}{2} = 1410554953728$$

Dla komputera jest to bardzo niewiele. Jeden rdzeń procesora AMD Opteron jest w stanie zrealizować 10^8 deszyfrowań AESa na sekundę. Atakujący dysponujący takim rdzeniem byłby więc w stanie wykonać miliard deszyfrowań w czasie poniżej 11 sekund. W praktyce atak zająłby więcej czasu (z uwagi na dodatkowe operacje, jak key scheduling, i sprawdzanie, czy deszyfrowanie się udało), ale ciągle jest wykonalny, nawet na zwykłym domowym sprzęcie.⁶

Tak naprawdę jest jeszcze gorzej. Jeżeli zamiast łamać szyfrogram AESa, spróbujemy tego samego z podpisem HMAC-SHA256, dobry GPU jest w stanie policzyć około 2.9 miliarda hashy na sekundę. Ponieważ jeden HMAC składa się z dwóch zagnieżdżonych hashy, w ciągu niecałej sekundy, dysponując takim sprzętem, da się złamać jedno hasło.⁷

Jak widać, zaproponowany sposób działania sieci WLAN jest implementowalny, ale z punktu widzenia bezpieczeństwa nie ma żadnego sensu. W tym momencie konieczne jest podjęcie decyzji o „spowolnieniu” któregoś z elementów całej maszyny. Przeanalizujemy:

Nie możemy wykonywać żadnych kosztownych obliczeniowo transformacji na oryginalnej wiadomości, która jest szyfrowana i podpisywana, ponieważ obie strony komunikacji będą musiały wykonywać intensywne obliczenia związane z każdym wysłanym lub otrzymanym pakietem, a to w nieakceptowalny sposób obniży przepustowość sieci.

Analogiczne konsekwencje miałoby wprowadzenie iterowanego szyfrowania/podpisywania, albo jakkolwiek inna operacja wprowadzająca istotny „koszt” związany z wysłaniem/odczytaniem pakietu.

Można natomiast wykonać bardzo kosztowną obliczeniowo operację na hasle, a jej wynik uznać za klucz do szyfrowania. Oficjalny termin opisujący taką czynność brzmi „wyprowadzanie klucza z oparcia o hasło” (*ang. Password-Based Key Derivation*).

Korzystne właściwości wynikające z takiego wyprowadzenia:

- » dobra transformacja powinna spowodować, że wszystkie wynikowe bity wyprowadzonego klucza będą w tym samym

6. <https://security.stackexchange.com/a/9083>

7. <https://gist.github.com/epixoip/a83d38f412b4737e99bbe804a270c40>

stopniu pseudolosowe, co powoduje, że atak brute force przeciwko kluczowi jest całkowicie niepraktyczny;

- » nawet jeżeli sieć używa słabego hasła, atakujący, chcąc wykonać atak brute force przeciwko hasłu, musi każdorazowo przeprowadzać bardzo kosztowną operację wyprowadzenia klucza, co obniża prędkość łamania hasła o kilka rzędów wielkości;
- » funkcje z możliwością regulacji kosztu będą użyteczne również w przyszłości – wraz ze wzrostem mocy obliczeniowej możliwe będzie podwyższenie tego współczynnika, aby używać ten sam poziom ochrony, co kiedyś.

CTF

Jeżeli rozumiecie błąd wynikający z takiej implementacji protokołu do szyfrowania pakietów, możecie spróbować go złamać samodzielnie: <https://var.tailcall.net/wlansec>

Wyprowadzanie klucza

Jak można taką funkcję utworzyć? Przeanalizujmy, jak działa algorytm PBKDF2⁸ wynaleziony przez RSA Laboratories:

Definicja [pbkdf2]. Pełny, generyczny wzór funkcji PBKDF2

$$U_{i,0} = \text{salt} \parallel \text{int32BigEndian}(i)$$

$$U_{i,n} = \text{PRF}(\text{password}, U_{n-1})$$

$$T_i = U_{i,1} \oplus U_{i,2} \oplus U_{i,3} \oplus \dots \oplus U_{i,\text{cost}}$$

$$\text{PBKDF2}(\text{PRF}, \text{password}, \text{salt}, \text{cost}, \text{dkLen}) = T_1 \parallel T_2 \parallel \dots \parallel T_{\text{dkLen}/hLen}$$

Gdzie *PRF* to „dwuargumentowa funkcja pseudolosowa” (dobrym kandydatem jest HMAC), *password* to hasło otrzymane od użytkownika, *salt* to losowa sól kryptograficzna, *cost* to regulowany koszt wyprowadzenia pojedynczego klucza, a *dkLen* to oczekiwana długość wyprowadzonego klucza. Do parametru *salt* doklejamy jest numer iteracji „i”, aby poszczególne iteracje różniły się czymś od siebie.

„Solenie” hashy

Dosyć poważnym atakiem przeciwko hashom generowanym z krótkich ciągów wejściowych jest wykorzystanie wcześniej wyliczonych tabel lookupu. Prawidłowe hashowanie krótkich wiadomości polega na wykorzystaniu techniki „solenia”, czyli dodawaniu na jej końcu losowego ciągu bajtów, który jest potem zapisywany w formie jawnej. Nawet jeżeli potencjalny atakujący wie, że wiadomość ma długość 3 bajtów, zna dodany do niej *salt* o długości 64 bajtów oraz hash wiadomości, to nie jest w stanie posłużyć się tabelami lookupu, ponieważ wcześniejsze wygenerowanie tabel dla ciągów 69-bajtowych jest wysoce niepraktyczne. Oczywiście zawsze istnieje możliwość złamania takiego hashu metodą brute force, ale wymaga to znacznie większych nakładów mocy obliczeniowej.

Jeżeli założymy, że funkcja *PRF* to HMAC-SHA256 oraz że koszt wynosi 3, a $\text{dkLen} = hLen = 32$ bajty (tzn. chcemy wyprowadzić klucz o takiej samej długości, jaką ma pojedynczy wynik SHA256), otrzymujemy bardziej czytelną postać:

Definicja [pbkdf_reduced]. Zredukowana wersja PBKDF2

$$U_1 = \text{HMAC}_{\text{SHA256}}(\text{password}, \text{salt} \parallel \text{int32BigEndian}(1))$$

$$U_2 = \text{HMAC}_{\text{SHA256}}(\text{password}, U_1)$$

$$U_3 = \text{HMAC}_{\text{SHA256}}(\text{password}, U_2)$$

$$\text{PBKDF2}'(\text{password}, \text{salt}) = U_1 \oplus U_2 \oplus U_3$$

Jak widać, cały pomysł polega na „zagnieżdżonym” obliczaniu wyniku funkcji HMAC-SHA256, a potem xorowaniu ze sobą wszystkich pośrednich wyników. Polecamy również zbadać inny aspekt tego algorytmu, poprzez zredukowanie oryginalnej definicji równaniami:

$$\text{PRF} = \text{HMAC}_{\text{SHA256}}; \text{cost} = 1; \text{dkLen} = 3 \cdot hLen = 96 \text{ bytes}$$

Przedstawione powyżej wzory dotyczą algorytmu PBKDF2. Istnieją również inne powszechnie znane algorytmy o podobnym zastosowaniu, między innymi bcrypt, scrypt oraz Argon2 (zwycięzca konkursu na funkcję do hashowania haseł⁹).

Wspomniane algorytmy umożliwiają regulowanie kosztu obliczeniowego (wszystkie trzy), pamięciowego (scrypt oraz Argon2) oraz stopnia równoległości (tylko Argon2). Różnorodne parametry kosztu są wprowadzane celem obniżenia podatności na ataki z użyciem GPU albo dedykowanych układów scalonych. Jest to odpowiedź na problem z algorytmem PBKDF2, który jest trudny dla CPU, ale nie wymaga zbyt dużych zasobów pamięciowych i łatwo go zrównoleglić na GPU.

Algorytm PBKDF2 wciąż jest używany, chociaż nie jest zalecanym wyborem do nowych projektów. Wspominamy jego detale implementacyjne w celu ułatwienia zrozumienia, jak działa wyprowadzanie klucza (inne algorytmy są nieco trudniejsze). Ponadto algorytm scrypt wewnętrznie wykorzystuje PBKDF2.

Warto zapamiętać, że hasło to ciąg znaków pochodzący od użytkownika, który najczęściej nie spełnia restrykcyjnych wymagań stawianych wobec klucza szyfrującego. Należy poważnie rozważyć wykorzystanie algorytmu wyprowadzania klucza, kiedy zamierzamy szyfrować dane hasłem.

Istotne jest, że wspomniane wyżej algorytmy świetnie sprawdzą się również w zakresie przechowywania haseł użytkowników – tak jak pokazano powyżej, wykorzystanie powolnego algorytmu może „uratować” nawet osoby z dosyć słabymi hasłami przed atakiem brute force.

PODSUMOWANIE

Celem artykułu było podniesienie świadomości programistów związanej z zastosowaniami poszczególnych rozwiązań kryptograficznych, takich jak hash, podpis i algorytmy wyprowadzania klucza. Podając praktyczne przykłady, udowodniliśmy, że projektowanie zabezpieczeń z wykorzystaniem intuicji bywa zgubne, a niekiedy potrafi obniżyć bezpieczeństwo do zera, nawet pomimo użycia sprawdzonych, „niełamaalnych” narzędzi.

Michał Leszczyński, Jarosław Jedynak

8. Pewne zamieszanie wprowadza fakt, że termin PBKDF (albo samo KDF) odnosi się do samego konceptu, a prawie identycznie brzmiące PBKDF2 to już nazwa konkretnego algorytmu.

