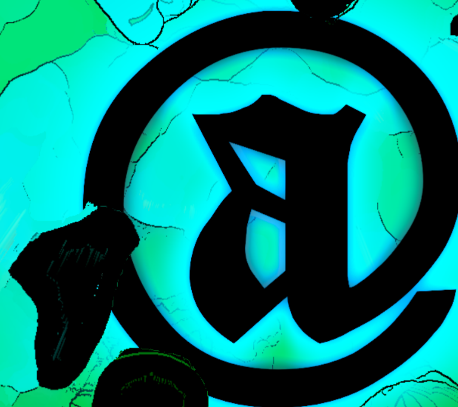
ethical.blue Magazine

ЗОНА://fe80::ae54:3ccc:1c43:7a33

Connecting to laboratory... Please wait...



Dawid Farbaniec

- зона://pwsh.exe, czyli podstawy PowerShell
- Pliki skrótu (*.LNK) w systemach Microsoft Windows
- Мова Асемблера (Asembler x64/arm64)
- Niezwykłe anomalie w językach C#/IL
- Technologia łańcucha bloków (ang. blockchain)

0xae58

Wszystkie materiały zawarte w tym czasopiśmie są chronione prawem autorskim. Kopiowanie i rozpowszechnianie opublikowanych tu materiałów bez zgody autora jest surowo zabronione. Wszystkie opublikowane tutaj materiały (w tym kody źródłowe i programy) mają charakter informacyjny i powstały wyłącznie w celach edukacyjnych. Autor niniejszego czasopisma nie ponosi odpowiedzialności za nielegalne wykorzystanie udostępnionych tu materiałów. Czytelnik niniejszego magazynu oświadcza, że wykorzystuje udostępnione materiały na własne ryzyko. Wszystkie znaki towarowe i zarejestrowane nazwy zostały użyte wyłącznie w celach informacyjnych i należą wyłącznie do ich prawnych właścicieli. Czcionki Agency FB (learn.microsoft.com/en-us/typography/font-list/agency-fb), Archivo (3fonts.google.com/specimen/Archivo/license), Open Sans (fonts.google.com/specimen/Open+Sans/license), Papyrus (learn.microsoft.com/en-us/typography/font-list/papyrus) oraz Tangerine (fonts.google.com/specimen/Tangerine/license) zostały użyte zgodnie z licencją dostarczoną od dystrybutora. Autor tego magazynu, w momencie tworzenia materiału nie działa w imieniu firm, których technologie lub produkty opisuje — za wyjątkiem, gdzie zostało to wyraźnie oznaczone. Produkty i rozwiązania opisywane w tekstach są wybierane losowo. Autor nie otrzymał żadnych pieniędzy za opisanie tych produktów lub rozwiązań — za wyjątkiem, gdzie jest to wyraźnie zaznaczone w tekście.

Spis treści

Pierwszy dzień w strefie i spotkanie 3° 4
Podstawy PowerShell dla użytkowników systemu Microsoft Windows 5
Skrypt C2.E!undead 19
Dowiązania (*.SYMLINK) oraz skróty (*.LNK) w systemie Microsoft Windows . . . 21
Tworzenie pliku skrótu (*.LNK) za pomocą PowerShell 22
Analiza statyczna pliku skrótu (*.LNK) za pomocą dotLNK 23
Anomaly.IL.ethical.blue.LNK.Z!3050 24
Zapomniana wiedza. Asembler x86-64 (x64) w systemach Microsoft Windows . . . 25
Niezwykłe anomalie w językach C#/IL dla platformy .NET 74
Nieznane rozkazy. Asembler arm64 w systemach Microsoft Windows 96
Leśny sad. Asembler arm32/arm64 na urządzeniach Raspberry Pi 124
Technologia łańcucha bloków (ang. blockchain) z przykładami w języku C# 132

Wykaz literatury **143**

Pierwszy dzień w strefie 3OHA

Zapomniane laboratorium strefy określanej terminem 3OHA (wym. zona). Morph powoli odzyskiwał siły, ale całą przeszłość pamiętał jak przez mgłę. Nie inaczej było z wiedzą, którą zdobywał przez długie noce pełne eksperymentów technologicznych.

Spotkanie 3°

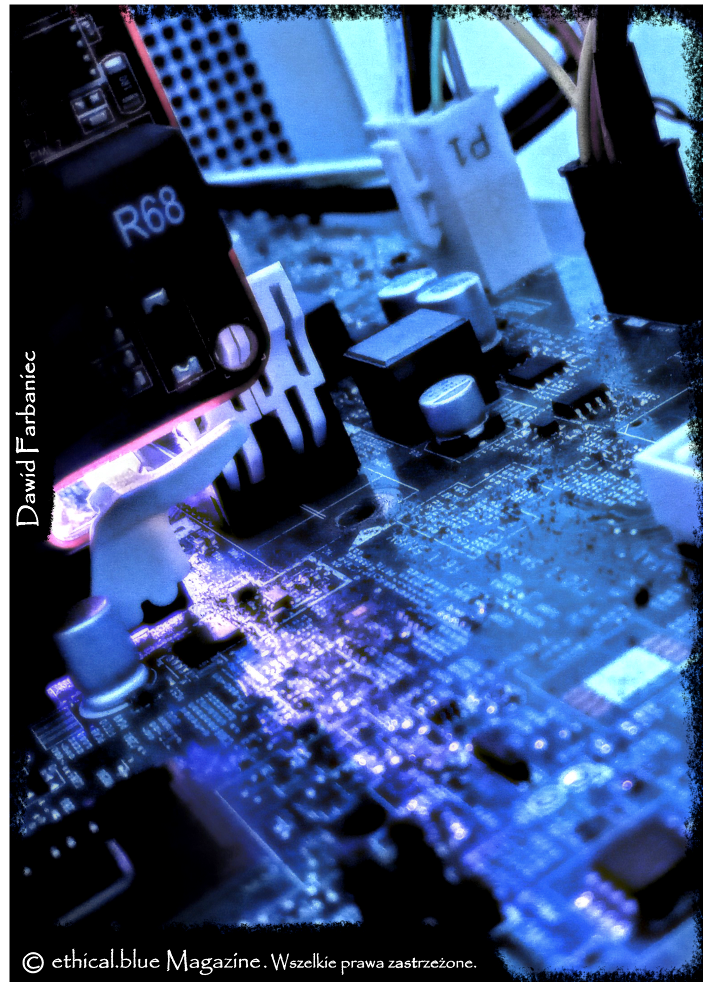
— Myślałem, że nie żyjesz Morph, ale na szczęście tylko straciłeś przytomność. — oznajmił pracownik laboratorium.

— Skąd znasz moje imię? — odpowiedział Morph powoli odzyskując przytomność i rozglądając się dookoła, gdzie znajdował się stalowy stół, pełno odłamków szkła laboratoryjnego, a w kolbach Erlenmeyera wrzały nieznane substancje.

— Masz plakietkę z imieniem. Co kilka dni znajduję jakiegoś nieszczęśnika, którego spotkała anomalia Otchłań zapomnienia (przez niektórych nazywana Niepamięć). W sumie to takie szczęście w nieszczęściu, gdyż w strefie 3OHA są o wiele groźniejsze anomalie. — oznajmił pracownik laboratorium.

BUILT. NOT BOUGHT.

Pracownik laboratorium w stercie różnych urządzeń znalazł części z których złożył prymitywny i trochę przestarzały zestaw komputerowy (Fotografia 1).



Fotografia 1. Fragment płyty głównej

— Na początek wystarczy. Z czasem dostaniesz lepszy sprzęt. — powiedział pracownik laboratorium.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Podstawy PowerShell dla użytkowników systemu Microsoft Windows

Powłoka systemowa (ang. shell) pełni rolę pośrednika pomiędzy użytkownikiem, a systemem operacyjnym i programami. Może mieć postać tekstową i przyjmować polecenia np. Wiersz polecenia (`%SystemRoot%\System32\cmd.exe`), Windows PowerShell (`%SystemRoot%\System32\Windows\PowerShell\v1.0\powershell.exe`, *wbudowany w system Microsoft Windows*), PowerShell Core (*oprogramowanie o otwartym kodzie źródłowym*) lub formę graficzną jak np. Eksplorator Windows (`%SystemRoot%\explorer.exe`).

Powłoka Windows PowerShell jest wbudowana w system Windows, ale istnieje też środowisko, które jest nazywane PowerShell Core lub po prostu PowerShell. PowerShell bardzo różni się od zwykłego Wiersza polecenia (cmd.exe), ponieważ jest oparty na środowisku wykonawczym .NET, a polecenia to nie są zwykłe pliki wykonywalne, lecz skrypty stworzone w PowerShell lub innym języku zgodnym z .NET. Warto też zaznaczyć, że polecenia PowerShell (ang. cmdlet) wykonują operacje na obiektach platformy .NET, a nie zwykłym tekście.

Instalacja PowerShell w systemach z rodziny Microsoft Windows

W celu rozpoczęcia eksperymentów należy zainstalować PowerShell w wersji odpowiedniej dla swojego systemu. W przypadku systemu Windows w celu instalacji PowerShell można użyć wbudowanego Windows PowerShell, wpisać polecenie `winget search Microsoft.PowerShell`, aby odnaleźć dostępne wersje, a następnie użyć polecenia `winget install -id Microsoft.PowerShell -source winget` w celu rozpoczęcia instalacji.

Format polecenia (ang. cmdlet) w PowerShell

Polecenia PowerShell nazywane są po ang. *command-let*, w skrócie *cmdlet*. Znajdują się w modułach i mogą być tworzone w języku skryptowym PowerShell lub innym języku zgodnym z .NET. Istnieją też funkcje oraz aliasy, czyli inne nazwy dla istniejących poleceń. Format polecenia to `Verb-Noun`, czyli `Czasownik-Rzeczownik`. Niech przykładem będzie `Get-Command` za pomocą którego można przeglądać zainstalowane na urządzeniu polecenia. W celu znalezienia poleceń z określonym czasownikiem można użyć parametru `-Verb`, a z rzeczownikiem parametru `-Noun`, czyli `Get-Command -Noun C*` odnajdzie polecenia PowerShell z rzeczownikami rozpoczynającymi się na literę `C`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Uzyskiwanie pomocy na temat poleceń

W celu uzyskania pomocy na temat określonego polecenia można użyć `Get-Help` podając dalej nazwę szukanego polecenia oraz parametr `-Online`. Na przykład: `Get-Help Format-Hex -Online`.

Polityka wykonywania skryptów

Nieuregulowane pozwolenie na wykonywanie skryptów może być zagrożeniem dla urządzenia, ponieważ ułatwia uruchomienie nieznanego kodu. Informacje na temat bieżących ustawień można uzyskać poleceniem `Get-ExecutionPolicy`.

Restricted

Pozwala uruchamiać bez problemów pojedyncze polecenia. Nie pozwala na uruchamianie skryptów.

Default

Ustawienia domyślne. `Restricted` dla systemów typu klient, a `RemoteSigned` dla serwerów z Windows.

AllSigned

Skrypty mogą być wykonywane. Wymaga podpisania skryptów przez zaufanego wydawcę, włączając w to skrypty tworzone na lokalnej maszynie.

RemoteSigned

Skrypty mogą być wykonywane. Dla skryptów uzyskanych z internetu wymagany jest podpis. Skrypty utworzone lokalnie lub niepochodzące z internetu nie muszą być podpisane. Można uruchomić niepodpisane skrypty pobrane z internetu, jeśli się je odblokuje odpowiednim poleceniem.

Undefined

Brak ustawień oznacza `Restricted` dla systemów typu klient i `RemoteSigned` dla systemów Windows w wersji serwerowej.

Unrestricted

Domyślne ustawienie dla maszyn z systemem innym, niż Windows. Istnieje bardzo duże ryzyko uruchomienia niebezpiecznego kodu.

Bypass

Skrypty nie są blokowane. Brak komunikatów ostrzegających. Istnieje bardzo duże ryzyko uruchomienia niebezpiecznego kodu.

Politykę wykonywania skryptów można zmienić poleceniem `Set-ExecutionPolicy`, czyli w przypadku chęci wyłączenia możliwości uruchamiania skryptów należy wywołać `Set-ExecutionPolicy Restricted`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Typy danych

Za pomocą operatorów -is oraz -isnot można sprawdzić czy obiekt jest określonego typu. Należy zaznaczyć, że wszystkie typy w .NET dziedziczą po object (klasa System.Object), czyli liczba całkowita też jest obiektem, co można zweryfikować za pomocą polecenia przedstawionego poniżej.

```
PS C:\> 4 -is [object]
True
PS C:\>
```

Idąc dalej można zauważyć, że liczba całkowita nie jest ciągiem znaków (napisem).

```
PS C:\> 4 -is [string]
False
PS C:\>
```

Napisy są umieszczone w cudzysłowie lub otoczone apostrofami. Ciągi znaków w apostrofach nazywane są dosłownymi, czyli umieszczone w nich wyrażenia nie są interpretowane.

```
PS C:\> Write-Host "$(2+2)"
4
PS C:\> Write-Host '$(2+2)'
$(2+2)
PS C:\>
```

Pojedynczy znak domyślnie jest typu string, ale nic nie stoi na przeszkodzie, aby wykonać konwersję typu ze string na char. Zmiana typu przez rzutowanie może zostać wykonana umieszczając nazwę typu w nawiasach prostokątnych przed obiektem.

```
PS C:\> '4' -is [char]
False
PS C:\> [char]'4' -is [char]
True
PS C:\>
```

Ciąg znaków jest obiektem typu string, czyli za pomocą kropki można uzyskać dostęp do jego właściwości. W ten sposób można np. odczytać długość (ang. length) określonego napisu.

```
PS C:\> 'ETHICAL.BLUE'.Length;
12
PS C:\>
```

Definiowanie zmiennych

Zmienna to w prostych słowach miejsce w pamięci o określonej nazwie (adresie) oraz zawartości. W celu utworzenia zmiennej o określonej nazwie i nadania jej wartości należy użyć znaku dolara.

Na przykład `$ethicalblue = 3;` tworzy zmienną typu liczba całkowita o nazwie `ethicalblue` i wartości trzy.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Jeśli zmienna ma skomplikowaną nazwę, to można użyć nawiasów klamrowych. Na przykład `${ethical.blue Magazine} = 'ethical.blue';` tworzy zmienną o podanej nazwie, która zawiera ciąg znaków *ethical.blue*.

Nic nie stoi na przeszkodzie, aby utworzyć tablicę obiektów np. `$a = 4,3,'a',1,512` (pięć elementów). Można też stworzyć kolekcję np. `$c = @{ x = 'ethical'; y = 'blue'; }` (przykładowa kolekcja zawiera dwa elementy, takie jak `x` i `y`).

Obiekt może przyjąć wartość `$null`, która oznacza, że nie przypisano żadnej wartości. Zmienne i ich wartości można wyświetlić poleceniem `Get-Variable` *nazwa-zmiennej*.

Dostęp do wartości składowej kolekcji jest możliwy za pomocą kropki.

```
PS C:\> $c = @{ x = 'ethical';
>> y = 'blue'; }
PS C:\> $c.x
ethical
PS C:\>
```

Podobnie można uzyskiwać dostęp do właściwości i metod innych typów. Przykładem może być uzyskanie dostępu do właściwości `Now` struktury `DateTime` z przestrzeni nazw `System` w celu uzyskania bieżącego czasu.

```
PS C:\> [System.DateTime]::
>> Now.ToString("HH:mm");
23:54
PS C:\>
```

W niektórych przykładach zastosowano nieznaczące dla działania kodu przełamanie linii w konsoli za pomocą klawiszy SHIFT+ENTER, które powoduje przejście do nowej linii i pojawienie się znaków >>, które nie są częścią polecenia do wpisania.

Komentarze

Fragmenty kodu źródłowego oznaczane jako komentarze jednoliniowe i wieloliniowe są pomijane przy wykonywaniu i służą do tworzenia opisów działania. Istnieje też specjalny rodzaj komentarza [14] (`#Requires`) w którym określa się wymagania dla skryptu.

Komentarz jednoliniowy zaczyna się znakiem kratki (#) i kończy wraz z napotkaniem znaku nowej linii. Natomiast komentarz wieloliniowy rozpoczyna się od znaków <# i kończy znakami #>. Może zawierać wewnątrz znaki nowej linii.

Jednostki informacji pamięci komputerowej

Przykład poniżej prezentuje jak za pomocą PowerShell można przeliczać jednostki informacji pamięci komputerowej takie jak bajty, kilobajty, megabajty, gigabajty, terabajty i petabajty. Użycie tej składni jest bardzo proste np. wpisując 4KB zwrócona zostanie liczba bajtów odpowiadająca czterem kilobajtom.

```
PS C:\> Write-Host "4 kilobajty
>> to $(4KB) bajtów.";
4 kilobajty
to 4096 bajtów.
PS C:\> 1KB
1024
PS C:\> 1MB
1048576
PS C:\> 1GB
1073741824
PS C:\> 1TB
1099511627776
PS C:\> 1PB
1125899906842624
PS C:\>
```

ForEach-Object

Polecenie `ForEach-Object`, którego alias to %, pozwala iterować po obiektach kolekcji, które można przekazać np. przez potok (|).

Przykład poniżej przekazuje kolekcję dwóch napisów przez potok do polecenia `ForEach-Object` uzyskując dostęp do poszczególnych elementów za pomocą `$_;`.

```
PS C:\> 'ethical', '.blue'
>> | % { $_; }
ethical
.blue
PS C:\>
```

Inny przykład przekazuje do pętli `ForEach-Object` (alias %) utworzoną tablicę liczb całkowitych od jeden do pięć.

```
PS C:\> 1..5 | % { $_; }
1
2
3
4
5
PS C:\>
```

Odczytywanie podstawowych informacji o urządzeniu

Systemy operacyjne z rodziny Microsoft Windows są wyposażone w Common Information Model (CIM), który pozwala na odpytywanie o sprzęt, konfigurację czy zużycie zasobów przez określoną maszynę lokalną lub zdalną. Informacje zawarte w CIM można odczytać poleceniem `Get-CimInstance`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Na przykład wywołanie `Get-CimInstance -ClassName CIM_Processor` zwróci informacje o procesorze.

Inne przykładowe wywołania to m.in.

```
Get-CimInstance -ClassName
CIM_BIOSElement

Get-CimInstance -ClassName
CIM_VideoController

Get-CimInstance -ClassName
CIM_NetworkAdapter

Get-CimInstance -ClassName
Win32_SystemUsers
```

Istnieje też podobne polecenie takie jak `Get-ComputerInfo`. Jeśli pełna nazwa właściwości nie jest dokładnie znana, to można zastosować symbol wieloznaczny (ang. wildcard), który dopuszcza dowolny znak lub ciąg znaków. Przykładowe polecenie `Get-ComputerInfo -Property '*User'` uruchomione na urządzeniu w laboratorium zwraca nazwę użytkownika przedstawioną poniżej.

```
OsRegisteredUser
--------
magazine@ethical.blue
```

Zmienne automatyczne

Środowisko PowerShell zawiera specjalne zmienne, które przechowują informacje na temat stanu związanego z wykonywaniem poleceń. Modyfikacja tych obiektów jest niezalecana [14], a ich stan powinno się kontrolować udostępnianymi metodami. Zmienne automatyczne można wyświetlić poleceniem `Get-Variable`.

Najmniejszym elementem leksykalnym skryptu jest token. Przykład poniżej konwertuje liczbę binarną `01111` na wartość szesnastkową o długości czterech bajtów. Pierwszy token z ostatniej linii można pobrać poleceniem `$^`, a ostatni token za pomocą składni `$$`. Natomiast zmienna `$?` zawiera status czy ostatnie polecenie się powiodło (`True`) czy nie (`False`).

```
PS C:\> "0x{0:X8}" -F 0b1111
0x0000000F
PS C:\> $^
0x{0:X8}
PS C:\> $$
0b1111
PS C:\> $?
True
PS C:\>
```

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Instrukcja warunkowa if

Typ logiczny [bool] może przyjmować jedną z dwóch wartości: prawda (ang. true) lub fałsz (ang. false). Bardzo często od stanu zmiennej logicznej zależy ścieżka wykonania kodu np. w instrukcji warunkowej `if`. Poniższe polecenie przypisuje do zmiennej o wymyślonej nazwie `$v` wynik zwrócony przez *command-let* `Test-Path` z parametrem `-IsValid`. Następuje tutaj sprawdzenie czy podana ścieżka `C:\` jest poprawna oraz czy istnieje na urządzeniu.

```
$v = Test-Path -IsValid 'C:\';
```

Wartość zmiennej logicznej można sprawdzić prosto instrukcją warunkową `if`. Poniżej następuje weryfikacja czy zmienna `$v` przechowuje wartość `True` i jeśli tak, to na ekranie wyświetlany jest tekst *Exists.*, czyli ścieżka `C:\` istnieje oraz została w podstawowym stopniu przefiltrowana, że nie zawiera nieodpowiednich znaków.

```
if ($v) { Write-Host 'Exists.'; }
```

Dalej przedstawiono trywialny przykład w którym do zmiennej `$x` przypisywana jest liczba całkowita trzy. Następnie za pomocą instrukcji `if` oraz operatora porównania `-eq` następuje sprawdzenie czy wartość zmiennej `$x` wynosi jeden. Wtedy wykonałby się pierwszy blok kodu otoczony nawiasami klamrowymi, ale w tym przypadku zmienna `$x` wynosi trzy, więc sprawdzany jest kolejny warunek. Przy `elseif` następuje sprawdzenie czy wartość zmiennej `$x` wynosi zero. Nie tym razem, gdyż wynosi trzy. Rozgałęzień z użyciem `elseif` może być więcej. Wszystkie pozostałe przypadki są obsługiwane przez `else` i to rozgałęzienie zostanie wybrane w tym przykładzie.

```
PS C:\> $x = 3;
PS C:\> if ($x -eq 1)
>> { Write-Host "`$x = 1"; }
>> elseif ($x -eq 0)
>> { Write-Host "`$x = 0"; }
>> else
>> { Write-Host "`$x = $x"; }
$x = 3
PS C:\>
```

Warty uwagi jest zapis `"`$x = $x"`, który za pomocą znaku ucieczki (ang. escape) w postaci odwrotnego apostrofu (ang. backtick, `` ` ``) pozwala wypisać nazwę zmiennej poprzedzoną znakiem dolara. A dalej po znaku równości wyświetlana jest wartość zmiennej, ponieważ nie zastosowano znaku ucieczki.

Dzięki temu możliwe jest wyświetlenie tekstu z bieżącą wartością zmiennej w postaci

```
#...
$x = 3
PS C:\>
```

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Operator warunkowy (ang. ternary)

W niektórych przypadkach instrukcję warunkową `if` warto zastąpić operatorem nazywanym w ang. *ternary* [14], którego składnia jest następująca *warunek ? jeśli-prawda : jeśli-fałsz;*

Nie można przy tej okazji pomijać informacji o tym, że zmienne różnych typów mogą ewaluować do typu logicznego `[bool]`. Bez zagłębienia się w szczegóły implementacji mogłoby się wydawać, że zero to fałsz, a jeden to prawda. Jest tak, ale z małą poprawką. Liczba całkowita zero ewaluuje do wartości logicznej fałsz. Natomiast inne liczby po konwersji na typ `[bool]` stają się wartością logiczną prawda.

```
PS C:\> [bool]0;
False
PS C:\> [bool]1;
True
PS C:\> [bool]-1;
True
PS C:\> [bool]100;
True
PS C:\>
```

Podstawowe metody tworzenia pętli

Instrukcja `for` pozwala utworzyć klasyczną pętlę zawierającą przeważnie zmienną, która jest zwiększana lub zmniejszana oraz warunek, który jest sprawdzany wraz z każdą iteracją i dopóki jest on prawdziwy, to instrukcje we wnętrzu pętli są wykonywane.

```
PS C:\> for ($i = 0;
>> $i -lt 5;
>> $i++) {
>> Write-Host "$i";
>> }
0
1
2
3
4
PS C:\>
```

W powyższym przykładzie do zmiennej `$i` przypisywane jest zero. Dalej po średniku jest warunek `$i -lt 5;` (dopóki `$i` mniejsze od pięć). A za pomocą operatora inkrementacji (`$i++`) wartość zmiennej `$i` jest zwiększana wraz z każdą iteracją (nazywaną też obrotem).

Wnętrze pętli to instrukcje pomiędzy nawiasami klamrowymi. Dla celów zaprezentowania składni znajduje się tu tylko instrukcja wypisująca wartość zmiennej `$i`. Jak można zauważyć pętla odlicza od zero do czterech.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Inny rodzaj pętli to zastosowanie instrukcji `foreach`, którą w celu zrozumienia mechanizmu działania można opisać słowami *Dla każdego elementu w kolekcji wykonaj określone instrukcje.*

```
PS C:\> foreach($i in 0..4) {
>> Write-Host "$i";
>> }
0
1
2
3
4
PS C:\>
```

W rzeczywistych przypadkach iterowanie po liczbach jest mało spotykane. Przeważnie iteracja następuje po kolekcjach zwróconych przez poszczególne *command-let*.

Na przykład za pomocą polecenia `Get-ChildItem` można odczytać elementy potomne z wprowadzonej lokalizacji takiej jak system plików, ale nie tylko. Odczytywać można też klucze rejestru systemowego i inne [14].

W celu odczytania elementów z dysku `C:\` można zastosować polecenie przedstawione poniżej.

```
Get-ChildItem -Path "C:\"
```

lub stosując składnię bez nazw parametrów.

```
Get-ChildItem "C:\"
```

Po umieszczeniu powyższego polecenia w pętli można uzyskać dostęp do elementów z dysku `C:\` takich jak pliki i foldery.

A wewnątrz przykładowej pętli zaprezentowano mały fragment oferowanych możliwości. Do każdego przetwarzanego elementu można uzyskiwać dostęp jak do obiektu określonej klasy. Przykład poniżej odczytuje nazwy plików i folderów za pomocą znaku kropki, który tutaj jest operatorem dostępu do składowej (`$i.Name`).

```
PS C:\> foreach($i in
>> Get-ChildItem -Path "C:\") {
>> Write-Host $i.Name;
>> }
eSupport
inetpub
Program Files
Program Files (x86)
Users
Windows
appverifUI.dll
vfcompat.dll
PS C:\>
```

Na systemie laboratoryjnym przykład zwrócił nazwy sześciu folderów oraz dwóch plików *.dll*.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

ForEach-Object -Parallel { }

W przypadku ciężkich operacji, które przetwarzają wiele elementów oraz działania te mogą być wykonywane niezależnie warto stosować *command-let* o nazwie `ForEach-Object` z parametrem `-Parallel`, którego alias to znak procent (%).

Przykład poniżej jest trywialny, ponieważ ma na celu jedynie zaprezentowanie mechanizmu działania.

```
PS C:\> 1..4 | % -Parallel {
>> [System.Threading.Thread]::
>> CurrentThread.ManagedThreadId;
>> } -ThrottleLimit 8
40
42
43
44
PS C:\>
```

Do pętli `ForEach-Object` przez potok (|) przekazywana jest kolekcja liczb całkowitych od jeden do cztery. Za pomocą parametru `-Parallel` aktywowane jest przetwarzanie równoległe, a parametr `-ThrottleLimit` ogranicza w tym przypadku maksymalną ilość wątków do liczby osiem. Jako dowód, że utworzone zostanie wiele wątków można wyświetlić identyfikator `ManagedThreadId` przy każdej iteracji. Wartości te są różne od siebie, czyli uruchomione zostały oddzielne wątki.

Różne znaczenie tych samych operatorów

Typy wbudowane (np. `int`, `string` etc.) oraz klasy użytkownika mają zaimplementowane różne operatory, które zależnie od rodzaju operandów mają określony mechanizm działania.

Na przykład operator + w przypadku liczb całkowitych wykona dodawanie. Natomiast w przypadku napisów połączy je w jeden ciąg znaków.

```
("ethical" + ".blue" +
[Environment]::NewLine) * 3
```

Przykład powyżej łączy trzy napisy w całość, czyli `"ethical"`, `".blue"` oraz zdefiniowany w środowisku znak nowej linii (właściwość `NewLine` klasy `Environment`). Wyrażenie to jest zgrupowane za pomocą nawiasów okrągłych, aby zwrócony ciąg znaków został powielony trzy razy za pomocą operatora mnożenia.

Na konsoli po wykonaniu prezentowanego polecenia pojawi się

```
ethical.blue
ethical.blue
ethical.blue

PS C:\>
```

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Get-FileHash

W celu weryfikacji integralności danych można użyć polecenia `Get-FileHash`, które zwróci wynik określonej funkcji skrótu (ang. hash). Przykładowe wywołanie przedstawione poniżej oblicza funkcję SHA512 z pliku o nazwie *ethicalblue.txt* z aktualnego katalogu oznaczonego tutaj znakiem kropki. Wywołanie *command-let* jest zgrupowane nawiasami okrągłymi, aby za pomocą znaku dostępu do składowej (kropka) wyświetlić właściwość `Hash` (Fotografia 4).

W przypadku nowicjuszy pomocnym może okazać się polecenie `Show-Command`, które pomaga użytkownikowi wyświetlając okno graficznego interfejsu zawierające parametry wywoływanego *command-let*. Składnia jest następująca `Show-Command Get-FileHash`, gdzie `Get-FileHash` może być zastąpione innym wybranym poleceniem dla którego ma pojawić się okno pomagające wybrać parametry (Fotografia 4).

Test-Connection

Polecenie `Test-Connection` pozwala w łatwy sposób zweryfikować możliwość połączenia się z określoną maszyną podając adres sieciowy (Fotografia 2).

```
PS C:\> Test-Connection ::1 |
>> Format-Table Source,
>> Address, Status;

Source       Address  Status
------       -------  ------
ETHICALBLUE  ::1      Success
ETHICALBLUE  ::1      Success
ETHICALBLUE  ::1      Success
ETHICALBLUE  ::1      Success

PS C:\>
```

Fotografia 2. Przykładowe użycie polecenia Test-Connection

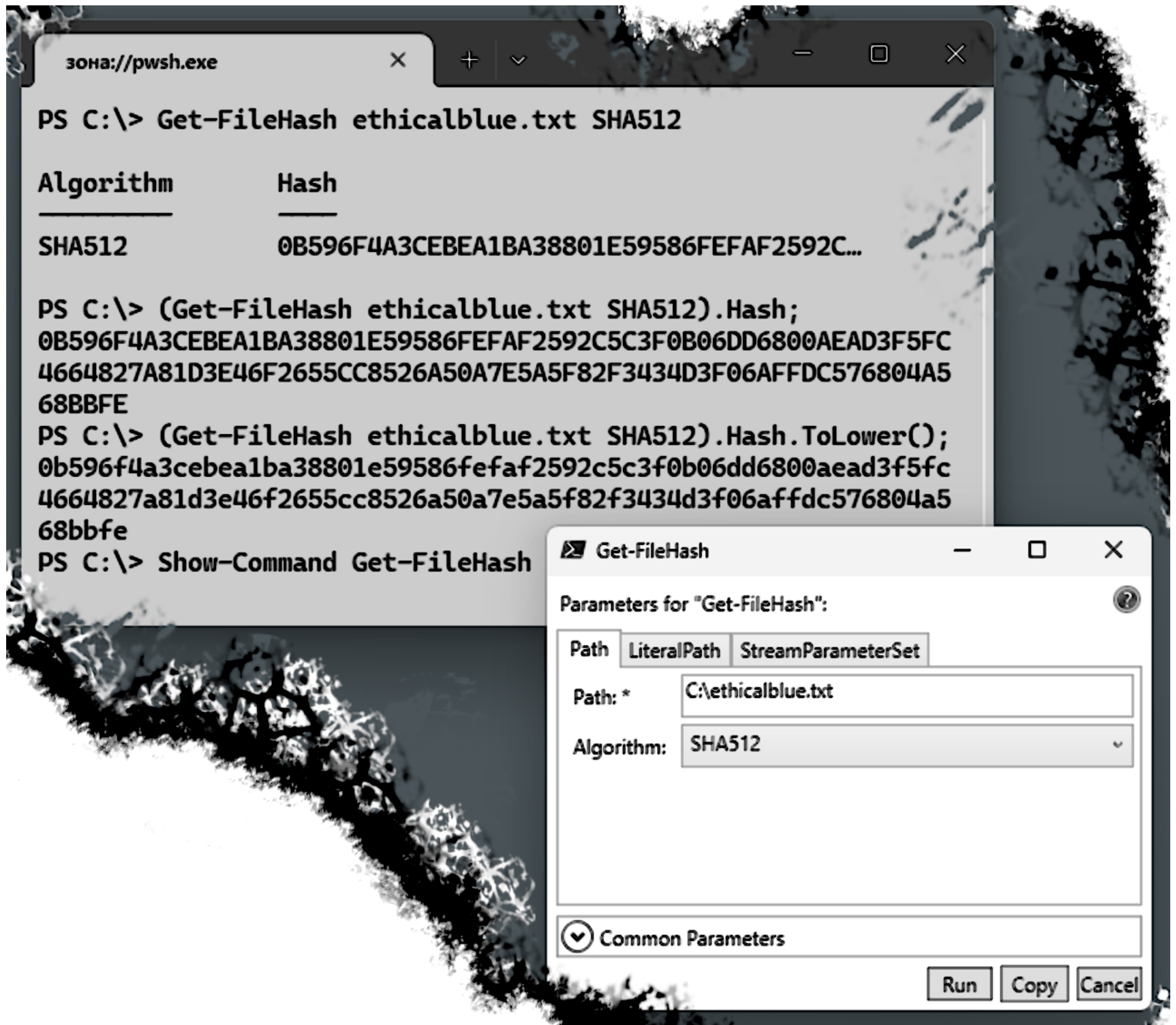
Get-HotFix

Za pomocą polecenia `Get-HotFix` można zweryfikować czy lokalne lub zdalne maszyny mają zainstalowane najnowsze poprawki bezpieczeństwa (Fotografia 3).

```
PS C:\> Get-HotFix -Description 'Security*'
>> | FT HotFixID, InstalledOn

HotFixID  InstalledOn
--------  -----------
KB
KB

PS C:\>
```

Fotografia 3. Przykładowe użycie polecenia Get-HotFix

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.



Fotografia 4. Przykładowe użycie polecenia Get-FileHash

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Format-Hex

Za pomocą polecenia `Format-Hex` można w łatwy sposób wyświetlać obiekty w postaci heksadecymalnej. Przykład poniżej przedstawia przekazanie potokiem ciągu znaków, aby wyświetlić wartości poszczególnych bajtów.

```
PS C:\> "ethical" | Format-Hex

   Label: String (System.String) <76F45CAD>

          Offset Bytes                                           Ascii
                 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
          ------ ----------------------------------------------- -----
0000000000000000 65 74 68 69 63 61 6C                            ethical

PS C:\>
```

Polecenie `Format-Hex` znajduje też zastosowanie przy odczytywaniu fragmentów plików, aby sprawdzić np. określone właściwości, zawartość czy rozpoznać typ przechowywanych danych (format pliku). Poniżej przedstawiono odczytanie sygnatury `MZ`, która jest charakterystyczna dla plików wykonywalnych PE (ang. Portable Executable). Następuje tutaj odczytanie dwóch bajtów (parametr `-Count`) z pliku `program.exe` (dysk `C`) zaczynając od przesunięcia 0x00000000, czyli od początku.

```
PS C:\> Format-Hex -Path program.exe -Offset 0x00000000 -Count 2

   Label: C:\program.exe

          Offset Bytes                                           Ascii
                 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
          ------ ----------------------------------------------- -----
0000000000000000 4D 5A                                           MZ

PS C:\>
```

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Uruchomienie polecenia PowerShell zakodowanego algorytmem Base64

Narzędzie PowerShell posiada możliwość uruchomienia poleceń zakodowanych algorytmem Base64. Ciąg w postaci Base64 można uzyskać za pomocą kodu

```
[System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.
   GetBytes("..."));
```

Niegroźny ładunek (ang. payload) może być następujący

```
[System.Convert]::ToBase64String([System.Text.Encoding]::Unicode.
   GetBytes("Add-Type -AssemblyName System.Windows.Forms; [System.Windows
   .Forms.MessageBox]::Show('ethical.blue Magazine', 'ethical.blue', 'OK
   ', 'Information');"));
```

Teraz polecenia zakodowane za pomocą Base64 to

QQBkAGQALQBUAHkAcABlACAALQBBAHMAcwBlAG0AYgBsAHkATgBhAG0AZQAgAFMAeQBzAHQA
ZQBtAC4AVwBpAG4AZABvAHcAcwAuAEYAbwByAG0AcwA7ACAAWwBTAHkAcwB0AGUAbQAuAFcA
aQBuAGQAbwB3AHMALgBGAG8AcgBtAHMALgBNAGUAcwBzAGEAZwBlAEIAbwB4AF0AOgA6AFMA
aABvAHcAKAAnAGUAdABoAGkAYwBhAGwALgBiAGwAdQBlACAATQBhAGcAYQB6AGkAbgBlACcA
LAAgACcAZQB0AGgAaQBjAGEAbAAuAGIAbAB1AGUAJwAsACAAJwBPAEsAJwAsACAAJwBJAG4A
ZgBvAHIAbQBhAHQAaQBvAG4AJwApADsA

Uruchomienie przykładowego ładunku (ang. payload) jest możliwe za pomocą

```
powershell.exe -EncodedCommand QQBkAGQALQBUAHkAc...
```

lub

```
powershell.exe -enc QQBkAGQALQBUAHkAc...
```

Oczywiście zamiast trzech kropek powyżej należy wprowadzić cały ładunek (ang. payload) w postaci ciągu zakodowanego algorytmem Base64.

Skrypt C2.E!undead

— Nasz personel co jakiś czas znajduje implanty w urządzeniach na terenie, gdzie przeprowadzamy eksperymenty. — oznajmił pracownik laboratorium.

