ethical.blue Magazine

Spreading knowledge like a virus.

Dawid Farbaniec

Volume 2

Spis treści

LEGAL.NFO

Common Weakness Enumeration (CWE™)

CWE™ is free to use by any organization or individual for any research, development, and/or commercial purposes, per these CWE Terms of Use. Accordingly, The MITRE Corporation hereby grants you a non-exclusive, royalty-free license to use CWE for research, development, and commercial purposes. Any copy you make for such purposes is authorized on the condition that you reproduce MITRE's copyright designation and this license in any such copy. CWE is a trademark of The MITRE Corporation. Please contact cwe@mitre.org if you require further clarification on this issue.

https://cwe.mitre.org/

Common Vulnerabilities and Exposures (CVE®)

CVE® is a registered trademark of the MITRE Corporation and the authoritative source of CVE details are MITRE's CVE pages. CWE is a registred trademark of the MITRE Corporation and the authoritative source of CWE details are MITRE's CWE pages.

https://www.cve.org/

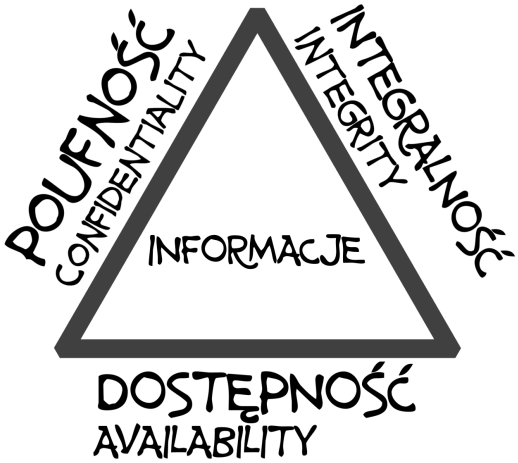
Podatności (ang. vulnerability)

Autor: Dawid Farbaniec

Niniejszy dokument powstał, aby podstawowa wiedza o możliwych podatnościach w aplikacjach rozprzestrzeniała się jak infekcja wirusowa. Informacje te mogą być szczególnie przydatne dla tych, którzy uczą się tworzyć programy oraz tych, których interesuje odkrywanie podatności w celu zwiększania bezpieczeństwa systemów informatycznych.

Podatność (ang. vulnerability)

W sensie bezpieczeństwa systemów informatycznych słowo podatność można wyjaśnić jako słabość czy wrażliwość, które mogą zostać wykorzystane (ang. exploited) i mieć negatywny wpływ (ang. negative impact) na triadę: Poufność (ang. confidentiality), Integralność (ang. integrity) i Dostępność (ang. availability) lub pozwalać na zachowania naruszające jawną lub domyślną politykę bezpieczeństwa.



Odpowiedzialne ujawnienie (ang. coordinated disclosure)

Podejście, które przeważnie polega najpierw na poinformowaniu o podatności przez prywatny kanał komunikacji, ale z zaznaczeniem, że po określonym czasie (ang. deadline) lub po wydaniu poprawki, szczegóły dotyczące odkrytej podatności staną się publicznie dostępne.

Pełne ujawnienie (ang. full disclosure)

Podejście nazywane pełnym ujawnieniem (ang. full disclosure) to ujawnienie informacji o podatności publicznie zaraz po jej odkryciu często z fragmentem kodu udowadniającym możliwość dokonania naruszenia. Pozytywną stroną tego rodzaju poglądu jest presja na producenta, aby nie lekceważył błędu i priorytetowo wydał poprawkę.

Brak publicznego ujawnienia (ang. private disclosure)

W tym podejściu odkrywca podatności informuje prywatnym kanałem komunikacji organizację o znalezionym problemie w zabezpieczeniach. Pozytywną stroną tego rodzaju poglądu jest ograniczenie informacji o podatności do zawężonej grupy podmiotów. Niestety podejście to może spowodować ignorowanie podatności przez organizację, opóźnienia w wydaniu poprawki czy wykorzystywanie podatności przez zawężoną grupę podmiotów zagrażających (ang. threat actor) przez długi czas.