9. <https://password-hashing.net/>

Ekoparty CTF 2017 – OldPC

W połowie września miała miejsce trzecia edycja zawodów EKOPARTY CTF organizowanych przez drużynę Null Life (zawodnicy pochodzą z Ameryki Południowej i Środkowej). Ja w EKOPARTY brałem udział po raz drugi i nie mogę się doczekać kolejnej edycji. Zadań zawsze jest dużo, są różnorodne i ciekawe. A ponieważ większość z nich jest raczej prosta, nawet osoby rozpoczynające przygodę z CTFami mają szansę zdobyć kilka flag, a przy okazji czegoś nowego się nauczyć.

```

✓ 20:29 ~/priv/ctf/ekoparty-2017-09/misc-old_pc $ r2 -a sparc -e cfg.bigendian=true oldpc.bin
[00] 0 0 < Disable these messages with 'e cfg.fortunes = false' in your ~/.radare2rc
[00] 0 0 /
[00] 0 0 -

[0x00000000]> pd
0x00000000 9de3bf90 save sp, -0x70, sp
0x00000004 90102001 mov l, o0
0x00000008 03000041 sethi 0x41, g1
0x0000000c 9210621c or g1, 0x21c, o1
0x00000010 94102004 mov 4, o2
0x00000014 82102004 mov 4, g1
0x00000018 91d02010 ta icc, g0 + 0x10
0x0000001c 01000000 nop
0x00000020 92102083 mov 0x83, o1
0x00000024 9410207f mov 0x7f, o2
0x00000028 96102071 mov 0x71, o3
0x0000002c 9810206d mov 0x6d, o4
0x00000030 1b2b908a sethi 0x2b908a, o5
0x00000034 9a03623e add o5, 0x23e, o5
0x00000038 40000043 call 0x144
0x0000003c 01000000 nop
0x00000040 9210206b mov 0x6b, o1
0x00000044 94102067 mov 0x67, o2
0x00000048 96102065 mov 0x65, o3
0x0000004c 98102061 mov 0x61, o4
0x00000050 1b0785c0 sethi 0x785c0, o5
0x00000054 9a136013 or o5, 0x13, o5
0x00000058 4000003b call 0x144
0x0000005c 01000000 nop
0x00000060 92102059 mov 0x59, o1
0x00000064 94102053 mov 0x53, o2
0x00000068 9610204f mov 0x4f, o3
0x0000006c 98102049 mov 0x49, o4
0x00000070 1b018188 sethi 0x18188, o5
0x00000074 9a03633d add o5, 0x33d, o5
0x00000078 40000033 call 0x144

```

Poniżej prezentuję rozwiązanie zadania *OldPC*, które polegało na przeanalizowaniu kodu maszynowego „niepopularnej” architektury sprzętowej, konkretnie był to SPARC. Gorąco polecam artykuł na Wikipedii przedstawiający podstawowe informacje techniczne i historię tej architektury. Przy okazji spróbuję zachęcić czytelników do zapoznania się ze świetnym frameworkiem *radare2*.

Chciałbym, żeby ten artykuł był zrozumiały dla osób absolutnie początkujących, stąd wspominam o rzeczach oczywistych dla doświadczonych reverserów – wybaczenie!

Ponieważ niedługo po zakończeniu zawodów strona internetowa z zadaniami przestała działać, pliki potrzebne do samodzielnego zmierzenia się z zadaniem (do czego bardzo zachęcam!) umieściłem na GitHubie pod adresem: <https://github.com/jnahorny/ctf-writeups/tree/master/ekoparty-2017-09>.

CTF	EKOPARTY CTF 2017 https://ctf.ekoparty.org
Waga CTFtime.org	55.24 (https://ctftime.org/event/483)
Liczba drużyn (z niezerową liczbą punktów)	184
System punktacji zadań	Dynamiczny (liczba punktów zależna od liczby rozwiązań)
Liczba zadań	> 35
Podium	1. CodiSec (Polska) – 13511 pkt. 2. TokyoWesterns (Japonia) – 12002 pkt. 3. 217 (Tajwan) – 11520 pkt.
Zadanie	OldPC (Misc 478)

OLDPC? ALE JAKI KONKRETNIE...

Plik `misc-oldpc.hex`, który musimy zbadać, znajduje się we wspomnianym wyżej repozytorium GitHub, w katalogu `OldPC`. Jak można szybko sprawdzić, nie jest to plik binarny, lecz ciąg bajtów zapisanych w ASCII. Najlepsza rzecz, jaką można zrobić na początek, to wziąć kilka pierwszych bajtów i wrzucić w wyszukiwarkę. Jednym z pierwszych wyników, jakie dostałem w odpowiedzi na zapytanie o „9D E3 BF 90”, był fragment książki na [books.google.com](https://books.google.com/books?id=9D_E3_BF_90): Alan Clements – *Computer Organization & Architecture: Themes and Variations*, z kodem przedstawionym na Rysunku 1.

that % indicates a register. The disassembled SPARC code generated by the UOBT package is given by

```
fib()
8050b54: 9d e3 bf 90      save    %sp, -112, %sp
8050b58: 80 a6 20 01      cmp     %i0, 1
8050b5c: 04 80 00 08      ble     0x8050b7c
8050b60: 01 00 00 00      nop
8050b64: 7f ff ff fc      call    fib
```

Rysunek 1. Fragment zdeasemblowanego kodu maszynowego SPARC

Wygląda obiecująco! Ponieważ *radare2* pozwala na deasemblację kodu SPARCa, spróbujmy go użyć. Zaczniemy od instalacji.

INSTALACJA RADARE2

Tu mała dygresja: *radare2* można pobrać w postaci binarnej ze strony projektu (<http://rada.re/r/down.html>) dla większości popularnych systemów operacyjnych, w tym iOS czy Androida. I na „pierwszy raz” jest to niezła opcja. Podobnie jak instalacja *radare2* z paczek dystrybucji Linuksa. Jeśli jednak chcemy poważniej poznać *radare2*, zdecydowanie lepiej skompilować najnowszą wersję z gita. Chociażby dlatego, że codziennie do repozytorium trafia kilka, a czasem kilkadziesiąt commitów. Jeśli zechcemy skorzystać z pomocy społeczności skupionej wokół *radare2* (kanał #*radare2* na sieci IRC *freenode* i komunikator Telegram, szczegóły w dziale *Contact* na stronie projektu <http://radare.org/r/>), pierwsze pytanie, jakie dostaniemy, brzmi: „którą masz wersję *radare2*”. Jeśli nie mamy najnowszej wersji z gita, nikt nie będzie chciał z nami rozmawiać ;)

Cały proces w najprostszym wariantcie sprowadza się do wykonania trzech komend (*radare2* nie ma długiej listy zależności, w zasadzie potrzebujemy kompilatora i narzędzia *make*):

```
$ git clone https://github.com/radare/radare2
$ cd radare2
$ ./sys/install.sh
```

PRZYGOTOWANIE

Zaczynamy od ustalenia, jak wewnątrz w *radare2* nazywa się interesująca nas architektura:

```
$ rasm2 -L | grep -i sparc
_dAe 32 64   sparc   BSD   Capstone SPARC disassembler
_dA_ 32 64   sparc.gnu GPL3 Scalable Processor Architecture
```

Jak widać, do wyboru są dwie implementacje obsługi tej architektury; proponuję użyć *sparc*, którego format wyjścia jest trochę bardziej czytelny. Spróbujmy zdeasemblować pierwszą instrukcję:

```
$ rasm2 -a sparc -e -d "9de3bf90"
save sp, -0x70, sp
```

gdzie `-a` służy do podania architektury, `-e` zmienia kolejność bajtów na *big-endian* (z domyślnej na architekturze x86 *little-endian*), a `-d` oznacza, że chcemy deasemblować. Faktycznie, otrzymaliśmy poprawny wynik, jeśli uwzględnimy, że szesnastkowo `0x70` to 112 dziesiętnie.

Kolejnym krokiem jest stworzenie pliku binarnego z ASCII. Na przykład takim krótkim programem w języku Ruby:

```
$ cat a2bin.rb
o = File.open("oldpc.bin", "wb")
File.read("misc-oldpc.hex").split(" ")
  .each { |b| o.putc(b.to_i(16)) }
o.close
```

ANALIZA KODU

Wyjściowy plik `oldpc.bin` możemy wczytać w *radare2*:

```
$ r2 -a sparc -e cfg.bigendian=true oldpc.bin
```

Będąc w linii komend *radare2*, zaczniemy od dokonania podstawowej analizy pliku za pomocą komendy: `aa` (od *analyze all*). Jeśli następnie wpisujemy `af1` (od *analyze function list*), zobaczymy, że *radare2* „wykrył” jedną funkcję, zaczynającą się od adresu `0x0000`. Wpiszmy `pdf` (*print disassembly function*) i przejrzymy wynik.