— Doceniam Twoje zaangażowanie Morph, ale nie jesteś jeszcze przygotowany, aby dołączyć do zespołu zajmującego się sprawą ukrytych kanałów komunikacyjnych. — stwierdził.

— Kontynuujmy naukę. Za chwilę utworzymy prosty kanał komunikacyjny w celach eksperymentalnych. — dodał.

Poniższym poleceniem można uruchomić nowe okna konsoli PowerShell z prawami administratora systemu.

```
Start-Process pwsh -Verb RunAs
Start-Process pwsh -Verb RunAs
```

Dla zachowania przejrzystości wszystko odbywa się na lokalnej maszynie i przez protokół nieszyfrowany. Przykład (Scriptum 1) nasłuchuje na połączenia na dostępnych interfejsach sieciowych na porcie 65535 (`http://+:65535/`).

```
[System.Net.HttpListener]::new() |
  % { $z = $_; $z.Prefixes.Add("
  http://+:65535/"); $z.Start();
  while($z.IsListening){ $c = $z.
  GetContext(); $s = $c.Request.
  InputStream; $x = [System.IO.
  StreamReader]::new($s, $c.Request
  .ContentEncoding); $m = $x.
  ReadToEnd(); Write-Output $m; $s.
  Close(); $x.Close(); $c.Response.
  Close(); } }
```

Scriptum 1. Edukacyjny skrypt Command and Control nasłuchujący na połączenia

```
Invoke-WebRequest -Uri http
  ://[::1]:65535/ -UserAgent "
  ethical.blue Magazine" -Body "
  Undead here." | Out-Null
```

Scriptum 2. Edukacyjny skrypt wysyłający żądanie (ang. request)

Wysłanie żądania (ang. request) następuje przez *command-let* o nazwie `Invoke-WebRequest` (Scriptum 2). Za pomocą parametru `-UserAgent` można ustawić własną nazwę przeglądarki internetowej, parametr `-Body` to wiadomość, a za pomocą `| Out-Null` wyłączane jest zwrócenie rezultatu na ekran konsoli.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

W oknie zatytułowanym *Undead* następuje trzykrotna próba wysłania żądania o treści *Undead here*. Natomiast okno nazwane *Command & Control* nasłuchuje za pomocą utworzonego obiektu typu `[System.Net.HttpListener]` w pętli `while` i wyświetla odebrane dane przez `Write-Output` (Fotografia 5).

— Najbardziej zagrożone są porzucone lub zapomniane fragmenty infrastruktury naszego laboratorium. Urządzenia, które działają i są podłączone do sieci, ale nikt nie monitoruje ich stanu stają się niczym zombie. Implanty niewiadomego pochodzenia w postaci skryptów powodują nietypowe i niebezpieczne zachowania całej sieci. W połączeniu z anomaliami strefy 3OHA temat staje się na prawdę mocno skomplikowany. — zaznaczył pracownik laboratorium.

Command & Control

```
PS C:\> [System.Net.HttpListener]
::new() | % { $z = $_; $z.Prefixe
s.Add("http://+:65535/"); $z.Star
t(); while($z.IsListening){ $c =
$z.GetContext(); $s = $c.Request.
InputStream; $x = [System.IO.Stre
amReader]::new($s, $c.Request.Con
tentEncoding); $m = $x.ReadToEnd(
); Write-Output $m; $s.Close(); $
x.Close(); $c.Response.Close(); }
 }
Undead here.
Undead here.
Undead here.
```

Undead

```
PS C:\> Invoke-WebRequest
 -Uri http://[::1]:65535/
 -UserAgent "ethical.blue
 Magazine" -Body "Undead
here." | Out-Null
PS C:\> Invoke-WebRequest
 -Uri http://[::1]:65535/
 -UserAgent "ethical.blue
 Magazine" -Body "Undead
here." | Out-Null
PS C:\> Invoke-WebRequest
 -Uri http://[::1]:65535/
 -UserAgent "ethical.blue
 Magazine" -Body "Undead
here." | Out-Null
PS C:\>
```

Fotografia 5. Przykład przesłania wiadomości z implantu Undead do skryptu nasłuchującego typu Command and Control

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dowiązania (*.SYMLINK) oraz skróty (*.LNK) w systemie Microsoft Windows

Odwołania do obiektów w systemie plików mogą przyjmować różną postać. Dowiązanie twarde (ang. hard link) to bezpośrednie łącze do danych. Nie ulega uszkodzeniu w przypadku usunięcia pliku do którego utworzono dowiązanie twarde, a dane są łatwo dostępne dopóki istnieje do nich choć jeden hard link.

Natomiast dowiązanie miękkie (*.SYMLINK) zawiera informacje dotyczące ścieżki pod którą powinny być dane. Z tego powodu usunięcie danych powoduje, że dowiązanie miękkie wskazuje na nieistniejący obiekt.

Jeszcze innym rodzajem odwołania się do pliku czy folderu jest skrót (*.LNK), jednak ten rodzaj łącza to po prostu plik z rozszerzeniem .LNK i metadanymi, którego zawartość można łatwo odczytać, a nawet modyfikować.

Przed rozpoczęciem analizy statycznej może być potrzebna zmiana rozszerzenia pliku skrótu z .LNK, na przykład na .LNK.anomaly, aby uniknąć odczytania miejsca docelowego na które skrót wskazuje, zamiast pliku .LNK.

Można to wykonać za pomocą następującego polecenia w aplikacji cmd.exe

```
rename C:\m.LNK C:\m.LNK.anomaly
```

Ewentualnie za pomocą PowerShell następującym poleceniem

```
Rename-Item -Path C:\m.LNK -NewName
    C:\m.LNK.anomaly
```

Format plików skrótu (*.LNK)

Podstawowe struktury z których składa się plik skrótu w systemie Windows przedstawia dokumentacja *[MS-SHLLINK] Shell Link (.LNK) Binary File Format*.

Schemat wspomnianego formatu w postaci struktur języka C można przedstawić tak jak poniżej.

```
struct WindowsShortcut {
  struct ShellLinkHeader sHeader;
  struct LinkTargetIDList sTarget;
  struct LinkInfo sInfo;
  struct StringData sStringData;
  struct ExtraData sExtraData;
};
```

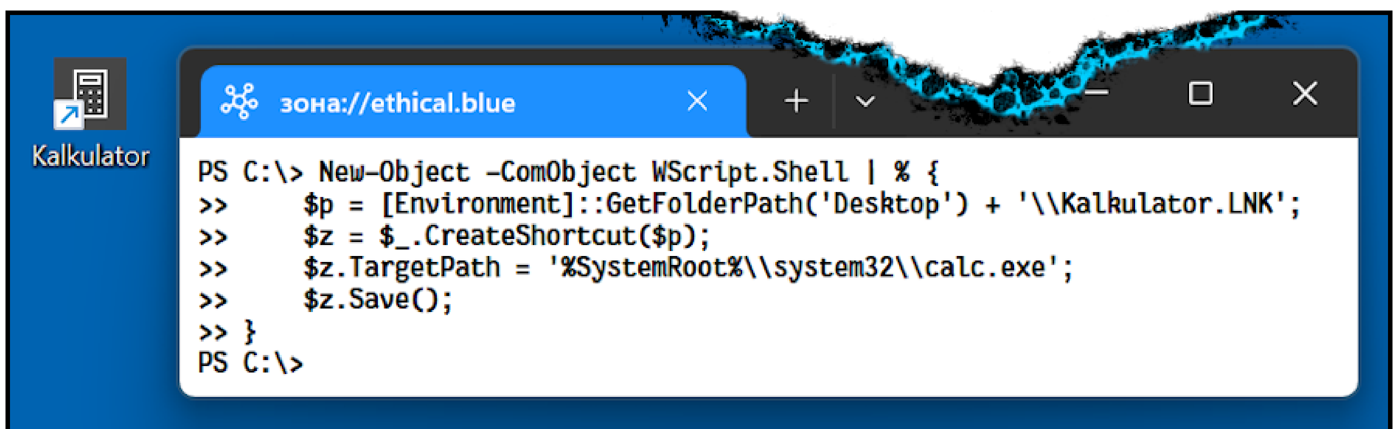
W celu zrozumienia prezentowanych kodów źródłowych należy wstępnie zapoznać się ze wspomnianą wcześniej dokumentacją.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Tworzenie pliku skrótu (*.LNK) za pomocą PowerShell

Zwykłe skróty można utworzyć klikając prawym przyciskiem myszy i wybierając Nowy / Skrót. W przypadku anomalii lub plików skrótu o nietypowych właściwościach wymagana będzie umiejętność napisania kodu w PowerShell lub innym języku programowania.

Poniżej przedstawiono przykładowy skrypt w języku PowerShell, który pozwala utworzyć plik skrótu w katalogu Pulpit. Przykład ustawia element docelowy tworzonego skrótu na program Kalkulator (calc.exe).

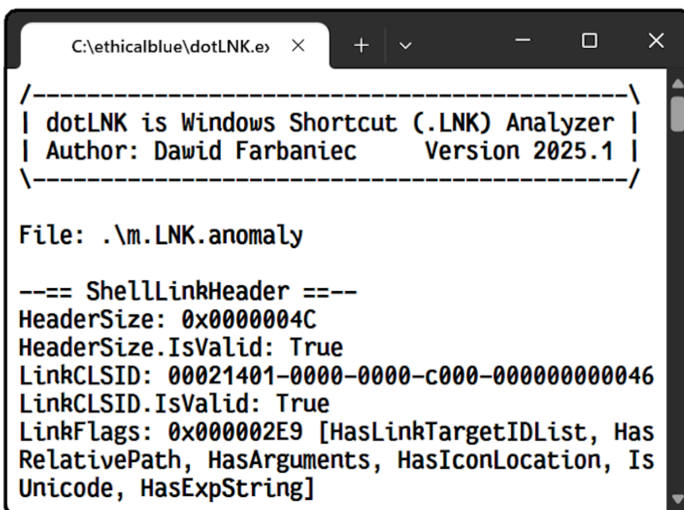



```
New-Object -ComObject WScript.Shell | % {
    $p = [Environment]::GetFolderPath('Desktop') + '\\Kalkulator.LNK';
    $z = $_.CreateShortcut($p);
    $z.TargetPath = '%SystemRoot%\\system32\\calc.exe';
    $z.Save();
}
```

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Analiza statyczna pliku skrótu (*.LNK) za pomocą narzędzia dotLNK

Podstawowe informacje o pliku skrótu (*.LNK) można odczytać narzędziem dotLNK, które jest aplikacją konsolową stworzoną w języku programowania o nazwie C# (Fotografia 6).



Fotografia 6. Narzędzie dotLNK uruchomione w konsoli PowerShell

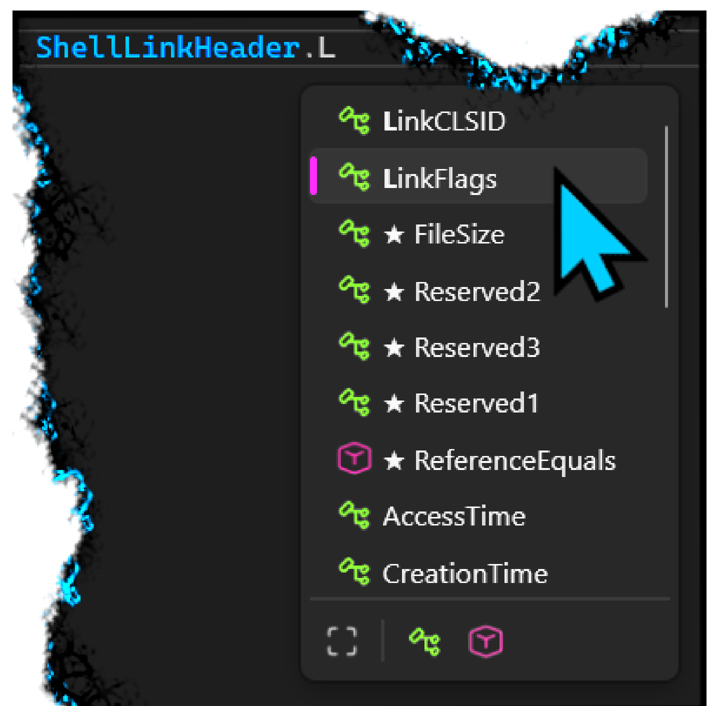
Kod źródłowy narzędzia dotLNK powinien być dostępny pod adresem internetowym przedstawionym poniżej.

//ethical.blue/n/dotLNK.zip

Powyższe archiwum zawiera rozwiązanie dla środowiska Microsoft Visual Studio. Udostępnienie kodu źródłowego pozwala rozszerzać możliwości narzędzia dotLNK.

Dodatkowo nic nie stoi na przeszkodzie, aby użyć dotLNK w postaci biblioteki we własnych programach (Fotografia 7).

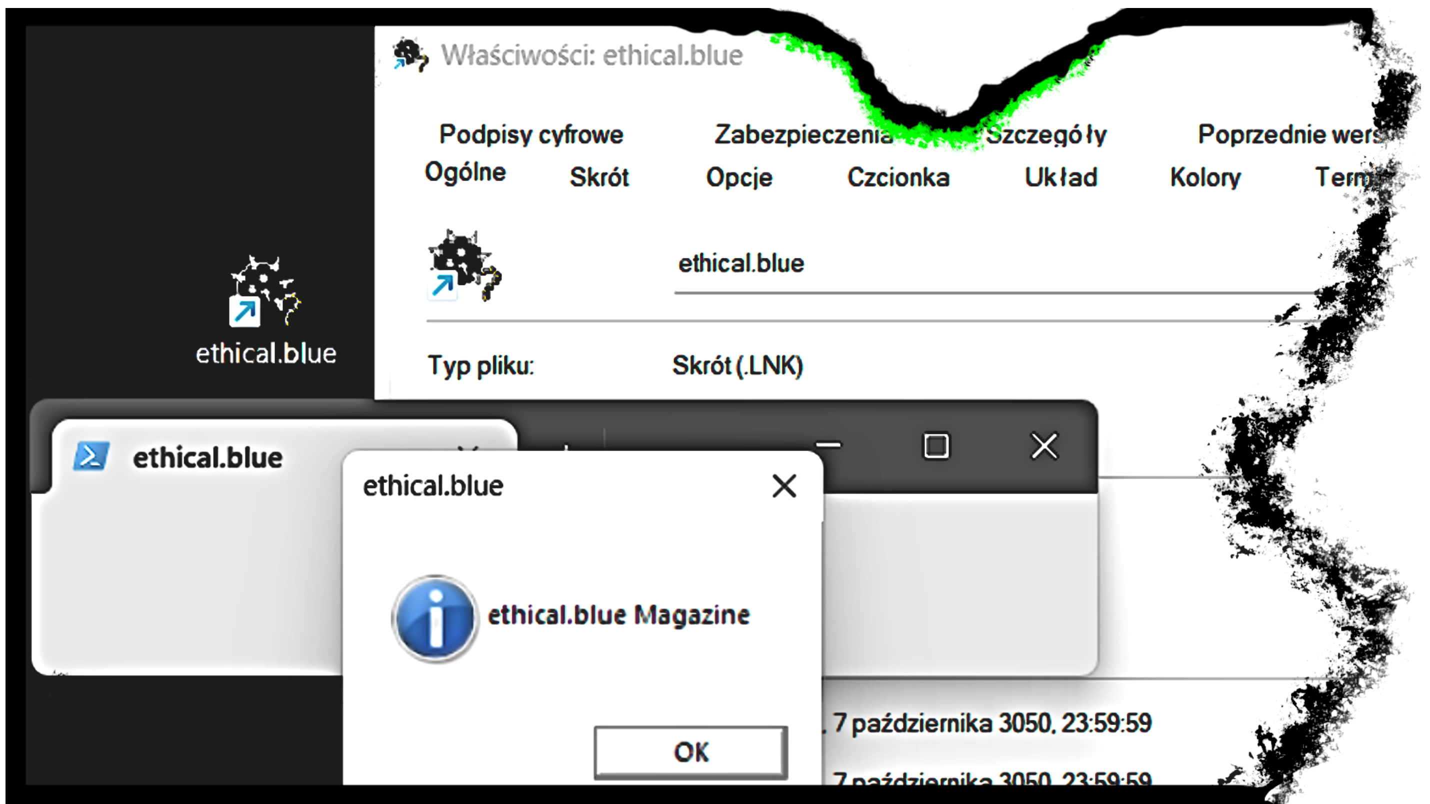
Dostęp do określonej struktury lub pola formatu plików *.LNK jest bardzo prosty w zastosowaniu. Wystarczy w środowisku Microsoft Visual Studio wprowadzić nazwę struktury i po kropce powinny wyświetlić się podpowiedzi dostępnych elementów.



Fotografia 7. dotLNK_Library pozwala na łatwy dostęp do elementów formatu pliku skrótu (*.LNK)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

ETHICAL.BLUE // ART // Anomaly.IL.ethical.blue.LNK.Z!3050




```
#Anomaly.IL.ethical.blue.LNK.Z!3050
New-Object -ComObject WScript.Shell | % {
    $p = [Environment]::GetFolderPath('Desktop') + '\\ethical.blue.LNK';
    $z = $_.CreateShortcut($p);
    $z.TargetPath = '%SystemRoot%\\system32\\WindowsPowerShell\\v1.0\\
   powershell.exe';
    $z.Arguments = '-Command "Add-Type -AssemblyName System.Windows.
   Forms; [System.Windows.Forms.MessageBox]::Show(''ethical.blue Magazine
   '', ''ethical.blue'', ''OK'', ''Information'');"';
    $z.IconLocation = "pifmgr.dll,36"; $z.Save();
    $i = Get-Item $p; $i.CreationTime = "7 October 3050 23:59:59"; $i.
   LastWriteTime = "7 October 3050 23:59:59"; $i.LastAccessTime = "7
   October 3050 23:59:59";
}
```

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

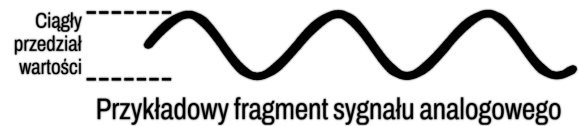
Zapomniana wiedza. Asembler x86-64 (x64) w systemach Microsoft Windows

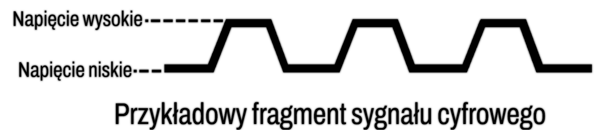
Powszechne systemy komputerowe działają na wartościach binarnych (dwójkowych). Oznacza to, że informacje mogą być reprezentowane w postaci ciągów bitów.

Bit jest najmniejszą jednostką informacji w pamięci komputerowej. Może on przyjmować jeden z dwóch stanów: wysoki lub niski. Spotyka się też inne określenia, np. jeden lub zero, ustawiony lub nieustawiony czy też prawda lub fałsz.

Osiągnięcie tylko dwóch stanów jest możliwe dzięki zastosowaniu sygnału cyfrowego zamiast analogowego (Fotografia 8). Sygnał analogowy, nawet jeśli jest w określonych granicach napięcia, to posiada ciągły przedział wartości.

Za pomocą kwantyzacji, czyli zmniejszenia precyzji możliwe jest uzyskanie wartości napięcia w postaci liczb całkowitych. Dzięki temu stan wysoki to np. 5 V (wolt), a stan niski to 0 V (wolt). Należy oczywiście wziąć poprawkę, że te wartości napięcia nie są idealne, a ich odchylenie od stałej wartości jest nazywane szumem (ang. noise), czyli 4.95 V to nadal stan wysoki, a 0.05 V to stan niski.





© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

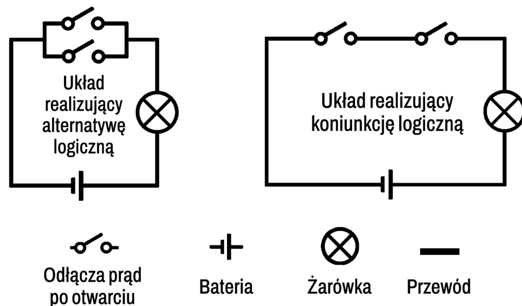
Fotografia 8. Sygnał analogowy oraz cyfrowy

Wartości binarne (dwójkowe) jako, że przyjmują tylko jeden z dwóch możliwych stanów: zero lub jeden, nazywane są też wartościami logicznymi. W celu przetwarzania tego rodzaju danych stosuje się mechanizmy nazywane bramkami logicznymi (ang. logic gate).

Elementy tego typu mogą realizować różne funkcje logiczne np. alternatywa czy koniunkcja.

Koniunkcja (iloczyn logiczny) zwraca w wyniku prawdę, wtedy i tylko wtedy, gdy wszystkie przyjmowane argumenty są prawdą. Natomiast alternatywa (suma logiczna) zwraca w wyniku prawdę, gdy chociaż jeden argument jest prawdą.

W celu lepszego przedstawienia działania funkcji logicznych można stworzyć prymitywny układ edukacyjny. Wystarczy bateria, przewody oraz żarówka (Fotografia 9).



Fotografia 9. Przykładowe układy edukacyjne realizujące funkcje OR (alternatywa) i AND (koniunkcja)

Tablice prawdy (ang. truth tables)

Tablice prawdy nazywane też matrycami logicznymi są przejrzystym sposobem na prezentację działania poszczególnych funkcji logicznych.

Bity, bajty i słowa

Bity połączone w oktet (8 bitów) tworzą bajt. Z kolei dwa bajty (16 bitów) komponują się w słowo maszynowe (ang. word). Jednak nie w każdym przypadku słowo maszynowe jest rozmiaru 16 bitów.

Na platformie sprzętowej x86-64 (x64) słowo ma rozmiar 16 bitów, ale np. w urządzeniach opartych o architekturę arm64 (AArch64) słowo ma rozmiar 32 bity.

Podstawowe rozmiary danych w architekturze x86-64 (x64)

Poniżej przedstawiono podstawowe rozmiary danych bez określonego typu w architekturze x64.

- bit – przechowuje wartość jeden lub zero,
- półbajt (ang. nibble) – 4 bity,
- bajt (ang. byte) – 8 bitów,
- słowo maszynowe (ang. word) – 2 bajty, czyli 16 bitów,
- podwójne słowo maszynowe (ang. doubleword) – 32 bity,
- poczwórne słowo maszynowe (ang. quadword) – 64 bity,
- ośmiokrotne słowo maszynowe (ang. octword) – 128 bitów,
- dwa ośmiokrotne słowa maszynowe (ang. double octword) – 256 bitów,
- cztery ośmiokrotne słowa maszynowe (ang. quad octword) – 512 bitów.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Matryce logiczne

Koniunkcja (iloczyn logiczny)

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Alternatywa (suma logiczna)

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

Alternatywa wykluczająca

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Zaprzeczenie (negacja)

A	not A
0	1
1	0

Dysjunkcja

A	B	A nand B
0	0	1
0	1	1
1	0	1
1	1	0

Zaprzeczona alternatywa

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

Zaprzeczona alternatywa wykluczająca

A	B	A xnor B
0	0	1
0	1	0
1	0	0
1	1	1

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Systemy liczbowe binarny i heksadecymalny

System binarny oparty o dwie cyfry, zero i jeden, jest przyjazną formą reprezentowania bitów. Jednak większe wartości liczbowe zapisywane w tym systemie są dość długie. Na przykład 65535 w systemie dziesiętnym to 1111111111111111 w binarnym (wszystkie bity ustawione). Natomiast system heksadecymalny oparty o szesnaście znaków, czyli 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F jest przyjazną formą reprezentowania bajtów. Na przykład 65535 w systemie dziesiętnym to 0FFFFh w heksadecymalnym. Mamy tutaj dwa bajty o wartościach 0FFh, czyli 255 traktując wartość jako liczbę bez znaku. Liczby binarne w asemblerze MASM x64 (ml64.exe) zapisuje się z przyrostkiem b, a wartości heksadecymalne z przyrostkiem h. Inną powszechnie stosowaną notacją zapisu liczb heksadecymalnych jest przedrostek 0x, czyli np. 0xFFFF. Wspomniany sposób reprezentowania liczb szesnastkowych obowiązuje m.in. w językach C++ i C#.

Liczby ze znakiem i bez znaku

Jeśli przechowywana wartość jest traktowana jak liczba całkowita ze znakiem, to najstarszy bit (najbardziej znaczący bit) nazywany bitem znaku określa czy wartość jest ujemna czy dodatnia.

Oznacza to, że 0FFh w systemie heksadecymalnym może oznaczać 255 w systemie dziesiętnym jako liczba całkowita bez znaku albo -1 jako liczba całkowita ze znakiem.

Dopełnienie zerami oraz rozszerzenie z zachowaniem znaku

Zmiana typu danych na większy wymaga odpowiedniego potraktowania wartości liczbowej. W przypadku liczby ze znakiem musi nastąpić powielenie bitu znaku na starsze bity. Natomiast dla wartości bez znaku stosowane jest dopełnienie zerami (Fotografia 10).

Rozszerzenie z zachowaniem znaku (ang. sign extension)

Dopełnienie zerami (ang. zero extension)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Fotografia 10. Rozszerzenie z zachowaniem znaku oraz dopełnienie zerami

Przykład. Jeśli bajt o wartości 11110011 (-13) zostanie zamieniony na słowo, to należy powielić bit znaku na starsze bity. Dzięki takiej operacji liczba to nadal -13 (1111111111110011), a nie 243 (0000000011110011).

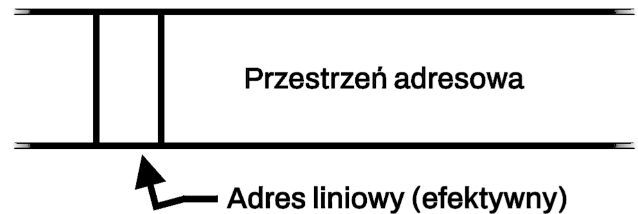
Skróty związane z poszczególnymi wersjami architektury x86/x64

Rodziny procesorów w architekturze 16-bitowej to 8086, 8186 i 8286. Natomiast w architekturze 32-bitowej to 8386, 8486, 8586, 8686 etc. Architektura 64-bitowa określana jest jako x86-64. Inne określenia dla architektury 64-bitowej to x86 (64-bit), x64, Intel 64 i AMD64. Natomiast dla architektury 32-bitowej to x86-32, x86 (32-bit), x32 i IA-32.

Pamięć operacyjna

Pamięć fizyczna to moduły sprzętowe zainstalowane w urządzeniu, czyli np. dyski twarde, pamięci zewnętrzne czy kości pamięci operacyjnej. Natomiast pamięć wirtualna to ciągła przestrzeń adresowa od 0 do 2^{64} - 1 (od 0000000000000000h do FFFFFFFFFFFFFFFFh). Wirtualne obszary pamięci dostępne dla programów są tłumaczone na mniejsze obszary pamięci fizycznej za pomocą różnych mechanizmów sprzętowych i programowych. Dawniej w systemie MS-DOS (programy 16-bitowe) pamięć była podzielona na segmenty takie jak np. segment kodu, danych czy stosu. Adresowanie odbywało się poprzez podanie selektora segmentu i wartości przesunięcia. W trybie 64-bitowym adresy są liniowe, a adresem bazowym selektorów segmentowych jest zero.

Fragmenty pamięci często są przedstawiane jako ciągi bajtów wraz z odpowiadającymi im adresami (Fotografia 11).



Fotografia 11. Płaski model pamięci (ang. flat)

Rejestry ogólnego przeznaczenia

Rejestry to nieadresowane elementy podobne do pamięci, ale mniejszych rozmiarów (najczęściej od 8 bitów do 512 bitów).

— Nie rozumiem. Po co są te wszystkie rejestry? — zapytał Morph.

Przykładowe zastosowania rejestrów to

- Wyświetlone funkcją `MessageBox` okno komunikatu zwróci w rejestrze `RAX` informacje jaki przycisk kliknął użytkownik.
- System operacyjny Windows będzie spodziewał się kodu wyjścia w rejestrze `RCX` przy kończeniu pracy programu funkcją `ExitProcess`.
- Jeśli operacja dodawania wykonana rozkazem `ADD` zwróci zero, to zostanie ustawiony bit nazywany `ZF` w rejestrze `RFLAGS`.
- Konwencja wywoływania funkcji może stawiać wymagania, aby zachować na stosie i przywrócić niektórym rejestrom poprzednią wartość przed powrotem z funkcji, aby uniknąć błędów.

— Wszystko zależy od kontekstu, ale jak widać rejestry mogą informować o stanie różnych operacji, przechowywać wyniki czy służyć przekazywaniu operandów. — wyjaśnił pracownik laboratorium.

Rejestr akumulatora

Rejestr ulotny (ang. volatile). Nie ma konieczności przywrócenia rejestrowi poprzedniej wartości przed wyjściem z funkcji.

Rejestr akumulatora `RAX` jest 64-bitowy i dzieli się na mniejsze części takie jak

- `EAX` o rozmiarze 32 bity
  (młodsze podwójne słowo rejestru `RAX`)
- `AX` o rozmiarze 16 bitów
  (młodsze słowo rejestru `EAX`)
- `AH` o rozmiarze 8 bitów
  (starszy bajt rejestru `AX`)
- `AL` o rozmiarze 8 bitów
  (młodszy bajt rejestru `AX`)

Domyślne zastosowanie to

- Przechowywanie operandów i rezultatów operacji arytmetycznych i logicznych.
- Przechowywanie operandów i rezultatów wraz z rejestrem danych (`EDX`), gdy wynik nie mieści się w pojedynczym rejestrze.
- Odczytywanie informacji o procesorze rozkazem `CPUID`.
- Przechowywanie wartości zwracanej przez funkcje Windows API.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestr bazowy

Rejestr nieulotny (ang. nonvolatile) — jego wartość powinna zostać zachowana na początku funkcji i przywrócona przed wyjściem z funkcji, która używa tego rejestru.

Rejestr bazowy RBX jest 64-bitowy i dzieli się na mniejsze części takie jak

- EBX o rozmiarze 32 bity (młodsze podwójne słowo rejestru RBX)
- BX o rozmiarze 16 bitów (młodsze słowo rejestru EBX)
- BH o rozmiarze 8 bitów (starszy bajt rejestru BX)
- BL o rozmiarze 8 bitów (młodszy bajt rejestru BX)

Domyślne zastosowanie to

- Do przechowywania adresu bazowego w starszym kodzie.
- Odczytywanie informacji o procesorze rozkazem CPUID.

Rejestr licznika

Rejestr ulotny (ang. volatile). Nie ma konieczności przywrócenia rejestrowi poprzedniej wartości przed wyjściem z funkcji.

Rejestr licznika RCX jest 64-bitowy i dzieli się na mniejsze części takie jak

- ECX o rozmiarze 32 bity (młodsze podwójne słowo rejestru RCX)
- CX o rozmiarze 16 bitów (młodsze słowo rejestru ECX)
- CH o rozmiarze 8 bitów (starszy bajt rejestru CX)
- CL o rozmiarze 8 bitów (młodszy bajt rejestru CX)

Domyślne zastosowanie to

- W rozkazach przesunięć i obrotów bitowych jako indeks (o ile bitów przesunąć czy obrócić wartość).
- W pętlach i powtarzalnych operacjach jako licznik iteracji.
- Odczytywanie informacji o procesorze rozkazem CPUID.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestr danych

Rejestr ulotny (ang. volatile). Nie ma konieczności przywrócenia rejestrowi poprzedniej wartości przed wyjściem z funkcji.

Rejestr danych RDX jest 64-bitowy i dzieli się na mniejsze części takie jak

- EDX o rozmiarze 32 bity (młodsze podwójne słowo rejestru RDX)
- DX o rozmiarze 16 bitów (młodsze słowo rejestru EDX)
- DH o rozmiarze 8 bitów (starszy bajt rejestru DX)
- DL o rozmiarze 8 bitów (młodszy bajt rejestru DX)

Domyślne zastosowanie to

- Do przechowywania operandów i rezultatów w operacjach mnożenia i dzielenia.
- Przechowywanie operandów i rezultatów wraz z rejestrem akumulatora (EAX), gdy wynik nie mieści się w pojedynczym rejestrze.
- Do przechowywania numeru portu w operacjach wejścia-wyjścia.

Rejestry dodatkowe R8..R15

Rejestry R12, R13, R14 oraz R15 są nieulotne (ang. nonvolatile) — ich wartość powinna zostać zachowana na początku funkcji i przywrócona przed wyjściem z funkcji, która używa tych rejestrów.

Rejestry dodatkowe od R8 do R15 są 64-bitowe i dzielą się na mniejsze części takie jak

- R8D..R15D o rozmiarze 32 bity (młodsze podwójne słowo rejestru R8..R15)
- R8W..R15W o rozmiarze 16 bitów (młodsze słowo rejestru R8D..R15D)
- R8B..R15B o rozmiarze 8 bitów (młodszy bajt rejestru R8W..R15W)

Domyślne zastosowanie to

- Rejestry pomocnicze.
- Rejestry R10 i R11 są ulotne (ang. volatile) i ich wartość może być utracona przy wywołaniach systemowych [10]. Dlatego może być konieczność zachowania ich aktualnej wartości przed wywołaniem. Rejestr R11 przechowuje wartość flag (RFLAGS) w rozkazach SYSCALL i SYSRET.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestr indeksowy źródła

Rejestr nieulotny (ang. nonvolatile) — jego wartość powinna zostać zachowana na początku funkcji i przywrócona przed wyjściem z funkcji, która używa tego rejestru.

Rejestr indeksowy źródła `RSI` jest 64-bitowy i dzieli się na mniejsze części takie jak

- `ESI` o rozmiarze 32 bity (młodsze podwójne słowo rejestru `RSI`)
- `SI` o rozmiarze 16 bitów (młodsze słowo rejestru `ESI`)
- `SIL` o rozmiarze 8 bitów (młodszy bajt rejestru `SI`)

Domyślne zastosowanie to

- Przechowywanie adresu operandu źródłowego dla rozkazów operujących na ciągach znaków.

Rejestr indeksowy docelowy

Rejestr nieulotny (ang. nonvolatile) — jego wartość powinna zostać zachowana na początku funkcji i przywrócona przed wyjściem z funkcji, która używa tego rejestru.

Rejestr indeksowy docelowy `RDI` jest 64-bitowy i dzieli się na mniejsze części takie jak

- `EDI` o rozmiarze 32 bity (młodsze podwójne słowo rejestru `RDI`)
- `DI` o rozmiarze 16 bitów (młodsze słowo rejestru `EDI`)
- `DIL` o rozmiarze 8 bitów (młodszy bajt rejestru `DI`)

Domyślne zastosowanie to

- Przechowywanie adresu operandu docelowego dla rozkazów operujących na ciągach znaków.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestr wskaźnika bazowego

Rejestr nieulotny (ang. nonvolatile) — jego wartość powinna zostać zachowana na początku funkcji i przywrócona przed wyjściem z funkcji, która używa tego rejestru.

Rejestr wskaźnika bazowego RBP jest 64-bitowy i dzieli się na mniejsze części takie jak

- EBP o rozmiarze 32 bity (młodsze podwójne słowo rejestru RBP)
- BP o rozmiarze 16 bitów (młodsze słowo rejestru EBP)
- BPL o rozmiarze 8 bitów (młodszy bajt rejestru BP)

Domyślne zastosowanie to

- Przechowywanie adresu bazowego wskaźnika ramki stosu (ang. stack frame).

Rejestr wskaźnika stosu

Rejestr nieulotny (ang. nonvolatile) — jego wartość powinna zostać zachowana na początku funkcji i przywrócona przed wyjściem z funkcji, która używa tego rejestru.

Rejestr wskaźnika stosu RSP jest 64-bitowy i dzieli się na mniejsze części takie jak

- ESP o rozmiarze 32 bity (młodsze podwójne słowo rejestru RSP)
- SP o rozmiarze 16 bitów (młodsze słowo rejestru ESP)
- SPL o rozmiarze 8 bitów (młodszy bajt rejestru SP)

Domyślne zastosowanie to

- Przechowywanie adresu wskaźnika wierzchołka stosu (adresu ostatniej wartości odłożonej na stosie).
- W połączeniu z odpowiednim przesunięciem (ang. offset) pozwala na uzyskanie dostępu do określonych wartości na stosie programu.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestr wskaźnika instrukcji

Rejestr wskaźnika instrukcji `RIP` jest 64-bitowy i dzieli się na mniejsze części takie jak

- `EIP` o rozmiarze 32 bity
  (młodsze podwójne słowo rejestru `RIP`)
- `IP` o rozmiarze 16 bitów
  (młodsze słowo rejestru `EIP`)

Domyślne zastosowanie to

- Przechowuje adres następnego rozkazu do wykonania. Na przykład. Przy odłożeniu tej wartości na stosie przez rozkaz `CALL` przed wywołaniem procedury możliwy jest później powrót za oznaczone miejsce. Inny przykład. W przypadku kodu niezależnego od miejsca w pamięci (np. wstrzykiwanego w proces) możliwe jest adresowanie względem wskaźnika instrukcji (ang. RIP relative addressing) co zapobiega popsuciu adresów do danych, gdy kod zmieni swoje miejsce w pamięci operacyjnej.

Rejestry segmentowe

Rejestry segmentowe to `CS`, `DS`, `ES`, `SS`, `FS` i `GS`. Rejestry segmentowe są o rozmiarze słowa (16 bitów).