Warto przeczytać

+ https://www.cve.org/ResourcesSupport/Glossary [dostęp: 2024-11-13 00:02]

+ https://www.cve.org/ResourcesSupport/FAQs [dostęp: 2024-11-13 00:02]

+ https://certcc.github.io/CERT-Guide-to-CVD/ [dostęp: 2024-11-13 00:02]

+ https://www.cvedetails.com/cwe-definitions/ [dostęp: 2024-11-13 00:02]

Podatność dnia zerowego (ang. zero day)

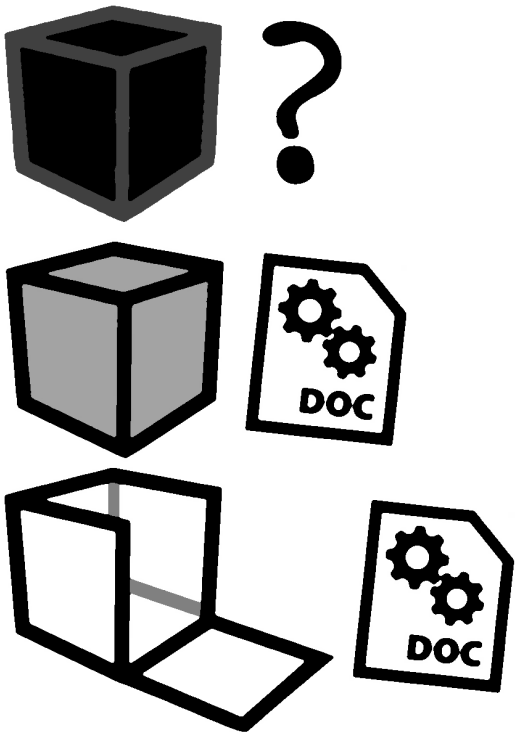
Niezawsze szczegóły dotyczące podatności są wysyłane do producenta czy nawet publikowane. Istnieją podatności o których wiedzą tylko niektóre podmioty i to właśnie tego rodzaju ataki są bardzo niebezpieczne.

Czarna, szara i biała skrzynka (ang. black, grey and white box)

Testy czarnej skrzynki są prawdopodobnie najczęstsze wśród hakerów. Bez wiedzy o systemie, który jest celem następują próby znalezienia podatności.

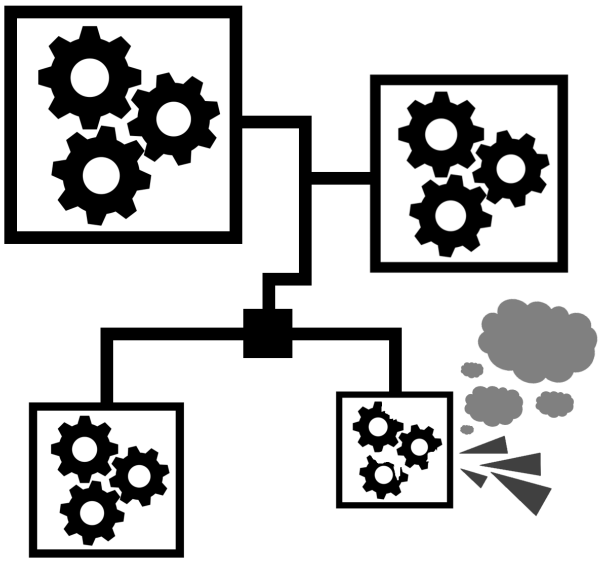
Testy szarej skrzynki to sytuacje w których osoba testująca posiada ograniczoną wiedzę o systemie, który jest celem.

Testy białej skrzynki to weryfikacja zabezpieczeń posiadając często dokumentację systemu, a nawet dostęp do kodów źródłowych w przypadku aplikacji.



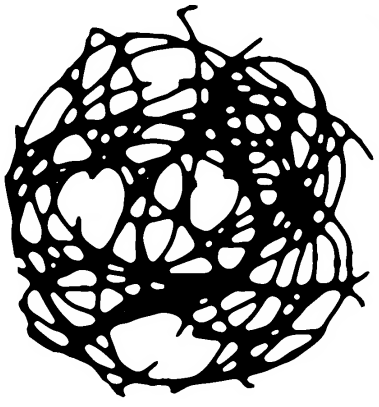
Próba dymu (ang. smoke test)

Test nazywany próbą dymu to przeważnie uruchomienie wczesnej wersji programu w celu sprawdzenia czy krytyczna funkcjonalność jest dostępna.
Inna sytuacja to uruchomienie aplikacji po wprowadzeniu dość znaczących zmian, aby zweryfikować czy aktualizacja nie uszkodziła podstawowej funkcjonalności.