Aby zrozumieć sens analizowanego kodu, warto skorzystać z dokumentacji asemblera SPARCa, na przykład pozycja [1] z bibliografii na końcu artykułu. Od adresu `0x001c` mamy kilka powtórzeń sekwencji instrukcji kończących się skokiem pod adres `0x0144` (identycznie z każdego kolejnego bloku):

```
0x0000001c  01000000  nop
0x00000020  92102083  mov 0x83, o1
0x00000024  9410207f  mov 0x7f, o2
0x00000028  96102071  mov 0x71, o3
0x0000002c  9810206d  mov 0x6d, o4
0x00000030  1b2b908a  sethi 0x2b908a, o5
0x00000034  9a03623e  add o5, 0x23e, o5
0x00000038  40000043  call 0x144
```

Rzut oka do dokumentacji i wiemy, że `o1` – `o5` to rejestry wyjściowe ogólnego przeznaczenia. Zatem do rejestrów `o1` – `o4` wstawiane są pojedyncze bajty. A co dzieje się z rejestrem `o5`? Instrukcja `sethi` pozwala załadować do rejestru liczbę 32-bitową. Ponieważ na SPARCu wszystkie instrukcje są 32-bitowe, normalnie byłoby to niemożliwe. Dlatego ładowanie liczby odbywa się w dwóch krokach: za pomocą `sethi` ustawiamy najstarsze 22 bity, następnie młodsze 10 bitów dodajemy za pomocą instrukcji `or` lub `add`. Przy okazji chciałbym zwrócić uwagę na to, jak drobne szczegóły czasem decydują o powodzeniu próby analizy kodu. Jeśli wybierzemy w *radare2* architekturę `sparc.gnu` zamiast `sparc`, instrukcje z adresów `0x0030` i `0x0034` przedstawione zostaną w ten sposób:

```
0x00000030  1b2b908a  sethi %hi(0xae422800), %o5
0x00000034  9a03623e  add %o5, 0x23e, %o5
```

Liczba `0x2b908a` (architektura `sparc`) to faktycznie 22 najstarsze bity liczby `0xae422800` (architektura `sparc.gnu`), co możemy szybko sprawdzić:

```
$ ruby -e 'puts "#{0xae422800}, #{0x2b908a << 10}'
2923571200, 2923571200
```

Należy zatem pamiętać przy analizie o konieczności przesunięcia argumentu `sethi` w lewo o 10 bitów (jeśli wybraliśmy architekturę `sparc`), inaczej otrzymamy błędne wyniki.

Możemy z dużą pewnością założyć, że pod adresem 0x0144 znajduje się funkcja, która wykonuje jakieś ciekawe operacje na przekazanych danych. I faktycznie, na początku mamy coś, co wygląda jak stworzenie miejsca na stosie (`save sp, -0x40, sp`), a pod koniec `restore`.

Nie musimy tego robić, ale ponieważ chcemy przy okazji poznać *radare2*, spróbujmy oznaczyć ten fragment jako nową funkcję (już teraz nazwiemy ją `decrypt`, nieco wybiegając w przyszłość):

```
[0x00000000]> afr @0x144
[0x00000000]> afn decrypt @0x144
```

Teraz możemy wyświetlić kod naszej nowej funkcji `decrypt` następująco:

```
[0x00000000]> pdf @decrypt
```

W GĄSZCZU KOMEND RADARE2

To dobry moment na refleksję na temat używania *radare2*. Jak widać (mam nadzieję), komendy tylko pozornie są trudne do zapamiętania i bezsensowne. Faktycznie framework używa swego rodzaju „drzewa” komend, gdzie pierwsza litera określa grupę, a kolejne – bardziej szczegółowe operacje. Bardzo przydatna jest funkcja `?`, która wyświetla pomoc. Wpisanie samego `?` zwróci informację o tym, że np. komendy dotyczące analizy zaczynają się od litery `a`, komendy do debugowania – od litery `d`.

Aby wyświetlić komendy z danej podgrupy, dodajemy znak zapytania do odpowiedniej litery. Na przykład `a?` informuje nas m.in. o tym, że:

```
| aa[?]      analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| ab [hexpairs] analyze bytes
| af[?]      analyze Functions
```

Z kolei `af?` wyświetli informację na temat komend „o poziom niżej”, dotyczących analizowania funkcji. Znajdziemy tam informację np. o funkcjach `afr` i `afn`, których użyłem wyżej. Z kolei eksplorując pomoc dla komend z grupy `print`, powinniśmy odkryć, że użyte przed chwilą `pdf` oznacza *print disassembly (of) function*. W przypadku problemów (które z pewnością na początku używania *radare2* się pojawiają) zachęcam do odwiedzenia strony [3], gdzie znajdują się odnośniki do dokumentacji oraz kilku (z dzieł istniejących) świetnych artykułów na temat tego narzędzia.

ANALIZA FUNKCJI DESZYFRUJĄCEJ

Wróćmy jednak do analizowanego kodu. Sprawdzamy w dokumentacji potrzebne instrukcje: `st` (*store*) zapisuje w pamięci słowo 32-bitowe, `stb` (*store byte*) – jeden bajt. Z kolei `ldub` (*load unsigned byte*) pobiera z pamięci do rejestru jeden bajt. Poniżej kod z moimi komentarzami (pamiętajmy, że zawartość rejestrów wejściowych `i1` – `i5` odpowiada temu, co przekazaliśmy funkcji `decrypt` w rejestrach `o1` – `o5`):

```
(fcn) decrypt 88
  decrypt ();
; XREFS: CALL 0x00000038 CALL 0x00000058 (...)
0x00000144    save sp, -0x40, sp
0x00000148    st i5, [sp]      ; słowo 32b z i5 na stos
0x0000014c    ldub [sp], i5      ; pierwszy bajt ze stosu do i5
0x00000150    xor i5, i1, i5      ; i5 = i5 ^ i1
0x00000154    stb i5, [sp]      ; nowa zawartość i5 na stos
0x00000158    ldub [sp+1], i5    ; drugi bajt ze stosu do i5
0x0000015c    xor i5, i2, i5      ; i5 = i5 ^ i2
0x00000160    stb i5, [sp+1]     ; nowa zawartość i5 na stos
0x00000164    ldub [sp+2], i5    ; analogicznie dla kolejnych
bajtów
```

```
0x00000168    xor i5, i3, i5
0x0000016c    stb i5, [sp+2]
0x00000170    ldub [sp+3], i5
0x00000174    xor i5, i4, i5
0x00000178    stb i5, [sp+3]
```

Uzbrojeni w te informacje nie powinniśmy mieć problemu z napisaniem skryptu, który policzy, a następnie wyświetli wynikowe bajty. Zachęcam czytelników do zrobienia tego samodzielnie. Moje rozwiązanie jest znów w Rubym. Obliczenia da się zapisać prościej, jednak zależało mi na tym, by kod jak najdokładniej imitował to, co dzieje się w analizowanym programie:

```
def decrypt_it(i1, i2, i3, i4, i5)
  r = []
  r[0] = ((i5 >> 24) & 0xff) ^ i1
  r[1] = ((i5 >> 16) & 0xff) ^ i2
  r[2] = ((i5 >> 8) & 0xff) ^ i3
  r[3] = ((i5 >> 0) & 0xff) ^ i4
  r.each { |i| puts i.chr }
end

decrypt_it(0x83, 0x7f, 0x71, 0x6d, 0x2b908a << 10 | 0x23e)
decrypt_it(0x6b, 0x67, 0x65, 0x61, 0x785c0 << 10 | 0x13)
decrypt_it(0x59, 0x53, 0x4f, 0x49, 0x18188 << 10 | 0x33d)
decrypt_it(0x47, 0x43, 0x3d, 0x3b, 0xd4898 << 10 | 0x276)
decrypt_it(0x35, 0x2f, 0x2b, 0x29, 0x19207 << 10 | 0x376)
decrypt_it(0x25, 0x1f, 0x1d, 0x17, 0x1957d3 << 10 | 0x247)
decrypt_it(0x13, 0x11, 0xd, 0xb, 0x1490d3 << 10 | 0x24b)
decrypt_it(0x7, 0x5, 0x3, 0x2, 0x11d60f << 10 | 0x22f)
puts ""
```

W wyniku uruchomienia skryptu dostajemy flagę:

```
$ ruby solver_new.rb
--[Super_Ultra_M3g4_@@SPARC@@]=-
```

PODSUMOWANIE

Decydując się na omówienie tego zadania, miałem następujące cele: pokazać, jak zacząć przygodę z *radare2*, oraz jak zabrać się do analizy kodu nieznannej architektury sprzętowej. Tego typu wyzwanie może stanąć przed nami niezależnie od tego, czy pracujemy jako programista, administrator czy specjalista bezpieczeństwa. Przy okazji to kolejny dowód na to, że warto brać udział w zawodach typu CTF. Ta „zabawa” doskonale rozwija umiejętność analizy i rozwiązywania problemów z różnych dziedzin, często zupełnie dla nas nowych, w krótkim czasie.

Jarosław Górny



Dragon Sector

Rozwiązanie zadania *OldPC* zostało nadesłane przez Jarosława Górnego, grającego gościnnie w Dragon Sector – jedną z polskich drużyn CTFowych. <http://www.dragonsector.pl/>

W sieci

- [1] <https://docs.oracle.com/cd/E19641-01/802-1947/802-1947.pdf>
- [2] <https://goo.gl/pZrZCt>
- [3] <http://radare.today/posts/radare2-is-documented/>



BEZPIECZEŃSTWO SYSTEMÓW IT

SECURITUM

Zapraszamy na autorskie szkolenia
z zakresu **bezpieczeństwa IT**

{ Bezpieczeństwo aplikacji WWW }

{ Offensive HTML, SVG, CSS and other Browser-Evil }

{ Wprowadzenie do bezpieczeństwa IT }

{ Szkolenie przygotowujące do egzaminu CEH }

(Certified Ethical Hacker)

www.securitum.pl/oferta/szkolenia

Patroni medialni: sekurak.pl



rozwal.to



Jak ciekawość nas napędzała

Jakiś czas temu pojawił się na łamach naszego pisma nowy dział „Wspomnienia zgreda”. Jego istnienie zainicjowane zostało przytoczoną pewną kontrowersyjną historią, lekko podkolorowaną, tego, co działo się ponad piętnaście lat temu na scenie. Może współcześnie wyglądać to troszkę dziwnie. Jednak wtedy nie było to aż tak nietypowe, jak może się wydawać autorowi tamtego tekstu.