Domyślne zastosowanie to

- W płaskim modelu pamięci rejestry `CS`, `DS`, `ES` i `SS` są traktowane jakby adresem bazowym segmentów było zero.
- W przypadku systemu operacyjnego Windows rejestry `FS` i `GS` w trybie użytkownika (ang. user mode, ring 3) wskazują na wewnętrzne struktury systemowe takie jak Thread Environment Block (TEB) czy Process Environment Block (PEB).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestr flag

Rejestr znaczników RFLAGS jest nazywany również rejestrem flag procesora [10]. Rejestr flag jest 64-bitowy i dzieli się na mniejsze części takie jak

- EFLAGS o rozmiarze 32 bity (młodsze podwójne słowo rejestru RFLAGS)
- FLAGS o rozmiarze 16 bitów (młodsze słowo rejestru EFLAGS)

Znaczenie poszczególnych bitów to

- Bit 0 (CF): Flaga przeniesienia (ang. carry flag) — Flaga jest ustawiana na wartość jeden, jeśli ostatnie dodawanie spowodowało przeniesienie poza najstarszy bit w wyniku lub ostatnie odejmowanie spowodowało pożyczenie spoza tego bitu. W przeciwnym wypadku flaga jest zerowana. Instrukcje logiczne (AND, OR, XOR) powodują wyzerowanie tej flagi. Flagę można ustawić rozkazem STC, a wyzerować za pomocą CLC.
- Bit 2 (PF): Flaga parzystości (ang. parity flag) — Flaga jest ustawiana na wartość jeden, jeśli (dla niektórych operacji) najmłodszy bajt zawiera parzystą liczbę bitów o wartości jeden. W przeciwnym wypadku flaga jest zerowana.
- Bit 4 (AF): Flaga przeniesienia pomocnicznego (ang. auxiliary carry flag) — Flaga jest ustawiana na wartość jeden, jeśli operacja arytmetyczna spowoduje przeniesienie poza trzeci bit wyniku lub pożyczenie spoza tego bitu. W przeciwnym wypadku flaga jest zerowana.
- Bit 6 (ZF): Flaga zerowa (ang. zero flag) — Flaga jest ustawiana na wartość jeden, jeśli ostatnia operacja arytmetyczna zwróciła w wyniku zero. W przeciwnym wypadku flaga jest zerowana. Rozkazy porównań (CMP, TEST etc.) mają wpływ na tę flagę.
- Bit 7 (SF): Flaga znaku (ang. sign flag) — Flaga jest ustawiana na wartość jeden, jeśli ostatnia operacja arytmetyczna zwróciła wartość ujemną. W przeciwnym wypadku (dla wartości dodatniej) flaga jest zerowana. Innymi słowy: Flaga jest ustawiana zgodnie z najstarszym bitem wyniku (bitem znaku).
- Bit 10 (DF): Flaga kierunku (ang. direction flag) — Flaga decyduje w jakim kierunku mają być przetwarzane ciągi znaków. Dotyczy to rozkazów operacji na napisach (MOVSx, SCASx, LODSx, CMPSx, OUTSx, INSx). Flagę można ustawić rozkazem STD, a wyzerować rozkazem CLD. Jeśli flaga jest ustawiona, to wskaźnik na fragment napisu jest zm-

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

niejszany. W przeciwnym wypadku jest zwiększany.

- Bit 11 (`OF`): Flaga przepełnienia (ang. overflow flag) — Flaga jest ustawiana na jeden, jeśli w ostatniej operacji arytmetycznej najstarszy bit wyniku (bit znaku) jest różny od bitu znaku operandów. W przeciwnym wypadku flaga jest zerowana. Innymi słowy: Ustawienie flagi oznacza, że wynik operacji nie mieści się w typie danych (rozmiarze) operandu docelowego. Dla rozkazu dzielenia `DIV` flaga ta jest niezdefiniowana. Instrukcje logiczne powodują wyzerowanie tej flagi.

- Bit 8 (`TF`): Flaga pułapki (ang. trap flag) — Programy ustawiają tę flagę na jeden, aby aktywować tryb wykonywania rozkazów krok po kroku na potrzeby debugowania. Wyzerowanie tego bitu wyłącza ten tryb. Gdy tryb wykonywania krok po kroku jest aktywny, to natychmiast po wykonaniu pojedynczego rozkazu generowany jest wyjątek (ang. debug exception), który obsługują odpowiednie procedury używane przez narzędzia typu debugger.

- Bit 9 (`IF`): Flaga przerwania (ang. interrupt flag) — Programy ustawiają tę flagę na jeden, aby aktywować odpowiedzi na przerwania. W przypadku wyzerowania tej flagi przerwania są wstrzymywane.

- Bity 13:12 (`IOPL`): Flaga uprzywilejowania I/O (ang. I/O privilege level) — Pole `IOPL` jest rozmiaru dwóch bitów i określa poziom uprzywilejowania wymagany do wykonywania rozkazów uzyskujących dostęp do adresów w przestrzeni wejścia/wyjścia. Dla programów, które wykonują tego rodzaju rozkazy, aktualny poziom uprzywilejowania (`CPL`) musi być większy bądź równy od wartości pola `IOPL`. Aktualny poziom uprzywilejowania dla programu jest przechowywany w dwóch najmłodszych bitach rejestru segmentowego `CS`. Tryb użytkownika to poziom 3 (ang. ring 3), natomiast tryb jądra to poziom 0 (ang. ring 0). Istnieją też poziomy 1 i 2.

- Bit 14 (`NT`): Flaga zagnieżdżenia zadań (ang. nested task) — Rozkaz `IRET` odczytuje pole `NT`, aby określić czy aktualne zadanie jest zagnieżdżone w innym. Jeśli flaga `NT` jest ustawiona na jeden, to aktualne zadanie jest zagnieżdżone. W przeciwnym wypadku aktualne zadanie jest „na samej górze" (niezagnieżdżone). Procesor ustawia tę flagę, gdy następuje przełączenie wynikające z rozkazu `CALL`, przerwania czy wyjątku.

- Bit 16 (`RF`): Flaga wznowienia wykonania (ang. resume flag) — Jeśli flaga jest ustawiona na jeden, to tymczasowo wstrzymuje raportowanie pułapek w try-

bie debugowania, aby zapobiec wielokrotnym rzucaniem wyjątków (ang. debug exception). Pozwala to na wznowienie wykonania rozkazu, który został wstrzymany z powodu wyjątku debugowania. Flaga jest zerowana po każdej pomyślnie wykonanej instrukcji poza `JMP`, `CALL`, `INTn` oraz `IRET` (może ustawiać flagę `RF`).

- Bit 17 (`VM`): Flaga wirtualnego trybu 8086 (ang. virtual-8086 mode) — Programy ustawiają tę flagę na jeden, aby aktywować tryb wirtualny procesora 8086. W przypadku wyzerowania tej flagi następuje wyjście z tego trybu.

- Bit 18 (`AC`): Flaga sprawdzenia wyrównania (ang. alignment check) — Programy ustawiają tę flagę na jeden (wraz z bitem `AM` w rejestrze `CR0`), aby wykonywać automatyczne sprawdzanie wyrównania. W prostych słowach: Dostęp do danych niewyrównanych (np. słowo maszynowe pod adresem nieparzystym) powoduje wyjątek typu *alignment-check*.

- Bit 19 (`VIF`): Wirtualna flaga przerwania (ang. virtual interrupt flag) — Wirtualna flaga przerwania, która pozwala programom w trybie wirtualnym 8086 korzystać z flagi przerwania bez powodowania wyjątków, które mogłyby wystąpić przy modyfikacji flagi `IF` z bitu 9 rejestru `RFLAGS`.

- Bit 20 (`VIP`): Flaga oczekującego przerwania (ang. virtual interrupt pending) — Flaga informuje czy przerwanie oczekuje wykonania (wartość jeden). W przypadku, gdy flaga jest wyzerowana nie ma oczekujących przerwań.

- Bit 21 (`ID`): Flaga identyfikacji możliwości procesora (ang. ID flag) — Jeśli program może modyfikować tę flagę, to oznacza, że wspierana jest instrukcja `CPUID`, która pozwala odczytać wiele informacji o możliwościach procesora.

Rejestry jednostki zmiennoprzecinkowej x87

Nazwa koprocesor pochodzi z dawnych czasów, kiedy to procesor posiadał osobną, dodatkową jednostkę wspomagającą obliczenia. Rozszerzenie nazywane x87 zawiera własny zestaw rejestrów i instrukcji do operacji na liczbach zmiennoprzecinkowych. `FPR0..FPR7` to rejestry fizyczne, których młodsze 64-bity są nałożone na rejestry `MM0..MM7`, czyli modyfikacja rejestrów MMX powoduje zmiany w rejestrach FPU (ang. floating point unit). Rejestry `ST0..ST7` są wewnętrznie zorganizowane w formie stosu, gdzie `ST0` wskazuje na wierzchołek. Podczas odkładania wartości na wierzchołek stosu rejestrów istniejące wartości są przesuwane dalej. [10]

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestry rozszerzenia MMX

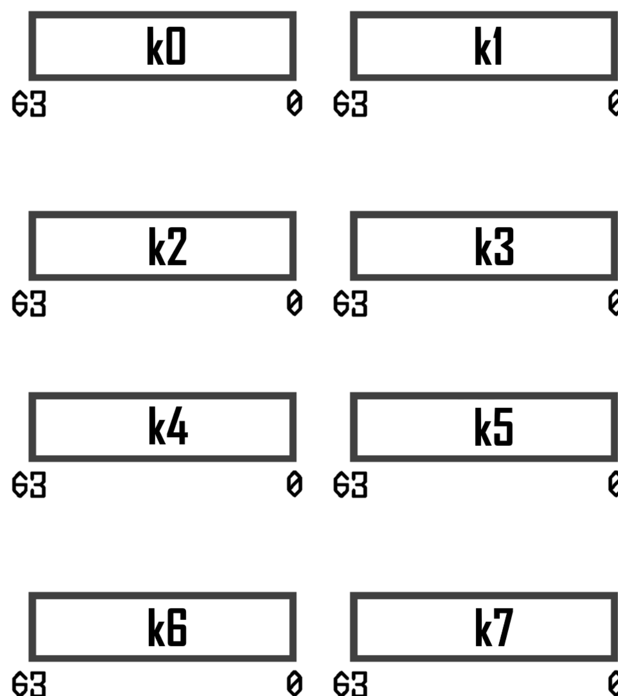
MMX (ang. MultiMedia eXtensions) to rozszerzenie zestawu instrukcji o rozkazy pozwalające operować na rejestrach 64-bitowych od MM0 do MM7.

Rejestry rozszerzeń SSE, AVX i AVX-512

SSE (ang. Streaming SIMD Extensions) to rozszerzenia zestawu instrukcji o rozkazy pozwalające operować na rejestrach 128-bitowych od XMM0 do XMM31.

AVX (ang. Advanced Vector eXtensions) to rozszerzenia zestawu instrukcji o rozkazy pozwalające operować na rejestrach 256-bitowych od YMM0 do YMM31.

AVX-512 (ang. Advanced Vector eXtensions) to rozszerzenie zestawu instrukcji o rozkazy pozwalające operować na rejestrach 512-bitowych od ZMM0 do ZMM31. Wprowadzane kolejne wersje rozszerzeń miały za podstawowy cel możliwość równoległego przetwarzania danych, czyli przeprowadzania operacji na wektorach o coraz to większych rozmiarach. Należy również dodać, że AVX zawiera rozkazy o trzech operandach, a AVX-512 pozwala stosować maski bitowe, które decydują o tym, które bity wyzerować, a które zostawić. Rejestry używane jako maski bitowe w rozkazach rozszerzenia AVX-512 są oznaczone od k0 do k7 i są 64-bitowe (Fotografia 12).



Fotografia 12. Rejestry używane jako maski bitowe w rozkazach rozszerzenia AVX-512

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Kolejność bajtów

Bajt jest najmniejszą możliwą do zaadresowania jednostką pamięci. Dane w pamięci operacyjnej, w rejestrach ogólnego przeznaczenia czy operandy rozkazów procesora są przedstawiane w formie ciągu bajtów.

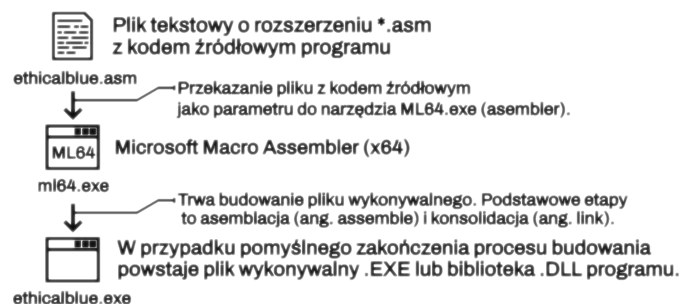
Urządzenia oparte o architekturę x86/x64 korzystają z kolejności bajtów nazywanej Little Endian (LE). Oznacza to, że najmłodszy bajt (określany też najmniej znaczącym bajtem, ang. low byte) jest zapisywany jako pierwszy, czyli przykładowe słowo maszynowe o wartości `0xDEAD` w postaci bajtów to `0xAD 0xDE`.

Istnieje też inna konwencja określająca kolejność bajtów — Big Endian (BE). W przypadku BE najstarszy bajt (określany też najbardziej znaczącym bajtem, ang. high byte) jest zapisywany jako pierwszy, czyli przykładowe słowo maszynowe o wartości `0xDEAD` w postaci bajtów to `0xDE 0xAD`.

Budowanie programu, czyli asemblacja i konsolidacja

W języku Asembler zamianę kodu źródłowego z postaci tekstowej na plik wykonywalny można podzielić na dwa podstawowe etapy takie jak asemblacja (ang. assemble) i konsolidacja (ang. link). Narzędzie asembler tworzy pliki binarne, ale to konsolidator (ang. linker) łączy te pliki z bibliotekami w celu stworzenia pliku wykonywalnego (*.exe).

Konsolidator (ang. linker) może być oddzielą aplikacją lub modułem wbudowanym w narzędzie asembler i uruchamianym przez odpowiedni parametr (Fotografia 13).



Fotografia 13. Narzędzie asembler tworzy plik wykonywalny z kodu źródłowego

Microsoft Macro Assembler (x64)

Narzędzie Microsoft Macro Assembler (x64) jest dołączone do środowiska programistycznego Microsoft Visual Studio. Pod adresem *visualstudio.microsoft.com* powinna być dostępna bezpłatna wersja nazywana Community.

Po zainstalowaniu środowiska Microsoft Visual Studio Community należy odnaleźć w systemie Windows skrót o nazwie x64 Native Tools Command Prompt for VS. Pozwoli on uruchomić specjalnie skonfigurowany Wiersz polecenia dla środowiska Visual Studio.

W celu ustawienia bieżącego katalogu na Pulpit w konsoli x64 Native Tools Command Prompt for VS należy wpisać polecenie przedstawione poniżej (Scriptum 3).

```
cd %USERPROFILE%\Desktop
```

Scriptum 3. Zmiana katalogu bieżącego (x64 Native Tools Command Prompt for VS)

W niniejszym eksperymencie pliki będą tworzone na Pulpicie, ponieważ w tej lokalizacji nie są wymagane prawa administratora do wykonywania operacji zapisu.

Teraz można dodatkowo uruchomić okno powłoki PowerShell, aby sprawnie utworzyć mały przykładowy program w języku Asembler.

Najpierw należy ustawić katalog bieżący na Pulpit poleceniem przedstawionym poniżej (Scriptum 4).

```
[Environment]::GetFolderPath('
   Desktop') | % { cd $_; }
```

Scriptum 4. Zmiana katalogu bieżącego (PowerShell)

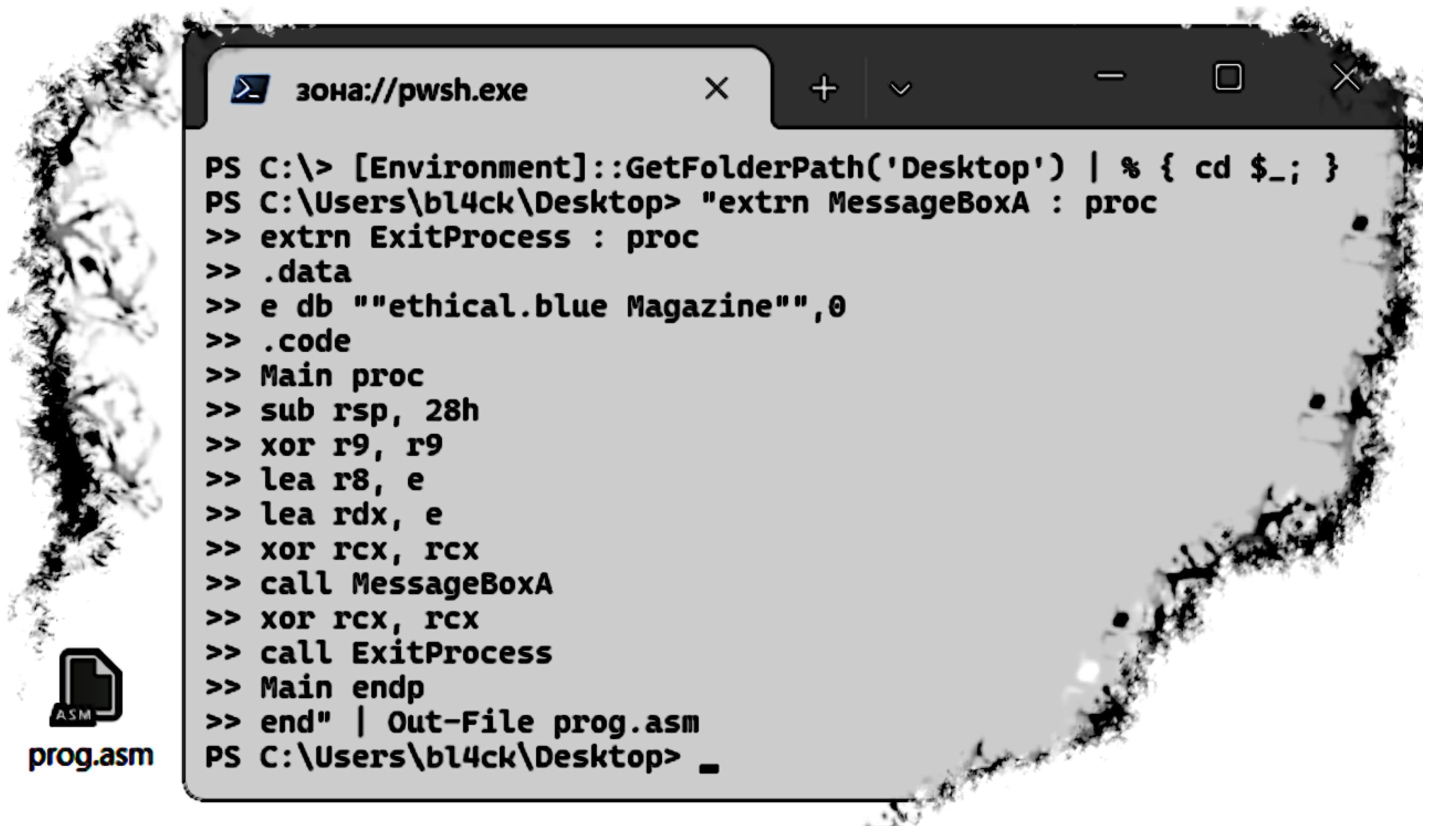
W przypadku małych programów plik z kodem źródłowym można utworzyć za pomocą polecenia PowerShell (Scriptum 5).

```
"extrn MessageBoxA : proc
extrn ExitProcess : proc
.data
e db ""ethical.blue Magazine"",0
.code
Main proc
sub rsp, 28h
xor r9, r9
lea r8, e
lea rdx, e
xor rcx, rcx
call MessageBoxA
xor rcx, rcx
call ExitProcess
Main endp
end" | Out-File prog.asm
```

Scriptum 5. Polecenie PowerShell do utworzenia w bieżącym katalogu pliku z kodem źródłowym

Rezultat polecenia przedstawiono dalej (Fotografia 14).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.



Fotografia 14. Utworzenie pliku z kodem źródłowym w języku Asembler za pomocą PowerShell

Po ustawieniu katalogu bieżącego na Pulpit i przesłaniu przez potok (|) kodu źródłowego do polecenia `Out-File` (Fotografia 14) można przejść do konsoli *x64 Native Tools Command Prompt for VS*, aby przeprowadzić budowanie pliku wykonywalnego (Fotografia 15).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Przykładowe wywołanie programu `ml64.exe` przedstawiono poniżej (Scriptum 6).

```
ml64.exe /quiet prog.asm /link /subsystem:windows /defaultlib:kernel32.
   lib /defaultlib:user32.lib /entry:Main /out:prog.exe
```

Scriptum 6. Przykładowe polecenie wywołujące ml64.exe (x64 Native Tools Command Prompt for VS)

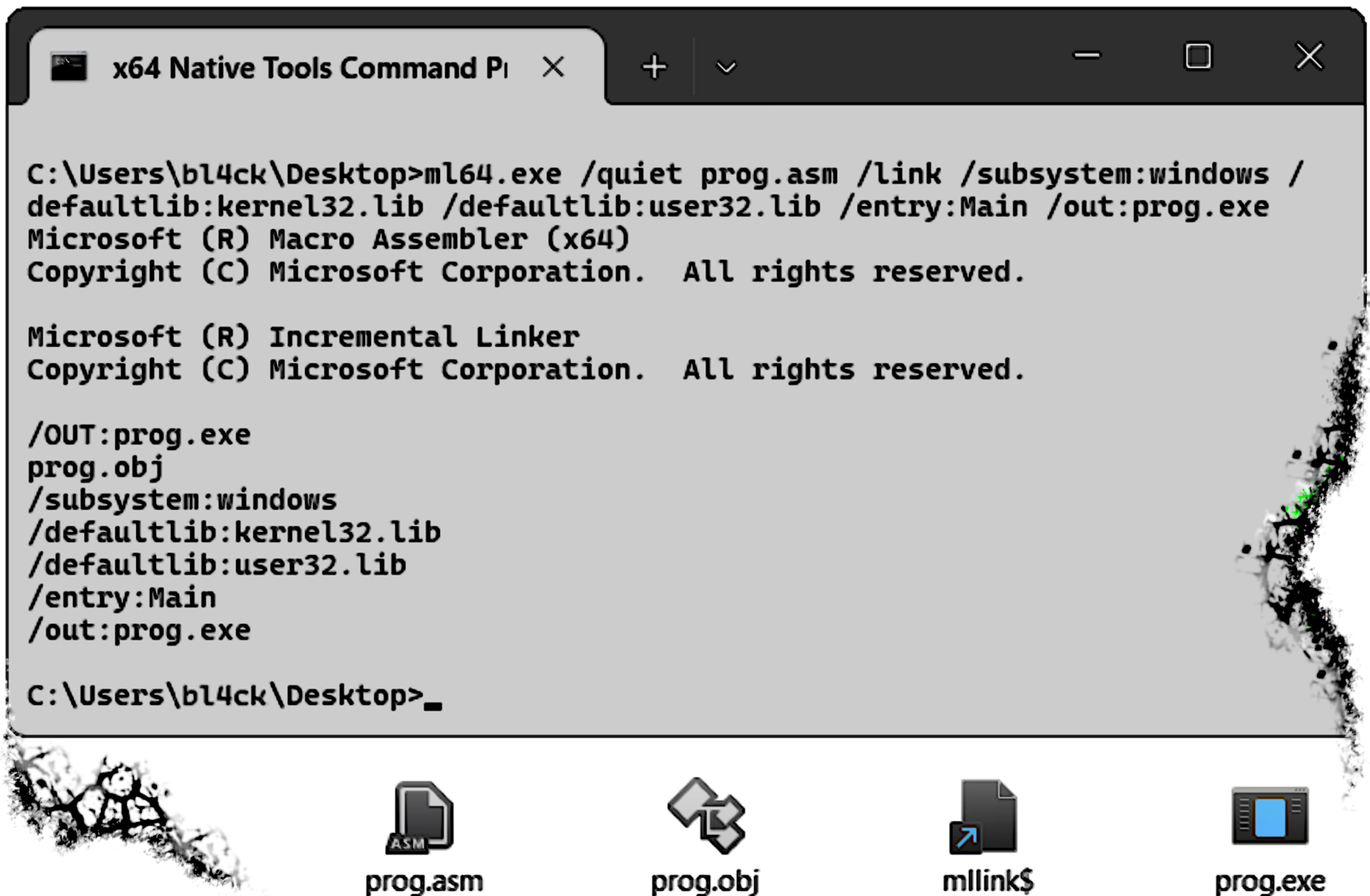
Opis parametrów wywołania zaprezentowano poniżej.

- `ml64.exe` – Microsoft Macro Assembler (x64)
- `/quiet` – mniej wyświetlanych szczegółów o procesie budowania.
- `prog.asm` – nazwa pliku z kodem źródłowym w bieżącym katalogu.
- `/link` – ustawienia konsolidatora.
- `/subsystem:windows` – program z graficznym interfejsem użytkownika (GUI).
- `/defaultlib:kernel32.lib` – dołączenie biblioteki statycznej o nazwie `kernel32.lib` w której jest funkcja `ExitProcess`.
- `/defaultlib:user32.lib` – dołączenie biblioteki statycznej o nazwie `user32.lib` w której jest funkcja `MessageBoxA`.
- `/entry:Main` – punkt wejścia (ang. entry point) to procedura o nadanej nazwie `Main`.
- `/out:prog.exe` – wymyślona nazwa pliku wykonywalnego, który powstanie w procesie budowania.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

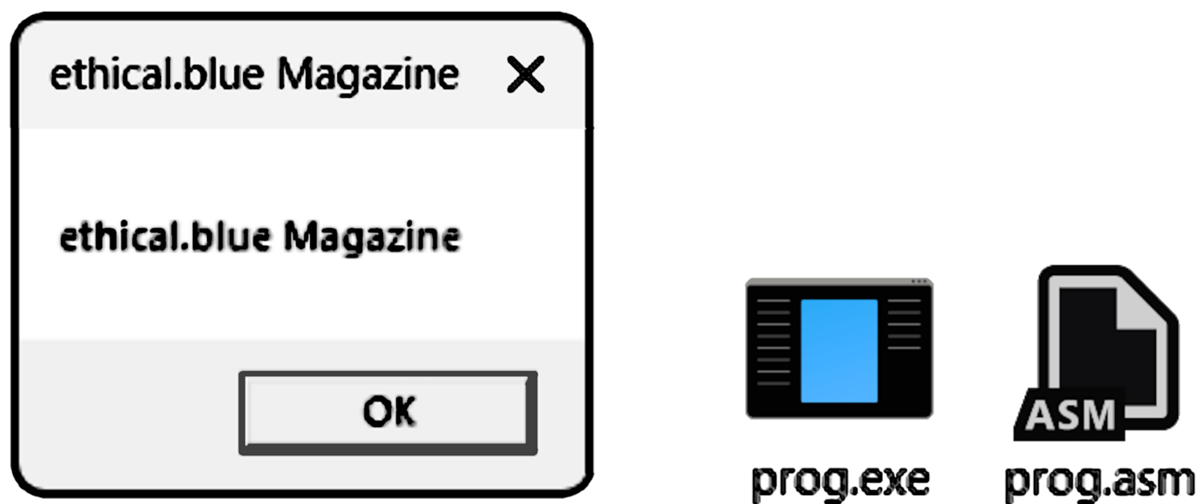
Jeśli budowanie pliku wykonywalnego zakończy się sukcesem, to w katalogu bieżącym powinien powstać nowy plik z rozszerzeniem *.exe (Fotografia 15).

Mogą pojawić się też pozostałości po procesie asemblacji i konsolidacji takie jak plik z rozszerzeniem *.obj. Nie są one potrzebne do działania programu i można je usunąć.



Fotografia 15. Przykładowe wywołanie ml64.exe – Microsoft Macro Assembler (x64)

Po uruchomieniu zbudowanego wcześniej programu na ekranie powinno pojawić się okno komunikatu z treścią *ethical.blue Magazine* (Fotografia 16). Plik wykonywalny `prog.exe` dzięki zastosowaniu języka Asembler ma rozmiar jedynie 2,50 KB (bajtów: 2 560).



Fotografia 16. Program w języku Asembler wyświetlający okno funkcją MessageBox

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

— Dobra, Morph. Pora przyjrzeć się przykładowi przedstawionemu na Scriptum 7. — oznajmił pracownik laboratorium.

```
extrn MessageBoxA : proc
extrn ExitProcess : proc

.data
  e db "ethical.blue Magazine", 0

.code
  Main proc
    sub rsp, 28h
    xor r9, r9
    lea r8, e
    lea rdx, e
    xor rcx, rcx
    call MessageBoxA
    xor rcx, rcx
    call ExitProcess
  Main endp
end
```

Scriptum 7. Przykładowy program w języku Asembler (ml64.exe)

Słowo kluczowe `extrn` to dyrektywa informująca o odwołaniu do funkcji zewnętrznej, a w tym przypadku Windows API.

Dokumentacja dyrektyw MASM x64 powinna być dostępna pod adresem

`learn.microsoft.com/en-us/cpp/assembler/masm/directives-reference`

Dawniej dane umieszczało się na końcu kodu (`.code`). Jednak, aby nie zostały one błędnie zinterpretowane jako instrukcje i wykonane, to lepiej umieszczać je w sekcji z danymi `.data`. Zmienna o nazwie `e` to ciąg znaków (bajtów) zakończony bajtem zerowym informującym o końcu napisu. Dyrektywa `db` to w dosłownym tłumaczeniu *define byte*. Natomiast `sub`, `xor`, `lea` oraz `call` to rozkazy procesora x64 w postaci tekstowej, czyli mnemoniki.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

W dokumentacji pod adresem `learn.microsoft.com` można znaleźć, że funkcja `MessageBoxA` posiada składnię, którą przedstawiono poniżej.

```
int MessageBoxA(
  HWND   hWnd,
  LPCSTR lpText,
  LPCSTR lpCaption,
  UINT   uType
);
```

Wartość zwracana to liczba całkowita (`int`) i zawiera informacje o tym co kliknął użytkownik w oknie dialogowym. Funkcja `MessageBoxA` przyjmuje cztery parametry.

- `hWnd` – uchwyt okna nadrzędnego lub zero.
- `lpText` – adres do napisu z treścią okna dialogowego.
- `lpCaption` – adres do napisu z tytułem okna dialogowego.
- `uType` – ustawienia ikon i dostępnych przycisków w oknie dialogowym.

W językach wysokiego poziomu abstrakcji takich jak C++ czy C# wartości parametrów można by było przekazać w operatorze wywołania (nawiasy okrągłe), rozdzielając je przecinkiem.

Schodząc niżej w poziomie abstrakcji przekazywane parametry lądują w specjalnych rejestrach. Jeśli parametry funkcji są cztery lub mniej, to przekazuje się je przez rejestry `RCX/RDX/R8/R9` (dla wartości całkowitoliczbowych). Jeśli parametrów jest więcej niż cztery, to pozostałe przekazuje się przez umieszczenie ich na stosie programu.

Niezwykle ważne jest, aby zgodnie z konwencją wywoływania procedur x64 zarezerwować na stosie programu tak zwane *shadow space* (miejsce na przekazywane parametry funkcji) oraz, aby stos programu był wyrównany do 16 bajtów. W przykładowym programie parametrów jest cztery, ale na stosie jest także wcześniej odłożony adres powrotu do Windows po zakończeniu programu. Na stosie jest zatem pięć wartości o rozmiarze 8 bajtów: $(4 \times 8) + 8 = 40$. Stos wtedy nie jest wyrównany do 16 bajtów (liczba 40 nie dzieli się na 16 bez reszty). W celu wyrównania należy zarezerwować dodatkowe 8 bajtów, czyli będzie: $(4 \times 8) + 8 + 8 = 32 + 8 + 8 = 48$. Teraz 48 jest podzielne przez 16 bez reszty, czyli stos jest wyrównany. Miejsce na stosie można zarezerwować np. rozkazem odejmującym określoną wartość od rejestru wskaźnika stosu, czyli `sub rsp, 28h`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

— Jeszcze raz. Nie rozumiem. — westchnął Morph.

Stos jest wyrównywany rozkazem `sub rsp, 28h`. Wartość `28h` szesnastkowo to w systemie dziesiętnym 40. Składają się na nią cztery parametry (każdy o rozmiarze 8 bajtów) i dodatkowe wyrównanie (o rozmiarze 8 bajtów), które my uznaliśmy za potrzebne. Adres powrotu na stosie jest domyślnie i też ma rozmiar 8 bajtów, czyli stos jest wyrównany. Program kończy działanie poprzez wywołanie funkcji `ExitProcess`, która posiada tylko jeden parametr — kod wyjścia. Przyjmuje się, że program zakończony z sukcesem zwraca wartość zero. Należy tutaj zauważyć, że wspomniane wcześniej miejsce *shadow space* nie musi być zwalniane, ponieważ funkcja `ExitProcess` zamyka program poprzez zakończenie procesu. Jeśli wyjście z funkcji głównej *Main* i powrót do systemu Windows byłby realizowany rozkazem `RET` (ang. return), to należy zwolnić zarezerwowane wcześniej miejsce na stosie instrukcją odwrotną do odejmowania, czyli dodaniem wartości `28h` do wskaźnika wierzchołka stosu rozkazem `add` (Scriptum 8).

```
    add rsp, 28h
    ret
  Main endp
end
```

Scriptum 8. Przykładowe zwolnienie shadow space

Tworzenie projektu dla języka Asembler w środowisku Microsoft Visual Studio

Utworzenie programu w języku Asembler x64 za pomocą środowiska Microsoft Visual Studio polega na modyfikacji projektu dla technologii Visual C++ — należy wykonać kilka prostych czynności, które pozwolą na korzystanie z asemblera MASM x64.

Po uruchomieniu środowiska należy wybrać z górnego menu `File / New / Project...` (Fotografia 17).

Fotografia 17. Microsoft Visual Studio. Nowy projekt

Uruchomienie kreatora aplikacji dla Windows Desktop jest jednym ze sposobów utworzenia projektu Visual C++ (Fotografia 18).

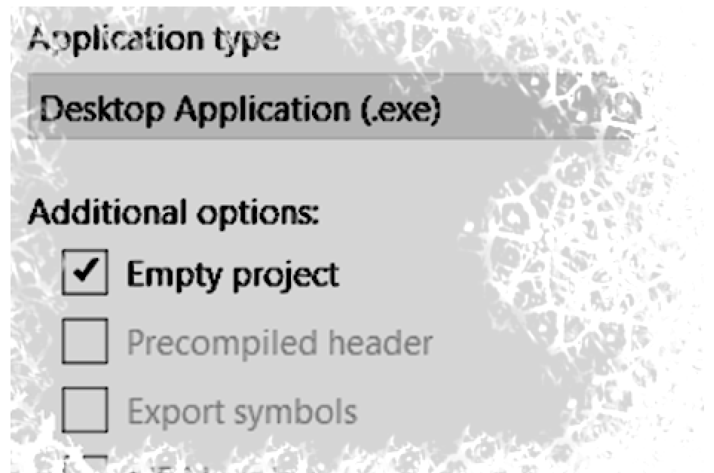
Fotografia 18. Microsoft Visual Studio. Kreator aplikacji dla Windows Desktop

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Od wpisanej nazwy projektu będzie między innymi zależała nazwa pliku wykonywalnego *.exe czy *.dll (Fotografia 19).

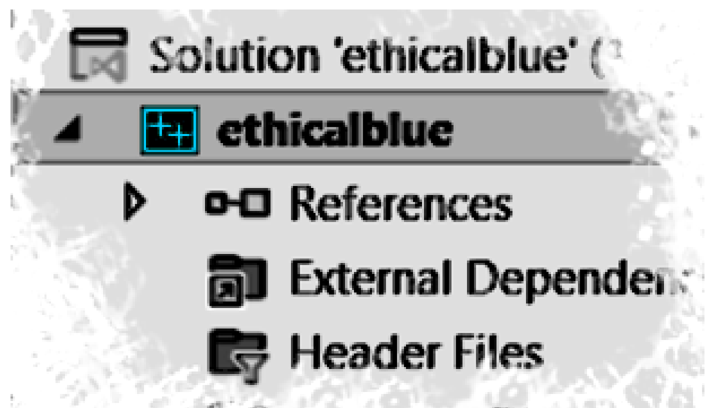
Fotografia 19. Microsoft Visual Studio. Nazwa projektu oraz wybranie lokalizacji dla plików rozwiązania

W celu wyłączenia generowania kodu początkowego w Visual C++ należy wybrać `Empty project`. Wybranie Console Application utworzy domyślnie program z widoczną konsolą tekstową. Natomiast wybranie Desktop Application utworzy program dla Windows bez konsoli tekstowej. Możliwe jest też tworzenie bibliotek (Fotografia 20).



Fotografia 20. Microsoft Visual Studio. Pusty projekt aplikacji typu Desktop

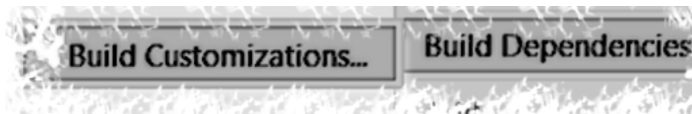
W oknie dokowanym Eksploratora rozwiązań (ang. Solution Explorer) należy kliknąć prawym przyciskiem myszy na nazwie projektu (Fotografia 21).



Fotografia 21. Microsoft Visual Studio. Przykładowy projekt o nazwie ethicalblue

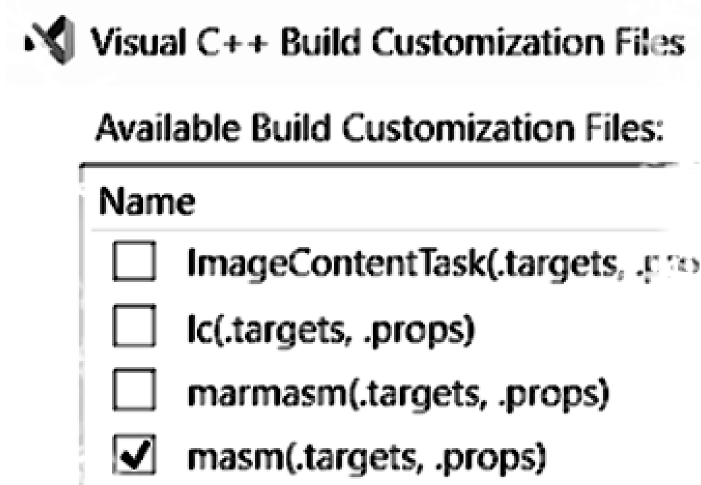
© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dalej należy włączyć obsługę Microsoft Macro Assembler poprzez kliknięcie prawym przyciskiem myszy na gałęzi projektu w oknie Solution Explorer i wybranie `Build Dependencies / Build Customizations` (Fotografia 22).