Testowanie odporności programu na błędne dane (ang. fuzzing)

Technika określana terminem fuzzing polega na zautomatyzowanym sprawdzaniu odporności oprogramowania lub innego systemu na błędne dane i skrajne przypadki, które mogą wystąpić.



KŁACZEK (ANG. FUZZ)

Podstawowe wektory ataku na aplikacje

Wszelkie wejścia do programu mogą stać się wektorem ataku i być narażone na wprowadzenie błędnych danych.

W przypadku aplikacji internetowych należy odpowiednio zabezpieczyć wszelkie parametry możliwe do zmodyfikowania przez użytkownika końcowego.

Jeśli dla celów eksperymentalnych pod adresem https://ethical.blue/fuzz/ działała by aplikacja internetowa przyjmująca parametr np. https://ethical.blue/fuzz/3, wtedy jest to potencjalny wektor ataku.

Zależnie od funkcjonalności metody /fuzz można użyć skanera lub stworzyć własne narzędzie, aby sprawdzić reakcję programu na nietypowe dane np.

https://ethical.blue/fuzz/0

https://ethical.blue/fuzz/-1

https://ethical.blue/fuzz/ffffffff

https://ethical.blue/fuzz/text

https://ethical.blue/fuzz/..

https://ethical.blue/fuzz/a.svg

Zdeformowane i skrajne wartości podawane do programu nazywane są ładunkami (ang. payload).

Inne wektory ataku to np. pola tekstowe (wpisanie tekstu zamiast liczby itp.), użycie kontrolek interfejsu użytkownika w nietypowej kolejności lub w nieprzewidziany sposób, parametry podawane przez Wiersz polecenia, wysyłane żądania przez sieć (przy weryfikacji protokołu lub interfejsu nazywanego API), importowany plik niezgodny z formatem, który później trafia do parsera, niepoprawne dane umieszczone na stosie programu i inne zależne od projektu.

Testowanie określane anglojęzycznym terminem fuzzing pozwala oszczędzić czas dzięki automatyzacji. Jednak nie zastępuje indywidualnego podejścia na które składają się m.in. analiza kodu źródłowego (lub inżynieria odwrotna) oraz samodzielnie napisane generatory ładunków (ang. payload) ukierunkowane na określoną aplikację.

Zasięg testów (ang. coverage)

W przypadku testów istotna jest możliwość zmierzenia jaka część kodu została przetestowana. Może to być np. sprawdzenie liczby wykonanych linii kodu i później zamiana ułamka na wartość procentową.

```
internal int XzvnmX73axDe00(int xvMz)
{
    if (xvMz < 0)
        return -1;
    else if (xvMz > 0)
        return 1;
    else
        return 0;
}
```

ZASIĘG TESTÓW (COVERAGE):
8 LINII Z WSZYSTKICH 8 LINII (100%)

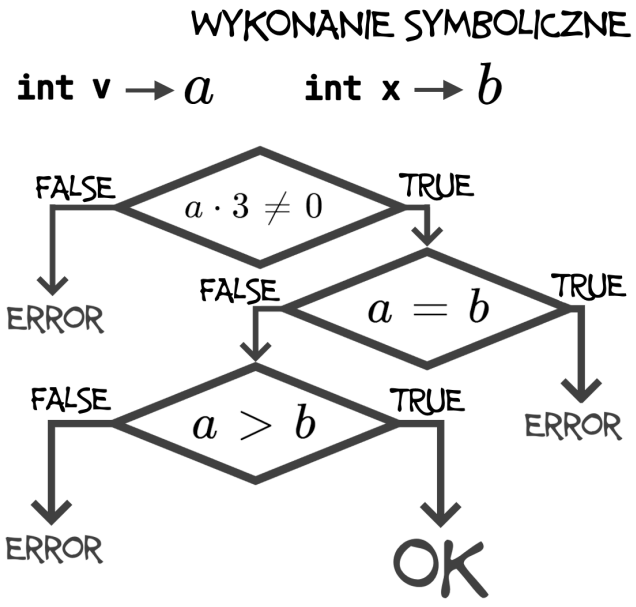
Zabieg tego rodzaju daje ogólny pogląd na ilość wykonanych linii kodu, wywołanych metod czy też które ścieżki w przypadku instrukcji warunkowych się wykonały (zostały przetestowane).