Ogólnie tamte czasy były inne. Sam próg wejścia w informatykę był dużo wyżej ustawiony niż obecnie. Dzisiaj praktycznie każdy ma dostęp do komputera od maleńkiego. Ba, właściwie każdy ma dostęp do Internetu, gdzie znajdzie wszystko, co tylko potrzebuje. W tamtych czasach było inaczej. Większość ówczesnych uczestników wszelakich list dyskusyjnych, osób prześiadujących na IRC¹ była bliska pełnoletności, lub tuż po 18 roku życia. Zaczynali przygodę w czasach, gdy o Internecie w kraju słyszała garstka naukowców pracujących na uczelniach, a zwykły obywatel jak miał styczność z komputerem, to albo w szkole, albo u bogatego kolegi. Ale po kolei.

Jak już wspominałem, początek lat 90-tych w naszym kraju pod względem dostępności nowych technologii był bardzo ciężki. Jedynie komputery, jakie się widziało na żywo, to albo Odry w bogatych zakładach pracy, albo cichcem przemycone przez granicę od naszych zachodnich sąsiadów komputery Atari, ZX Spectrum oraz Commodore C64. Część z nich można było zresztą kupić legalnie w kraju w sklepach Pewex², jednak ich cena w porównaniu do ówczesnych zarobków powalała i stać na nich było tylko najbardziej zamożnych obywateli. Stąd dostęp do tego typu urządzeń mieli głównie ci, których rodzice lub najbliższa rodzina pracowała za granicą, bądź pływała na naszych statkach PŻM³. Sam pamiętam, jak będąc w szkole podstawowej, po raz pierwszy na oczy ujrzałem ZX Spectrum przyniesionego przez kolegę z klasy na kółko matematyczne. Praktycznie całe tamte zajęcia wszyscy siedzieli i grali, łącznie z nauczycielką.

Już wtedy w większości społeczeństwa krążyły opinie o zbędności tego sprzętu. Poza rozrywkowymi funkcjami mało kto widział sens posiadania komputera w domu i wydawania na jego zakup równowartości kilku/kilkudziesięciu ówczesnych pensji. Stąd praktycznie każdy, kto zaczynał przygodę z tymi urządzeniami w tamtych czasach, był albo ciekawskim dzieciakiem, którego pociągały nowe nieznanne rzeczy, albo miał w rodzinie kogoś, kto pracował na komputerach i w umiejętny sposób zasiewał w młodym człowieku ziarenko zainteresowania, które po jakimś czasie kiełkowało. Poza problemem z dostępnością fizyczną urządzeń był jeszcze praktycznie brak polskojęzycznych opracowań książkowych z podstaw obsługi komputerów. Oczywiście istniała prasa, która się regularnie ukazywała w kioskach, taka jak „Komputer”, kultowy obecnie „Bajtek” i „PCKurier” skierowany bardziej do profesjonalistów, choć w dopiero co rodzącej się branży w naszym kraju ciężko było trafić na kogoś bez odpowiednich umiejętności. Można w sumie stwierdzić z bardzo dużą dozą pewności, że każdy zaczynał „zarażony”

bakcyłem przez kogoś innego. Ot, taki pierwszy polski wirus komputerowy o nazwie fascynacja.

Tamte czasy były zdominowane przez wspomniane komputery, choć była pewna grupa osób, która miała dostęp do sprzętu z górnej półki. Tym sprzętem były komputery klasy PC. Wyobraźcie sobie pudło ważące dobre 15 kg, wysokie na około 60 cm, kosztujące po 2500 dolarów amerykańskich z podłączonym dwukolorowym monitorem. To był pierwszy komputer, z którym miałem dłuższą styczność. Słynny PC-AT 286, z monochromatycznym Herculesem (o rozdzielczości 720x348 pikseli) na pokładzie, bursztynowym ekranem, dyskiem twardym 40 MB, dwoma stacjami dyskietek i aż 1 MB pamięci RAM. Mój pierwszy komputer. Tak, tego się nie zapomina, przynajmniej nasze pokolenie tak ma ;) Całość działała w oparciu o system MS-DOS 3.3, w późniejszym czasie uaktualnionym do wersji 5.0, w którym sporadycznie uruchamiałem Windowsa 3.1. Ciekawostką jest to, że karta ta potrafiła w trybie „emulacji” z wykorzystaniem programu color.com udawać kolorową kartę CGA, dzięki czemu działały niektóre gry, które do działania wymagały kolorowego ekranu, np. *Prehistorik*, *Blues Brothers* i słynna *Prince of Persia*. Z tego powodu, praktycznie od samego początku (i za delikatną sugestią mojego sąsiada, pana Mirosława, pod postacią stosu PCKurierów i Bajtków), interesowałem się bardziej twórczym wykorzystaniem komputera.

W obu pismach można było znaleźć sporo kodów źródłowych. Na komputerze miałem do dyspozycji kilka różnych języków programowania, począwszy od różnych dialektów BASICa, skończywszy na Turbo Pascalu (strasznie lubianego w późniejszych czasach przez nauczycieli w szkołach). Cała przygoda z programowaniem zaczęła się praktycznie standardowo, jak dla każdego młodego adepta, od przepisywania na ślepo poszczególnych instrukcji z czasopisma do komputera, następnie metodą prób i błędów zmienianie kodu i sprawdzanie po każdej zmianie, jaki jest jej efekt. Równolegle uczyłem się ogólnej obsługi komputera, ze szczególnym naciskiem na obsługę systemu operacyjnego. W późniejszym czasie poruszanie się po czeluściach dysku ułatwiał, zapewne znany większości czytelników, program o nazwie Norton Commander oraz przygotowane własne skrypty BAT. Wtedy też pojawiła się myśl, że dobrze byłoby zrozumieć, jak to całe ustrojstwo (tak komputer był określany przez moją Mamę) tak naprawdę działa. Szukałem wszelkiej dostępnej dokumentacji i opracowań w bibliotekach i w czytelnich. Będąc wtedy dziesięcioletkiem, nie sądziłem, że ten temat jest aż tak złożony, wszystko wydawało się wtedy proste i łatwe do zrozumienia. Przygoda z poznawaniem komputera od środka zaczęła się od wspólnego jego rozebrania z sąsiadem, wtedy to zafascynowany zobaczyłem pierwszy raz na własne oczy, jak wygląda płyta główna, gdzie znajdują się pamięci, jak wygląda kość BIOSu z magicznym okienkiem resetowania jej zawartości

1. Internet Relay Chat – usługa sieciowa umożliwiająca wspólną rozmowę z innymi użytkownikami na utworzonych kanałach tematycznych.

2. Przedsiębiorstwo Eksportu Wewnętrzznego „Pewex”, sieć sklepów walutowych w czasach PRL, obowiązującą w nich walutą był dolar.

3. Polska Żegluga Morska.

za pomocą źródła światła oraz ogromny kawał płytki z szeregiem układów scalonych będący kartą graficzną. Sąsiad pozwolił mi złożyć komputer samodzielnie z powrotem, podłączyć wszystkie przewody i uruchomić go. Niby niewiele, ale dla dziecka to było coś. I tak mijały dni i tygodnie na klepaniu kodu w BASICu, przeskakując z QBASICa do GW BASICa, kończąc na Turbo BASICu. W międzyczasie pojawiła się fascynacja Pascaliem podparta podstawowymi w tamtych czasach podręcznikami z żółtej serii autorstwa śp. Jana Bieleckiego, z jego występującym w każdej książce słynnym „Hello, I'm Jan B.”. To był ten etap największej fascynacji, spędzałem przy klawiaturze praktycznie każdy wolny wieczór, pisząc różnego rodzaju narzędzia, bawiąc się w tworzenie prostych tekstowych gier, próbując pojąć, jak dokładnie działa procesor, i wyduszając ze sprzętu wszystko, co się da. Nie było rzeczy niemożliwych, człowiek do wszystkiego dochodził samodzielnie. Nie było dostępnego Internetu tak jak obecnie, nie można było w dowolnym momencie wyszukać, jak rozwiązać dany problem, wszystko trwało długie godziny, zmuszało do myślenia twórczego, analizowania problemu i rozbierania go na czynniki pierwsze. Łączyło się wtedy praktycznie nieświadomie wiedzę ze wszystkich przedmiotów w jedno, aby móc wyprowadzić równania, które służyły do opisu np. ruchu. Podobnie było z efektami graficznymi, animacją i całą resztą około-multimedialną. Bardzo popularne były wtedy tzw. copy party. Spotykało się ze znajomymi u jednego z kolegów, wszyscy przychodzili z dyskietkami pełnymi różnych dziwnych narzędzi, ciekawostek, gier. W tamtych czasach u nas praktycznie nie istniało pojęcie praw autorskich, więc kopiowanie gier i programów było na porządku dziennym. Ba, nawet w radiu była audycja, która kończyła się transmisją gry lub programu (pamiętacie to?) nagrywanego na kasetę magnetofonową. Wtedy każdy użytkownik komputera miał wiedzę, potrafił sobie poradzić z wieloma problemami samodzielnie. Nie było czegoś takiego, co spotyka się obecnie, że umie się tylko jedno, a reszta nie jest w ogóle potrzebna do pracy zawodowej.