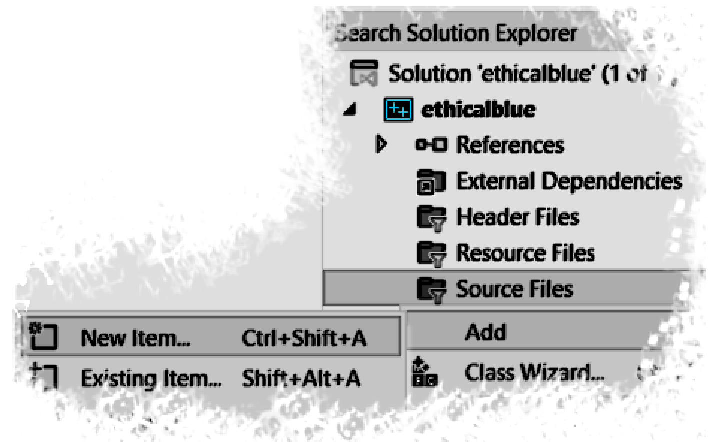
Fotografia 22. Microsoft Visual Studio. Dostosowanie budowania projektu przez wybranie Build Customizations

Zaznaczenie elementu `masm` pozwoli na korzystanie z narzędzia Microsoft Macro Assembler w projekcie (Fotografia 23).



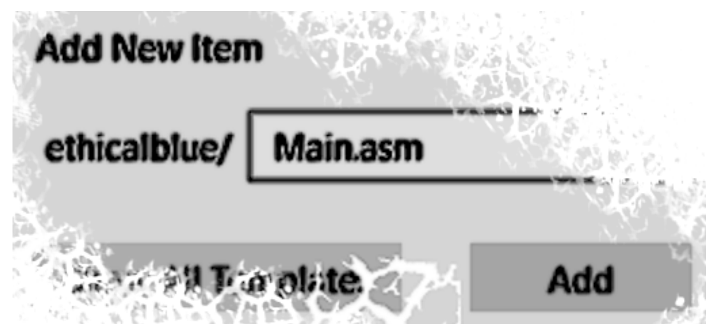
Fotografia 23. Microsoft Visual Studio. Microsoft Macro Assembler (masm)

Plik z kodem źródłowym można dodać poprzez kliknięcie prawym przyciskiem myszy na `Source Files` w oknie `Solution Explorer` i wybranie `Add / New Item` (Fotografia 24).



Fotografia 24. Microsoft Visual Studio. Dodawanie nowego elementu do projektu

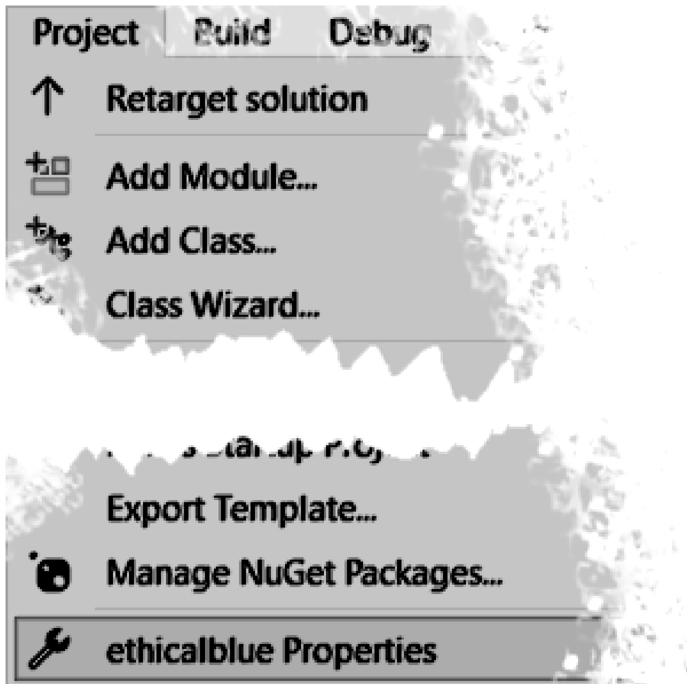
W oknie dialogowym należy podać nazwę pliku źródłowego (np. Main) z rozszerzeniem .asm (Fotografia 25).



Fotografia 25. Microsoft Visual Studio. Nazwa pliku źródłowego (.asm)

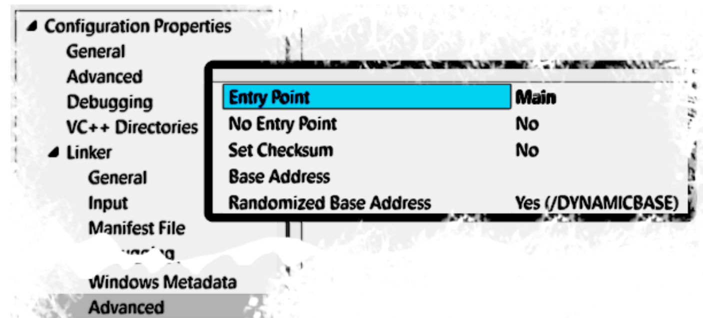
© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Właściwości projektu można dostosować poprzez wybranie `Project / Properties` (Fotografia 26).



Fotografia 26. Microsoft Visual Studio. Właściwości projektu

W zaawansowanych ustawieniach konsolidatora (`Project / ethicalblue Properties / Linker / Advanced / Entry Point`) należy wpisać odpowiednią nazwę procedury dla punktu wejścia programu (Fotografia 27).



Fotografia 27. Microsoft Visual Studio. Ustawienie punktu wejścia

Należy pamiętać, aby do pliku Main.asm wprowadzić kod źródłowy programu w języku Asembler. Można użyć zaprezentowany wcześniej przykład wyświetlający okno dialogowe za pomocą funkcji `MessageBoxA`. W celu zbudowania projektu i uruchomienia w trybie debugowania należy kliknąć przycisk Local Windows Debugger (Fotografia 28).

Fotografia 28. Microsoft Visual Studio. Uruchomienie projektu w trybie debugowania

Program można zakończyć klikając kwadratowy przycisk zatrzymujący (Fotografia 29).

Fotografia 29. Microsoft Visual Studio. Zatrzymanie debugowania

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Podstawy składni Microsoft Macro Assembler (MASM x64)

Podstawowe elementy składni Microsoft Macro Assembler (MASM x64, ML64.exe).

Dane lokalne (LOCAL)

Dane lokalne tworzone dyrektywą `local` przechowywane są na stosie i istnieją na czas wykonywania procedury w której są zarezerwowane.

Przykład Scriptum 9 zawiera dane lokalne o nazwach `var1` oraz `var2`. Odwoływanie się do danych lokalnych następuje za pomocą nazwy nadanej im przez nas. Na przykład rozkaz `mov var1, eax` kopiuje wartość z rejestru `EAX` (32-bity) do danej o nazwie `var1` (rozmiar 32-bity), a rozkaz `mov var2, rax` kopiuje wartość z rejestru `RAX` (64-bity) do danej o nazwie `var2`.

Tekst poprzedzony znakiem średnika (`;`) to nasze komentarze do kodu, które są pomijane przez narzędzia budujące program do pliku wykonywalnego.

Istotną kwestią jest prolog i epilog procedury. Składnia wyższego poziomu abstrakcji (dyrektywy) podczas budowania pliku wykonywalnego zamieniana jest na określone instrukcje kodu maszynowego.

```
.code
Main proc
  local var1:dword, var2:qword

  mov rax, 0ADD1C7EDADD1C7EDh

  ;var1 = ADD1C7EDh
  mov var1, eax

  ;var2 = 0ADD1C7EDADD1C7EDh
  mov var2, rax

  ret
Main endp
end
```

Scriptum 9. Przykład transferu wartości między rejestrem a zmienną lokalną (ml64.exe)

W wielu przypadkach, jeśli kod nie jest chroniony, to można zobaczyć jakie instrukcje kodu maszynowego zostały wygenerowane z dyrektyw i kodu źródłowego. Pora zbudować przykładowy kod z Scriptum 9 do pliku wykonywalnego *.exe. Polecenie zostało przedstawione wcześniej, poniżej przypomnienie (Scriptum 10).

```
ml64.exe /quiet prog.asm /link /
   subsystem:windows /defaultlib:
   kernel32.lib /defaultlib:user32.
   lib /entry:Main /out:prog.exe
```

Scriptum 10. Polecenie budowania prog.asm do prog.exe za pomocą ml64.exe (x64 Native Tools Command Prompt for VS)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Ze sklepu Microsoft Store można zainstalować bezpłatne narzędzie WinDbg, które jest debugger'em z opcją deasemblacji, czyli pozwala m.in. na wykonywanie i analizę programu instrukcja po instrukcji, weryfikowanie stanu rejestrów czy odtworzenie rozkazów w języku Asembler z bajtów kodu maszynowego (Fotografia 30).

Instrukcje generowane na początku procedury określane są terminem prolog. Następuje tutaj odłożenie na stos nieulotnego rejestru RBP (`push rbp`), a wartość wskaźnika wierzchołka stosu RSP jest kopiowana do rejestru RBP (`mov rbp, rsp`). Natomiast rozkaz `add rsp, 0FFFFFFFFFFFFFFF0h` rezerwuje na stosie miejsce na dane lokalne. Zwracająca uwagę wygenerowana przez narzędzie instrukcja `add rsp, 0FFFFFFFFFFFFFFF0h` w celu zwiększenia przejrzystości kodu mogłaby być zastąpiona przez `sub rsp, 0Fh`, czyli zamiast dodawać (rozkaz `add`) do RSP wartość ujemną -16, to można po prostu odjąć (rozkaz `sub`) od RSP wartość 16.

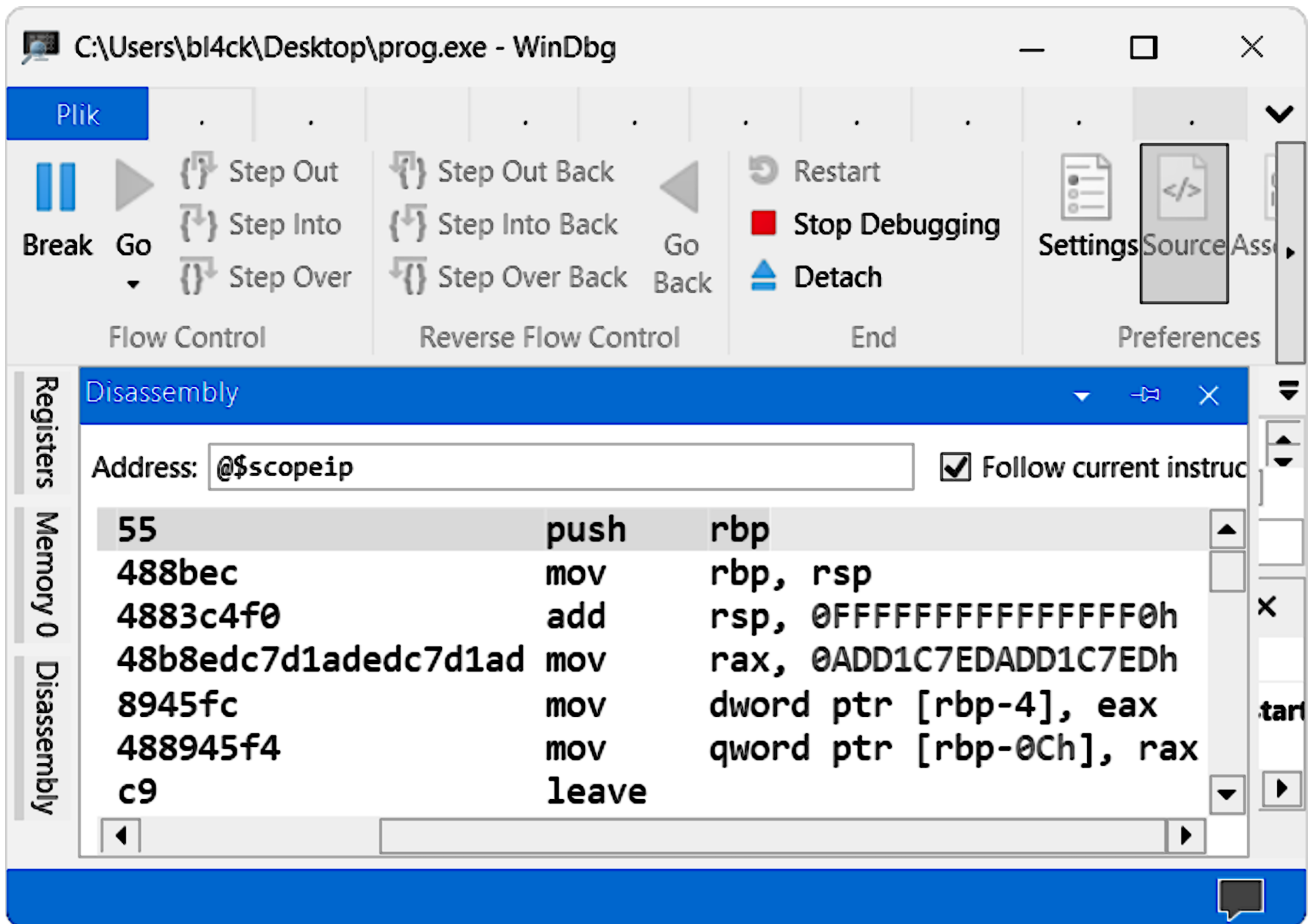
Instrukcje usuwające dane lokalne ze stosu umieszczane na końcu procedury nazywane są terminem epilog. W tym przypadku została wygenerowana instrukcja `leave`. Rozkaz `leave` odpowiada `mov rsp, rbp` oraz `pop rbp`. Niektóre narzędzia budujące mogą stosować `leave`, a inne alternatywną, dłuższą, bardziej niskopoziomową formę.

Za pomocą rozkazu `mov rsp, rbp` przywracana jest początkowa wartość wskaźnika stosu RSP (przed wykonaniem `add` wskaźnik stosu RSP został zachowany w rejestrze RBP przez instrukcję `mov rbp, rsp` w prologu). Rozkaz `pop rbp` w epilogu procedury zdejmuje ze stosu ostatnio odłożoną wartość i umieszcza ją w rejestrze RBP. Następuje tu po prostu przywrócenie wartości rejestrowi RBP, gdyż zgodnie z konwencją wywoływania procedur, rejestr RBP jest nieulotny (ang. non-volatile) i należy przywrócić jego początkową wartość przed wyjściem (`ret`, ang. return) z procedury, która go modyfikuje, używa.

Warty zanotowania jest także sposób odwoływania się do danych lokalnych. Nazwy `var1` czy `var2` obowiązujące w kodzie źródłowym są zamieniane na odwołania względem wierzchołka stosu, czyli rozkaz `mov qword ptr [rbp-0Ch], rax` oznacza co przedstawiono poniżej.

- Odwołanie do wartości o rozmiarze poczwórnego słowa (qword ptr ..., ang. qword pointer).
- Adres wartości to aktualny wskaźnik wierzchołka stosu (RSP), zachowany przez instrukcje prologu procedury w rejestrze RBP, zmniejszony o wartość 0Ch.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.



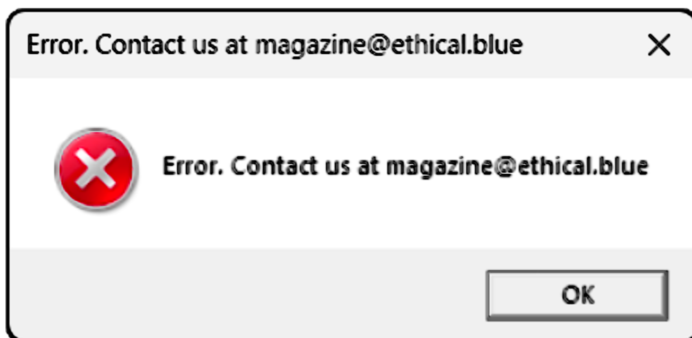
Fotografia 30. Odzyskanie przykładowego kodu w Asemblerze z pliku wykonywalnego *.exe za pomocą narzędzia WinDbg

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Stałe

Sekcja .const może zawierać wartości stałe, czyli nazwy do których za pomocą dyrektywy equ (ang. equal) przypisano wartość liczbową lub wyrażenie. Pozwala to uniknąć w kodzie źródłowym tzw. *magic numbers*, czyli wartości liczbowych, których znaczenie może być niejasne dla osoby czytającej. Warto zanotować, że stałą może być nie tylko wartość liczbowa, ale też napis w postaci ciągu bajtów.

Przykład ze Scriptum 11 przedstawia, że wartość 10h jako czwarty parametr funkcji MessageBoxA może wymagać sięgnięcia do dokumentacji w celu sprawdzenia co ta liczba oznacza. Dlatego dobrym zwyczajem jest zastosowanie stałej MB_ICONERROR z nazwy której możliwe jest odczytanie, że wartość ta ustawia ikonę błędu dla okna dialogowego (Fotografia 31).



Fotografia 31. Przykładowe okno dialogowe MessageBox z ikoną błędu

```
extrn ExitProcess : proc
extrn MessageBoxA : proc

.const
   NULL equ 0
   MB_ICONERROR equ 10h

.data
  szText db "Error. "
   db "Contact us at "
   db "magazine@ethical.blue"
   db NULL

.code
Main proc
  sub rsp, 28h

  mov r9, MB_ICONERROR
  lea r8, szText
  lea rdx, szText
  xor rcx, rcx
  call MessageBoxA

  xor rcx, rcx
  call ExitProcess
Main endp
end
```

Scriptum 11. Przykład dla wartości stałych i dyrektywy equ (ml64.exe)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dane o rozmiarze bajta

Dane z nadaną wartością początkową definiuje się w sekcji `.data`. Za pomocą dyrektywy `byte` lub w skrócie `db` (ang. define byte) możliwe jest stworzenie danej o rozmiarze bajta lub danych w postaci ciągu bajtów. Bajt ze znakiem można zdefiniować za pomocą dyrektywy `sbyte` (ang. signed byte). Nic nie stoi też na przeszkodzie, aby zdefiniować kilobajt np. `kilobyte db 1024 dup(0)`.

Nie należy zapominać, że ciągi znaków (napisy) to również bajty. Dlatego możliwe jest wstawianie bajtów wewnątrz napisu. Przykładem może być umieszczenie znaku nowej linii obowiązującego w systemie Windows, czyli dwóch bajtów o wartościach trzynaście (`0Dh`) oraz dziesięć (`0Ah`). Przykład poniżej (Scriptum 12).

```
szText db "ethical.blue"
       db 0Dh, 0Ah, "Magazine", 0
```

Scriptum 12. Przykładowy napis ze znakiem nowej linii (ml64.exe)

Dane o rozmiarze słowa maszynowego

Dane o rozmiarze słowa maszynowego (16 bitów) można definiować dyrektywami `WORD` lub w skrócie `DW` (ang. define word). Dla typów danych ze znakiem można użyć dyrektywy `SWORD` (ang. signed word). Przykład poniżej (Scriptum 13).

```
.data
  wVar1 dw 0DEADh
  wVar2 word 0BEEFh
  wSigned sword -1
```

Scriptum 13. Przykłady zmiennych o rozmiarze słowa maszynowego (ml64.exe)

Dane o rozmiarze podwójnego słowa maszynowego

Dane o rozmiarze podwójnego słowa maszynowego (32 bity) można definiować dyrektywami `DWORD` lub w skrócie `DD` (ang. define dword). Dla typów danych ze znakiem można użyć dyrektywy `SDWORD` (ang. signed dword). Przykład poniżej (Scriptum 14).

```
.data
  dwVar1 dd 0ADD1C7EDh
  dwVar2 dword 0C0FFEEh
  dwSigned sdword -1
```

Scriptum 14. Przykłady zmiennych o rozmiarze podwójnego słowa maszynowego (ml64.exe)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dane o rozmiarze poczwórnego słowa maszynowego

Dane o rozmiarze poczwórnego słowa maszynowego (64 bity) można definiować dyrektywami QWORD lub w skrócie DQ (ang. define qword). Dla typów danych ze znakiem można użyć dyrektywy SQWORD (ang. signed qword). Przykład poniżej (Scriptum 15).

```
.data
  dqVar1 dq 0ADD1C7EDADD1C7EDh
  dqVar2 qword 0ADD1C7EDADD1C7EDh
  dqSigned sqword -1
```

Scriptum 15. Przykłady zmiennych o rozmiarze poczwórnego słowa maszynowego (ml64.exe)

Dane typu MMWORD (64 bity)

Typ danych do pracy z technologią MultiMedia eXtensions. Przykład poniżej (Scriptum 16).

```
.data
  mem64 mmword 1.25
```

Scriptum 16. Przykłady zmiennych typu mmword (ml64.exe)

Dane typu XMMWORD (128 bitów)

Typ danych do pracy z instrukcjami rozszerzeń Streaming SIMD Extensions (SSE). Niniejsza dyrektywa wymaga nieco bardziej szczegółowego omówienia. Podwójne słowo maszynowe ma rozmiar 32 bity, czyli wektor czterech takich wartości ma rozmiar 128 bitów (4×32).

Wewnątrz procedury Main następuje wczytanie adresu danej mem128 do rejestru danych RDX, aby dalej przekazać te wartości do rejestru XMM0 za pomocą rozkazu movdqu (Scriptum 17).

```
.data
  mem128 dword 1.0, 2.0, 3.0, 4.0

.code
Main proc
  mov rdx, offset mem128
  movdqu xmm0, xmmword ptr [rdx]

  ret
Main endp
end
```

Scriptum 17. Przykład transferu wektora liczb zmiennoprzecinkowych między pamięcią a rejestrem XMM0 (ml64.exe)

Dane typu YMMWORD (256 bitów)

Typ danych do pracy z instrukcjami rozszerzeń Advanced Vector eXtensions (AVX). Mechanizm działania jest podobny jak w poprzednim przykładzie (Scriptum 17). Jednak zamiast mnemonika `movdqu` zastosowano `vmovdqu`, wektor wartości jest rozmiaru 256 bitów, a rejestr to nie `xmm0`, tylko większy `ymm0` (Scriptum 18).

```
.data
  mem256 dword 1.0, 2.0, 3.0
         dword 4.0, 5.0, 6.0
         dword 7.0, 8.0

.code
Main proc
  mov rdx, offset mem256
  vmovdqu ymm0, ymmword ptr [rdx]

  ret
Main endp
end
```

Scriptum 18. Przykład transferu wektora liczb zmiennoprzecinkowych między pamięcią a rejestrem YMM0 (ml64.exe)

Dane typu ZMMWORD (512 bitów)

Typ danych do pracy z instrukcjami rozszerzenia AVX-512 (Advanced Vector eXtensions). Scriptum 19 zawiera przykład dla dyrektywy `zmmword`.

```
.data
  mem512 dword 1.0, 2.0, 3.0
         dword 4.0, 5.0, 6.0
         dword 7.0, 8.0, 9.0
         dword 10.0, 11.0, 12.0
         dword 13.0, 14.0, 15.0
         dword 16.0

.code
Main proc
  mov rdx, offset mem512
  vmovdqu64 zmm0, \
   zmmword ptr [rdx]

  ret
Main endp
end
```

Scriptum 19. Przykład transferu wektora liczb zmiennoprzecinkowych między pamięcią a rejestrem ZMM0 (ml64.exe)

Jeśli procesor nie obsługuje prezentowanego rozszerzenia, to podczas wykonania wystąpi wyjątek nieprawidłowej instrukcji (ang. illegal instruction).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Aliasy, czyli nowe nazwy dla istniejących typów

Utworzenie nowych nazw (aliasów) dla wbudowanych typów danych może zwiększyć przejrzystość kodu źródłowego. Przykładem może być stworzenie typu logicznego, który przyjmuje jedną z dwóch wartości: prawda lub fałsz. Tego rodzaju nowy typ danych można utworzyć dyrektywą `typedef`. Dla typu logicznego bazującego na podwójnym słowie maszynowym (32 bity) zapis wygląda następująco: `bool typedef dword`. Wartościami dla typu logicznego mogą być stałe np. `false equ 0` oraz `true equ 1`.

Teraz dzięki tym zabiegom możliwe jest utworzenie danej nowego typu. Pamięć o nazwie `isValid` może teraz być traktowana podobnie do typu logicznego `bool` znanego z języków takich jak C++ czy C# (Scriptum 20).

```
bool typedef dword

.const
  false equ 0
  true equ 1

.data
  ;dane typu bool
  isValid bool false
```

Scriptum 20. Utworzenie w języku Asembler typu logicznego bool znanego z języków C++ i C# (ml64.exe)

Etykiety (ang. labels)

Etykietami można oznaczać miejsca w kodzie źródłowym do których przekazywane jest wykonanie programu. Dzięki temu mechanizmowi możliwe jest wielokrotne wykonywanie oznaczonych instrukcji (pętla) czy też pominięcie fragmentu kodu poprzez skok za ten kod. Etykiety anonimowe oznaczane są znakami `@@:`. Natomiast etykiety nazwane mogą mieć nadaną nazwę wymyśloną przez piszącego kod w stylu `_exit`, `_next`, `_skip` czy inne słowa opisujące znaczenie określonego miejsca w kodzie.

Na przykład instrukcja `JNE @b` (`@b`, czyli ang. backward) wykonuje warunkowy (jeśli nierówne, ang. jump if not equal) skok wstecz do najbliższej anonimowej etykiety. Podobnie jest z przejściem naprzód skokiem bezwarunkowym `JMP @f` (ang. jump forward). W przypadku etykiety nazwanej, zamiast `@b` (ang. backward) czy `@f` (ang. forward) należy podać jej nazwę jako operand rozkazu zmieniającego przepływ wykonania (Scriptum 21).

```
.code
Main proc
    mov rax, 3 ;rejestr RAX przyjmuje wartość trzy
@@:
    dec rax ;zmniejsz wartość rejestru RAX o jeden
    test rax, rax ;sprawdź czy rejestr RAX równy zero
    jne @b ;jeśli nie, to skocz wstecz do anonimowej etykiety @@

    mov rax, 9 ;rejestr RAX przyjmuje wartość dziewięć
    jmp @f ;bezwarunkowy skok w przód do anonimowej etykiety @@
    mov rax, 5 ;to się nie wykona (jest przeskoczone)
@@:
    sub rax, rax ;wyzerowanie rejestru RAX
    jmp _exit ;bezwarunkowy skok do etykiety nazwanej _exit

_exit:
    ret ;powrót do systemu Windows
Main endp
end
```

Scriptum 21. Przykład tworzenia etykiet nazwanych i anonimowych oraz skoki warunkowe i bezwarunkowe do miejsc w kodzie programu (ml64.exe, Microsoft Visual Studio)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Konwencja wywoływania stdcall

Wywołanie funkcji w Asemblerze x64 nazywane też wywołaniem podprogramu przenosi sterowanie do innego miejsca w kodzie. Gdy blok kodu określany funkcją się wykona, to następuje powrót, który jest możliwy poprzez odłożony wcześniej na stosie programu adres powrotny. Funkcje wywołuje się instrukcją procesora `call`. To ona odkłada na stos wspomniany wcześniej adres powrotny i przekazuje kontrolę do wywoływanej procedury.

Nie ma jednego uniwersalnego sposobu na wywołanie funkcji. Zależne jest to od architektury, a to jak działa wywołanie i związane z nim operacje określają konwencje wywołania (ang. calling conventions).

Przypomnijmy sobie, że w Microsoft Macro Assembler (MASM) dla architektury x86-32 przeważnie korzysta się z konwencji *stdcall*, która jest domyślna dla API systemu Windows (Scriptum 22 i 23). Wyczyszczenie stosu programu jest obowiązkiem funkcji, która jest wywoływana (ang. callee), czyli korzystający z takiej funkcji ma „z głowy" czyszczenie stosu. Argumenty (nazywane też parametrami) przekazywane są przez stos „od końca", czyli od prawej do lewej strony.

Jeśli funkcja zwraca jakiś rezultat, to znajdzie się on w rejestrze akumulatora `EAX`. Niektóre funkcje, gdy wynik jest większy niż 32 bity zwracają wynik w parze rejestrów `EDX:EAX`. W konwencji `stdcall`, jeśli chcemy modyfikować wartości rejestrów `ESI`, `EDI`, `EBP` i `EBX`, to powinniśmy zachować ich wartości np. na stosie, a następnie je przywrócić przed powrotem do Windows.

```
hFile = CreateFile(
  szFileName,
  GENERIC_WRITE,
  0, 0,
  CREATE_NEW,
  FILE_ATTRIBUTE_NORMAL, 0
);
```

Scriptum 22. Wywołanie funkcji CreateFile (Microsoft Visual C++)

```
push 0
push FILE_ATTRIBUTE_NORMAL
push CREATE_NEW
push 0
push 0
push GENERIC_WRITE
push offset szFileName
call CreateFileA
mov hFile, eax
```

Scriptum 23. Wywołanie funkcji CreateFileA (Microsoft Macro Assembler 32-bit)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Konwencja wywoływania funkcji Microsoft x64

Asembler MASM x64 (ml64.exe) dla architektury x86-64 (w skrócie x64) korzysta z konwencji wywoływania funkcji nazwanej Microsoft x64 (Scriptum 24). Wyczyszczenie stosu programu jest obowiązkiem funkcji wywołującej (ang. caller).

Argumentów nie przekazuje się wyłącznie przez stos, ale przez wybrane rejestry takie jak: R9, R8, RDX, RCX. Ze stosu korzysta się, gdy argumentów jest więcej niż cztery i odkłada się je „od końca", czyli od prawej do lewej strony. Rezultat funkcji, jeśli ma rozmiar mniejszy niż 64-bity, zwracany jest w rejestrze akumulatora RAX.

W konwencji Microsoft x64, jeśli chcemy modyfikować wartości rejestrów RBP, RBX, RDI, RSI, RSP, R12, R13, R14 i R15, to powinniśmy zachować ich wartości np. na stosie, a następnie je przywrócić przed powrotem do Windows.

Należy również pamiętać o wyrównaniu stosu do okrągłych 16 bajtów, co zostało dokładnie opisanej wcześniej.

```
sub rsp, 38h
mov qword ptr [rsp+30h], 0
mov qword ptr [rsp+28h], \
  FILE_ATTRIBUTE_NORMAL
mov qword ptr [rsp+20h], \
  CREATE_NEW
xor r9, r9
xor r8, r8
mov rdx, GENERIC_WRITE
mov rcx, offset szFileName
call CreateFileA
mov hFile, rax
```

Scriptum 24. Wywołanie funkcji CreateFileA (ml64.exe)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

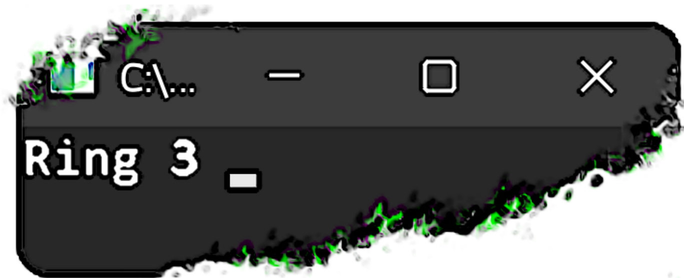
Poziomy uprzywilejowania (ang. rings)

Architektura systemu Windows zapewnia izolację programów użytkownika od tych działających w trybie jądra systemu. Możliwe jest to dzięki zaimplementowanym w procesorze zabezpieczeniom, które udostępniają cztery poziomy uprzywilejowania (ang. rings) dla wykonywanego kodu [10].

Najbardziej uprzywilejowany jest poziom zerowy w którym działa jądro systemu operacyjnego (ang. kernel). Dwa kolejne poziomy (*ring 1* oraz *ring 2*) w którym to działają komponenty krytyczne dla systemu operacyjnego jak np. sterowniki są o umiarkowanym poziomie uprzywilejowania. Natomiast poziom użytkownika (*ring 3*) to tryb w którym działają standardowe programy, które nie mają bezpośredniego dostępu do krytycznych zasobów systemowych czy sprzętowych. W dokumentacji procesorów AMD64 oraz Intel 64 bieżący poziom uprzywilejowania oznaczany jest skrótem *CPL* (*Current Privilege Level*). Można go odczytać z rejestru segmentu kodu CS (Fotografia 32).

Scriptum 25 zawiera przykładowy kod odczytujący poziom uprzywilejowania. Rdzeniem wspomnianego przykładu jest zdefiniowanie ciągu bajtów tak jak poniżej.

```
szT db "Ring ",03Fh,0
```



Fotografia 32. Odczytanie poziomu uprzywilejowania z rejestru CS przez program w Asemblerze

Teraz za pomocą rozkazu MOV można odczytać poziom uprzywilejowania z rejestru CS i umieścić w rejestrze RAX.

```
mov rax, cs
```

W celu zmiany bajtu ze znakiem zapytania na odczytaną wartość można zastosować instrukcje, która zamienia piąty bajt w zmiennej szT na wartość z rejestru AL (najmłodszy bajt rejestru RAX).

```
mov byte ptr [szT+5], al
```

Przykład Scriptum 25 zawiera kod źródłowy całego programu.

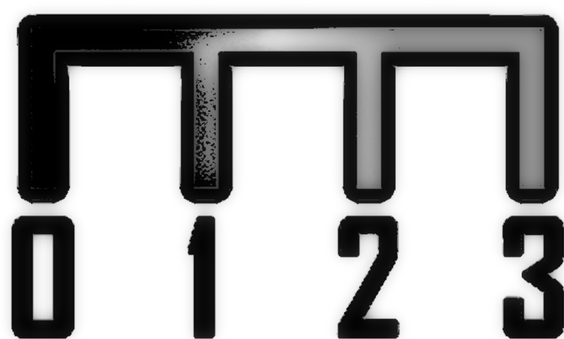
© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

```
extrn AllocConsole : proc
extrn GetStdHandle : proc
extrn WriteConsoleA : proc
extrn ReadConsoleA : proc
extrn FreeConsole : proc
extrn ExitProcess : proc

.const
  STD_OUTPUT_HANDLE equ \
  0FFFFFFFFFFFFFFF5h
  STD_INPUT_HANDLE equ \
  0FFFFFFFFFFFFFFF6h

.data
  szE db "ethical.blue",0
  szT db "Ring ",03Fh,0
   dwW dd 0
  szC db 0

.code
Main proc
  push r12
  push r13
  push r14
  push r15
  sub rsp, 38h

  mov rax, cs
  mov byte ptr [szT+5], al

  call AllocConsole
  mov rcx, STD_OUTPUT_HANDLE
  call GetStdHandle

  mov qword ptr [rsp+20h], 0
  lea r9, dwW
  mov r8, sizeof szT
  lea rdx, szT
  mov rcx, rax
  call WriteConsoleA

  mov rcx, STD_INPUT_HANDLE
  call GetStdHandle

  mov qword ptr [rsp+20h], 0
  lea r9, dwW
  mov r8, sizeof szC
  lea rdx, szC
  mov rcx, rax
  call ReadConsoleA

  call FreeConsole
  add rsp, 38h
  pop r15
  pop r14
  pop r13
  pop r12
  ret
Main endp
end
```

Scriptum 25. Odczytanie poziomu uprzywilejowania z rejestru CS i wyświetlenie na konsoli tekstowej (ml64.exe)

Poziomy uprzywilejowania w procesorach o architekturze x86-64 (x64)

Poziomy uprzywilejowania



Najbardziej uprzywilejowany — **Najmniej uprzywilejowany**

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Wywołania systemowe (ang. syscalls)

Interfejs programowania aplikacji Windows (w skrócie WinAPI) to w uproszczeniu zestaw funkcji zawartych w bibliotekach, które są przeważnie częścią systemu. Udostępniane są piszącym kod, aby mogli używać tych funkcji w swoich programach. Istniejące od bardzo dawna Windows API (nazywane też Win32 API) składa się z warstw i często jedne funkcje są „opakowaniami" na znajdujące się abstrakcyjnie niżej wywołania.

Podstawowe funkcje dostępne są w bibliotekach `kernel32.dll` oraz `user32.dll`.

Przykład. Morph tworzy program korzystający z Win32 API i potrzebuje utworzyć plik. Może do tego celu skorzystać z funkcji `CreateFile`.

Co jednak dzieje się wewnątrz funkcji? Znajdzie tam m.in. różne rozkazy procesora, ale wykonanie zmierza do wywołania funkcji `NtCreateFile`. Funkcja `NtCreateFile` znajduje się w bibliotece `ntdll.dll` i jest abstrakcyjnie niżej niż Win32 API, a zbiór funkcji z przedrostkiem `Nt` tam zawartych określa się jako Native API. Interfejs ten jest mostem pomiędzy aplikacjami użytkownika, a „wnętrznościami" systemu Windows.

Native API zawiera też elementy, które dotyczą trybu bardziej uprzywilejowanego takie jak wersje funkcji z przedrostkiem `Zw`, czyli idąc za przykładem będzie to `ZwCreateFile`. Korzysta się z nich podczas tworzenia programów działających w trybie mocno uprzywilejowanym.

Warto wspomnieć też o bibliotece `hal.dll`, która jest powiązana z warstwą abstrakcji sprzętowej (ang. Hardware Abstraction Layer) i odpowiada m.in. za ukrycie szczegółów implementacji obsługi sprzętu i urządzeń, co zwiększa przenośność kodu oraz dostarcza interfejs do komunikacji ze sprzętem oraz usługami jądra systemu Windows. Biblioteka znajduje się w katalogu przedstawionym poniżej.

```
%WINDOWS%\System32\hal.dll
```

Funkcje, które udostępnia ta biblioteka mają przedrostek `Hal`. Na przykład `HalExamineMBR`.