Wykonanie symboliczne (ang. symbolic execution)

Podejście tego rodzaju polega w skrócie na użyciu symboli zamiast konkretnych wartości podczas testów programu.

Pozwala to utworzyć drzewo wykonania, które uwzględnia różne ścieżki kodu.

```
WYKONANIE SYMBOLICZNE
/// <summary>
/// Jeśli powodzenie, to zwraca true.
/// W przeciwnym wypadku zwraca false.
/// </summary>
internal static bool Simple(int v, int x)
{
    int z = v * 3;

    if (z != 0)
    {
        if (v == x)
            return false;
        if (v > x)
            return true;
    }

    return false;
}
```



Niech za przykład posłuży metoda `Simple(int v, int x)`, która w przypadku poprawnych wartości zwraca prawdę (ang. true), a w przypadku błędu zwraca fałsz (ang. false).

Metoda ta przyjmuje dwa parametry od wartości których zależna jest zwracana wartość typu logicznego `bool`.

Niech zmienna `v` będzie oznaczona symbolem `a`, natomiast zmienna `x` niech będzie oznaczona symbolem `b`. Te nazwy symboli mogą być dowolne.

Zmienna `z` przyjmuje wartość `v` razy trzy. Pamiętamy, że zmienna `v` to symbol `a`. Instrukcja warunkowa `if` weryfikuje czy zmienna `z` jest różna od zero, a `z` to `v` razy trzy. Dlatego pierwszy blok to będzie sprawdzenie czy `a` razy trzy jest różne od zero.

Kolejny warunek to weryfikacja czy wartość zmiennej `v` jest równa wartości zmiennej `x`, czyli używając nadanych wcześniej symboli oznacza to sprawdzenie czy `a` jest równe `b`.

Kolejna instrukcja warunkowa to sprawdzenie czy wartość zmiennej `v` jest większa od wartości zmiennej `x`. Używając nadanych symboli oznacza to weryfikację czy `a` jest większe od `b`.

Wykonanie metody używając konkretnych wartości

Przykładowy kod można też wykonać używając konkretnych wartości zamiast symboli. Np. `v = 0`, `x = 3` czy też `v = 3`, `x = 1`. Te dwa podejścia współpracują ze sobą podczas testów, a ich połączenie określa się po angielsku **concolic** (słowa **concrete** i **symbolic**).

```
Console.WriteLine(
$"{Simple(0, 3)}"
); //False

Console.WriteLine(
$"{Simple(3, 1)}"
); //True
```

Instrumentacja kodu programu (ang. code instrumentation)

Technika ta polega na wstrzyknięciu lub wstawieniu specjalnych procedur do kodu aplikacji, których zadaniem jest nadzorowanie wykonania programu.

Dodatkowe procedury mogą być umieszczone ręcznie np. w formie instrukcji wypisujących wartości parametrów na konsolę debugowania. Mogą też np. mierzyć czas wykonania kodu w celu optymalizacji.

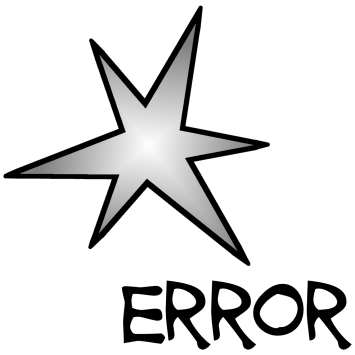
W przypadku testowania odporności programu na błędne dane (ang. fuzzing) wstrzykiwane procedury analizują wykonanie programu i mogą wspomagać decydowanie o tym w jaki sposób dalej testować poszczególne ścieżki wykonania kodu.

W celu zwiększenia zasięgu testów (ang. coverage) wykonanie techniką concolic pozwala zebrać informacje o ograniczeniach (ang. constraint), aby weryfikować nieprzetestowane dotąd ścieżki wykonania.

Przypadki brzegowe i skrajne

Przypadki brzegowe (ang. edge case) to sytuacje lub problemy występujące przy minimalnej lub maksymalnej wartości parametru weryfikowanego systemu. Na przykład: Doświadczenie postaci w grze RPG reprezentuje 16-bitowa liczba całkowita bez znaku i przypadkiem brzegowym może być uzyskanie doświadczenia równego 65535 dziesiętnie (0xFFFF szesnastkowo).