Jednak najwięcej nauczyłem się po zmianie komputera. Następnym sprzętem był Pentium 133, 16 MB ramu, dysk 2 GB i karta graficzna S3 VIRGE. To był potwór. Nadal najczęściej używanym systemem operacyjnym był DOS, nadal programy głównie pozyskiwało się na copy party. Jednak zmienił się wachlarz używanych narzędzi i troszkę zainteresowań. Zamiast Nortona używaliśmy w naszym środowisku Dos Navigatora. Programowanie z wykorzystaniem Pascala zamieniło się w programowanie z wykorzystaniem środowiska Delphi. Doszło poznawanie WinApi, głębsza analiza zasady działania procesora i zabawa z Assemblerem. To były też czasy, gdy powoli i nieśmiało pod strzechy zaczynał zaglądać Linux. Najczęściej jako dystrybucja RedHat albo Mandrake. Reszta była bardzo trudno dostępna w kraju. Wtedy też pojawiło się, jakoś samoistnie, grzebanie hexeditorem w plikach. I znowu zabawa w zmienianie zawartości plików, próby podmieniania tekstów w grach, zabawy wszelakiego typu narzędziami służącymi do wyszukiwania ukrytych plików w innych plikach. Ogólnie to był czas beztróskiego grzebania we wszystkim, jak coś się zepsuło tak, że nie było możliwości naprawy, formatowałem dysk i wszystko od nowa. Wtedy też pierwszy raz spotkałem się z padem dysku twardego. Pamiętam te kilka tygodni, zanim przyjechał nowy dysk, pracy tylko na dyskietkach na dwóch stacjach. Całe szczęście, że był wtedy dostępny pewien magiczny trick o nazwie 2MGUI, umożliwiający zapisanie na zwykłej dyskietce troszkę więcej danych. To było zbawienie, odpowiednio przygotowane pliki startowe systemu DOS, plus do tego kilka dyskietek z potrzebnymi do pracy narzędziami okrojonymi ze

zbędnych plików równało się w miarę wygodne środowisko pracy, jedyną wadą tego rozwiązania to konieczność wkładania określonych dyskietek w określonej kolejności do napędu. Ale przy prawie non stop włączonym komputerze nie było to większym kłopotem. Był 1995 rok: przełomowy rok dla branży – pojawił się w sierpniu nowy Windows zrywający w sporym stopniu z przeszłością. Od tego czasu częściej niestety pracowało się na okienkach niż pod czystym DOSem. Reszta pozostała bez zmian, no, poza chwilowymi fascynacjami tworzeniem grafiki 3D i próbami stworzenia własnego modelera.

Minęło kilka lat zabawy z komputerem, pojawiały się coraz to nowsze procesory, karty graficzne. Już wtedy widać było, jak bardzo dynamiczna jest ta branża. Zmiany w technologii postępowały lawinowo. Dziś kupiony komputer, za pół roku był już w znacznym stopniu przestarzały. Ba, nadal komputery w naszym kraju kosztowały kilkukrotność średnich pensji. Wtedy też zaczęło docierać do nas, że to może być w przyszłości dość ciekawe zajęcie. I tak z hobby powoli zaczęło się rodzić coś, co można obecnie nazwać ukierunkowaniem zawodowym. To były czasy końcówki szkoły podstawowej, wtedy bardzo popularne były tzw. testy kompetencji zawodowych. Przeprowadzała je pani pedagog szkolny. Polegało to na odbyciu około godzinnej rozmowy z panią, na podstawie której typowane były szkoły, do których powinniśmy iść. W moim przypadku na liście znajdowały się prawie same technika. Jednak finalnie wyładowałem w zwykłym liceum, co okazało się w perspektywie czasu chyba najlepszym rozwiązaniem.

Wtedy to poznałem kilka nowych osób, każdy z nich był magikiem komputerowym. Jeden miał umiejętności około-muzyczne, inny bawił się grafiką. Każdy umiał coś innego i każdy miał ogromną potrzebę dzielenia się wiedzą z resztą. Spędzaliśmy całe godziny na wymyślaniu, jak zbawić świat i stać się milionerami. To był też czas, kiedy pojawiły się pierwsze zlecenia na napisanie oprogramowania użytkowego. Oczywiście bez wynagrodzenia. Pisałem wtedy po prostu dla wprawy i praktyki. Do dzisiaj pamiętam pierwsze dzieło dla pracowni fizycznej (pozdrawiam panią Ewę P), przeprotowany kod narzędzia służącego do rysowania wykresu odchylenia statystycznego pomiarów czasu podczas staczania z równi pochyłej. Brzmi skomplikowanie, a w rzeczywistości narzędzie było dość proste, jednak przeprotowanie go z Commodore na PC było już ciekawym wyzwaniem. Port zakończył się pełnym sukcesem, co zaowocowało rozbudową dostępnej funkcjonalności. Wtedy też pojawił się u mnie w domu Internet. To był szal. Cały świat w zasięgu ręki, można szukać, co tylko się chce. Można komunikować się z innymi bez wychodzenia z domu. Szok!

Początkowy okres korzystania z tego nowego medium był chyba taki sam jak u większości. Po prostu szperanie po sieci i czytanie wszystkiego, co brzmiało ciekawie. W ten sposób trafiłem pod koniec lat 90-tych na stronę poświęconą ciemnej stronie wiedzy informatycznej – usuwaniu zabezpieczeń zwanego potocznie crackowaniem. Tą stroną była witryna crack.pl. Zafascynowało to mnie całkowicie. Raptem cała wiedza, jaką przez lata nabywałem, złożyła się w jedną całość. Czytałem wtedy tutorial za tutorialiem, usiłując zrozumieć, jak to się robi, jak temu zapobiegać i jak wykorzystać tę wiedzę w swoich projektach. Wtedy razem z kolegą zapisaliśmy się na listy dyskusyjne CPL (CrackPL). Ilość wiadomości generowanych przez członków tej społeczności była powalająca. Poza standardowymi mailami z prośbą o crack (kończącą się banem na liście) było bardzo dużo merytorycznych dyskusji, które kończyły się najczęściej na IRCu. Odkrycie tego sposobu komunikacji było przełomem. Nie dość, że przyczyniło się do wzrostu tempa pisania na klawiaturze

rze (!), to umożliwiało bezpośredni kontakt z chłopakami z poszczególnych grup. Sama obecność na kanale była już dość prestiżowa, gdyż za byle przewinienie wylatywało się stamtąd z banem, a za pytanie o cracki dostawało się permanentnego bana. Szanse mieli ci, co coś potrafilo i dzielili się tą wiedzą, lub siedzieli cicho. My należeliśmy do tej pierwszej grupy. Z biegiem czasu mieliśmy dostęp do coraz większej ilości materiałów, umiejętności nasze analizy oprogramowania rosły bardzo szybko. Na komputerach mieliśmy non stop uruchomionego debuggera (w tamtych czasach najczęściej był to SoftICE). Świeżo co nabyte umiejętności bardzo przydawały się przy tworzeniu własnego oprogramowania. Pierwszym przypadkiem takiego dziwnego mariażu był program do obsługi stanów magazynowych na promie wolnocłowym. Znajomość SICE bardzo uprościło wyszukiwanie błędów. Wtedy też przeprowadziłem się do innego miasta i rozpocząłem studia. Z jednym przyjacielem pewnego pięknego zimowego wieczoru, czytając kolejny tutek z rzędu i ucząc go, jak samodzielnie analizować w debuggerze kod programu, wpadliśmy na pomysł założenia własnej grupy. Grupa była przez jakiś czas tajna, na IRCu wyłapywaliśmy ludzi mających jakąś wiedzę, albo chcących się czegoś nauczyć, i proponowaliśmy im na priv członkostwo. W ten sposób w dość krótkim czasie pozyskaliśmy kilku bardzo zdolnych ludzi. Teoretycznie grupa powinna zajmować się łamaniem zabezpieczeń, jednak my nietypowo zajmowaliśmy się bardziej programowaniem (w nazwie literka C była oficjalnie rozwijana jako Cracked, a w rzeczywistości było to Coders), nie narzucając praktycznie żadnego kodeksu członkom grupy (jeśli chcieli zajmować się łamaniem – mogli, ich sprawa). Ta wolność sumienia była dla wielu kusząca. Wtedy to doszło do kilku niezbyt sympatycznych incydentów na scenie, w tym tego, który został opisany przez kolegę w pierwszym tekście z tego działu. Założyciele grup wiedzieli oczywiście więcej niż zwykli „szeregowcy”, jednak to była wiedza przeznaczona tylko dla nich – nigdzie nie publikowana. Tajemnicą poliszynela jest współpraca niektórych grup z autorami oprogramowania. Zresztą nie jest sekretem, że wtedy na grupie newsowej pojawił się Rafał, autor jednego z bardziej znanych polskich programów, z otwartą propozycją legalizacji zdobytych licencji w zamian za ich niepublikowanie. Jedynym warunkiem było po prostu udowodnienie, że się ten numer zdobyło samodzielnie. Sporo grup miało kilka honorowych stanowisk. W naszej grupie było też kilku takich członków, były to osoby, które naszym zdaniem zasłużyły się dla rodzimej sceny, między innymi Coxoc, GustawKit, Pit^R – ich tutoriale były dla wielu pierwszym źródłem wiedzy tajemnej, oknem na magiczny świat grzebania na poziomie rejestrów i instrukcji procesora. Zresztą już wtedy co niektóre osoby bardzo wyróżniały się na scenie umiejętnościami. Byli to magicy, którzy poprawiali nawet kernel systemu Windows podczas jego działania – coś, co dla wielu ludzi wydawało się całkowicie niemożliwe. Wtedy też pojawiały się różne projekty wyprzedzające znacznie swoje czasy (np. DNC – czyli Distributed Network Cracker). Zresztą tajemnicą nie jest, że niektóre osoby nadal korzystają ze swoich starych umiejętności, tworząc własne narzędzia i udostępniając je komercyjnie.