Jeśli Native API jest mostem pomiędzy trybem użytkownika, a trybem mocno uprzywilejowanym, to powinno zawierać jakiś mechanizm przekazywania wywołań. Możliwe jest to za pomocą rozkazu `SYSCALL`. A najlepszym wyjaśnieniem będzie podejrzenie wnętrza wybranej funkcji Native API.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

— Pora uruchomić narzędzie WinDbg. Staramy się, aby było na każdym urządzeniu typu Windows Desktop w naszym laboratorium. — oznajmił pracownik.

W celu uruchomienia programu w narzędziu WinDbg należy wybrać `File / Launch executable`.

Na potrzeby prostego eksperymentu próbką może być nawet systemowy Notatnik (`notepad.exe`), który powinien się znajdować pod ścieżką zaprezentowaną poniżej.

```
%WINDOWS%\System32\notepad.exe
```

W oknie poleceń (ang. command) powinny pojawić się informacje o wczytanych modułach wśród których jest `ntdll.dll`.

Poprzez wydanie polecenia `u` (ang. unassemble) możliwe jest wyświetlenie wnętrza funkcji `NtCreateFile` w języku Asembler.

```
u ntdll!NtCreateFile
```

Polecenie `u` (ang. unassemble) w narzędziu WinDbg pozwala dokonać deasemblacji, czyli otrzymania odzyskanego kodu w języku Asembler wybranego fragmentu programu takiego jak funkcja czy inny adres w pamięci.

We wnętrzu funkcji `NtCreateFile` przedstawionym jako kod w Asemblerze można zauważyć podstawowe elementy, które opisano poniżej.

- Przekazanie argumentu z rejestru `RCX` do `R10`, czyli rozkaz `SYSCALL` niszczy poprzednią zawartość rejestru `RCX`.
- Wczytanie numeru wywołania systemowego do rejestru `EAX`. Wartość ta może się zmieniać wraz z nowymi wersjami systemu operacyjnego.
- Instrukcja `TEST` sprawdza czy architektura systemu operacyjnego to 64-bity.
- Instrukcja `JNE`, skocz jeśli nierówne (ang. jump if not equal), wykonuje przejście warunkowe do adresu 0x15 licząc od początku analizowanej funkcji, czyli do rozkazu `INT 2Eh`.
- Jeśli skok się nie wykona, to system operacyjny jest architektury 64-bitowej, więc do wywołania systemowego jest używany rozkaz `SYSCALL`, a nie `INT 2Eh`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Registers | Memory 0 | Disassembly

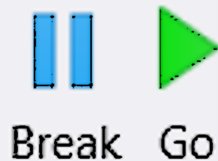
Command

```
4c8bd1           mov     r10,rcx
b855000000       mov     eax,55h
f604250803fe7f01 test     byte ptr [SharedUserDa
7503             jne     ntdll!NtCreateFile+0x15
0f05             syscall
c3               ret
cd2e             int     2Eh
c3               ret
```

```
0:000> u ntdll!NtCreateFile
```

C:\Wind

Plik



Break Go

Flow

Locals | Name

Locals | Watch

Threads

TID	Index	Thread
0x22ce4	0x0	notepad
0x24c8c	0x1	ntdll!Tpr

Threads | Stack | Breakpoints

Registers | Memory 0 | Disassembly

Command

```
4c8bd1           mov     r10,rcx
b836000000       mov     eax,36h
f604250803fe7f01 test     byte ptr [SharedUserDa
7503             jne     ntdll!NtQuerySystemInfo
0f05             syscall
c3               ret
cd2e             int     2Eh
c3               ret
```

```
0:000> u ntdll!NtQuerySystemInformation
```

Locals

Threads

Index	Thread
0x0	notepad

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Uzyskanie kodu w Asemblerze z pliku źródłowego w języku Visual C++

Kod programu w języku Asembler jest często formą pośrednią między językiem wysokiego poziomu abstrakcji a plikiem wykonywalnym. Pozwala to na tworzenie kodu źródłowego w Visual C++ lub podobnej technologii i uzyskanie pliku z kodem w języku Asembler.

Przykładowe wywołanie programu `cl.exe` przedstawiono poniżej (Scriptum 26).

```
cl.exe /FA /GA /GL /GS- main.cpp
```

Scriptum 26. Przykładowe polecenie wywołujące cl.exe (x64 Native Tools Command Prompt for VS)

Opis parametrów wywołania zaprezentowano poniżej.

- `cl.exe` – Microsoft C/C++ Optimizing Compiler for x64
- `/FA` – wygenerowanie pliku w języku Asembler.
- `/GA` – włączenie optymalizacji. Nie używać w przypadku bibliotek dynamicznych (.DLL).
- `/GL` – włączenie optymalizacji całego programu.
- `/GS-` – wyłącza ochronę przed *buffer overrun*. Można wyłączyć to sprawdzanie jeśli aplikacja nie będzie wystawiona na zagrożenia tego typu.
- `main.cpp` – nazwa pliku z kodem źródłowym w bieżącym katalogu.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

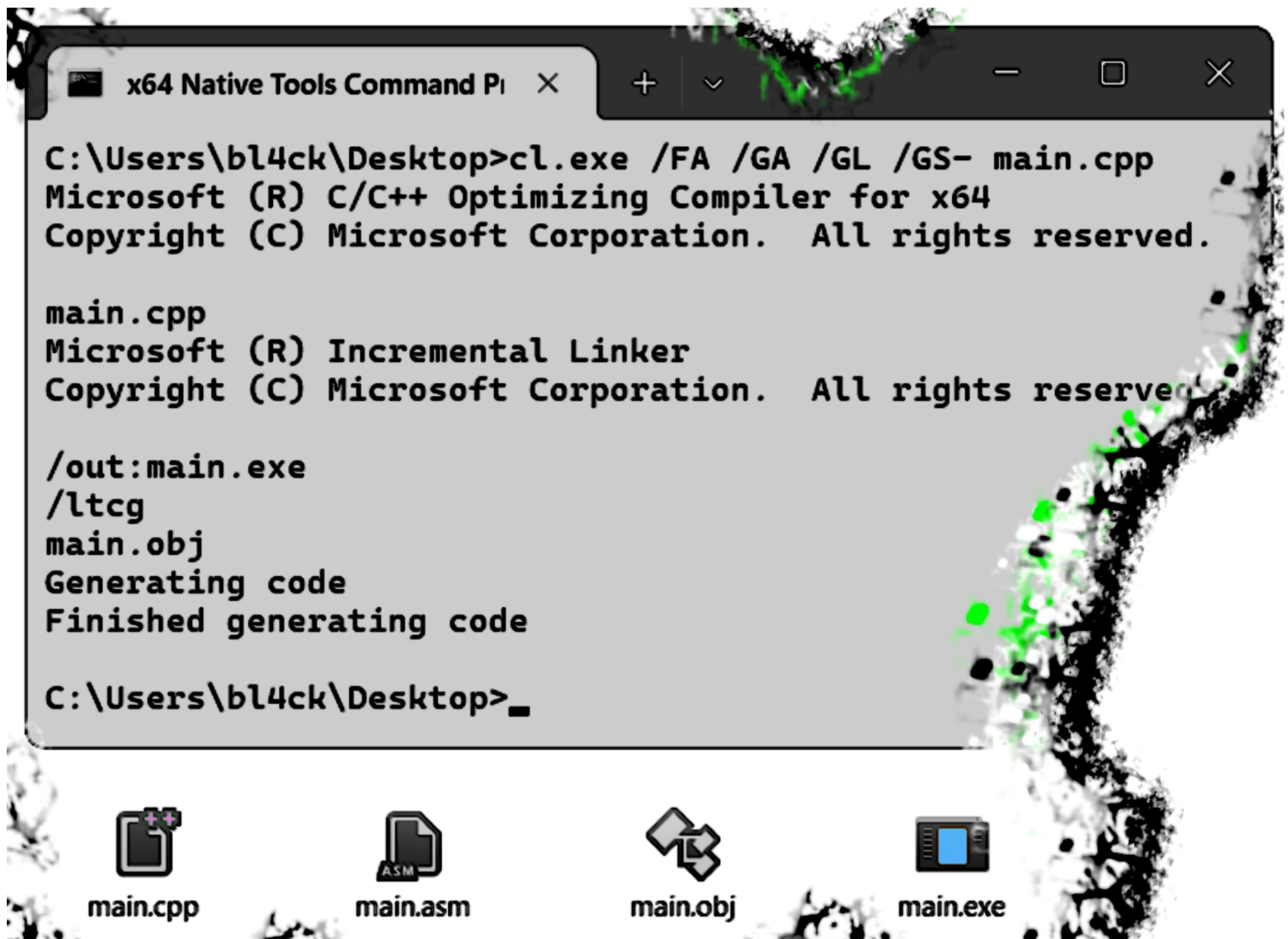
Poniżej przedstawiono program w Microsoft Visual C++ dla systemu Windows wyświetlający okno dialogowe za pomocą funkcji MessageBox z biblioteki user32 (Scriptum 27).

```
#include <Windows.h>

#pragma comment(lib, "user32.lib")

int WINAPI wWinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  PWSTR pCmdLine,
  int nCmdShow)
{
  MessageBox(0,"ethical.blue", "3OHA", MB_OK);
  ExitProcess(0);
}
```

Scriptum 27. Przykładowy program wyświetlający okno dialogowe MessageBox (Microsoft Visual C++)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Jeśli generowanie pliku w Asemblerze zakończy się sukcesem, to w katalogu bieżącym powinien powstać nowy plik z rozszerzeniem *.asm (Fotografia 33).



Fotografia 33. Przykładowe wywołanie cl.exe – Microsoft C/C++ Optimizing Compiler for x64

Scriptum 28 zawiera przykładowy kod pośredni wygenerowany przez narzędzie Microsoft C/C++ Optimizing Compiler.

Bardzo szybko można zauważyć, że jest to kod stworzony przez narzędzie, ponieważ zawiera elementy typowe dla kodu źródłowego generowanego automatycznie.

```
include listing.inc

INCLUDELIB OLDNAMES

EXTRN __imp_ExitProcess:PROC
EXTRN __imp_MessageBoxA:PROC
$SG4294967290 DB '3OHA',00H
 ORG $+3
$SG4294967289 DB 'ethical.blue',00H
CONST ENDS
PUBLIC wWinMain
pdata SEGMENT
$pdata$wWinMain DD imagerel $LN3
 DD imagerel $LN3+63
 DD imagerel $unwind$wWinMain
pdata ENDS
xdata SEGMENT
$unwind$wWinMain DD 011801H
 DD 04218H
xdata ENDS

_TEXT SEGMENT
hInstance$ = 48
hPrevInstance$ = 56
pCmdLine$ = 64
nCmdShow$ = 72
wWinMain PROC

$LN3:
 mov DWORD PTR [rsp+32], r9d
 mov QWORD PTR [rsp+24], r8
 mov QWORD PTR [rsp+16], rdx
 mov QWORD PTR [rsp+8], rcx
 sub rsp, 40

 xor r9d, r9d
 lea r8, OFFSET FLAT:$SG4294967290
 lea rdx, OFFSET FLAT:$SG4294967289
 xor ecx, ecx
 call QWORD PTR __imp_MessageBoxA

 xor ecx, ecx
 call QWORD PTR __imp_ExitProcess
 npad 1
$LN2@wWinMain:

 add rsp, 40
 ret 0
wWinMain ENDP
_TEXT ENDS
END
```

Scriptum 28. Kod pośredni w Asemblerze wygenerowany przez narzędzie Microsoft C/C++ Optimizing Compiler

Wykaz literatury

Dokumenty przedstawione poniżej to literatura obowiązkowa dla chcących zgłębić tematykę języka Asembler x86-64 (x64).

- Advanced Micro Devices, Inc. (AMD), *AMD64 Architecture Programmer's Manual*, 2024.
- Intel Corporation, *Intel Advanced Vector Extensions 10.2 Architecture Specification*, 2025.
- Intel Corporation, *Intel Architecture Instruction Set Extensions and Future Features*, 2025.
- Intel Corporation, *The Intel 64 and IA-32 Architectures Software Developer's Manual*, 2025.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Niezwykłe anomalie w językach C#/IL dla platformy .NET

Kod źródłowy napisany w językach wysokiego poziomu abstrakcji, takich jak C# czy inne języki platformy .NET, jest tłumaczony do języka pośredniego, którego wykonywaniem zajmuje się maszyna wirtualna. [5]

Inne spotykane nazwy języka pośredniego CIL to .NET IL oraz dawniej MSIL. Język pośredni IL przypomina Asembler, jednak nie używa się tutaj rejestrów — jest oparty na stosie i dodatkowo pozwala operować na obiektach. Maszyna wirtualna (ang. Virtual Execution System) to silnik odpowiedzialny za wykonywanie programów stworzonych dla infrastruktury języka pośredniego. [11]

Kod zarządzany (ang. managed) i niezarządzany (ang. unsafe)

Kod zarządzany (ang. managed code) nie jest tylko wczytywany do pamięci i uruchamiany jak w przypadku języków natywnych niskiego poziomu abstrakcji. Istnieje infrastruktura, która sprawuje nadzór, czyli zarządza wykonaniem kodu i ma dostęp do metadanych opisujących metody, monitoruje operacje na stosie czy zajmuje się obsługą wyjątków. Dane w kodzie zarządzanym też nie są pozostawione same sobie. Infrastruktura CLI zajmuje się rezerwacją pamięci oraz usuwaniem niepotrzebnych obiektów za pomocą mechanizmu nazywanego *garbage collector*, co w języku polskim nazywamy odśmiecaniem pamięci.

Podstawowe pojęcia

Poniżej przedstawiono pojęcia związane z językiem pośrednim IL. [11]

- Infrastruktura języka pośredniego (ang. Common Language Infrastructure) — specyfikacja środowiska, którego zadaniem jest wykonywanie kodu CIL.
- Język pośredni (ang. Common Intermediate Language) — zestaw instrukcji, których wykonaniem zajmuje się maszyna wirtualna VES. Inne spotykane nazwy języka pośredniego CIL to: .NET IL oraz MSIL. Język pośredni CIL przypomina Asembler, jednak nie używa się tutaj rejestrów — jest oparty na stosie i pozwala operować na obiektach.
- Maszyna wirtualna (ang. Virtual Execution System) — silnik odpowiedzialny za wykonywanie programów stworzonych dla infrastruktury języka pośredniego (ang. Common Language Infrastructure).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Narzędzie typu asembler dla języka pośredniego IL

W celu zbudowania pliku wykonywalnego *.exe lub *.dll z kodu źródłowego w języku IL można użyć narzędzia `ilasm.exe`. Należy w tym celu zainstalować środowisko programistyczne Microsoft Visual Studio i uruchomić Developer PowerShell for Visual Studio.

Przykładowe wywołanie programu `ilasm.exe` przedstawiono poniżej (Scriptum 29).

```
ilasm.exe ethicalblue.il /QUIET /SUBSYSTEM=2 /PE64 /X64
```

Scriptum 29. Przykładowe polecenie wywołujące ilasm.exe (Developer PowerShell for Visual Studio)

Opis parametrów wywołania zaprezentowano poniżej.

- `ilasm.exe` – Microsoft .NET Framework IL Assembler
- `ethicalblue.il` – nazwa pliku z kodem źródłowym w bieżącym katalogu.
- `/QUIET` – wyłączenie wyświetlanie postępu procesu budowania.
- `/SUBSYSTEM=2` – oznacza program z interfejsem graficznym użytkownika (GUI). Stała o wartości dwa oznacza dokładnie ustawienie `IMAGE_SUBSYSTEM_WINDOWS_GUI` w słowie maszynowym `Subsystem` w nagłówku `_IMAGE_OPTIONAL_HEADER` pliku wykonywalnego PE (ang. Portable Executable).
- `/PE64` – utworzenie pliku wykonywalnego PE64 (PE32+).
- `/X64` – ustawienie architektury docelowej na x64.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Poniżej przedstawiono kod źródłowy programu wyświetlającego okno dialogowe `MessageBox` z przestrzeni nazw `System.Windows.Forms` (Fotografia 34). Rdzeniem przykładu są instrukcje `ldstr` odkładające napisy na stos, instrukcje `ldc.i4.` odkładające wartości stałe na stos oraz wywołanie metody `Show` z klasy `MessageBox`, aby wyświetlić okno dialogowe.

```
.assembly extern mscorlib { .ver 0:0:0:0 }
.assembly extern System.Runtime {
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 4:0:0:0 }
.assembly extern System.Windows.Forms {
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0 }
.assembly ethicalblue { .ver 0:0:0:0 }
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    .custom instance void [System.Runtime]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
    .maxstack 8
    ldstr "WISDOM +1"
    ldstr "ethical.blue Magazine"
    ldc.i4.0
    ldc.i4.s 64
    call valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
        [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(
            string, string,
            valuetype [System.Windows.Forms]System.Windows.Forms.MessageBoxButtons,
            valuetype [System.Windows.Forms]System.Windows.Forms.MessageBoxIcon
        )
    pop
    ret
}
```

Fotografia 34. Kod źródłowy przykładowego programu w Asemblerze IL

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Etapy pozwalające na zbudowanie kodu źródłowego w Asemblerze IL do pliku wykonywalnego *.exe przedstawiono poniżej.

Najpierw należy uruchomić konsolę tekstową Developer PowerShell for Visual Studio.

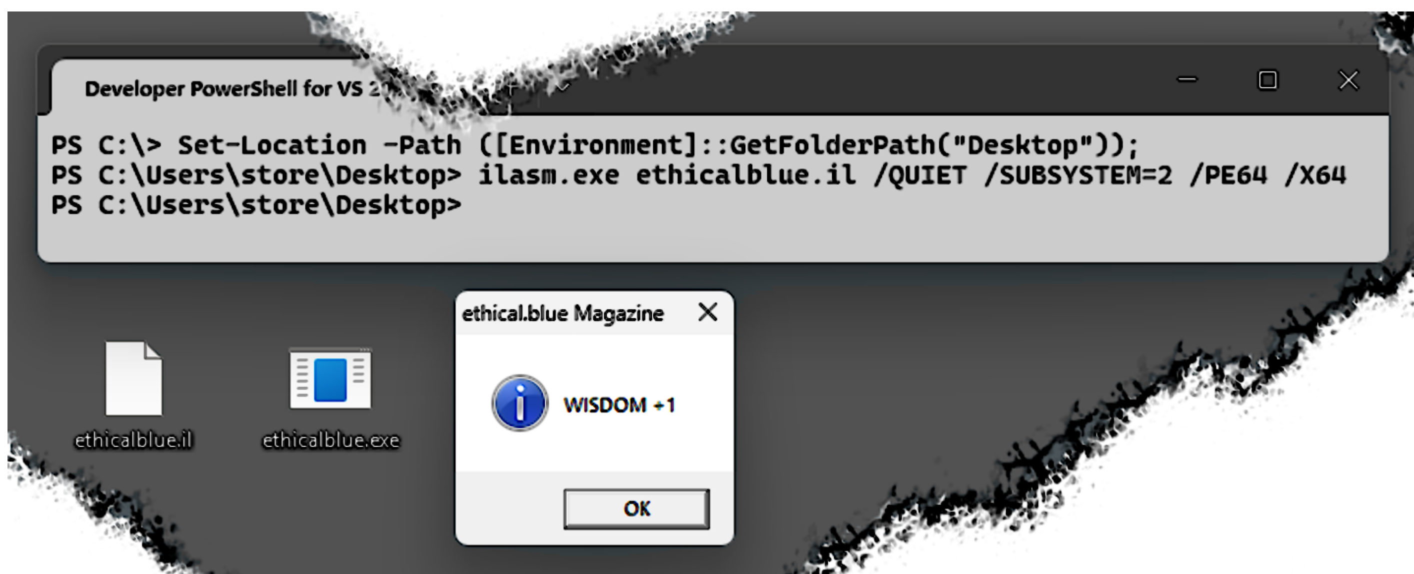
Można użyć poniższego polecenia, aby ustawić katalog bieżący na Pulpit.

```
Set-Location -Path ([Environment]::GetFolderPath("Desktop"));
```

Przykładowy kod źródłowy (np. Fotografia 34) należy wpisać do pliku tekstowego i nadać mu rozszerzenie `.il`. Jeśli plik został nazwany `ethicalblue.il` i architektura docelowa to x64, to polecenie rozpoczynające budowanie będzie takie jak poniżej.

```
ilasm.exe ethicalblue.il /QUIET /SUBSYSTEM=2 /PE64 /X64
```

Pojawienie się w bieżącym katalogu pliku `ethicalblue.exe` oznacza, że proces budowania zakończył się powodzeniem i można uruchomić stworzoną aplikację. Prezentowany przykład wyświetla okno dialogowe z tekstem *WISDOM +1* (Fotografia 35).

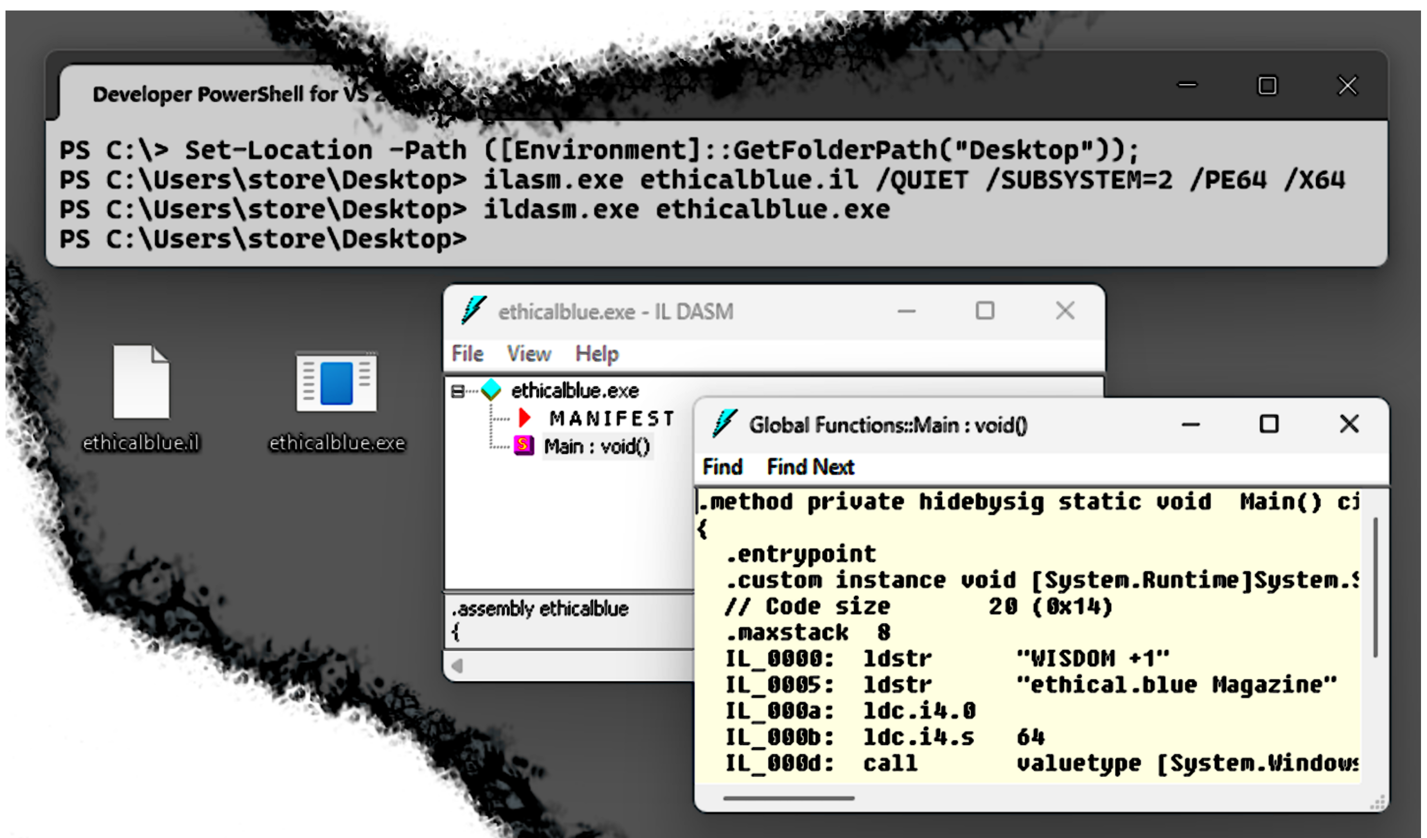


Fotografia 35. Budowanie przykładowego programu w Asemblerze IL do pliku wykonywalnego *.exe (Developer PowerShell for Visual Studio)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

— Istnieją o wiele bardziej zaawansowane deasemblery, ale na potrzeby niniejszego eksperymentu użyjemy `ildasm.exe` (Fotografia 36). Najważniejsze to zrozumieć mechanizm działania, Morph. — oznajmił pracownik laboratorium.

— Narzędzie, które przeprowadza proces odwrotny to w tym przypadku `ildasm.exe` i pozwala odzyskać z pliku wykonywalnego kod w języku pośrednim IL (Fotografia 36). Badanie uzyskanego w ten sposób kodu z pliku wykonywalnego nazywamy analizą statyczną. Oczywistym staje się fakt, że zarówno w strefie anomalii jak i poza obszarem ЗОНА (wym. `zona`) istnieją ukryte laboratoria, które pracują nad utrudnianiem procesu analizy znalezionych artefaktów. — kontynuował wyjaśnienia pracownik laboratorium.



Fotografia 36. Deasemblacja pliku wykonywalnego i odzyskanie kodu pośredniego w Asemblerze IL (Developer PowerShell for Visual Studio)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Anomaly.C#/IL!equ.A

```
/*-     Anomaly.C#/IL!equ.A     -*\
\*-    ethical.blue Magazine    -*/

T(@"
 0.1000000001F == 0.1000000009F
");
T($" {
 0.1000000001F == 0.1000000009F
} ");
T(@"
 0.3333333333F == 0.3333333333
");
T($" {
 0.3333333333F == 0.3333333333
} ");

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 30. Anomaly.C#/IL!equ.A

Konsola debugowania

```
0.1000000001F == 0.1000000009F

True

0.3333333333F == 0.3333333333

False
```

Scriptum 31. Anomaly.C#/IL!equ.A
(Konsola debugowania)

— Anomaly.C#/IL!equ.A. Fascynujące, nie sądzisz? Taka już jest natura liczb zmiennoprzecinkowych, które ze względu na swój format są nieprecyzyjne. — stwierdził pracownik laboratorium.

— Dlatego należy unikać porównywania liczb typu `float` czy `double`, gdyż prowadzi to często do nieoczekiwanego zachowania programu. — kontynuował.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Anomaly.C#/IL!adr.A

```
/*-     Anomaly.C#/IL!adr.A     -*\
\*-    ethical.blue Magazine    -*/

using System.Net;

List<string> v4 = [
 "0177.0.3.1",
 "0x7f.0.3.1",
 "127.769",
 "2130707201",
 "017700001401",
 "0x7F.0x0.0x3.0x1",
 "0x7f000301",
 "0x000000007F000301"
];

List<string> v6 = [
 "0000:0000:0000:0000" +
     ":0000:0000:0000:0001",
 "0:0:0:0:0:0:0:1",
 "[0:0:0:0:0:0:0:1]",
 "[::1]"
];

T($" .AllEqual() == {
 v4.All(z => addrEqual(z,
 "127.0.3.1"))
}, {
 v6.All(z => addrEqual(z,
 "::1"))
} ");
static bool addrEqual(
string a, string b) =>
IPAddress.Parse(a)
.Equals(IPAddress.Parse(b));
static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 32. Anomaly.C#/IL!adr.A

Konsola debugowania

```
.AllEqual() == True, True
```

Scriptum 33. Anomaly.C#/IL!adr.A (Konsola debugowania)

— Anomaly.C#/IL!adr.A to niesamowity przykład na jak wiele sposobów można przedstawić adres protokołu internetowego (ang. Internet Protocol Address, IP Address). — powiedział pełen entuzjazmu pracownik laboratorium.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

— W wersji czwartej protokołu poszczególne fragmenty mogą być zapisywane nie tylko w systemie dziesiętnym. Wartości mogą być oddzielone kropkami lub nie. Dodatkowo może dojść do przepełnienia (ang. overflow), czyli 127.769 oznacza 127.0.3.1 (3×255+4). Natomiast w wersji szóstej można pominąć wiodące zera, usunąć zera czy też umieścić adres IPv6 w nawiasach kwadratowych.
— kontynuował wyjaśnienia.

— Nieprzewidywanych zachowań może być mniej lub więcej. Zależy od implementacji obsługi przetwarzania adresu protokołu internetowego przez określone oprogramowanie.
— podsumował.

Wymienione formy adresu IP można sprawdzać następującym poleceniem (PowerShell).

```
Test-Connection 0x7f000301 -Count 1
| Select-Object Address


Address
-------
127.0.3.1
```

Przykładowe adresy protokołu internetowego przedstawiono poniżej.

- `127.0.3.1`
- `0177.0.3.1`
- `0x7f.0.3.1`
- `127.769`
- `2130707201`
- `017700001401`
- `0x7F.0x0.0x3.0x1`
- `0x7f000301`
- `0x000000007F000301`
- ...
- `0:0:0:0:0:0:0:1`
- `[0:0:0:0:0:0:0:1]`
- `[::1]`
- ...

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Bacteria.C#/IL!+=.A

```
/*-     Bacteria.C#/IL!+=.A      -*\
\*-    ethical.blue Magazine     -*/

string x = string.Empty;
M(); T(" -> ");
for (int i = 0; i < 3072; i++)
 x += "Bacterium?";
M(); T(Environment.NewLine);
T(x[..10]);
T(Environment.NewLine);

static void M() {
double c =
 GC.GetTotalAllocatedBytes();
 c /= 1048576.0; T($"{c:F4} MB");
}
static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

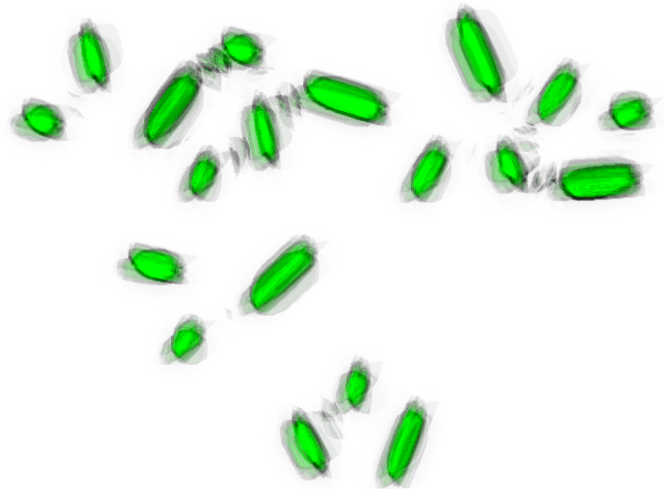
Scriptum 34. Bacteria.C#/IL!+=.A

Konsola debugowania

```
1,0713 MB -> 91,1548 MB
Bacterium?
```

Scriptum 35. Bacteria.C#/IL!+=.A (Konsola debugowania)

— Łączenie ciągów znaków za pomocą operatora + w pętli może mieć negatywny wpływ na wydajność aplikacji, ponieważ powstaje wiele nowych obiektów w pamięci. O wiele lepszym rozwiązaniem jest zastosowanie w takich przypadkach metody `Append` z klasy `System.Text.StringBuilder`. — wyjaśnił pracownik laboratorium.



© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Example.C#/IL!StringBuilder

```
/*- Example.C#/IL!StringBuilder -*\
\*-    ethical.blue Magazine    -*/

using System.Text;

string x = string.Empty;
M(); T(" -> ");
StringBuilder builder = new();
for (int i = 0; i < 3072; i++)
 builder.Append(
 "Use System.Text.StringBuilder."
);
M(); T(Environment.NewLine);
T(builder.ToString()[..30]);
T(Environment.NewLine);

static void M() {
double c =
 GC.GetTotalAllocatedBytes();
 c /= 1048576.0; T($"{c:F4} MB");
}
static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 36. Example.C#/IL!StringBuilder

Konsola debugowania

```
1,0713 MB -> 91,1548 MB
Bacterium?
```

Scriptum 37. Bacteria.C#/IL!+=.A (Konsola debugowania)

Konsola debugowania

```
1,0713 MB -> 1,2730 MB
Use System.Text.StringBuilder.
```

Scriptum 38. Example.C#/IL!StringBuilder (Konsola debugowania)

— Powyższe przykłady doskonale prezentują różnice w zużyciu pamięci przez aplikację. — podsumował pracownik laboratorium.

— Fragment `x += "Bacterium?";` tworzy nowe obiekty, co powoduje, że program zachowuje się jak bakteria. Zastosowanie `StringBuilder builder = new(); /*...*/ builder.Append(/*...*/ builder.ToString(/*...*/` pozwala zaoszczędzić trochę pamięci. — kontynuował wyjaśnienia pracownik laboratorium.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!Cmd.Inject

```
/*- Vulnerable.C#/IL!Cmd.Inject -*\
\*-    ethical.blue Magazine    -*/

using System.Diagnostics;

// string clean = "ethical.blue";
string anomaly =
    "::1 -n 1 && " +
    "shutdown /s /t 10 " +
    "/c \"ethical.blue\"" +
    " && exit";
Exec(anomaly);
T("cmd.exe /c ping " +
    $"{anomaly} -n 1");

static void Exec(string a) =>
 Process.Start(
  "cmd.exe", $"/c ping {a} -n 1"
 );

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 39. Vulnerable.C#/IL!Cmd.Inject

Konsola debugowania

```
cmd.exe /c ping ::1 -n 1 &&
  shutdown /s /t 10 /c "ethical.
  blue" && exit -n 1
```

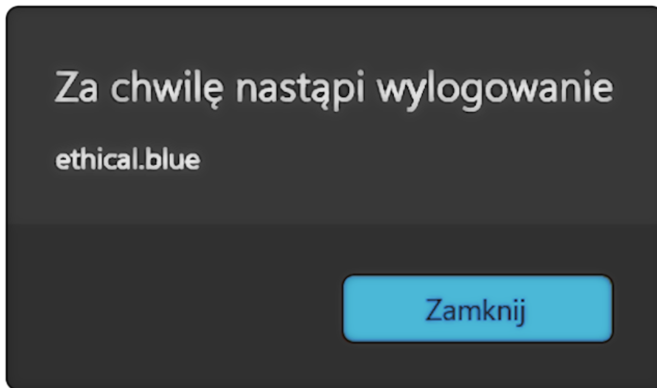
Scriptum 40. Vulnerable.C#/IL!Cmd.Inject (Konsola debugowania)

Próbka `Vulnerable.C#/IL!Cmd.Inject` to przykład podatności pozwalającej na wstrzyknięcie polecenia systemowego do tworzonego ciągu znaków. Za pomocą metody `Process.Start` z przestrzeni nazw `System.Diagnostics` uruchamiany jest Wiersz polecenia (cmd.exe) z parametrami *\/c ping {a} -n 1*.

W zwykłym scenariuszu parametr *{a}* to adres maszyny docelowej do której będą wysyłane pakiety, aby sprawdzić połączenie przez oczekiwanie na odpowiedź od urządzenia zdalnego.

Nietypowy scenariusz. Jeśli parametr *{a}* przyjmie wartość `::1 -n 1` to zweryfikuje połączenie z maszyną lokalną. Nic groźnego. Jednak dalej za pomocą znaków `&&` zostaje dołączone `shutdown /s /t 10 /c "ethical.blue" && exit -n 1`, co powoduje zamknięcie systemu operacyjnego.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.



Fotografia 37. Wywołanie nieplanowanego zamknięcia systemu Microsoft Windows

— Fotografia 37. Jeśli urządzenie jest fragmentem infrastruktury i podatność pozwoli na wstrzyknięcie polecenia wywołującego nieplanowane zamknięcie systemu operacyjnego, to działanie tego rodzaju nazywane jest odmową usługi (ang. Denial of Service). — wyjaśnił pracownik laboratorium.

— Czasem wyłączenie części infrastruktury to najmniejszy z możliwych problemów. Podatność pozwalająca na nieautoryzowane wykonanie kodu może przynieść katastrofalne skutki, a zdalne sterowanie maszynami przez ukryty kanał komunikacyjny spowodować wycieki wrażliwych informacji i niebezpieczne manipulowanie środowiskiem pracy całego laboratorium. — kontynuował.

— Teraz już wiesz w jakim celu przekazuję Ci tego typu wiedzę, Morph. — podsumował pracownik.

Wstrzyknięcie polecenia do ciągu znaków to jeden z prostszych scenariuszy. Jeśli fragment jest umieszczany wewnątrz innego polecenia, to, zależnie od środowiska, stosuje się znak kończący bieżące polecenie, a po nim umieszcza się ładunek (ang. payload). Czasem możliwe jest łączenie poleceń (ang. chaining) specjalnymi elementami składniowymi zależnymi od środowiska działania aplikacji (Fotografia 38).