Zastanawiająca nieco jest rosnąca popularność sceny. Może przyczynił się do tego projekt RP2K (Revolution Project 2000), do udziału w którym przystąpili członkowie z prawie każdej grupy. To był konkurs (tak to można obecnie nazwać), w którym wygrywał ten, co najwięcej wypuścił działających cracków. Projekt ten miał własną stronę WWW, własną listę dyskusyjną i grupę maniaków,

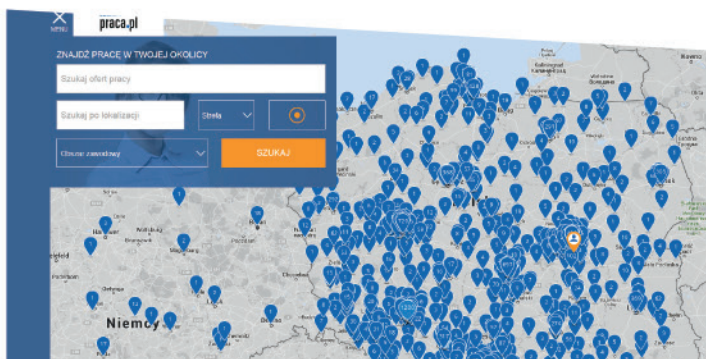
którzy za punkt honoru postawili sobie masowe łamanie programów. Wyobraźcie sobie sytuację, jest np. środa rano – pojawia się nowy numer PC World, mija kilka godzin, średnio tak do 12-14, na liście RP2K przychodzi kilkadziesiąt maili od jednego z crackerów, są to wszystko cracki do softu znajdującego się na płycie dołączonej do tego numeru (oj eax...). Mówiąc krótko, to masakra dla twórców Shareware, a raj dla piratów. Strasznie często też dochodziło do dziwnych najazdów ludzi na kanał IRCowy. Masowo się pojawiali i zadawali dziwne, irytujące pytania. Do tego dochodziły próby przejęcia kanału, co doprowadziło do spięcia kilku botnetów w jeden wielki i stworzenia własnych skryptów, które pilnowały porządku i w razie czego wzywały pozostałe maszyny na pomoc. Dziś to brzmi jak dobra zabawa młodych maniaków komputerów i w sumie tak większość z nas to traktowała. To była zabawa w poznawanie, jak to wszystko działa, do jakiego stopnia można kontrolować komputer, jak wycisnąć z niego wszystko, co się da. Zresztą pamiętam do dzisiaj pewną zabawną historię. Praktycznie zawsze miałem uruchomionego w tamtych czasach debuggera. Siedział sobie w tle spokojnie, nikomu nie wadząc, uruchamiany skrótem Ctrl+d (to ważne w tej historii). Pewnego pięknego dnia padła sieć. Dostawcą była pewna duża firma, której serwery wybitnie upodobali sobie Gumisie i C.Y.P.H.E.R.S. (r00bi, pamiętasz, jak chcieliśmy świat zmienić, uświadamiając ludziom różnice między crackerami a chakerami?), i miała niechlubny status monopolisty. Telefon na błękitną linię, rozmowa z konsultantem i ten prosi o uruchomienie programiku służącego do nawiązywania połączenia; jak się uruchomił, to poprosił o otwarcie jednego okna i wciśnięcie kombinacji klawiszy, jaką poda. Chyba się domyślacie, jaka to była kombinacja, która uruchamiała ukryte okno diagnostyki połączenia? Tak, Ctrl+d, zacząłem się śmiać w momencie, jak ten miły pan poprosił o powiedzenie, co widzę na ekranie. Ta cisza w słuchawce, po tym jak usłyszał opis tego, co widzę, była bardzo wymowna ;)

Zaskakujące może być to, że, wbrew pozorom, naprawdę bardzo dużo pożytecznej wiedzy z tamtych czasów wyniosłem. Nie dość, że nauczyłem się, jak nie należy zabezpieczać programów, to posiadałem umiejętność poprawiania istniejących narzędzi, poprzez dodawanie do nich brakujących funkcjonalności. Dzięki tej wiedzy dziś jestem w stanie dość szybko przeanalizować nieznaną binarkę i wyciągnąć swoje wnioski odnośnie jej działania. To, co było ponad 15 lat temu zwykłą zabawą, dziś okazuje się bardzo przydatną wiedzą w obecnie wykonywanej przeze mnie pracy. Obecnie praktycznie nikt w taki sposób już się nie uczy. Aktualni adepci wiedzy informatycznej kończąc studia, są wyuczeni tylko w tym kierunku, który ich interesował (o ile nie była to nauka na zasadzie „zdać i zapomnieć”). Brakuje im zaawansowanej wiedzy ogólnej, dzięki której ma się coś w stylu intuicji zawodowej. Coś, co sprawia, że nawet nowy problem, z którym przychodzi nam się zmierzyć, jesteśmy w stanie rozwiązać, bazując na posiadanych umiejętnościach, a nie jak wielu absolwentów-programistów nie potrafiących znaleźć przyczyny, czemu im kompilator zgłasza błędy linkowania (rzeczywista sytuacja u znajomego, programiści nie potrafili znaleźć przyczyny zgłaszania błędów podczas linkowania w Visual Studio – brakowało zainstalowanych w systemie starszych wersji .NET, na które target miał ustawiony projekt)...

Mariusz „maryush” Witkowski

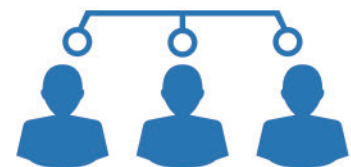


Chcesz dobrze zarobić?
Na Praca.pl codziennie znajdziesz ponad 3 000 ofert pracy
z obszaru IT i nowe technologie



Znajdź pracę w Twojej okolicy

Lokalna.praca.pl



Poleć znajomego do pracy
i zgarnij 1 000 zł

Praca.pl/rekomendacje.html

Kurs angielskiego dla programistów.

Lekcja 9

Przedstawiam dziewiątą lekcję minikursu angielskiego dla programistów. Tym razem tematem przewodnim jest bezpieczeństwo. Zachęcam do wielokrotnego wykonywania ćwiczeń, aby dobrze utrwalić sobie przyswojony materiał.

The Internet is a dangerous place! With great regularity we hear about websites becoming **unavailable due to denial of service** attacks, or displaying **modified** (and often **damaging**) information on their home pages. In other **high-profile** cases millions of passwords, email addresses and credit card **details** have been **leaked** into the **public domain**, **exposing** website users to **both** personal embarrassment and financial risk.

The purpose of **website security** is to **prevent** these (or any) sorts of **attacks**. More formally, website security is the act/practice of **protecting** websites from **unauthorized access, use, modification, destruction or disruption**.

Effective website security requires design effort across the whole of the website: in your web application, in the configuration of the web server, in your **policies** for creating and **renewing** passwords, and in client-side code. While that all sounds very **ominous**, the good news is that if you're using a server-side web framework, it will almost certainly already enable **robust** and **well-thought-out defence mechanisms** against a number of the more common attacks **"by default"**. Other attacks can be **mitigated** through your web server configuration, for example **enabling**

HTTPS. Finally, there are publically available **vulnerability scanner** tools that can help you find out if you've made any obvious mistakes.

Here are some of the most common types of attacks on websites.

CROSS-SITE SCRIPTING (XSS)

XSS is a term used to describe a class of attacks that allow an **attacker** to **inject** client-side scripts through the website into the browsers of other users. As the injected code comes to the browser from the site it is **trusted**, and can hence do things like sending the user's site **authorisation cookie** to the attacker. Once the attacker has the cookie they can **log into a site** as though they were the user and do anything the user can.

SQL INJECTION

SQL injection **vulnerabilities** enable **malicious users** to **execute arbitrary SQL code** on a database, allowing data to be **accessed, modified or deleted** irrespective of the user's **permissions**. A **suc-**

Słownik

administration rights – uprawnienia administracyjne
 attack – atak
 attacker – napastnik
 authorisation cookie – ciasteczko autoryzacyjne
 by default – domyślnie
 consent – zgoda
 consume – konsumować
 credentials – dane uwierzytelniania, dane logowania
 defence – obrona
 defence mechanism – mechanizm obronny
 denial of service – odmowa usługi
 destruction – destrukcja
 details – dane
 disruption – zakłócenie
 due to – z powodu
 effective – skuteczny
 high-profile – wysokiej klasy
 legitimate user – legalny użytkownik
 malicious user – szkodliwy użytkownik
 modification – modyfikacja
 numerous – liczny
 ominous – złowieszczy
 permissions – uprawnienia
 policy – polityka
 public domain – domena publiczna
 robust – solidny, niezawodny
 spoof identities – podrabiać tożsamości
 spurious – fałszywy
 successful – udany

to access – uzyskać dostęp
 to achieve – osiągnąć
 to damage – zniszczyć
 to delete – usunąć
 to destroy – zniszczyć
 to disrupt – zakłócać
 to enable – umożliwić
 to execute actions – wykonywać działania
 to execute arbitrary code – wykonywać dowolny kod
 to expose – ujawnić
 to flood – zalewać
 to inject – wstrzykiwać
 to leak – wyciekać
 to log into a site – zalogować się w witrynie
 to mitigate – zminimalizować, zmniejszyć
 to modify – modyfikować
 to prevent – zapobiegać
 to protect – chronić
 to renew – odnowić
 to use – używać
 traffic – ruch
 trusted – zaufany
 unauthorized access – nieuprawniony dostęp
 unavailable – niedostępny
 unusable – bezużyteczny
 vulnerability – podatność (na ataki)
 vulnerability scanner – skaner podatności na ataki
 website security – bezpieczeństwo witryny internetowej
 well-thought-out – dobrze zaplanowany

successful injection attack might **spoof identities**, create new identities with **administration rights**, access all data on the server, or **destroy/modify** the data to make it **unusable**.

CROSS SITE REQUEST FORGERY (CSRF)

CSRF attacks allow a malicious user to **execute actions** using the **credentials** of another user without that user's knowledge or **consent**.

DENIAL OF SERVICE (DOS)

DoS attack is usually **achieved** by **flooding** a target site with **spurious** requests so that access to a site is **disrupted** for **legitimate users**. The requests may simply be **numerous**, or they may individually **consume** large amounts of resource (e.g. slow **reads**, uploading of large files, etc.) DoS **defences** usually work by identifying and blocking "bad" **traffic** while allowing legitimate messages through.

ĆWICZENIA

1. Dopasuj słowa lub wyrażenia z lewej kolumny do słów lub wyrażeń z prawej. Potrafisz je wszystkie przetłumaczyć na język polski?

vulnerability	cookie
website	identities
unauthorized	arbitrary code
to execute	mechanism
spoof	rights
malicious	scanner
defence	access
authorisation	user
administration	security
high-profile	

2. Odpowiedz na poniższe pytania. Możesz cytować tekst. Powtarzaj, aż będziesz w stanie udzielić odpowiedzi bez patrzenia na tekst.

1. What do we regularly hear about websites?
2. What is the purpose of website security?
3. What is website security?
4. What does effective website security require?
5. What advantage do server-side web frameworks offer?
6. How can you mitigate some types of attacks?
7. What are vulnerability scanners used for?
8. What is Cross-Site Scripting?
9. What is SQL injection?
10. What is Cross Site Request Forgery?
11. What is Denial of Service?

3. Dokończ zdania zaczynające się od wyrażenia „The purpose of...”, jak w poniższym przykładzie.

website security -> The purpose of website security is to prevent many sorts of attacks.

1. Cross-Site Scripting attacks
2. SQL injection attacks
3. Cross Site Request Forgery attacks
4. Denial of Service attacks
5. executing SQL code on attacked database
6. modify the compromised data
7. flooding a target site with spurious requests
8. acquiring the authorisation cookie by an attacker

Źródło tekstu: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security (CC-BY-SA 2.5)



ŁUKASZ PIWKO

piwko.lukas@gmail.com

Tłumacz angielskiej i francuskiej literatury programistycznej z około 70 książkami na koncie, nauczyciel, wykładowca i maniak technologii programistycznych.

Czy programista musi być pasjonatem?

Uwaga, uwaga! Oto ON! Wybraniec! Zbawca! Czysty, płonący, zapamiętały. W ognistym rydwanie, z seledynowymi, zero-jedynkowymi lejcami. Programista pasjonat. Poświęcony programowaniu. Prawdziwie.

PRAWDZIWY PASJONAT

W internetach wizerunek programisty idealnego rysuje się bardzo wyraźnie. Celem jego życia jest programowanie. Tak długo jak to robi, jest spełniony. To jego główne zainteresowanie, jego hobby, pasja. Jest w tym oczywiście bardzo dobry i swoją działalność zna na wylot. Jednocześnie stara się nieustannie poszerzać horyzonty i czerpać z konkurencyjnych rozwiązań.

Nie dla niego praca od ósmej do szesnastej. Tak robią tylko lenie! Słabeusze! Przecież do poznania jest ciągle tyle nowych rzeczy. Taki ktoś to w pewnych okolicznościach skarb. Dobrze jest mieć go czy to w zespole, czy to jako wykonawcę projektu.

Jednak czasami wolę mieć z nimi jak najmniej do czynienia. Wiesz, dlaczego? Jeśli ktoś uważa się za takiego właśnie programistę, to w bardzo wielu przypadkach na tym nie poprzestaje. To, co robi, jest wyjątkowe, może nawet godne podziwu, ale jemu to nie wystarcza. Jeśli on jest taki, to tego samego wymaga od wszystkich innych. Gardzi pseudoprogramistami nadającymi wysoki priorytet czemuś tak banalnemu jak czas dla rodziny, wakacje bez komputera czy wreszcie... wysokość pensji. Najchętniej wyrwałby flaki licealiście pytającemu: „czy opłaca się pójść na informatykę?”. Wszyscy inni, kwestionujący świętość jego postawy i świętość czynności kodowania oraz jej „najważniejszość”, to materialistyczne plugawce. A to przecież Wielkie Powołanie! Na pohybel im!

Przyznam się z własnej nieprzymuszonej woli: doskonale taką postawę znam. Sam taki byłem z jakiegoś powodu.

CUKIER W CUKRZE

Ilu mamy pasjonatów? Niekoniecznie aż tak ekstremalnych, ale jednak pasjonatów? To zależy, jak na to spojrzeć.

Kiedyś myślałem, że prawdziwi programistyczni pasjonaci to norma. Że po prostu tak wygląda ta branża. To były czasy, gdy nowych ludzi poznawałem zwykle na konferencjach i meetupach. Czyli tam, gdzie bywają... pasjonaci właśnie!

Prawdziwy obraz polskiego IT ujrzałem, gdy zacząłem regularnie odwiedzać „normalne” firmy. Zobaczyłem tam, że prawdziwy zapalenie to właściwie rzadkość. I mało tego – zrozumiałem wartość postawy prezentowanej przez tych „normalnych”, rzemieślników! Takich, którzy pracują z kodem. Dostają zadania, wykonują je, a potem idą do domu. Trzeba się czegoś nauczyć? To się uczą. Nie trzeba? To poczytają kryminał, pobawią się z dziećmi albo pójdą do kina.

I tyle. Robią swoje. Jak dziewięćdziesiąt pięć procent populacji.

Programowanie to przecież po prostu profesja. Praca. I nie ma niczego złego w tym, że „tylko” pracą pozostanie.

Żałuję, że tak późno otworzyły mi się oczy. Mam nadzieję, że między innymi dzięki tej książce Twoje otworzą się, zanim będzie za późno.

Bez umiaru, bez przystanku, bez chwili refleksji. Latami żyłem tylko i wyłącznie kodem. Jak rozmowy, to tylko o kodowaniu. Jak lektury, to tylko techniczne. Do kina chodziłem i tam drzemałem. Oglądając serial, wyczekiwałem końca, by wskoczyć z powrotem przed klawiaturę.

Anekdota

Poznałem drugie oblicze pasji o godzinie trzeciej trzydziści w nocy. Mój dev-maraton trwał w najlepsze. Drugi czy trzeci miesiąc programowania po szesnaście, osiemnaście godzin dziennie. Bo tak trzeba. Samo się nie zrobi. Pasja! Power! Kawa, energetyk, fajka, znowu kawa – i jazda! Nie jestem przecież jakiś mięczak. Mugol.

Nagle: kap. Kap. Kap... Co to? A to wesoło na spację chlupocze krew. Prosto z mojego nosa, bez ostrzeżenia. Jak zmaltretowany musiał być organizm, by w taki sposób błagać mnie o litość? Cudem nie wyładowałem wtedy w szpitalu.

Przecież wszystko inne to marnowanie czasu!

A jednak. Skończyło się. Po ponad dekadzie – nie mogłem pa-trzeć na kod. Potrzebowałem kilkumiesięcznej przerwy, by w ogóle być w stanie myśleć o programowaniu. Straszne uczucie, uwierz mi.

Owszem, daj się ponieść. Owszem, poszukuj pasji. Owszem, celebrować odkrywanie tego wspaniałego świata. W końcu:

Passion is something that likes to be conquered
Julia Marcell

Ale... zachowaj odrobinę umiaru. Nie zmarnij tego.

WNIOSKI?

Najlepiej robić coś z pasją i mieć z tego dużą kasę.

Gorzej robić coś z pasją i nie mieć kasy.

Jeszcze gorzej robić coś bez pasji i mieć z tego kasę.

A najgorzej robić bez pasji i nie mieć kasy.

Czy lepiej być pasjonatem? Oczywiście, że tak! Jakie większe szczęście może spotkać człowieka niż realizowanie się w swoim hobby i możliwość ustawienia sobie dzięki temu życia? Na pewno stosunkowo niewielka część ludzi może się tym cieszyć. A programowanie jest zawodem, który może takie coś dać. Umiarkowany pasjonat to pozycja idealna. Świadome trzymanie dzikich zapędów w ryzach nie jest proste, ale to najlepsza droga.