```
::1 -n 1 && shutdown /s /t 10 /c "ethical.blue" && exit

$"cmd.exe /c ping {domain} -n 1"
```

Fotografia 38. Przykład wstrzyknięcia polecenia systemu operacyjnego Microsoft Windows do ciągu tekstowego w języku C#

W celu łagodzenia (ang. mitigate) tego typu zagrożeń powinno się odpowiednio neutralizować elementy specjalne biorąc pod uwagę środowisko wykonania.

Fragmenty z zewnątrz zawsze należy traktować jako niezaufane i starać się utworzyć surową listę dozwolonych wartości (ang. known good), zamiast skanować ciąg w poszukiwaniu niebezpiecznych wzorców. [13]

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!Path.Trv.A

```
/*- Vulnerable.C#/IL!Path.Trv.A -*\
\*-    ethical.blue Magazine    -*/

string n = Environment.NewLine;
T(".      = " + R(".") + n);
T("..     = " + R("..") + n);
T("...    = " + R("...") + n);
T("../../ = " + R("../../")
 + n + n);
T(@"C:\Windows\.. = ");
T(R(@"C:\Windows\..") + n);
T(@"C:\..\..\..\  = ");
T(R(@"C:\..\..\..\") + n);
T(@"C:\\\\\\\\\// = ");
T(R(@"C:\\\\\\\\\//") + n);
T(@"C:\Windows\..\Data = ");
T(R(@"C:\Windows\..\Data")
 + n);
static string R(string x) =>
 System.IO.Path.GetFullPath(x);
static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 41. Vulnerable.C#/IL!Path.Trv.A

Konsola debugowania

```
.      = C:\Laboratory\Sample
..     = C:\Laboratory
...    = C:\Laboratory\Sample\
../../ = C:\

C:\Windows\.. = C:\
C:\..\..\..\  = C:\
C:\\\\\\\\\// = C:\
C:\Windows\..\Data = C:\Data
```

Scriptum 42. Vulnerable.C#/IL!Path.Trv.A (Konsola debugowania)

Ścieżka do określonego zasobu może mieć formę bezpośrednią, czyli taką jak np. `C:\Windows` czy `\\HEAVEN\C:\`. Pierwszy przykład kieruje do folderu systemowego, a drugi przykład prezentuje ścieżkę do dysku oznaczonego literą C na maszynie o nazwie HEAVEN.

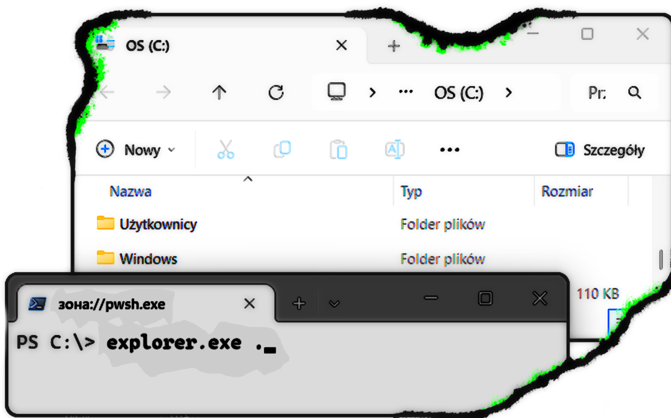
Istnieją też ścieżki względne i w ich przypadku należy wspomnieć o dwóch specjalnych nazwach katalogów. Pojedyncza kropka (`.`) oznacza folder bieżący, a dwie kropki (`..`) to folder o jeden poziom wyżej od bieżącego.

Próbka `Vulnerable.C#/IL!Path.Trv.A` prezentuje przykładowe ścieżki względne rozwinięte do pełnej, bezpośredniej postaci.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

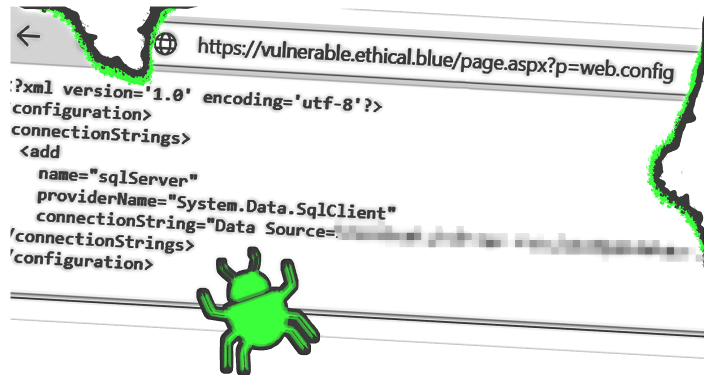
Pojedynczą kropkę często spotyka się w konsoli tekstowej wywołując skrypt PowerShell np. `.\Skrypt.ps1` czy też w systemach Linux uruchamiając plik wykonywalny z bieżącego katalogu np. `./ethical.elf`.

Inny przykład to uruchomienie okna Eksploratora Windows (`%SystemRoot%\explorer.exe`) z otwartym folderem bieżącym za pomocą polecenia `explorer.exe .` (*po explorer.exe jest znak kropki*) przedstawionego poniżej (Fotografia 39).



Fotografia 39. Znak kropki w ścieżce oznacza aktualny folder, czyli przedstawione polecenie uruchomi explorer.exe z ustawionym bieżącym katalogiem

Istnienie prezentowanych elementów specjalnych jasno pokazuje, że niezamierzone przejście między katalogami w przypadku aplikacji przyjmujących ścieżkę jako parametr może zagrażać bezpieczeństwu (Fotografia 40).



Fotografia 40. Odczytanie pliku web.config przestarzałej aplikacji w technologii ASP.NET z powodu wielu podatności i błędnej konfiguracji

Najbardziej rozsądnym pomysłem na łagodzenie (ang. mitigate) tego rodzaju zagrożenia wydaje się brak ekspozycji parametru reprezentującego ścieżkę na zewnątrz oraz utworzenie ścisłej listy dozwolonych wartości.

Jeśli kod aplikacji będzie zawierał błędy bezpieczeństwa i podatności, to nawet filtry wyszukujące niebezpieczne wzorce mogą okazać się nieskuteczne.

Nie można też zapominać, że w niektórych środowiskach poszczególne znaki mogą być różnie interpretowane. Na przykład dla adresów URL znak kropki można przedstawić jako ciąg `//ethical.blue/..` lub w innych formatach takich jak m.in. `//ethical.blue/%2e%2e`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!Path.Trv.B

```
/*- Vulnerable.C#/IL!Path.Trv.B -*\
\*-    ethical.blue Magazine    -*/

const string r = @"C:\Laboratory";
var anomaly = @"..\..\..\..\.." +
 @"\Windows\system.ini";
T(Vulnerable(anomaly));
T(Mitigated("readme"));
string Vulnerable(string i)
 => File.ReadAllText(
  Path.Combine(r, i));
string Mitigated(string i)
 => i switch {
  "index" =>
   File.ReadAllText(
       $"{r}\\index.txt"),
  "readme" =>
   File.ReadAllText(
       $"{r}\\readme.txt"),
  _ => string.Empty };
static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 43. Vulnerable.C#/IL!Path.Trv.B

Konsola debugowania

```
[386Enh]
...

[drivers]
...

[mci]
```

Scriptum 44. Vulnerable.C#/IL!Path.Trv.B (Konsola debugowania)

Anomalia `Vulnerable.C#/IL!Path.Trv.B` zawiera dwie przykładowe metody odczytujące plik tekstowy. Pierwsza metoda nazwana `Vulnerable` udostępnia na zewnątrz parametr, który jest ścieżką do pliku i przez to jest podatna na niezamierzone przejścia między katalogami stosując sekwencje `..\` (dwie kropki i *backslash*).

Konsola debugowania (Scriptum 44) prezentuje nieautoryzowane odczytanie zawartości pliku `C:\Windows\system.ini` w środowisku laboratoryjnym. Natomiast druga metoda nazwana `Mitigated` zawiera surową listę dozwolonych wartości i łagodzi zagrożenie niezamierzonego przejścia między katalogami.

Bug.C#/IL!Useless.Increment

```
/*- Bug.C#/IL!Useless.Increment -*\
\*-    ethical.blue Magazine    -*/

int x = 0;
int z = 0;

T($"x++ = {x++} // useless");
T($"++z = {++z} // OK");

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 45. Bug.C#/IL!Useless.Increment

Konsola debugowania

```
x++ = 0 // useless
++z = 1 // OK
```

Scriptum 46. Bug.C#/IL!Useless.Increment (Konsola debugowania)

Operatory zwiększenia (++) i zmniejszenia o jeden (--) mogą być w formie prefiksu, czyli przed zmienną, właściwością, indekserem lub w postaci przyrostkowej nazywanej *postfix*.

W przypadku zwracania wartości z metod (`return`) należy używać prefiksu, ponieważ wtedy operacja zwiększenia lub zmniejszenia o jeden jest wykonywana najpierw.

`Bug.C#/IL!Useless.Increment` prezentuje bezużyteczną próbę zwiększenia wartości zmiennej `x` o jeden (`x++`). Operacja jest umieszczona w ciągu znaków przeplatanym wyrażeniami (ang. interpolated). Zwiększenie wartości w formie przyrostkowej jest bezużyteczne, ponieważ modyfikuje zmienną x dopiero po umieszczeniu jej w napisie.

Nawet jeśli zmienna jest używana później i zostanie przykładowo znów zwiększona o jeden, czyli `x++`, to w wyniku otrzymamy wartość dwa, co może być mylące.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!Random.A

```
/*-  Vulnerable.C#/IL!Random.A  -*\
\*-    ethical.blue Magazine    -*/

using System.Security.Cryptography;

byte[] buffer = [0x00];

var randomGenerator =
  RandomNumberGenerator.Create();
randomGenerator.GetBytes(buffer);
T($"0x{buffer[0]:X2}");
buffer[0] = 0x00; // avoid leak

#region Vulnerable (Predictable)
  int seed = 0xBAD;
  var random = new Random(seed);
  random.NextBytes(buffer);
  T($"0x{buffer[0]:X2}");
#endregion

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 47. Vulnerable.C#/IL!Random.A

Konsola debugowania

```
0xEA
0x43
```

Scriptum 48. Vulnerable.C#/IL!Random.A (Konsola debugowania)

Aplikacje mogą korzystać z losowych wartości w funkcjach kryptograficznych i różnych mechanizmach zabezpieczających. Z tego powodu bardzo ważne jest, aby generowane wartości były jak najmniej możliwe do przewidzenia.

Krytycznym błędem jest używanie ziarna (ang. seed), które da się przewidzieć lub uzyskać w inny sposób. Podatność tego typu powoduje, że losowe wartości nie są wystarczająco bezpieczne kryptograficznie.

W celu złagodzenia zagrożenia należy korzystać ze specjalnych klas kryptograficznych, takich jak m.in. `System.Security.Cryptography.RandomNumberGenerator`, zamiast zwykłego `System.Random`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!NaN.eq.A

```
/*-  Vulnerable.C#/IL!NaN.eq.A  -*\
\*-    ethical.blue Magazine    -*/

T("0.00 / 0.00 == 0.00 / 0.00");
T($"{0.00 / 0.00 == 0.00 / 0.00}");

T(Environment.NewLine);

T("float.NaN == float.NaN");
T($"{float.NaN == float.NaN}");

T(Environment.NewLine);

T("float.IsNaN(float.NaN)");
T($"{float.IsNaN(float.NaN)}");

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 49. Vulnerable.C#/IL!NaN.eq.A

Konsola debugowania

```
0.00 / 0.00 == 0.00 / 0.00
False


float.NaN == float.NaN
False


float.IsNaN(float.NaN)
True
```

Scriptum 50. Vulnerable.C#/IL!NaN.eq.A (Konsola debugowania)

NaN (ang. Not a Number) to wartość, która nie jest równa żadnej wartości, a nawet samej sobie.

W celu weryfikacji czy w wyniku powstał NaN należy używać specjalnej metody o nazwie `IsNaN`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!.Min.Max

```
/*-  Vulnerable.C#/IL!.Min.Max  -*\
\*-    ethical.blue Magazine    -*/

var max = int.MaxValue;

T("int.MaxValue");
T($"+{max} (0x{max:X2})");
T(Environment.NewLine);

T("int.MaxValue + 1");
T($"{max + 1} (0x{max + 1:X2})");
T(Environment.NewLine);

T("int.MaxValue + 4");
T($"{max + 4} (0x{max + 4:X2})");
T(Environment.NewLine);

T("Math.Clamp(50 + 90, 0, 100);");
T($"{Math.Clamp(50 + 90,
    0, 100)}");

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 51. Vulnerable.C#/IL!.Min.Max

Konsola debugowania

```
int.MaxValue
+2147483647 (0x7FFFFFFF)

int.MaxValue + 1
-2147483648 (0x80000000)

int.MaxValue + 4
-2147483645 (0x80000003)

Math.Clamp(50 + 90, 0, 100);
100
```

Scriptum 52. Vulnerable.C#/IL!.Min.Max (Konsola debugowania)

Jeśli wartość maksymalna 32-bitowej liczby typu `int` to +2147483647 i zostanie zwiększona o jeden, to nastąpi obrót i zmienna przyjmie wartość minimalną, czyli -2147483648.

Zabezpieczeniem przed błędami przepełnienia całkowitoliczbowego może być zawężenie wyniku do określonego przedziału. W programowaniu niskopoziomowym powoduje to mniej zakłóceń w przypadku przetwarzanych sygnałów. W składni wysokopoziomowej takiej jak C# można zastosować w tym celu metodę `Clamp` z klasy `System.Math`.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!unsafe.A

```
/*-  Vulnerable.C#/IL!unsafe.A  -*\
\*-    ethical.blue Magazine    -*/

unsafe
{
    var z = LocRetVuln();
    T($"0x{*z:X8}");
    z++;
    T($"0x{*z:X8}");
    z++;
    T($"0x{*z:X8}");
    z++;
}
unsafe static uint* LocRetVuln()
{
    uint* x = stackalloc[] {
        0xDEADBEEF, 0xDEADC0DE,
        0xADD1C7ED
    };
    return x;
}

static void T(string szT) =>
typeof(Console).Assembly
.GetType("\x53\x79\x73"
+ "\x74\x65\x6D\x2E\x43"
+ "\x6F\x6E\x73\x6F\x6C\x65")?
.GetMethod("\x57\x72\x69\x74"
+ "\x65\x4C\x69\x6E\x65",
[typeof(string)])?
.Invoke(null, [szT]);
```

Scriptum 53. Vulnerable.C#/IL!unsafe.A

Konsola debugowania

```
0x00000000
0x000000D1
0x00000002
```

Scriptum 54. Vulnerable.C#/IL!unsafe.A (Konsola debugowania)

Podatność prezentowana w anomalii `Vulnerable.C#/IL!unsafe.A` nazywana jest `Use After Free` i polega na próbie dostępu do pamięci, która została oznaczona jako zwolniona. [13]

We fragmentach kodu objętych słowem kluczowym `unsafe` nie występuje automatyczna weryfikacja bezpieczeństwa pamięci. Przykładowa anomalia `Vulnerable.C#/IL!unsafe.A` pokazuje statyczną metodę o wymyślonej nazwie `LocRetVuln`, w której następuje rezerwacja miejsca na stosie, gdzie umieszczone są trzy liczby całkowite bez znaku o charakterystycznie wyglądających wartościach.

Zwrócenie zmiennej lokalnej z metody za pomocą słowa kluczowego `return` powoduje, że zmienna wskazuje na pamięć, która najczęściej zawiera już inne wartości, co powoduje nieoczekiwane zachowanie programu.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Vulnerable.C#/IL!unsafe.B

```
/*-  Vulnerable.C#/IL!unsafe.B  -*\
\*-    ethical.blue Magazine    -*/

unsafe
{
  OutOfBounds();
}

  OutOfBoundsManaged();

unsafe static void OutOfBounds()
{
  int* x = stackalloc[] {
    0x00000000, 0x00000000,
    0x00000000
  };
  x[3] = 0xDEADEAD;
}
static void OutOfBoundsManaged()
{
  int[] z = [
    0x00000000, 0x00000000,
    0x00000000
  ];
  z[3] = 0xDEADEAD;
}
```

Scriptum 55. Vulnerable.C#/IL!unsafe.B

Konsola debugowania

```
Unhandled exception. System.
  IndexOutOfRangeException: Index
  was outside the bounds of the
  array.
```

Scriptum 56. Vulnerable.C#/IL!unsafe.B (Konsola debugowania)

W kodzie zarządzanym (ang. managed) wyjście poza zakres przy dostępie do elementów kolekcji jest kontrolowane przez infrastrukturę języka pośredniego. Powoduje to rzucenie wyjątku, który powinien być przechwycony i przekazany do odpowiedniej metody obsługującej.

Przykład `Vulnerable.C#/IL!unsafe.B` pokazuje, że fragment kodu niezarządzanego, oznaczony słowem kluczowym `unsafe`, powodujący wyjście poza zakres zmiennej tablicowej o nazwie `x`, nie skutkuje wyrzuceniem wyjątku.

Największy problem przy wyjściu poza zakres pojawia się wtedy, gdy w uszkodzonym miejscu pamięci znajdują się struktury lub kod maszynowy krytyczny dla bezpieczeństwa aplikacji.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Wykaz literatury

Dokument przedstawiony poniżej to literatura obowiązkowa dla chcących zgłębić tematykę podatności (ang. vulnerability) w językach programowania.

- ISO/IEC, *Programming languages. Avoiding vulnerabilities in programming languages*, 2024.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Nieznane rozkazy. Asembler arm64 w systemach Microsoft Windows

Rodzaj platformy sprzętowej na której zainstalowano system Windows można odczytać za pomocą polecenia PowerShell przedstawionego poniżej (Fotografia 41).

```
Get-WmiObject Win32_ComputerSystem
   | Select-Object SystemType
```

```
PowerShell
PS C:\Users\ethicalblue> Get-WmiObject -Class Win32_ComputerSystem
 | Select-Object -Property SystemType

SystemType
----------
ARM64-based PC

PS C:\Users\ethicalblue> [System.Environment]::OSVersion.Version

Major  Minor  Build  Revision
-----  -----  -----  --------

PS C:\Users\ethicalblue>
```

Fotografia 41. Odczytanie rodzaju architektury i wersji systemu operacyjnego (PowerShell)

Podstawy architektury arm64 (AArch64)

Architektura sprzętowa Arm cały czas się rozwija, dlatego jej kolejne generacje nazywane są z użyciem numeru wersji np. Armv7, Armv8 czy Armv9. Procesory Arm obsługują kolejność bajtów LE oraz BE. System operacyjny Windows dla architektury Arm działa na kolejności bajtów LE. Warto wspomnieć, że nazwy AArch64 i AArch32 dotyczą stanu w jakim jest procesor (ang. execution state). Stan AArch64 (nazywany arm64) oznacza możliwość korzystania z 64-bitowych rejestrów oraz zestawu rozkazów A64. Natomiast stan AArch32 (nazywany arm32) używa 32-bitowych rejestrów oraz zestawu rozkazów A32 oraz T32.

Dodatkowo architektura Arm definiuje następujące profile przedstawione poniżej.

- Application (pol. aplikacje),
- Real-time (pol. czasu rzeczywistego),
- Microcontroller (pol. mikrokontroler).

Dlatego często można spotkać zapis np. Armv9-A, który oznacza architekturę Arm dziewiątej generacji i profil Application.

Podstawowe rozmiary danych w architekturze arm64 (AArch64)

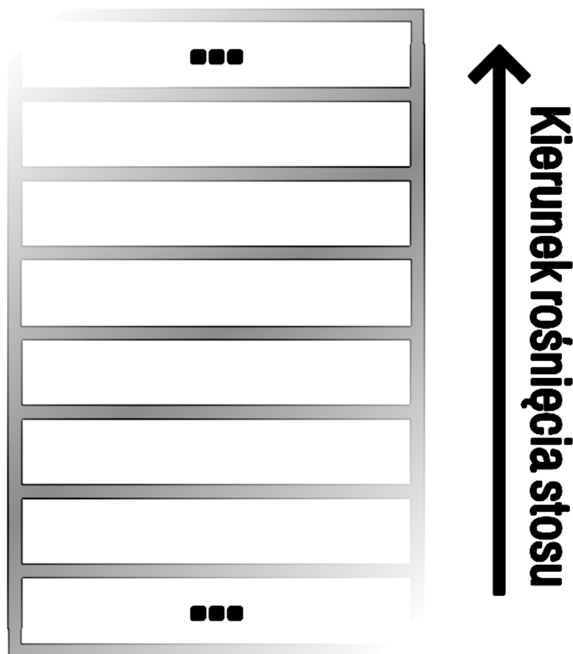
Poniżej przedstawiono podstawowe rozmiary danych bez określonego typu w architekturze arm64.

- Bajt (8 bitów)
- Pół słowa maszynowego (16 bitów)
- Słowo maszynowe (32 bity)
- Podwójne słowo maszynowe (64 bity)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Stos programu

Stos programu rośnie w kierunku niższych adresów pamięci. Oznacza to, że kolejne dane umieszczane na stosie będą miały niższe adresy, czyli wartość rejestru SP będzie odpowiednio zmniejszana (Fotografia 42).

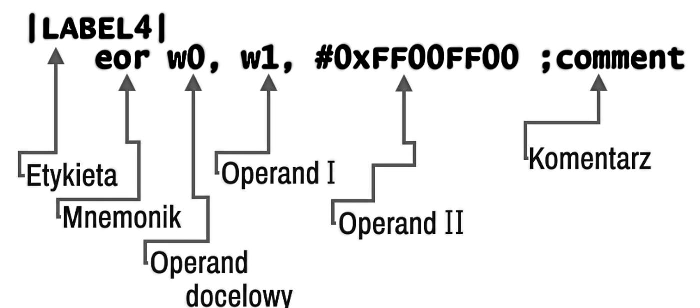


Fotografia 42. Kierunek rośnięcia stosu programu (arm64, AArch64)

Przykładowy rozkaz w formie tekstowej

Instrukcje procesora w formie tekstowej przyjaznej do odczytywania nazywane są mnemonikami. Podczas procesu budowania rozkazy te są zamieniane na ciągi bajtów według określonego formatu, czyli instrukcje w formie binarnej, które nazywane są kodem maszynowym.

Przykładowy rozkaz EOR (ang. Exclusive OR) wykonuje alternatywę wykluczającą na określonych wartościach. Działa na trzech operandach podawanych po przecinku, gdzie najpierw występuje operand docelowy w którym znajdzie się rezultat, a dopiero dalej operand pierwszy i operand drugi. Nie należy zapominać, że korzystając z Microsoft Arm Assembler wymagane jest, aby przed rozkazami było wcięcie w postaci znaku tabulatora lub czterech spacji (Fotografia 43).



Fotografia 43. Przykładowy rozkaz alternatywy wykluczającej (arm64, AArch64)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestry R0..R30

Rejestry od `R0` do `R30` są udostępniane w postaci `W0..W30` (32 bitowe) oraz `X0..X30` (64 bitowe). Zastosowanie poszczególnych rejestrów zostało opisane poniżej.

- Rejestr `X30` to *Link Register* (`LR`), który przechowuje adres powrotny wymagany do powrotu z wywołania procedury.
- Rejestr `X18` (`XPR`) w trybie użytkownika (ang. user mode, ring 3) umożliwia dostęp do struktur Thread Environment Block (TEB) i Process Environment Block (PEB).
- Parametry wywoływanej procedury lub funkcji są przekazywane przez rejestry od `X0` do `X7`. Jeśli parametrów jest więcej niż osiem, to pozostałe przekazywane są przez stos.
- Wartości w postaci liczb całkowitych są zwracane w rejestrze `X0`.
- Rejestry od `X19` do `X28` oraz rejestr `FP` są nieulotne (ang. nonvolatile), czyli jeśli są modyfikowane (używane), to ich wartości powinny zostać zachowane na początku funkcji i przywrócone przed wyjściem z funkcji.
- Rejestry od `X8` do `X15` są ulotne (ang. volatile) — nie ma potrzeby zachowania ich wartości.

Rejestry V0..V31 oraz FPSR i FPCR

Rejestry od `V0` do `V31` są udostępniane w postaci `B0..B31` (8 bitowe), `H0..H31` (16 bitowe), `S0..S31` (32 bitowe), `D0..D31` (64 bitowe) oraz `Q0..Q31` (128 bitowe). Zastosowanie poszczególnych rejestrów zostało opisane poniżej.

- Parametry w postaci liczb zmiennoprzecinkowych dla wywoływanej procedury lub funkcji są przekazywane przez rejestry od `V0` do `V7`. Jeśli parametrów jest więcej niż osiem, to pozostałe przekazywane są przez stos.
- Wartości w postaci liczb zmiennoprzecinkowych są zwracane w rejestrze `V0`.
- Młodsze 64-bity rejestrów od `V8` do `V15` są nieulotne (ang. nonvolatile), czyli jeśli są modyfikowane (używane), to ich wartości powinny zostać zachowane na początku funkcji i przywrócone przed wyjściem z funkcji.
- Rejestry `V16` do `V31` są ulotne (ang. volatile) — nie ma potrzeby zachowania ich wartości.

`FPCR` to rejestr kontrolny, a `FPSR` to rejestr statusu operacji zmiennoprzecinkowych.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rejestry Z0..Z31

Rejestry od Z0 do Z31 są używane w rozkazach Scalable Vector Extensions (SVE) i ich rozmiar jest zależny od implementacji.

Rejestr Program Counter

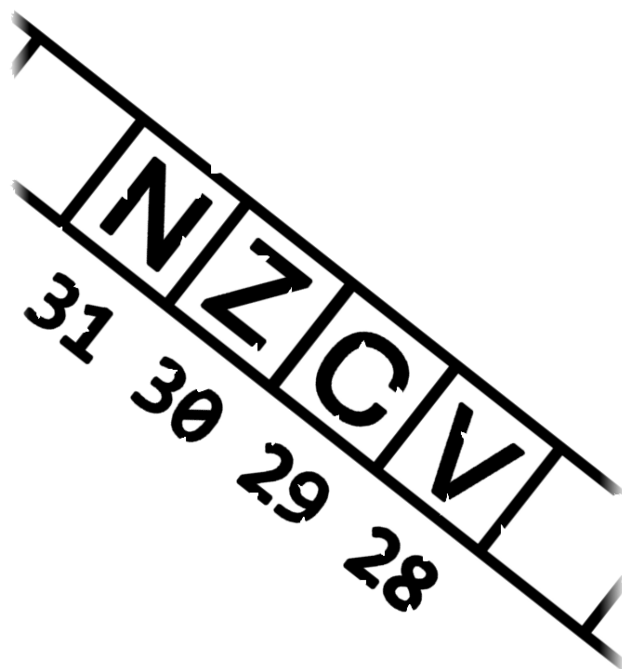
Rejestr PC pozwala określić, która instrukcja jest następna do wykonania. Nie jest to rejestr ogólnego przeznaczenia. Rejestr PC jest rejestrem specjalnym.

Rejestr Stack Pointer

Rejestr wskaźnika stosu. Nie jest to rejestr ogólnego przeznaczenia. Rejestr SP jest rejestrem specjalnym.

Flagi NZCV

- N (Negative) — Rezultat operacji jest wartością ujemną. Flaga ta jest ustawiana zgodnie z bitem znaku rezultatu (najstarszy bit).
- Z (Zero) — Rezultat operacji jest równy zero. Flaga jest ustawiana np. gdy porównujemy dwie wartości i są one równe. Gdyż porównanie dwóch wartości to wykonanie odejmowania. Inny przykład to dekrementacja, czyli zmniejszanie wartości o jeden. Jeśli osiągniemy zero, to flaga jest ustawiana.
- C (Carry) — Rezultat operacji spowodował przeniesienie (ang. carry). Przy użyciu rejestrów X0..X30 oznacza to, że rezultat nie mieści się w rejestrze (wartość jest większa niż 64 bity).
- V (Overflow) — Rezultat operacji spowodował przepełnienie. Oznacza to, że rezultat nie mieści się w rejestrze oraz zmienił się najstarszy bit (nazywany bitem znaku).



© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Podstawy składni Microsoft Arm Assembler

Podstawowe elementy składni Microsoft Arm Assembler (armasm64.exe).

Stałe (EQU)

Dyrektywa EQU, której synonimem jest znak gwiazdki (*) pozwala nadać wartości numerycznej przyjazną nazwę symboliczną. Pozwala to uniknąć tzw. magic numbers, czyli wartości liczbowych, których znaczenie nie jest zrozumiałe bez zagłębiania się w dokumentację (Scriptum 57).

```
NULL EQU 0
WM_SETTEXT EQU 12
```

Scriptum 57. Przykład definicji stałej wartości (Microsoft Arm Assembler)

Dane o rozmiarze bajta (DCB)

Dyrektywa DCB, której synonimem jest znak równości (=) pozwala zarezerwować jeden lub więcej bajtów z nadaną wartością początkową. Za pomocą tego rodzaju dyrektywy możliwe jest również definiowanie napisów zakończonych zerem — nazywanych *C string*. Jednak o kończący znak NULL należy zadbać samemu (Scriptum 58).

```
szText DCB "ethical.blue", 0
```

Scriptum 58. Przykład definicji ciągu bajtów (Microsoft Arm Assembler)

Dane o rozmiarze pół słowa maszynowego (DCW, DCWU)

Dyrektywa DCW pozwala zarezerwować pół słowa maszynowego (16 bitów) lub więcej takich pół słów z nadaną wartością początkową (Scriptum 59).

```
wVar1 DCW 0
```

Scriptum 59. Przykład definicji pół słowa maszynowego (Microsoft Arm Assembler)

Dane o rozmiarze słowa (DCD, DCDU)

Dyrektywa DCD pozwala zarezerwować słowo maszynowe (32 bity) lub więcej takich słów z nadaną wartością początkową (Scriptum 60).

```
dwVar1 DCD 0
```

Scriptum 60. Przykład definicji słowa maszynowego (Microsoft Arm Assembler)

Dane o rozmiarze podwójnego słowa (DCQ, DCQU)

Dyrektywa DCQ pozwala zarezerwować podwójne słowo maszynowe (64 bity) lub więcej takich podwójnych słów z nadaną wartością początkową (Scriptum 61).

```
dqVar1 DCQ 0
```

Scriptum 61. Przykład definicji podwójnego słowa maszynowego (Microsoft Arm Assembler)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dane o rozmiarze wielu bajtów

Dyrektywy SPACE oraz FILL pozwalają zarezerwować przestrzeń o rozmiarze wielu bajtów z nadaną wartością początkową. Poniżej przykładowe dane o nazwie mVar1 i rozmiarze 255 bajtów wypełnione zerami oraz dane o nazwie mVar2 o rozmiarze 50 bajtów wypełnione wartością 0xCC. Wartość jeden po przecinku to rozmiar wartości (Scriptum 62).

```
mVar1 SPACE 255
mVar2 FILL 50,0xCC,1
```

Scriptum 62. Przykład definicji wielu bajtów (Microsoft Arm Assembler)

Etykiety (ang. labels)

Etykiety oznaczają miejsca w kodzie programu, do których później można się odwoływać za pomocą określonych rozkazów. Pozwala to m.in. na zapętlanie fragmentów kodu czy przekazanie wykonania do fragmentu kodu i powrót za miejsce rozgałęzienia po wykonaniu (Scriptum 63).

W przypadku błędów budowania stosując Microsoft Arm Assembler, należy pamiętać, że etykiety nie mają mieć wcięcia, natomiast mnemoniki powinny mieć wcięcie w postaci tabulatora lub znaków spacji.

```
  AREA .drectve, DRECTVE

  AREA .rdata, DATA, READONLY

  EXPORT Main
  IMPORT |__imp_ExitProcess|

  AREA .text, CODE, ARM64

Main PROC

    stp fp,lr,[sp,#-0x10]!
    mov fp,sp

    mov x0,#0
    tst x0,x0
    beq |FAIL|
    b |EXIT|
|FAIL|
    mov w0,#-1
|EXIT|
    adrp x8,__imp_ExitProcess
    ldr x8,[x8,__imp_ExitProcess]
    blr x8

ENDP

END
```

Scriptum 63. Przykład zawierający rozgałęzienia i etykiety (Microsoft Arm Assembler)

Punktem wejścia przedstawionej wcześniej aplikacji (Scriptum 63) jest procedura `Main` rozpoczynająca się od prologu w którym tworzona jest ramka stosu.

Za pomocą rozkazu `STP` (ang. store pair) od wartości rejestru wskaźnika stosu `SP` jest odejmowana wartość `0x10`, ponieważ jest to adresowanie z preindeksowaniem, a na zarezerwowanym miejscu na stosie zachowywane są wartości rejestrów `LR` (*Link Register*) oraz `FP` (*Frame Pointer*). Później za pomocą instrukcji `mov fp, sp` w rejestrze `FP` zachowywany jest adres wskazujący na nową ramkę stosu (Scriptum 64).

```
stp fp,lr,[sp,#-0x10]!
mov fp,sp
```

Scriptum 64. Fragment tworzący ramkę stosu (Microsoft Arm Assembler)

Kolejny rozkaz (`mov x0,#0`) powoduje wyzerowanie rejestru `X0`.

Później, aby zobaczyć zmianę ścieżki wykonania można tę wartość zmodyfikować.

Dalej za pomocą instrukcji `tst x0,x0` następuje sprawdzenie czy rejestr `X0` jest wyzerowany.

Jeśli tak, to za pomocą rozkazu `beq |FAIL|` (ang. branch if equal) następuje przejście do etykiety `|FAIL|` w której przez instrukcję `mov w0,#-1` kod zwracany do systemu Windows przez funkcję `ExitProcess` jest ustawiany na minus jeden (zakończenie programu z błędem). Należy pamiętać, że to osoba pisząca program decyduje z jakim kodem zakończyć proces, a prezentowana aplikacja ma jedynie cel pokazania użycia etykiet i zmiany ścieżki wykonania. Zmiana rozkazu `mov x0,#0` na `mov x0,#1` spowoduje, że warunek (instrukcje `tst` i `beq`) nie będzie spełniony i nastąpi przejście od razu do etykiety `|EXIT|` (instrukcja `b |EXIT|`).

W miejscu oznaczonym etykietą `|EXIT|` następuje wczytanie adresu funkcji `ExitProcess` do rejestru `X8` (instrukcja `adrp x8,__imp_ExitProcess`).

W celu uzyskania pełnego adresu wymagana jest dodatkowo instrukcja `ldr x8,[x8,__imp_ExitProcess]`.

Zakończenie działania programu następuje przez wywołanie funkcji `ExitProcess`, której adres jest w rejestrze `X8` (rozkaz `blr x8`).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Proste operacje arytmetyczne

Proste operacje arytmetyczne można przeprowadzić za pomocą rozkazów przedstawionych poniżej.

- ADD (dodawanie),
- SUB (odejmowanie),
- SUBS (odejmowanie z ustawieniem flag),
- NEG (zero odjąć wartość),
- MUL (mnożenie),
- UDIV (dzielenie bez znaku),
- SDIV (dzielenie ze znakiem).

– Przykład dodawania (Scriptum 65) –
Do rejestru X8 wpiszmy wartość zero. Do rejestru X10 wpiszmy wartość cztery. Rejestr X8 przyjmuje wartość rejestru X10 dodać jeden, czyli pięć.

```
mov x8,#0
mov x10,#4
add x8,x10,#1
```

Scriptum 65. Przykład dodawania wartości (Microsoft Arm Assembler)

– Przykład odejmowania (Scriptum 66) –
Do rejestru X8 wpiszmy wartość zero. Do rejestru X10 wpiszmy wartość dwieście pięćdziesiąt pięć (0xFF). Rejestr X8 przyjmuje wartość rejestru X10 odjąć jeden, czyli dwieście pięćdziesiąt cztery (0xFE).

```
mov x8,#0
mov x10,#255
sub x8,x10,#1
```

Scriptum 66. Przykład odejmowania wartości (Microsoft Arm Assembler)

– Przykład zmiany znaku (Scriptum 67) –
Do rejestru X8 wpiszmy wartość zero. Do rejestru X10 wpiszmy wartość osiem. Rejestr X8 przyjmuje wartość zero odjąć wartość w rejestrze X10, czyli minus osiem (-8).

```
mov x8,#0
mov x10,#8
neg x8,x10
```

Scriptum 67. Przykład zmiany znaku (Microsoft Arm Assembler)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

– Przykład mnożenia (Scriptum 68) –
Do rejestru X8 wpiszmy wartość zero. Do rejestru X9 wpiszmy wartość dwa. Do rejestru X10 wpiszmy wartość cztery. Rejestr X8 przyjmuje wartość w rejestrze X9 (dwa) razy wartość w rejestrze X10 (cztery), czyli razem osiem.

```
mov x8,#0
mov x9,#2
mov x10,#4
mul x8,x9,x10
```

Scriptum 68. Przykład mnożenia wartości (Microsoft Arm Assembler)

Warto wspomnieć, że istnieje także mnożenie z dodawaniem `madd x8,x9,x10,x11`, czyli *x8 = x9 * x10 + x11*.

– Przykład dzielenia (Scriptum 69) –
Do rejestru X8 wpiszmy wartość zero. Do rejestru X9 wpiszmy wartość osiem. Do rejestru X10 wpiszmy wartość dwa. Rejestr X8 przyjmuje wartość w rejestrze X9 (osiem) dzielone przez wartość w rejestrze X10 (dwa), czyli wynik jest równy cztery.

```
mov x8,#0
mov x9,#8
mov x10,#2
udiv x8,x9,x10
```

Scriptum 69. Przykład dzielenia bez znaku (Microsoft Arm Assembler)

Aby wykonać dzielenie ze znakiem należy użyć rozkazu `sdiv`, zamiast `udiv`.

Proste operacje logiczne

Proste operacje logiczne można przeprowadzić za pomocą rozkazów przedstawionych poniżej.

- AND (koniunkcja),
- ORR (alternatywa),
- MVN (negacja, zaprzeczenie),
- EOR (alternatywa wykluczająca).

– Przykład koniunkcji (Scriptum 70) –
W przykładzie dla koniunkcji (AND) do rejestru docelowego W8 trafi wartość z rejestru W10 (0xFF0000FF) and 0xFF00FF00, czyli 0xFF000000.

```
mov w10,#0xFF0000FF
and w8,w10,#0xFF00FF00
```

Scriptum 70. Przykład koniunkcji (Microsoft Arm Assembler)

– Przykład alternatywy (Scriptum 71) –
W przykładzie dla alternatywy (ORR) do rejestru docelowego W8 trafi wartość z rejestru W10 (0xFF0000FF) orr 0xFF00FF00, czyli 0xFF00FFFF.

```
mov w10,#0xFF0000FF
orr w8,w10,#0xFF00FF00
```

Scriptum 71. Przykład alternatywy (Microsoft Arm Assembler)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

– Przykład zaprzeczenia (Scriptum 72) –
W przykładzie dla zaprzeczenia (mvn) do rejestru docelowego W8 trafi wartość 0xFFFFFF00.