Niepasjonat? Tak zwany rzemieślnik? Też jest potrzebny! Dopóki robota jest zrobiona i świat się kręci dalej, każdy wybiera swoją pasję. Można być programistą, a prawdziwe emocje zachować na coś innego.

Przede wszystkim – jako pasjonat nie wymagaj takiego samego podejścia od wszystkich wokół. Żyj i daj żyć innym. Programistyczny dżihad to zło.

I na koniec, ku refleksji:

Czy nie można być pasjonatem przez jedną trzecią doby?

Czy pasja faktycznie musi oznaczać harowanie po godzinach?

Maciej Anisierowicz

Powyższy tekst pochodzi z książki „Zawód Programista”. Więcej informacji: <http://zawodprogramista.pl/>.

Jesteś doświadczonym programistą?

Wyciśnij maxa z tej profesji!

Stawiasz pierwsze kroki w branży?

Mądrze pokieruj swoją karierą!

Dopiero zamierzasz wejść w świat IT?

Zobacz, co Cię czeka!



Ta książka to **dekada doświadczenia w jednym miejscu.**

Odpowie na Twoje pytania.
Pomoże podjąć właściwe decyzje.

To wszystko, czego potrzebuje
świadomy developer.



Game Engine Black Book. Wolfenstein 3D



Myślę, że nie rozminę się z prawdą, twierdząc, iż historia informatyki obejmuje szereg epickich bitew, chlubnych zwycięstw i sromotnych porażek. Przeciwnikami w tym nieustającym boju są wyzwania i problemy, których (pozornie) nie da się rozwiązać, ograniczenia – których (ponoć) nie da się przekroczyć, i złożoność, która wydaje się być nie do okiełznania. Pomimo to kolejne

zastępy dzielnych programistów oraz badaczy podejmują wyzwanie i odważnie stają w szranki, odkrywając i zdobywając kolejne dziewicze obszary Nieznanego. Problem w tym, że wiele z tych historii pozostaje anonimowych, a ich świadkami są jedynie osoby biorące udział w zmaganiach. Na szczęście od czasu do czasu znajdują się dzielni kronikarze, którzy nie szczędzą swojego czasu i energii po to, aby przybliżyć światu te niesamowite historie.

Jednym z takich kronikarzy jest niewątpliwie Fabien Sanglard. Na blogu tego znakomitego programisty (<http://fabiansanglard.net/>) od wielu lat znaleźć można dogłębne, a jednocześnie przystępne opisy silników przełomowych gier, które zmieniły bieg historii elektronicznej rozrywki: *Doom*, *Quake*, *Strike Commander*, *Prince of Persia*, *Duke Nukem 3D* czy *Another World*. Analizy te na przestrzeni lat zyskały wielką popularność i stały się natchnieniem wielu programistów, pokazując, że cierpliwość, ciężka praca, determinacja i wiara we własne możliwości pozwalają tworzyć produkty zmieniające świat. Nie ukrywam, że sam od wielu lat śledzę wpisy na blogu Fabiena i z wielkim entuzjazmem przyjąłem wiadomość o jego planach wydawniczych. W sierpniu 2014 roku Sanglard na swoim blogu obwieścił światu, iż zamierza wydać książkę opisującą silnik gry *Wolfenstein 3D*: protoplasty gatunku *First Person Shooter*. Pomimo tego, że wpis dotyczący książki brzmiał bardzo optymistycznie (zaczynał się słowami: *I am almost done...*), na zakończenie pracy przyszło nam czekać trzy lata. Warto jednak było, gdyż lektura *Game Engine Black Book. Wolfenstein 3D* to czysta przyjemność.

Z logicznego punktu widzenia książka Fabiena składa się z trzech głównych części: opis sprzętu, na który budowana była gra, przedstawienie Zespołu, który grę stworzył, i wreszcie – analiza kodu źródłowego gry. Część pierwsza książki (rozdział *Hardware*) zabiera nas na wycieczkę w lata 90-te poprzedniego wieku i przedstawia (dość ponury) krajobraz rynku sprzętowego ówczesnych PC-tów. Autor omawia tutaj (z punktu widzenia programisty gier, rzecz jasna) kolejne komponenty składające się na typowego blaszaka z połowy lat 90-tych: procesor, pamięć RAM, kartę graficzną, kartę muzyczną, kontrolery oraz szynę danych. Przyznam się szczerze, iż pomimo tego, że sam dość dobrze pamiętam te czasy, dopiero lektura wspomnianych rozdziałów uświadomiła mi, jak bardzo pod górkę mieli w tamtym okresie twórcy gier przeznaczonych na komputery PC. Trzeba w tym miejscu mocno podkreślić, że treść zawarta w tej części książki wychodzi bardzo daleko poza suchy opis sprzętu. Jest to umieszczona w szerszym kontekście kronikarska opowieść, ubarwiona szeregiem anegdot i komentarzy autorstwa kluczowych osób z branży IT działających w tamtym okresie. Myślę,

że dla wielu współczesnych programistów opowieść ta może być szokująca, przez co tym bardziej polecam im jej lekturę.

Tło zarysowane w pierwszym rozdziale książki stanowi świetne wprowadzenie do części kolejnej, w której autor opisuje Zespół pracujący nad grą. Młodość, pasja, talent, wiara we własne siły, determinacja, kreatywność i zwinność: przymiotniki te opisują dość dobrze charakter zespołu id Software w momencie przystępowania do pracy nad grą *Wolfenstein 3D*. W ramach drugiego rozdziału książki autor przybliżył nam ów zespół zarówno w kontekście charakterystyki poszczególnych jego członków, ale też (a może przede wszystkim) w odniesieniu do interakcji między nimi, które przekładają się bezpośrednio na kształt procesu wytwarzania gry.

W ramach kontekstu zbudowanego misternie za pomocą treści pierwszych dwóch rozdziałów autor przystępuje do meritum. W części trzeciej (rozdział *Software*) na warsztat wzięty jest oryginalny kod źródłowy gry *Wolfenstein 3D*. Autor krok po kroku opisuje poszczególne jego składowe, przy czym analiza ta odbywa się na zasadzie „od ogółu do szczegółu”, co niewątpliwie ułatwia przyswojenie prezentowanego materiału. Mamy więc po kolei: ogólny przegląd architektury silnika, procedurę startu, renderowanie 2D (menu) oraz 3D (rozgrywka), obsługę audio i efektów dźwiękowych oraz kontrolerów. Oddzielny podpunkt poświęcony jest sztuczkom zastosowanym w grze. W tej części książki w pełnej krasie objawia się talent autora polegający na umiejętnym i dogłębnym, a zarazem bardzo klarownym i przystępnym omawianiu eleganckich i niebanalnych rozwiązań trudnych problemów, z którymi zmagali się twórcy przełomowych gier. Na zakończenie, w ramach deseru, Fabien serwuje nam jeszcze dwa krótkie rozdziały (jeden traktujący na temat sequelów gry, drugi – prezentujący popularne porty *Wolfenstein'a 3D*), epilog oraz kilka dodatków, w których omówiono co ciekawsze poboczne zagadnienia związane z tworzeniem gry.

Jak podsumować całość pracy Fabiena włożonej w napisanie *Game Engine Black Book. Wolfenstein 3D*? Myślę, że słowo „klasyka” będzie w przypadku tej książki jak najbardziej na miejscu. Zresztą stuprocentowa ocena 5.0 w serwisie Amazon.com (bardzo rzadkie zjawisko w dzisiejszym świecie) mówi sama za siebie... Jak wiadomo – produkty idealne nie istnieją. *Game Engine Black Book* ma również jedną wadę. Zbyt szybko się czyta :) Książka ma 316 stron, jednakże – po części ze względu na dużą ilość zdjęć, diagramów oraz zrzutów ekranu itd.; po części zaś ze względu na wartką akcję i świetny styl autora – tempo lektury jest tak błyskawiczne, że zanim się czytelnik obejrzy, a tu już koniec. I wtedy rodzi się żal, a jednocześnie nadzieja na to, że w niedługim czasie Fabien uraczy nas kolejną częścią *Game Engine Black Book* i pozwoli nam raz jeszcze zajrzeć w przeszłość po to, aby odkryć kolejną niesamowitą, inspirującą historię!

Rafał Kocisz

Tytuł:	<i>Game Engine Black Book: Wolfenstein 3D</i>
Autor:	Fabien Sanglard
Stron:	316
Wydawnictwo:	CreateSpace Independent Publishing Platform
Data wydania:	2017-09-05



Szkolenia i warsztaty ekspertckie - **bo umysł to Twoje najważniejsze narzędzie**



DDD



ARCH



TEST&CRAFT



AGILE&SOFT



JAVA



.NET



C&CPP



WEB



BAZY



MOBILNE



EIP

SPRAWDŹ **200**
AUTORSKICH
PROGRAMÓW
SZKOLEŃ

JOIN THE E-TEAM

Praca

Chcesz mieć wpływ na to, w którym kierunku zmierza postęp technologiczny? Brać udział w prestiżowych projektach IT realizowanych przez jednego z kluczowych graczy branży ICT? Rozwijać się zawodowo i żyć pełnią życia po pracy?

Dołącz do Ericsson. Stwarzamy naszym zespołom możliwości rozwijania umiejętności i dzielenia się pasjami. U nas zrealizujesz ciekawe projekty w dobrej, przyjacielskiej atmosferze.

Dowiedz się więcej o pracy w Ericsson:
Ericsson.com/careerspoland



tiny.cc/theEteam