```
mvn w8,#0x000000FF
```

Scriptum 72. Przykład zaprzeczenia (Microsoft Arm Assembler)

– Przykład alt. wykl. (Scriptum 73) –
W przykładzie dla alternatywy wykluczającej (EOR) do rejestru docelowego W8 trafi wartość z rejestru W10 (0xFF0000FF) eor 0xFF00FF00, czyli 0x0000FFFF.

```
mov w10,#0xFF0000FF
eor w8,w10,#0xFF00FF00
```

Scriptum 73. Przykład alternatywy wykluczającej (Microsoft Arm Assembler)

Utworzenie i wywołanie procedury

Tworzenie procedur ma głównie na celu podzielenie kodu źródłowego na mniejsze fragmenty, które mogą być później wielokrotnie wywoływane, zamiast wklejania ich w miejsca gdzie są potrzebne. Przykładowy program prezentuje wydzieloną procedurę ProcedureExample, która wyświetla okno dialogowe za pomocą funkcji MessageBoxA.

Wywołanie procedury za pomocą rozkazu BL (ang. branch with link) zachowuje w rejestrze X30 (LR) adres powrotny, aby po wykonaniu fragmentu kodu zawartego w procedurze wrócić za miejsce rozgałęzienia.

Utworzona procedura posiada własny prolog i epilog z rozkazami odpowiedzialnymi za ramkę stosu (ang. stack frame).

Parametry do wywoływanej funkcji MessageBoxA przekazywane są przez rejestry X0, X1, X2 i X3.

Pełny kod źródłowy przykładu zaprezentowano poniżej (Scriptum 74).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

```
  AREA .drectve, DRECTVE

  AREA .rdata, DATA, READONLY
szText DCB "ethical.blue", 0x00

  EXPORT Main
  IMPORT |__imp_ExitProcess|
  IMPORT |__imp_MessageBoxA|

  AREA .text, CODE, ARM64

Main PROC

    stp fp,lr,[sp,#-0x10]!
    mov fp,sp

    bl ProcedureExample

    mov w0,#0
    adrp x8,__imp_ExitProcess
    ldr x8,[x8,__imp_ExitProcess]
    blr x8

ENDP

  AREA .text, CODE, ARM64

ProcedureExample PROC

    stp fp,lr,[sp,#-0x20]!
    mov fp,sp

    mov w3,#0
    adrp x8,szText
    add x2,x8, szText
    adrp x8,szText
    add x1,x8, szText
    mov x0,#0
    adrp x8,__imp_MessageBoxA
    ldr x8,[x8,__imp_MessageBoxA]
    blr x8

    ldp fp,lr,[sp],#0x20
    ret

ENDP
```

Scriptum 74. Utworzenie i wywołanie procedury (Microsoft Arm Assembler)

Podstawowe instrukcje (A64)

Podstawowe instrukcje zestawu A64 zostały opisane poniżej.

Instrukcje transferu danych

Pozostałe rozkazy są dokładnie opisane w dokumentacji. [3]

LDR

Przykładowa składnia
`LDR` X_t, [X_n, *expression*]
`LDR` X_t, [SP, *expression*]
`LDR` X_t, [W_n, *expression*]

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

Dostęp do wartości w pamięci wykonywany jest na podstawie adresu liniowego. Najprostszy sposób adresowania to odczytanie wartości spod adresu zawartego w rejestrze. Możliwe jest także uzyskanie dostępu do wartości w pamięci poprzez umieszczenie adresu bazowego w rejestrze i dodanie do tego adresu wartości przesunięcia (ang. offset). Przykładem zastosowania tego rodzaju adresowania może być uzyskiwanie dostępu do określonego pola struktury, gdzie adres bazowy struktury jest w rejestrze, a przesunięcia to adresy poszczególnych pól struktury względem adresu bazowego (początku).

W przypadku adresowania z preindeksowaniem najpierw następuje zwiększenie wartości w rejestrze o przesunięcie (ang. offset), a dopiero później odczytanie lub zapisanie wartości spod tego adresu. Natomiast adresowanie z postindeksowaniem powoduje modyfikację wartości rejestru zawierającego adres bazowy po wykonaniu rozkazu odczytu lub zapisu. Z tego powodu adresowania z postindeksowaniem używa się np. przy zdejmowaniu wartości ze stosu — najpierw dane są odczytywane spod adresu w rejestrze, a dopiero później następuje modyfikacja aktualnej wartości adresu. Inne sposoby adresowania, które można spotkać to odczytywanie wartości według przesunięcia dodanego do adresu etykiety lub nawet odczytywanie wartości spod adresu, który jest przedstawiony jako surowa wartość 32-bitowa lub 64-bitowa.

ADR

Przykładowa składnia
`ADR` X_d, *expression*

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

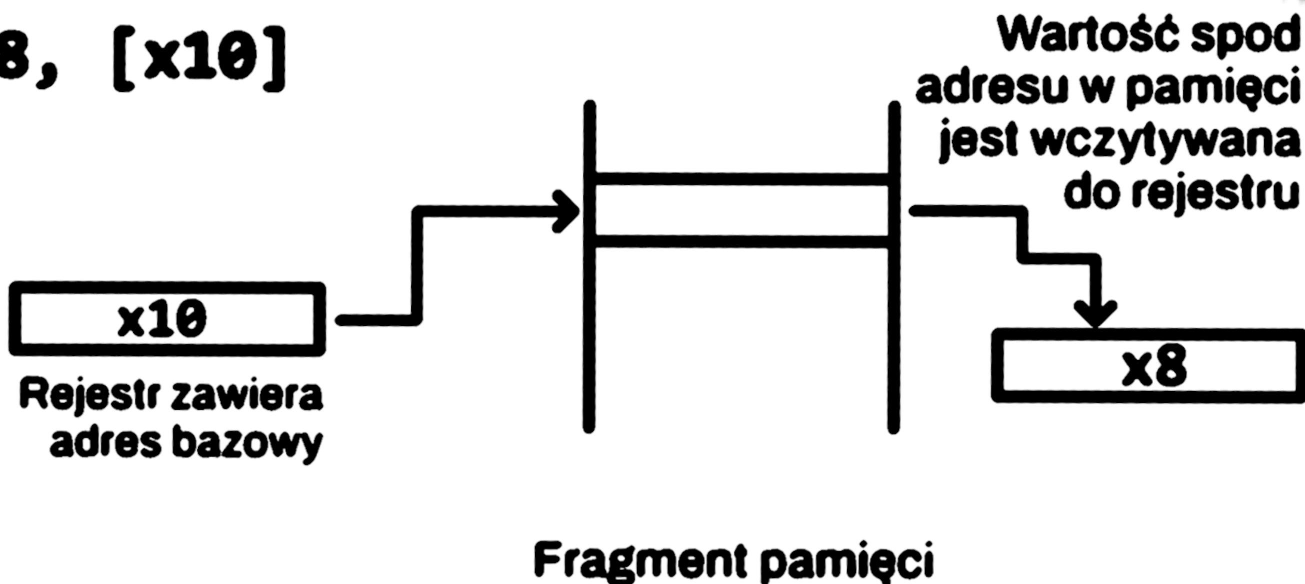
Rozkaz `ADR` pozwala uformować adres względem wartości rejestru `PC` poprzez dodanie odpowiedniego przesunięcia do aktualnej wartości `PC` i zapisanie uzyskanego adresu w 64-bitowym rejestrze ogólnego przeznaczenia.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Adresowanie pośrednie przez rejestr

Przykład

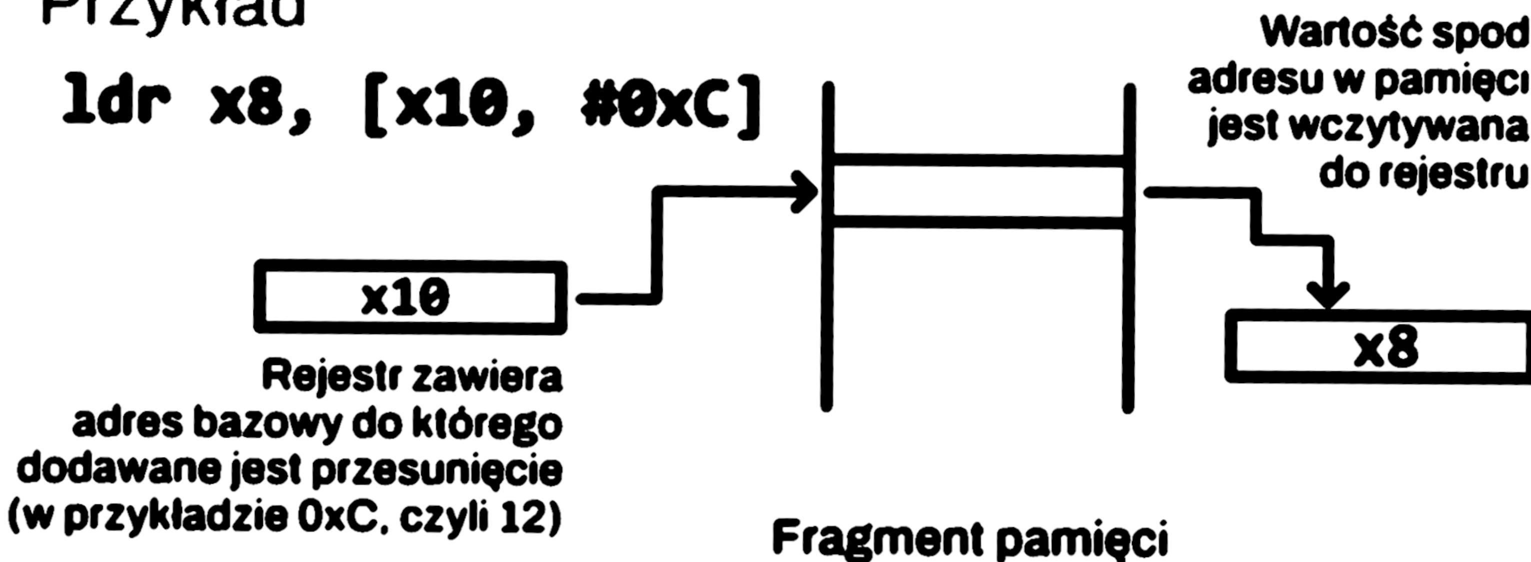
```
ldr x8, [x10]
```



Adresowanie pośrednie przez rejestr z przesunięciem

Przykład

```
ldr x8, [x10, #0xC]
```



© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

ADRL (pseudoinstrukcja)

Przykładowa składnia
`ADRL` X_d, *expression*
`ADRL` W_d, *expression*

Pseudorozkaz `ADRL` pozwala uformować adres względem wartości rejestru `PC` podobnie jak instrukcja `ADR`. Jednak pozwala sięgnąć dalej niż ±1MB, ponieważ mnemonik `ADRL` w kodzie maszynowym jest przeważnie tłumaczony na dwa rozkazy: instrukcja `ADR`, a po niej rozkaz `ADD`. Dzięki temu zasięg formowanych adresów jest większy.

ADRP

Przykładowa składnia
`ADRP` X_d, *expression*

Rozkaz `ADRP` pozwala uformować adres względem wartości rejestru `PC` poprzez dodanie odpowiedniego przesunięcia w zakresie ±4GB do aktualnej wartości `PC` i zapisanie uzyskanego adresu w 64-bitowym rejestrze ogólnego przeznaczenia.

MOV

Przykładowa składnia
`MOV` X_d, X_m
`MOV` W_d, W_m
`MOV` X_d, *#immediate*
`MOV` W_d, *#immediate*
`MOV` X_d, SP
`MOV` W_d, WSP
`MOV` SP, X_m
`MOV` WSP, W_m

Rozkaz `MOV` zawiera wiele wersji składniowych, a niektóre są nawet aliasami dla innych rozkazów. Podstawowe możliwości tej instrukcji to m.in.

- kopiowanie pomiędzy rejestrami ogólnego przeznaczenia,
- kopiowanie pomiędzy rejestrem ogólnego przeznaczenia a rejestrem wskaźnika stosu,
- kopiowanie wartości natychmiastowej (ang. immediate) do rejestru ogólnego przeznaczenia.

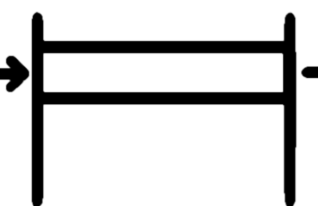
© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Adresowanie pośrednie przez rejestr z postindeksowaniem (ang. post-indexed)

Przykład

```
ldr x8, [x10], #4
```

Wartość spod adresu w pamięci jest wczytywana do rejestru



Fragment pamięci

Po wykonaniu rozkazu adres jest wpisywany do rejestru

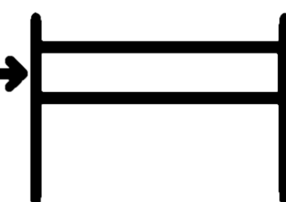
x10 = x10 + 4

Adresowanie pośrednie przez rejestr z preindeksowaniem (ang. pre-indexed)

Przykład

```
ldr x8, [x10, #4]!
```

Wartość spod adresu w pamięci jest wczytywana do rejestru



Wartość w rejestrze jest zwiększana o przesunięcie (w tym przykładzie 4)

→ x10 = x10 + 4

Fragment pamięci

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

STP

Przykładowa składnia
STP W_a, W_b, [X_n, #*offset*]
STP W_a, W_b, [SP, #*offset*]
STP X_a, X_b, [X_n, #*offset*]
STP X_a, X_b, [SP, #*offset*]

Przykładowe adresowanie
z preindeksowaniem
STP W_a, W_b, [X_n, #*offset*]!

Przykładowe adresowanie
z postindeksowaniem
STP W_a, W_b, [X_n], #*offset*

Rozkaz STP oblicza adres bazowy na podstawie wartości rejestru X_n/SP oraz wartości przesunięcia (ang. offset), aby do pamięci o tym adresie zapisać dwa 32-bitowe słowa pary rejestrów W_a/W_b lub dwa 64-bitowe podwójne słowa pary rejestrów X_a/X_b.

STR

Przykładowa składnia
STR X_t, [X_n, #*offset*]
STR X_t, [SP, #*offset2*]
STR W_t, [X_n, #*offset*]
STR W_t, [SP, #*offset2*]

Przykładowe adresowanie
z preindeksowaniem
STR X_t, [X_n, #*offset*]!

Przykładowe adresowanie
z postindeksowaniem
STR X_t, [X_n], #*offset*

Rozkaz STR oblicza adres bazowy na podstawie wartości rejestru X_n/SP oraz wartości przesunięcia (ang. offset), aby do pamięci o tym adresie zapisać słowo maszynowe lub podwójne słowo.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Instrukcje arytmetyczne

Pozostałe rozkazy są dokładnie opisane w dokumentacji. [3]

ADD

Przykładowa składnia
`ADD` X_d, X_a, X_b
`ADD` W_d, W_a, W_b
`ADD` X_d, X_a, *#immediate*
`ADD` W_d, W_a, *#immediate*

Rozkaz `ADD` dodaje wartości rejestrów X_a (lub W_a) i X_b (lub W_b) i zapisuje wynik w rejestrze docelowym X_d (lub W_d). Inna wersja składni umożliwia dodanie wartości rejestru X_a (lub W_a) oraz wartości natychmiastowej (ang. immediate) i zapisanie wyniku w rejestrze docelowym X_d (lub W_d).

CLS

Przykładowa składnia
`CLS` X_d, X_a
`CLS` W_d, W_a

Rozkaz `CLS` zlicza wiodące bity znaku w rejestrze X_a (lub W_a) i zapisuje rezultat operacji do rejestru docelowego X_d (lub W_d).

CLZ

Przykładowa składnia
`CLZ` X_d, X_a
`CLZ` W_d, W_a

Rozkaz `CLZ` zlicza wiodące bity zerowe w rejestrze X_a (lub W_a) i zapisuje rezultat operacji do rejestru docelowego X_d (lub W_d).

SUB

Przykładowa składnia
`SUB` X_d, X_a, X_b
`SUB` W_d, W_a, W_b
`SUB` X_d, X_a, *#immediate*
`SUB` W_d, W_a, *#immediate*

Rozkaz `SUB` odejmuje wartość rejestru X_a (lub W_a) od X_b (lub W_b) i zapisuje wynik w rejestrze docelowym X_d (lub W_d). Inna wersja składni umożliwia odjęcie wartości natychmiastowej (ang. immediate) od wartości rejestru X_a (lub W_a) i zapisanie wyniku w rejestrze docelowym X_d (lub W_d).

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

MUL

Przykładowa składnia
`MUL` X_d, X_a, X_b
`MUL` W_d, W_a, W_b

Rozkaz `MUL` mnoży wartość rejestru X_a przez X_b i zapisuje wynik w rejestrze docelowym X_d. Inna wersja składni (32-bitowa) pozwala użyć rejestrów W_n.

SDIV/UDIV

Przykładowa składnia
`SDIV/UDIV` X_d, X_a, X_b
`SDIV/UDIV` W_d, W_a, W_b

Rozkaz `SDIV` dzieli ze znakiem (`UDIV` dzieli bez znaku) wartość całkowitą z rejestru X_a (lub W_a) przez X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d). Instrukcja ta nie ma wpływu na flagi warunkowe procesora.

Instrukcje logiczne

Pozostałe rozkazy są dokładnie opisane w dokumentacji. [3]

AND

Przykładowa składnia
`AND` X_d, X_a, X_b
`AND` W_d, W_a, W_b
`AND` X_d, X_a, *#immediate*
`AND` W_d, W_a, *#immediate*

Rozkaz `AND` wykonuje koniunkcję logiczną wartości z rejestru X_a (lub W_a) i X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d). Inna wersja składni pozwala na wykonanie koniunkcji logicznej wartości z rejestru X_a (lub W_a) oraz wartości natychmiastowej (ang. immediate).

ASR

Przykładowa składnia
`ASR` X_d, X_a, X_b
`ASR` W_d, W_a, W_b
`ASR` X_d, X_a, *#immediate*
`ASR` W_d, W_a, *#immediate*

Rozkaz `ASR` wykonuje arytmetyczne przesunięcie w prawo wartości z rejestru X_a (lub W_a) o liczbę bitów zakodowaną na sześciu najmłodszych bitach rejestru X_b o zakresie od 0 do 63 (lub na pięciu najmłodszych bitach rejestru W_b o zakresie od 0 do 31).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Inna wersja składni ASR pozwala wykonać arytmetyczne przesunięcie w prawo o liczbę bitów podaną jako wartość natychmiastowa (ang. immediate) o zakresie od 0 do 63 (składnia 64-bitowa) lub od 0 do 31 (składnia 32-bitowa). Informacja: Przesunięcie arytmetyczne w prawo zachowuje bit znaku.

BIC

Przykładowa składnia
BIC X_d, X_a, X_b
BIC W_d, W_a, W_b

Rozkaz BIC wykonuje koniunkcję logiczną wartości z rejestru X_a (lub W_a) i zanegowanej wartości z rejestru X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d).

EON

Przykładowa składnia
EON X_d, X_a, X_b
EON W_d, W_a, W_b

Rozkaz EON wykonuje alternatywę wykluczającą wartości z rejestru X_a (lub W_a) i zanegowanej wartości z rejestru X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d).

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

EOR

Przykładowa składnia
EOR X_d, X_a, X_b
EOR W_d, W_a, W_b
EOR X_d, X_a, *#immediate*
EOR W_d, W_a, *#immediate*

Rozkaz EOR wykonuje alternatywę wykluczającą wartości z rejestru X_a (lub W_a) i wartości z rejestru X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d). Inna wersja składni pozwala na wykonanie tej samej operacji z użyciem wartości natychmiastowej (ang. immediate).

LSL

Przykładowa składnia
LSL X_d, X_a, X_b
LSL W_d, W_a, W_b
LSL X_d, X_a, *#immediate*
LSL W_d, W_a, *#immediate*

Rozkaz LSL wykonuje logiczne przesunięcie w lewo wartości z rejestru X_a (lub W_a) o liczbę bitów zakodowaną na sześciu najmłodszych bitach rejestru X_b o zakresie od 0 do 63 (lub na pięciu najmłodszych bitach rejestru W_b o zakresie od 0 do 31).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Inna wersja składni `LSL` pozwala wykonać logiczne przesunięcie w lewo o liczbę bitów podaną jako wartość natychmiastowa (ang. immediate) o zakresie od 0 do 63 (składnia 64-bitowa) lub od 0 do 31 (składnia 32-bitowa). Informacja: Na miejsce przesuniętych w lewo bitów wpisywane są zera.

LSR

Przykładowa składnia
`LSR` X_d, X_a, X_b
`LSR` W_d, W_a, W_b
`LSR` X_d, X_a, *#immediate*
`LSR` W_d, W_a, *#immediate*

Rozkaz `LSR` wykonuje logiczne przesunięcie w prawo wartości z rejestru X_a (lub W_a) o liczbę bitów zakodowaną na sześciu najmłodszych bitach rejestru X_b o zakresie od 0 do 63 (lub na pięciu najmłodszych bitach rejestru W_b o zakresie od 0 do 31). Inna wersja składni pozwala wykonać logiczne przesunięcie w prawo o liczbę bitów podaną jako wartość natychmiastowa (ang. immediate) o zakresie od 0 do 63 (składnia 64-bitowa) lub od 0 do 31 (składnia 32-bitowa).

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

MVN

Przykładowa składnia
`MVN` X_d, X_a
`MVN` W_d, W_a

Rozkaz `MVN` wykonuje logiczne zaprzeczenie wartości z rejestru X_a (lub W_a) zapisując rezultat do rejestru docelowego X_d (lub W_d).

NEG

Przykładowa składnia
`NEG` X_d, X_a
`NEG` W_d, W_a

Rozkaz `NEG` odejmuje wartość z rejestru X_a (lub W_a) od zera zapisując rezultat do rejestru docelowego X_d (lub W_d).

ORN

Przykładowa składnia
`ORN` X_d, X_a, X_b
`ORN` W_d, W_a, W_b

Rozkaz `ORN` wykonuje alternatywę wartości z rejestru X_a (lub W_a) i zanegowanej wartości z rejestru X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

ORR

Przykładowa składnia
`ORR` X_d, X_a, X_b
`ORR` W_d, W_a, W_b
`ORR` X_d, X_a, *#immediate*
`ORR` W_d, W_a, *#immediate*

Rozkaz `ORR` wykonuje alternatywę logiczną wartości z rejestru X_a (lub W_a) i wartości z rejestru X_b (lub W_b) zapisując wynik w rejestrze docelowym X_d (lub W_d). Inna wersja składni pozwala na wykonanie tej samej operacji z użyciem wartości natychmiastowej (ang. immediate).

RBIT

Przykładowa składnia
`RBIT` X_d, X_a
`RBIT` W_d, W_a

Rozkaz `RBIT` odwraca kolejność bitów w rejestrze X_a (lub W_a) i zapisuje wynik w rejestrze docelowym X_d (lub W_d).

ROR

Przykładowa składnia
`ROR` X_d, X_a, X_b
`ROR` W_d, W_a, W_b
`ROR` X_d, X_a, *#immediate*
`ROR` W_d, W_a, *#immediate*

Rozkaz `ROR` wykonuje obrót w prawo bitów z rejestru X_a (lub W_a) o liczbę bitów zakodowaną na sześciu najmłodszych bitach rejestru X_b o zakresie od 0 do 63 (lub na pięciu najmłodszych bitach rejestru W_b o zakresie od 0 do 31). Inna wersja składni pozwala wykonać logiczne przesunięcie w lewo o liczbę bitów podaną jako wartość natychmiastowa (ang. immediate) o zakresie od 0 do 63 (składnia 64-bitowa) lub od 0 do 31 (składnia 32-bitowa). Informacja: Bity wyrzucone z prawej strony podczas obrotu wchodzą z lewej strony z powrotem do rejestru.

Dokładne informacje o składni można znaleźć w dokumentacji. [3]

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Rozgałęzienia (ang. branch)

Pozostałe rozkazy są dokładnie opisane w dokumentacji. [3]

B

Przykładowa składnia
`B` *label*

Rozgałęzienie. Rozkaz `B` (ang. branch) wykonuje przejście do określonego miejsca w kodzie oznaczonego etykietą. Możliwe jest przejście wstecz lub do przodu o zakresie ±128 MB licząc od adresu bieżącej instrukcji.

BL

Przykładowa składnia
`BL` *label*

Rozgałęzienie z adresem powrotnym. Rozkaz `BL` (ang. branch with link) wykonuje przejście do określonego miejsca w kodzie oznaczonego etykietą i ustawia wartość rejestru `X30` (`LR`) na wartość rejestru `PC + 4`. Możliwe jest przejście wstecz lub do przodu o zakresie ±128 MB licząc od adresu bieżącej instrukcji.

BLR

Przykładowa składnia
`BLR` X_n

Rozgałęzienie z użyciem rejestru z adresem powrotnym. Rozkaz `BLR` (ang. branch with link to register) wykonuje przejście do określonego miejsca w kodzie, którego adres jest w rejestrze ogólnego przeznaczenia X_n i ustawia wartość rejestru `X30` (`LR`) na wartość rejestru `PC + 4`. Możliwe jest przejście wstecz lub do przodu o zakresie ±128 MB licząc od adresu bieżącej instrukcji.

BR

Przykładowa składnia
`BR` X_n

Rozgałęzienie z użyciem rejestru. Rozkaz `BR` (ang. branch to register) wykonuje przejście do określonego miejsca w kodzie, którego adres jest w rejestrze ogólnego przeznaczenia X_n. Możliwe jest przejście wstecz lub do przodu o zakresie ±128 MB licząc od adresu bieżącej instrukcji.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

BEQ

Przykładowa składnia
`BEQ` *label*

Rozgałęzienie warunkowe, jeśli równe. Rozkaz `BEQ` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy ustawiona jest flaga zerowa (`Z`). Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BNE

Przykładowa składnia
`BNE` *label*

Rozgałęzienie warunkowe, jeśli różne. Rozkaz `BNE` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flaga zerowa (`Z`) nie jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BCS/BHS

Przykładowa składnia
`BCS/BHS` *label*

Rozgałęzienie warunkowe, jeśli przeniesienie. Rozkazy `BCS` i `BHS` wykonują warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy ustawiona jest flaga przeniesienia (`C`), co można rozumieć jako większe bądź równe bez znaku. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BCC/BLO

Przykładowa składnia
`BCC/BLO` *label*

Rozgałęzienie warunkowe, jeśli brak przeniesienia. Rozkazy `BCC` i `BLO` wykonują warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flaga przeniesienia (`C`) nie jest ustawiona, co można rozumieć jako mniejsze bez znaku. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

BMI

Przykładowa składnia
`BMI` *label*

Rozgałęzienie warunkowe, jeśli ujemny. Rozkaz `BMI` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy ustawiona jest flaga ujemna (`N`). Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BPL

Przykładowa składnia
`BPL` *label*

Rozgałęzienie warunkowe, jeśli dodatni lub zero. Rozkaz `BPL` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flaga ujemna (`N`) nie jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BVS

Przykładowa składnia
`BVS` *label*

Rozgałęzienie warunkowe, jeśli przepełnienie. Rozkaz `BVS` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy ustawiona jest flaga przepełnienia (`V`). Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BVC

Przykładowa składnia
`BVC` *label*

Rozgałęzienie warunkowe, jeśli brak przepełnienia. Rozkaz `BVC` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flaga przepełnienia (`V`) nie jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

BHI

Przykładowa składnia
`BHI` *label*

Rozgałęzienie warunkowe, jeśli większe bez znaku. Rozkaz `BHI` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flaga przeniesienia (`C`) jest ustawiona, a flaga zerowa (`Z`) nie jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BLS

Przykładowa składnia
`BLS` *label*

Rozgałęzienie warunkowe, jeśli mniejsze bądź równe bez znaku. Rozkaz `BLS` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flaga przeniesienia (`C`) nie jest ustawiona, a flaga zerowa (`Z`) jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BGE

Przykładowa składnia
`BGE` *label*

Rozgałęzienie warunkowe, jeśli większe bądź równe ze znakiem. Rozkaz `BGE` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flagi `N` i `V` mają taką samą wartość. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BLT

Przykładowa składnia
`BLT` *label*

Rozgałęzienie warunkowe, jeśli mniejsze ze znakiem. Rozkaz `BLT` wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flagi `N` i `V` mają różne od siebie wartości. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

BGT

Przykładowa składnia
BGT *label*

Rozgałęzienie warunkowe, jeśli większe ze znakiem. Rozkaz BGT wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flagi N i V mają taką samą wartość, a flaga Z nie jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BAL

Przykładowa składnia
BAL *label*

Rozkaz BAL wykonuje bezwarunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą. Przyrostek (kod warunkowy AL, ang. always) jest przeważnie pomijany i stosuje się instrukcję rozgałęzienia bezwarunkowego B. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

BLE

Przykładowa składnia
BLE *label*

Rozgałęzienie warunkowe, jeśli mniejsze ze znakiem. Rozkaz BLE wykonuje warunkowe przejście do określonego miejsca w kodzie oznaczonego etykietą, gdy flagi N i V mają różne od siebie wartości oraz flaga Z jest ustawiona. Możliwe jest przejście wstecz lub do przodu o zakresie ±1 MB licząc od adresu bieżącej instrukcji.

RET

Przykładowa składnia
RET X_n

Powrót z rozgałęzienia. Rozkaz RET wykonuje powrót do określonego miejsca w kodzie, którego adres jest w rejestrze ogólnego przeznaczenia X_n. Domyślnie adres instrukcji do której ma nastąpić powrót jest pobierany z rejestru X30 (LR), wtedy rozkaz jest wywoływany bez operandu.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Inne instrukcje (ang. misc)

Pozostałe rozkazy są dokładnie opisane w dokumentacji. [3]

REV

Przykładowa składnia
REV64 X_d, X_a
REV X_d, X_a
REV W_d, W_a

Odwrócenie kolejności bajtów w rejestrze. Rozkaz REV odwraca kolejność bajtów w rejestrze X_a (lub W_a) i zapisuje wynik w rejestrze docelowym X_d (lub W_d).

CMP

Przykładowa składnia
CMP X_d, X_a, X_b
CMP W_d, W_a, W_b
CMP X_d, X_a, *#immediate*
CMP W_d, W_a, *#immediate*

Rozkaz CMP wykonuje porównanie wartości dwóch operandów przez wykonanie odejmowania (SUB) z ustawieniem flag, ale bez zapisywania wyniku. Po tej instrukcji przeważnie występują rozkazy warunkowe.

TST

Przykładowa składnia
TST X_d, X_a, X_b
TST W_d, W_a, W_b
TST X_d, X_a, *#immediate*
TST W_d, W_a, *#immediate*

Rozkaz TST wykonuje porównanie wartości dwóch operandów przez wykonanie koniunkcji logicznej (AND) z ustawieniem flag, ale bez zapisywania wyniku. Po tej instrukcji przeważnie występują rozkazy warunkowe. Wskazówka: TST X_n, X_n pozwala łatwo sprawdzić czy wartość rejestru X_n to zero.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Wykaz literatury

Dokument przedstawiony poniżej to literatura obowiązkowa dla chcących zgłębić tematykę języka Asembler arm64 (AArch64) w systemach Microsoft Windows.

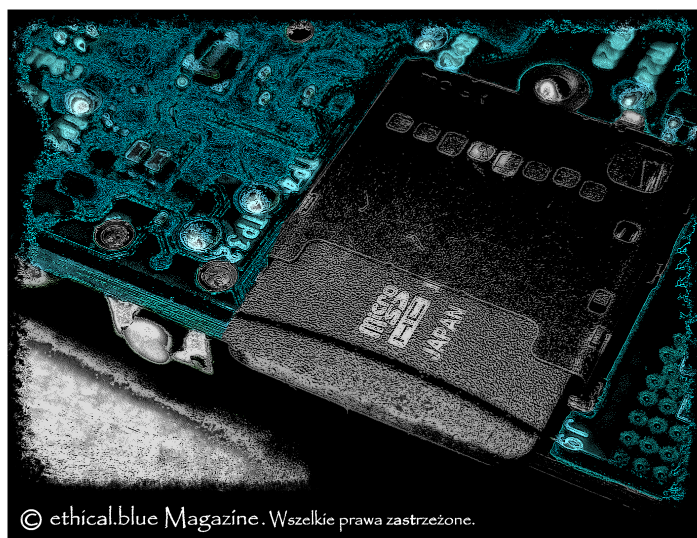
- Arm Limited, *Arm Architecture Reference Manual for A-profile architecture*, 2025.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

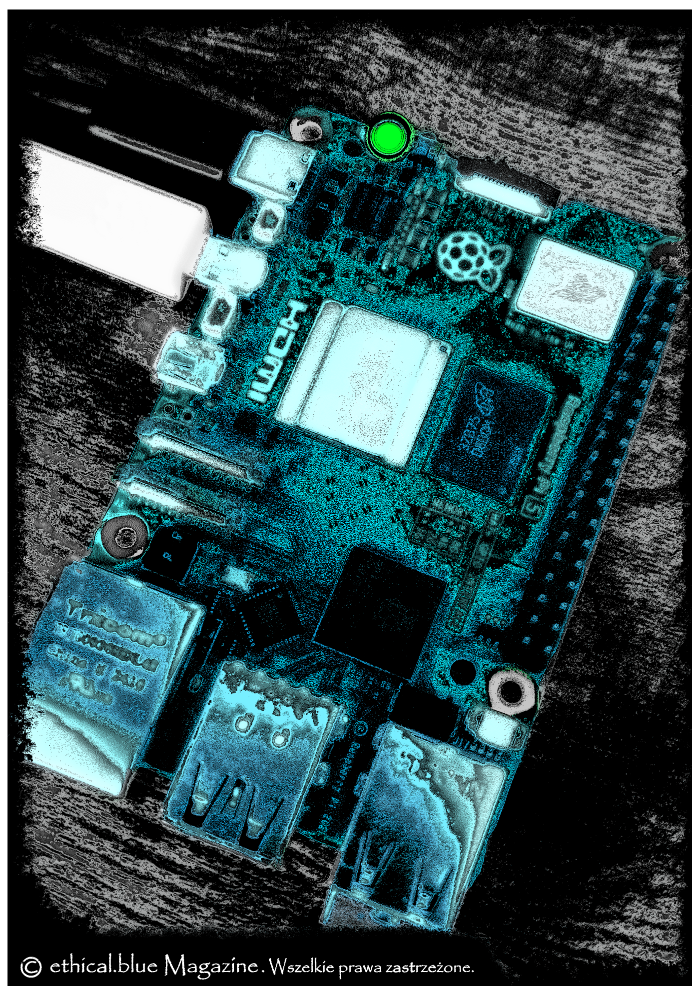
Leśny sad. Asembler arm32/arm64 na urządzeniach Raspberry Pi

— Urządzenia Raspberry Pi są niesamowite i zużywają mało energii elektrycznej. Te małe układy dają naprawdę ogromne możliwości. W kolejnym eksperymencie spróbujemy zbudować program w języku Asembler na urządzenie Raspberry Pi oraz nauczymy się używać deasemblera. — oznajmił pracownik laboratorium.

Fotografie 44 i 45 przedstawiają urządzenie Raspberry Pi.



Fotografia 44. Karta pamięci z systemem operacyjnym Raspberry Pi OS



Fotografia 45. Urządzenie Raspberry Pi

Po umieszczeniu karty pamięci z systemem w wyznaczonym miejscu, podłączeniu Raspberry Pi do zasilania oraz podłączeniu myszy, klawiatury i monitora można wcisnąć przycisk uruchamiający urządzenie.

Po uruchomieniu narzędzia Terminal można wpisać polecenie `ls`, aby zobaczyć foldery znajdujące się w bieżącym katalogu. Przykładowy plik z kodem źródłowym utworzymy w folderze Pulpit, czyli za pomocą polecenia `cd` (ang. change directory) należy zmienić katalog na `Desktop`.

Utworzenie pustego pliku jest możliwe np. poleceniem `touch ethical.s`. Teraz, aby wprowadzić kod źródłowy do pliku `ethical.s` można wpisać `mousepad ethical.s`, co spowoduje uruchomienie edytora tekstowego z graficznym interfejsem użytkownika (Fotografia 46).

Przykładowy program (Scriptum 75) wypisuje napis zakończony znakiem nowej linii i bajtem zerowym na standardowe wyjście, czyli w tym przypadku konsola tekstowa. Na początku dyrektywą `.global` eksportowany jest punkt wejścia (ang. entry point) o nazwie `_start`. Później w sekcji z danymi (`.data`) tworzona jest etykieta o nazwie `ethicalblueText`, która służy do oznaczenia miejsca ciągu znaków zdefiniowanego dyrektywą `.asciz`. Poniżej dyrektywa `.equ` definiuje symbol o nazwie `size`, który zawiera długość napisu `ethicalblueText` w bajtach. Rozmiar ten jest obliczany przez odjęcie (`-`) od bieżącego miejsca w programie (kropka, `.`), adresu etykiety o nazwie `ethicalblueText`.

Program zawiera dwa wywołania systemowe (ang. syscall) przeprowadzane za pomocą rozkazu `SVC`. W rejestrach `r0`, `r1`, `r2`... przekazywane są parametry, a w rejestrze `r7` numer wywołania systemowego. Wartość cztery (`mov r7, #4`) odpowiada za operację zapisu, w tym przykładzie na standardowe wyjście. Natomiast wartość jeden (`mov r7, #1`) powoduje zakończenie programu.

```
.global _start

.data
ethicalblueText:
  .asciz "ethical.blue Magazine\n"
.equ size, . - ethicalblueText

.text
_start:
  mov r0, #1
  ldr r1, =ethicalblueText
  ldr r2, =size
  mov r7, #4
  svc 0

  mov r0, #0
  mov r7, #1
  svc 0
```

Scriptum 75. Przykładowy program w języku Asembler (The GNU Assembler, Raspberry Pi)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Fotografia 46. Napisanie przykładowego programu w języku Asembler (Raspberry Pi)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

W celu zbudowania pliku wykonywalnego ELF można użyć narzędzia The GNU Assembler (Fotografia 47). Najpierw należy ustawić katalog bieżący poleceniem `cd` (ang. change directory). Jeśli w katalogu Pulpit znajduje się utworzony wcześniej plik z kodem źródłowym `ethical.s`, to wystarczy wywołać polecenia przedstawione poniżej (Scriptum 76).

```
as -march=armv9-a ethical.s -o ethical.o
ld ethical.o -o ethical.elf
```

Scriptum 76. Budowanie programu w języku Asembler (The GNU Assembler, Raspberry Pi)

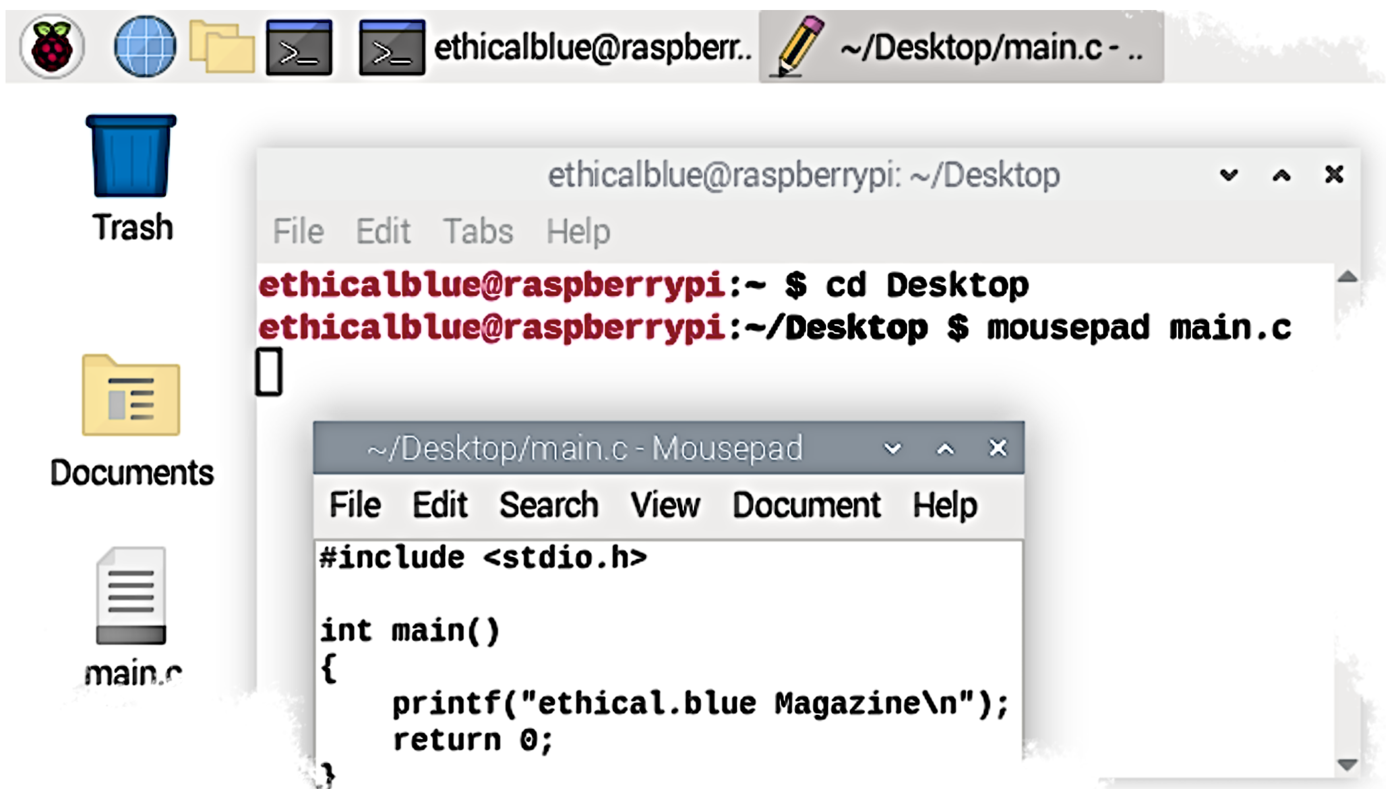
Teraz, aby uruchomić powstały plik ELF należy wydać polecenie `./ethical.elf`, które uruchamia program o nazwie `ethical.elf` z bieżącego katalogu (znak kropki, `./`).

Fotografia 47. Budowanie i uruchomienie przykładowego programu w języku Asembler (Raspberry Pi)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Odpowiednie parametry, ustawione podczas budowania programu w języku C lub podobnym, pozwalają na uzyskanie pliku z kodem w języku Asembler.

Za przykład może posłużyć prosty program w języku C, który wyświetla tekst na standardowe wyjście, czyli tutaj konsolę tekstową. Wystarczy otworzyć edytor taki jak np. mousepad i zapisać przykładowy kod źródłowy w pliku z rozszerzeniem .c, czyli domyślnie `main.c` (Fotografia 48).



Fotografia 48. Napisanie przykładowego programu w języku C (Raspberry Pi)

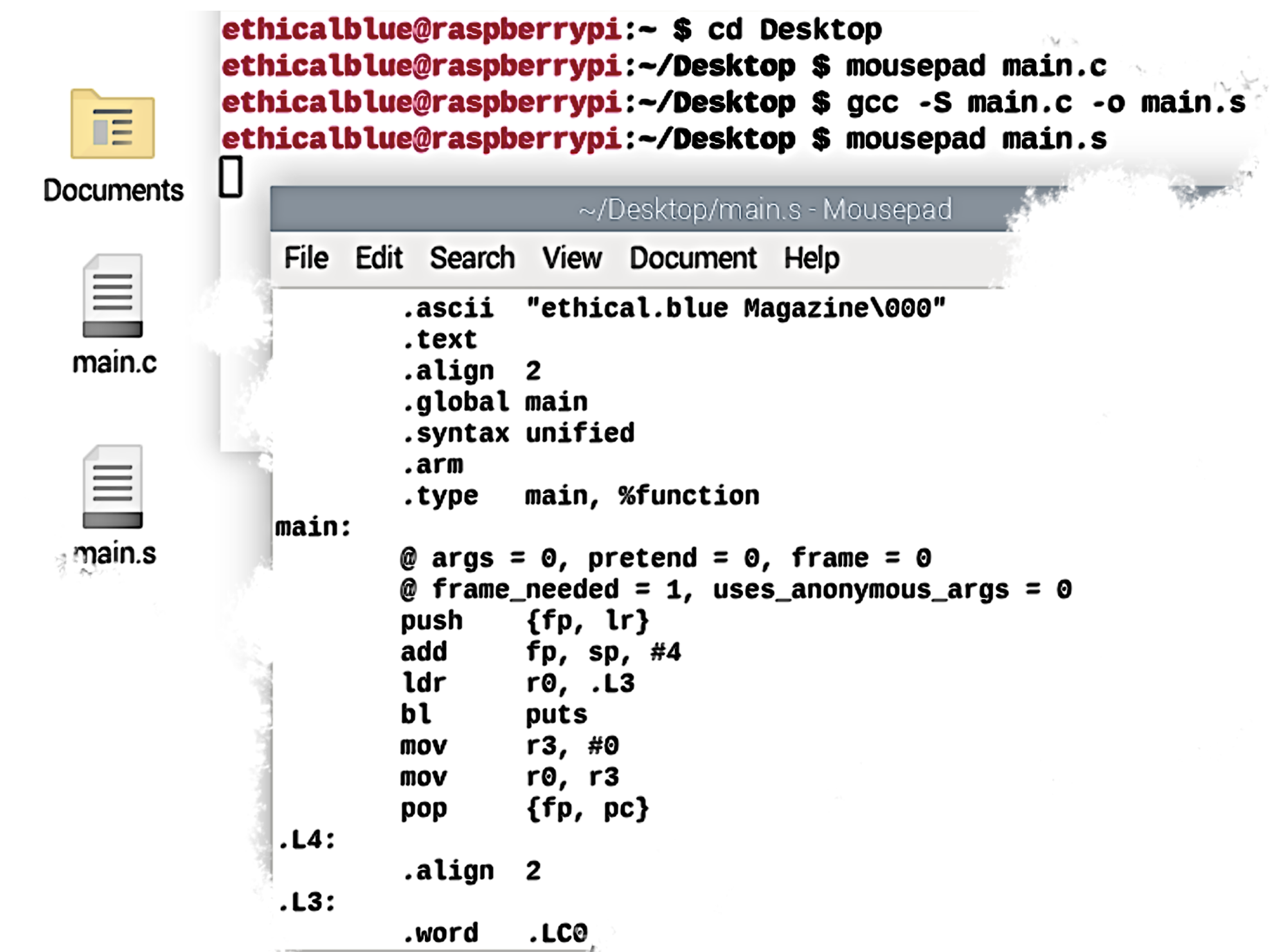
© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Uzyskanie kodu w Asemblerze z pliku źródłowego w języku C

W celu uzyskania kodu w języku Asembler można wywołać narzędzie gcc z parametrem -S (wielka litera S) tak jak przedstawiono poniżej (Fotografia 49, Scriptum 77).

```
gcc -S main.c -o main.s
```

Scriptum 77. Wywołanie narzędzia gcc z parametrem -S (GNU Compiler Collection)



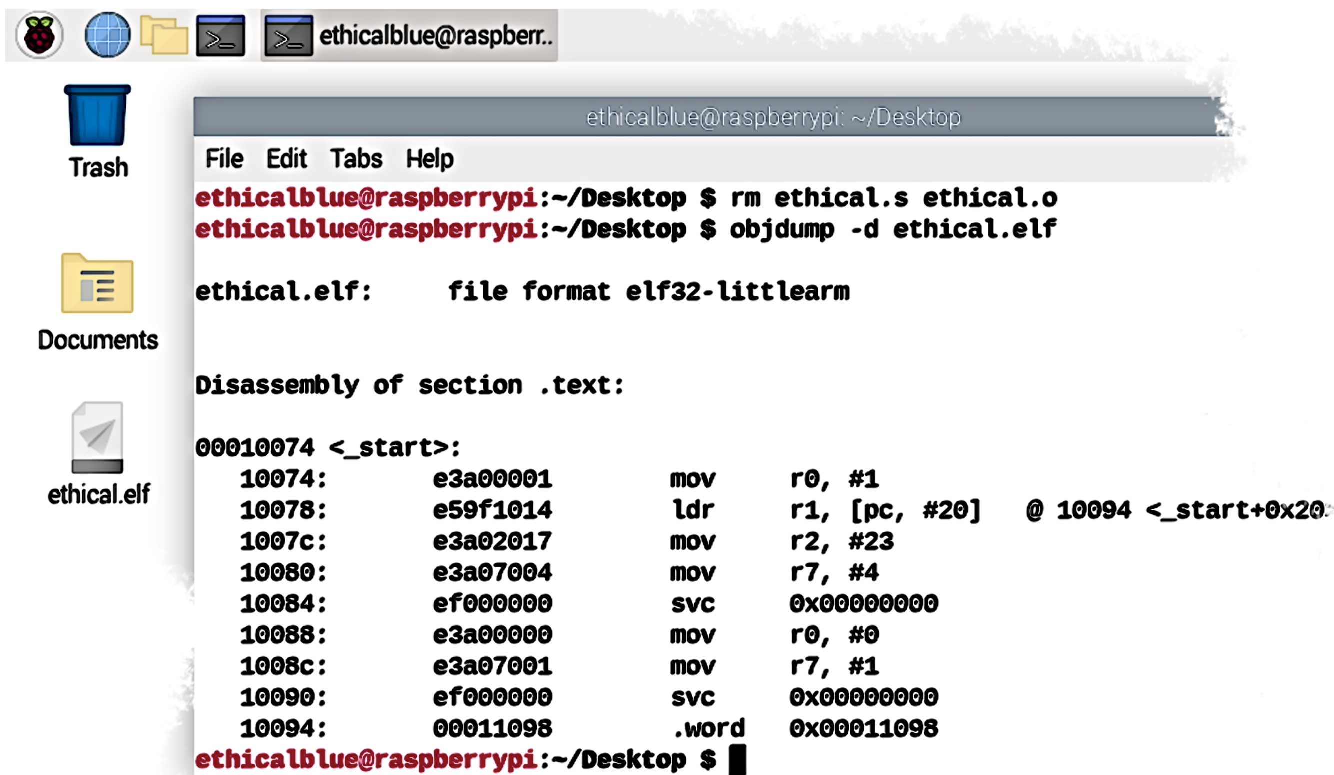
Fotografia 49. Przykładowe wywołanie narzędzia gcc, aby uzyskać kod w Asemblerze (Raspberry Pi)

Deasemblacja za pomocą narzędzia objdump

— Kolejny eksperyment będzie polegał na próbie odzyskania kodu w Asemblerze z pliku wykonywalnego ELF. — oznajmił pracownik laboratorium.

— Usuwamy pliki z kodem źródłowym i obiektowe za pomocą polecenia `rm ethical.s ethical.o` tak jak na Fotografii 50, żeby było bardziej realistycznie. — kontynuował.

— Teraz wywołaniem narzędzia `objdump -d ethical.elf` odzyskujemy kod w Asemblerze. Nazywane jest to inżynierią wsteczną (ang. reverse engineering) i często jest utrudniane na różne sposoby od prostego zaciemniania (ang. obfuscation), aż po tworzenie warstw abstrakcji implementując skomplikowane maszyny wirtualne i zaawansowaną ochronę kodu. — powiedział pracownik laboratorium podsumowując eksperyment.



Fotografia 50. Przykładowe wywołanie narzędzia objdump, aby odzyskać kod w Asemblerze (Raspberry Pi)

Wykaz literatury

Dokumenty przedstawione poniżej to literatura obowiązkowa dla chcących zgłębić tematykę języka Asembler arm64 (AArch64).

- Arm Limited, *Arm Architecture Reference Manual for A-profile architecture*, 2025.
- Arm Limited, *ELF for the Arm 64-bit Architecture (AArch64)*, 2025.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Technologia łańcucha bloków (ang. blockchain) z przykładami w języku C#

Solidne podstawy działania łańcucha bloków i opartych na nim walut kryptograficznych, takich jak Bitcoin i inne, mogą okazać się niezbędne w świecie przesiąkniętym coraz to nowszymi technologiami.

W latach 2008/2009 powstaje zabezpieczona kryptograficznie waluta o nazwie Bitcoin. Potwierdza to m.in. przykładowe zapytanie do bazy danych WHOIS.

```
C:\Sysinternals\whois.exe
bitcoin.org
...
Creation Date:
2008-08-18T13:19:55Z
...
```

Znak reprezentujący kryptowalutę Bitcoin jest zawarty w standardzie Unicode i ma wartość 0x20BF. W systemach z rodziny Microsoft Windows symbol ten można wygodnie wybrać z Tablicy znaków przez wpisanie `charmap.exe` w Wierszu polecenia. Inny sposób to wpisanie w konsoli PowerShell np. `[char]::ConvertFromUtf32(0x20BF);`, czyli wywołanie metody konwertującej wartość liczbową na typ znakowy `char`.

Podstawy kryptografii asymetrycznej

Algorytmy szyfrowania posiadające ten sam klucz zarówno do szyfrowania jak i deszyfrowania nazywane są symetrycznymi. Natomiast kryptografia asymetryczna nazywana też kryptografią z kluczem publicznym stosuje ich dwa rodzaje: prywatny i publiczny. Najczęstsze zastosowanie to szyfrowanie wiadomości kluczem publicznym przez nadawcę i deszyfrowanie kluczem prywatnym przez odbiorcę. A w przypadku podpisów: podpisywanie kluczem prywatnym (przez autora), a weryfikacja poprawności podpisu kluczem publicznym (przez każdego).

Jeśli mowa o zabezpieczonej kryptograficznie walucie, to klucz prywatny pozwala na podpisywanie transakcji i transfer środków. Natomiast klucz publiczny jest powiązany z adresem portfela i potrzebny do odebrania przesyłanych monet. Udostępnianie swojego adresu zbyt szerokiemu gronu może powodować śledzenie ile jest na nim środków.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Zgubione monety (ang. lost coins)

Podobnie jak zwykłe monety, kryptowaluta też może zostać zgubiona. W przypadku utraty klucza prywatnego do portfela można powiedzieć, że środki zapisane w łańcuchu bloków zostały zgubione. Brak do nich dostępu, czyli nie można ich przetransferować, więc tkwią w łańcuchu bloków jako praktycznie niczyje. Dość często popełnianym błędem przez nowicjuszy jest wybranie nieprawidłowej sieci dla transakcji, pomyłka podczas przepisywania adresu odbiorcy lub zainfekowanie swojego urządzenia złośliwym oprogramowaniem (ang. malware).

Sieć zdecentralizowana

Urządzenia połączone w sieć i tworzące logiczną całość nazywane są systemami rozproszonymi. Wiele rozwiązań korzysta z tego rodzaju architektury w mniejszym lub większym stopniu. Podstawowa korzyść to minimalizacja braku dostępności usługi w przypadku awarii części infrastruktury. Łańcuch bloków (ang. blockchain) implementowany w walutach zabezpieczonych kryptograficznie to zdecentralizowana baza danych. Urządzenie na którym uruchomiono program do przetwarzania transakcji staje się węzłem (ang. node). Mogą istnieć pełne węzły, które przetwarzają transakcje i posiadają kopię całości rozproszonej bazy danych. Spotyka się też węzły, które nie przechowują kopii całego łańcucha bloków, a zajmują się jedynie weryfikacją nowych transakcji.

W prostych słowach: Jeśli istnieje np. 100 tys. pełnych węzłów, to baza danych posiada podobną liczbę kopii zapasowych swojej zawartości. Utrudnia to jej całkowite usunięcie czy fałszowanie. Jednak wpisy powinny być w jakiś sposób weryfikowane, a węzły osiągać zgodność co do zawartości tego rozproszonego rejestru. Sytuacja tego typu porównywana jest do dowodzenia jednostkami na polu bitwy i nazywana *Problem generałów bizantyjskich* [15]. Powstałe na przestrzeni czasu rozwiązania stosują różne algorytmy i poziom tolerancji błędu, aby decyzja o przyjęciu bloku do łańcucha była słuszna.

Blok zerowy (ang. genesis block)

Bloki w łańcuchu są połączone ze sobą, a kolejne są zależne od poprzednich. Zabezpieczenia kryptograficzne i połączenie wpisów w ten sposób powodują, że nie można dowolnie modyfikować bloków bez naruszenia integralności całego łańcucha. Istnieje jednak blok nazywany *genesis* i jest to wpis w łańcuchu, który nie ma poprzednika. Blok zerowy jest przeważnie utrwalany i rozpowszechniany, aby powstał silny dowód, że wpis jest autentyczny i na prawdę istniał na przestrzeni czasu.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Integralność bloku danych

Podstawowym zastosowaniem funkcji skrótu (ang. hash) jest ochrona integralności bloku danych, a jest to możliwe, ponieważ operacja hash'owania jest jednokierunkowa oraz nawet minimalna zmiana w danych powoduje całkowicie inną wartość skrótu. Określenie, że wspomniana operacja jest jednokierunkowa oznacza brak odwrotnego algorytmu, który daje możliwość otrzymania oryginalnych danych posiadając wartość z funkcji skrótu.

Pozostają jedynie siłowe próby losowania potencjalnie prawidłowych danych i porównywanie czy otrzymana wartość skrótu jest zgodna lub znalezienie kolizji. Jeśli stosowany algorytm ma podatność związaną z bezpieczeństwem, to może wystąpić kolizja, która oznacza taką samą wartość skrótu (ang. digest) dla różnych danych.

Secure Hash Algorithm (SHA)

Wartość funkcji skrótu SHA-512 dla przykładowego tekstu `ethical.blue Magazine` to następujący ciąg bajtów w postaci szesnastkowej `0b596f4a3cebea1ba38801e5958 6fefaf2592c5c3f0b06dd6800aead3f5fc4 664827a81d3e46f2655cc8526a50a7e5a5f 82f3434d3f06affdc576804a568bbfe`.

Blokiem danych przekazywanym do funkcji skrótu (ang. hash) najczęściej jest ciąg tekstowy lub plik na dysku urządzenia i chociaż składnia wywoływanych poleceń jest różna, to nie należy zapominać, że *under the hood* informacje w pamięci komputerowej to po prostu bajty.

```
printf "ethical.blue Magazine" |
   sha512sum | tr -d " -"
```

Scriptum 78. Uzyskanie skrótu SHA-512 z przykładowego ciągu znaków (Kali Linux)

W systemie Kali Linux funkcję skrótu SHA-512 można obliczyć np. poleceniem `sha512sum`, które domyślnie jako parametr przyjmuje nazwę pliku. Jednak nic nie stoi na przeszkodzie, aby obliczyć też *hash* z ciągu tekstowego łącząc polecenia za pomocą znaku pionowej kreski (|).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

W ten sposób utworzony zostanie potok (ang. pipe), który standardowe wyjście polecenia `printf` przekaże bezpośrednio do wejścia `sha512sum` i dalej do `tr`, aby usunąć zbędne znaki: spacja oraz minus (Scriptum 78).

Natomiast w systemach z rodziny Microsoft Windows funkcję skrótu SHA-512 można obliczyć np. używając powłoki PowerShell.

```
[System.Convert]::ToHexString([
    System.Security.Cryptography.
    SHA512]::HashData([System.Text.
    Encoding]::UTF8.GetBytes("ethical
    .blue Magazine"))).ToLower();
```

Scriptum 79. Uzyskanie skrótu SHA-512 z przykładowego ciągu znaków (::HashData, PowerShell)

Przykład (Scriptum 79) korzystający z platformy .NET oparty jest o klasę `SHA512`. W nawiasach prostokątnych są typy (klasy) udostępniane przez środowisko, a za pomocą znaków `::` można uzyskać dostęp do wybranej metody i wywołać ją podając wymagane parametry w nawiasach okrągłych.

```
[System.IO.MemoryStream]::new([
    System.Text.Encoding]::UTF8.
    GetBytes("ethical.blue Magazine")
    ) | % { (Get-FileHash -
    InputStream $_ -Algorithm SHA512)
    .Hash.ToLower(); }
```

Scriptum 80. Uzyskanie skrótu SHA-512 z przykładowego ciągu znaków (Get-FileHash, PowerShell)

Inny przykład (Scriptum 80) pozwalający utworzyć *hash* z określonego ciągu znaków, oparty jest o potok |, iterację po obiektach `%` i *command-let* o nazwie `Get-FileHash`. Domyślnie polecenie `Get-FileHash` odczytuje dane z pliku na dysku, ale za pomocą parametru `InputStream` można zmienić odczytywany strumień danych.

`[System.IO.MemoryStream]::new` tworzy strumień z bajtów napisu w kodowaniu UTF-8, który potokiem (ang. pipe) jest przekazywany do pętli iterującej po obiektach (%). Blok instrukcji, którego początek i koniec jest oznaczony nawiasami klamrowymi zawiera wywołanie *command-let* o nazwie `Get-FileHash`. Parametr `InputStream` określa, że strumieniem jest bieżący obiekt `$_` przekazany potokiem | do pętli %. Natomiast parametr `Algorithm` określa użycie w tym przypadku *Secure Hash Algorithm* (SHA) o długości skrótu 512 bitów.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Z uwagi na fakt, że zwracany rezultat wywołanego *command-let* o nazwie `Get-FileHash` jest obiektem, to może zostać otoczony nawiasami okrągłymi w celu uzyskania dostępu do właściwości o nazwie `Hash`. Można też dodatkowo zamienić wszystkie litery w ciągu na małe za pomocą metody `ToLower`.

Dowód pracy (ang. Proof of Work)

Dla urządzeń z trzeciej dekady XXI wieku uzyskanie skrótu SHA-512 nie wymaga wiele mocy obliczeniowej. Dzięki temu można w mgnieniu oka weryfikować integralność danych.

W klasycznych rozwiązaniach, aby otrzymać pozwolenie na dodanie nowego bloku do łańcucha, program typu *miner* (pol. górnik) rozwiązuje puzzle kryptograficzne o określonej trudności. Nazywane jest to dowodem pracy (ang. Proof of Work) i polega na znalezieniu wartości określanej *nonce* dla której wynik funkcji skrótu spełnia wymagania sieci [15].

Dowód wkładu (ang. Proof of Stake)

Innym wymaganiem od dowodu pracy może być dowód wkładu. Oznacza to, że operacje na łańcuchu bloków mogą wykonywać węzły, które posiadają określoną liczbę danej kryptowaluty [15].

Dowód spalenia (ang. Proof of Burn)

Trochę podobnym wymaganiem do dowodu wkładu (ang. Proof of Stake) może być warunek wejścia w posiadanie, a następnie zniszczenie części kryptowaluty (ang. Proof of Burn), co może spowodować deflację, czyli większą wartość wirtualnych monet dla tych co je posiadają. Opisywane spalenie jest dokonywane przez wysłanie monet na specjalny adres.

System karuzelowy (ang. Round Robin)

Łańcuch bloków może wymagać dodatkowych uprawnień, aby dodawać nowe wpisy. Jednak nawet wśród zaufanych węzłów należy chronić rejestr przed szkodliwymi działaniami. Pomaga w tym system karuzelowy, dzięki któremu węzły dodają nowe wpisy w turach. Tak jakby były ustawione w kolejce i oczekiwały na swoją możliwość wprowadzenia danych [15].

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Autorytet jednostek (ang. Proof of Authority)

Możliwość publikowania nowych bloków może być oparta o autorytet węzłów, czyli ich powiązanie z jednostkami w rzeczywistym świecie. Dzięki temu możliwe jest zidentyfikowanie i ukaranie w jakiś sposób węzłów działających na szkodę sieci [15].

Dowód upływu czasu (ang. Proof of Elapsed Time)

Dodatkowym zabezpieczeniem przed dominacją w łańcuchu bloków przez niektóre węzły jest ustalanie losowego czasu oczekiwania na otrzymanie pozwolenia. Czas powinien być weryfikowany sprzętowo w bezpieczny sposób, aby nie było możliwe pominięcie oczekiwania na swoją turę [15].

Opłata paliwowa (ang. gas fee)

Przetworzenie i weryfikacja transakcji w łańcuchu bloków wymaga mocy obliczeniowej. Z tego powodu przyjęło się, aby ułamek kwoty przetwarzanej transakcji przeznaczyć na pokrycie kosztów operacji. W sieci o nazwie Ethereum opłata ta jest określana terminem *gas fee* (pol. koszty paliwa) [7].

Moneta stabilna (ang. stablecoin)

Różne kryptowaluty określane po angielsku jako *stablecoin* są w pewnym sensie zabezpieczone przed utratą wartości i niestabilnością. Na przykład USDC (USD Coin) czy USDT (USD Tether) mają odwzorowywać wartość dolara, a PAXG (Pax Gold) ma być zabezpieczone złotem. Często do waluty typu *stablecoin* zamienia się monety po wykonanej operacji handlu, aby zatrzymać osiągnięty zysk.

Kopalnie (ang. mining pools)

Wzrost poziomu trudności wydobywania kryptowaluty powoduje, że indywidualne próby mogą stawać się nieskuteczne. Istnieją jednak tak zwane kopalnie (ang. pools), które pozwalają wspólnie wspierać sieć i dzielą między uczestników (nazywanych górnikami czy kopaczami) otrzymane nagrody za zweryfikowane transakcje.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Zmiany w protokole i strukturach danych (ang. forks)

Waluty zabezpieczone kryptograficznie oparte o łańcuch bloków działają na zdefiniowanym protokole i strukturach danych. W uproszczeniu można to porównać do określonego przez twórców sposobów i kanałów komunikacji między węzłami w sieci.

Warto zaznaczyć, że nie tylko zmiany wprowadzane przez twórców mogą powodować tak zwane *rozwidlenia* (ang. forks). Fragment określonego projektu może się odłączyć od reszty np. z powodu braku akceptacji działań na szkodę sieci lub incydentu związanego z bezpieczeństwem. Może to mieć postać np. wydzielenia fragmentu łańcucha bloków do określonego momentu, porzucenie wpisów dodanych później i kontynuację prowadzenia rejestru od wyznaczonego miejsca przez część węzłów.

Post-Quantum Era

Jeśli urządzenia wykonujące obliczenia w oparciu o fizykę kwantową będą dostępne na szeroką skalę, to wyznaczą początek nowej ery dla kryptografii i bezpieczeństwa cybernetycznego. W przypadku zagrożenia ze strony urządzeń kwantowych prawdopodobnie będą potrzebne twarde zmiany w protokole i strukturach danych (ang. hard fork).

W przypadku funkcji skrótu takich jak SHA (Secure Hash Algorithm), aby zwiększyć odporność na zagrożenia *post-quantum*, należy stosować dłuższy *hash* [15].

Algorytmy kryptograficzne jak RSA (Rivest-Shamir-Adleman), ECDSA (Elliptic Curve Digital Signature Algorithm), ECDH (Elliptic-Curve Diffie–Hellman) i DSA (Digital Signature Algorithm) nie będą już wystarczająco bezpieczne, aby je stosować [16].

Natomiast używanie AES (Advanced Encryption Standard) w celu zachowania bezpieczeństwa będzie wymagało zwiększenie długości kluczy [16].

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

```
using System.Diagnostics;
using System.Security.Cryptography;
using System.Text;

var stopwatch = Stopwatch.StartNew();
var data = "/ETHICAL.BLUE/";
for(ulong nonce = 0x0000000000000000;
    nonce < ulong.MaxValue; nonce++)
{
    var prefix = "0xC0FFEE";
    var hash = SHA512.HashData(
            Encoding.UTF8.GetBytes(
                string.Format("{0}{1:X16}",
                  data, nonce)));
    var digest =
        $"0x{Convert.ToHexString(hash)}";
    Console.WriteLine(
        $"{digest[..10]}...");
    if (digest.StartsWith(prefix))
    {
        Console.WriteLine(string.Format(
            "SHA512({0}{1:X16})={2}...",
            data, nonce, digest[..10]));
        break;
    }
}
stopwatch.Stop();
Console.WriteLine(
    stopwatch.Elapsed.ToString());
```

Scriptum 81. Edukacyjny prototyp programu typu kopacz (ang. miner, C#)

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Kopanie (ang. mining)

Zarządzanie poziomem trudności można zrealizować wprowadzając wymagania dla otrzymywanego wyniku funkcji skrótu (ang. hash) [15]. Przykładem wymagania może być uzyskanie docelowego *hash*, który rozpoczyna się od ustalonej wartości np. 0x0000. Należy zaznaczyć, że im więcej bajtów posiada wzorzec wymaganych wartości, tym bardziej wzrasta poziom trudności, czyli potrzeba więcej czasu i mocy obliczeniowej na znalezienie rozwiązania.

Przykładowy program typu *miner* (pol. górnik) próbuje znaleźć *hash*, który rozpoczyna się od bajtów 0xC0FFEE. W tym celu wielokrotnie oblicza wartość funkcji skrótu (ang. hash) zmieniając *nonce* i weryfikując otrzymany *hash* pod względem dopasowania do wzorca. W prezentowanym przykładzie rozwiązanie udało się odnaleźć w 1 minutę i 24 sekundy (Fotografia 51).

Schemat wykonywanej operacji można zapisać w postaci `Funkcja skrótu ( Dane + Wartość typu nonce ) = 0xC0FFEE...` (Scriptum 81).

```
0x2440CDA4...
0x443E1578...
0x5EF6845C...
0x2C6FA7FB...
0x141866AB...
0xC1578640...
0xF41DE92A...
0xB79F12DB...
0x8D2FC30C...
0xA52C7D0E...
0x94CCFBF5...
0xC0FFEE3D...
SHA512(
 /ETHICAL.BLUE
 /0000000001AD69A
)=0xC0FFEE3D...
00:01:24.7590668
```

Fotografia 51. Konsola debugowania

Fraza odzyskiwania (ang. seed phrase)

Portfel sprzętowy (ang. cold wallet) przechowuje klucz prywatny do kryptowaluty znajdującej się w łańcuchu bloków (ang. blockchain). Rozwiązanie to jest o wiele bardziej bezpieczne, niż portfele internetowe (ang. hot wallet). Jednak urządzenie tego rodzaju jest układem elektronicznym, który wraz z upływem czasu może ulec uszkodzeniu. Podstawowe zagrożenia to ogień, woda czy uszkodzenia mechaniczne (np. upadek z wysokości).

Odzyskanie dostępu do kryptowaluty po utracie urządzenia możliwe jest dzięki tak zwanemu *seed phrase*, czyli wybranym, losowym słowom z języka angielskiego. Ciąg można zapisać na papierze, jednak taki nośnik nie jest odporny na wodę, ogień czy wyblaknięcie wraz z upływem czasu. Z tego powodu przyjęło się przechowywać *seed phrase* na metalowych płytkach. W celu wytłoczenia kodów odzyskiwania na metalowych podkładkach wystarczające będą zwykłe stemple (Fotografia 52).

Fotografia 52. Fraza wytłoczona w metalu

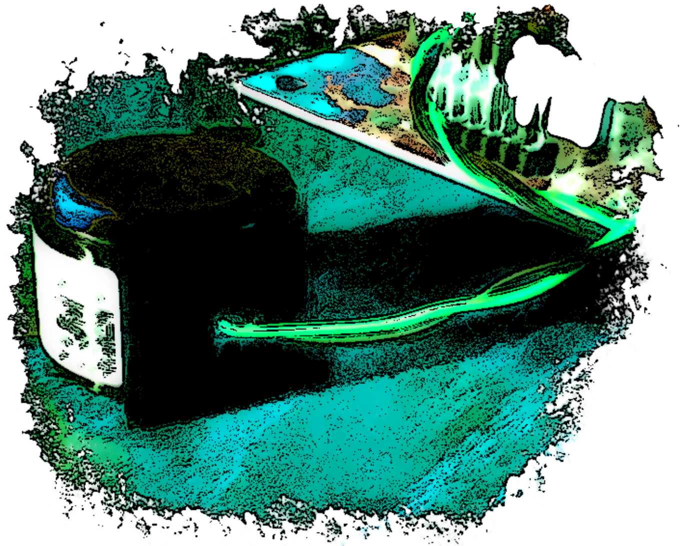
Kody odzyskiwania należy przechowywać w bezpiecznym miejscu. Ciągu słów nie powinno się fotografować. Dwie podkładki można przeznaczyć na zaślepki i wybić na nich dowolne znaki. Dla zwiększenia bezpieczeństwa metalowe podkładki można dodatkowo zaplombować.

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Melody.C#/IL!Prodigal.Son

```
/*-  Melody.C#/IL!Prodigal.Son  -*\
\*-    ethical.blue Magazine    -*/

List<(int freq, int d)> m = [
 new(1044, 237), new(1044, 475),
 new(1044, 237), new(1044, 475),
 new(833,  475), new(934,  475),
 new(934,  950), new(934,  475),
 new(934,  475), new(934,  237),
 new(1044, 475), new(1175, 237),
 new(1044, 950), new(1044, 237),
 new(1044, 475), new(1044, 237),
 new(1044, 475), new(833,  475),
 new(934,  475), new(934,  950),
 new(833,  475), new(833,  237),
 new(833,  237), new(783,  475),
 new(783,  475), new(698,  950)
];
m.ForEach(i => {
 Console.Beep(i.freq, i.d);
 Thread.Sleep(i.d);
});
```

Scriptum 82. Melody.C#/IL!Prodigal.Son



Fotografia 53. Głośniczek systemowy (ang. beeper) podłączony do płyty głównej

Program `Melody.C#/IL!Prodigal.Son` odtwarza na głośniczku systemowym początek melodii znanej pieśni kościelnej pod tytułem *Syn marnotrawny*.

Odgrywane dźwięki są bardzo prymitywne, ponieważ metoda `Console.Beep` (Scriptum 82) korzysta z funkcji Windows API, która pozwala jedynie na ustawienie częstotliwości oraz długości trwania sygnału dźwiękowego (Fotografia 53).

© ethical.blue Magazine. Wszelkie prawa zastrzeżone.

Wykaz literatury

[1] Advanced Micro Devices, Inc. (AMD), *AMD64 Architecture Programmer's Manual*, 2024.

[2] Advanced Micro Devices, Inc. (AMD), *System V Application Binary Interface*, 2025.

[3] Arm Limited, *Arm Architecture Reference Manual for A-profile architecture*, 2025.

[4] Arm Limited, *ELF for the Arm 64-bit Architecture (AArch64)*, 2025.

[5] Ecma International, *Common Language Infrastructure (CLI)*, 2012.

[6] Ecma International, *C# Language Specification*, 2023.

[7] Ethereum Foundation, *https://ethereum.org/*, 2013.

[8] Intel Corporation, *Intel Advanced Vector Extensions 10.2 Architecture Specification*, 2025.

[9] Intel Corporation, *Intel Architecture Instruction Set Extensions and Future Features*, 2025.

[10] Intel Corporation, *The Intel 64 and IA-32 Architectures Software Developer's Manual*, 2025.

[11] ISO/IEC, *Information technology. Common Language Infrastructure (CLI)*, 2012.

[12] ISO/IEC, *Information technology. Programming languages. C#*, 2018.

[13] ISO/IEC, *Programming languages. Avoiding vulnerabilities in programming languages*, 2024.

[14] Microsoft Corporation, *Windows PowerShell Language Specification Version 3.0*, 2012.

[15] National Institute of Standards and Technology (NIST), *Blockchain Technology Overview*, 2018.

[16] National Institute of Standards and Technology (NIST), *Report on Post-Quantum Cryptography*, 2016.

[17] National Institute of Standards and Technology (NIST), *SHA-3 Standard*, 2015.

[18] National Institute of Standards and Technology (NIST), *Specifications for Secure Hash Standard*, 2015.