Natomiast przypadki skrajne (ang. corner case) nazywane też patologicznymi mogą wystąpić, gdy wartości wielu parametrów powodują nieprawidłowe działanie systemu.



Mutacje danych (ang. mutation)

Całkowicie losowe dane wysyłane do programu czy innego testowanego systemu mogą być łatwo rozpoznane i odrzucone.
Testowanie odporności programu na błędne dane oparte na mutacjach pozwala modyfikować zbiór poprawnych wartości za pomocą kodu np. zamieniając jeden znak, wstawiając dodatkowy znak na koniec etc.

MUTACJE DANYCH

```
<a href="https://ethical.blue/">ethical.blue</a>
```

```
<a href="https://ethical.blue/">ethical
<a ИΩef="https://ethical.blue/">ethical.blue</a
<<a href=https://ethical.blue/">ethical.blue</a>
<A HREF="https://ethical.blue/">ethical.blue</A>
<@ href="https://ethical.blue/">ethical.blue</a>
<a3 href="https://ethical.blue/">ethical.blue</a>
<!-- href="https://ethical.blue/">ethical.blue</a>
<a href="""https://ethical.blue/>ethical.blue</a>
<a ?ref="https://ethical.blue/">ethical.blue</a>
<a href="{https://ethical.blue/">ethical.blue</a>
...
```

Generacje danych i gramatyka

Testowanie odporności programu na błędy za pomocą mutacji danych jest często niewystarczające. W przypadku przetwarzania np. pliku o określonym formacie należy wysyłać do programu lub testowanego systemu ładunki (ang. payload) zgodne z formatem lub bardzo mu bliskie.
W anglojęzycznej literaturze narzędzia oparte wyłącznie na mutacjach są określane jako głupie (ang. dumb), natomiast te oparte o generacje z obsługą gramatyki jako inteligentne (ang. intelligent).

Wprowadzenie do narzędzia AFL++ (American Fuzzy Lop)

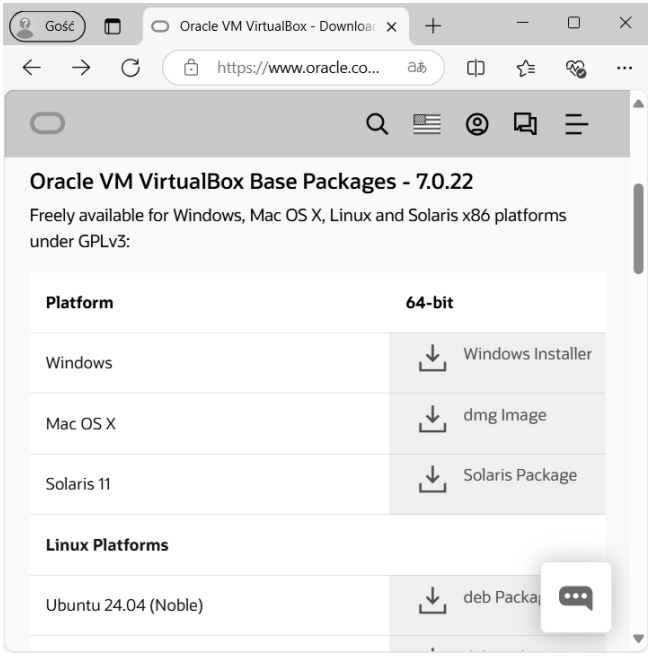
Narzędzie AFL++ jest oparte na znanym programie American Fuzzy Lop.

Instalacja Oracle VM VirtualBox

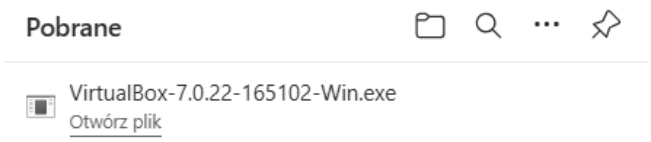
Program VirtualBox w niniejszym eksperymencie pozwoli zainstalować wirtualny system operacyjny Ubuntu Desktop w systemie Microsoft Windows.

https://www.oracle.com/virtualization/technologies/vm/downloads/virtualbox-downloads.html

W przypadku systemu Microsoft Windows należy wybrać łącze Windows Installer (64-bit).

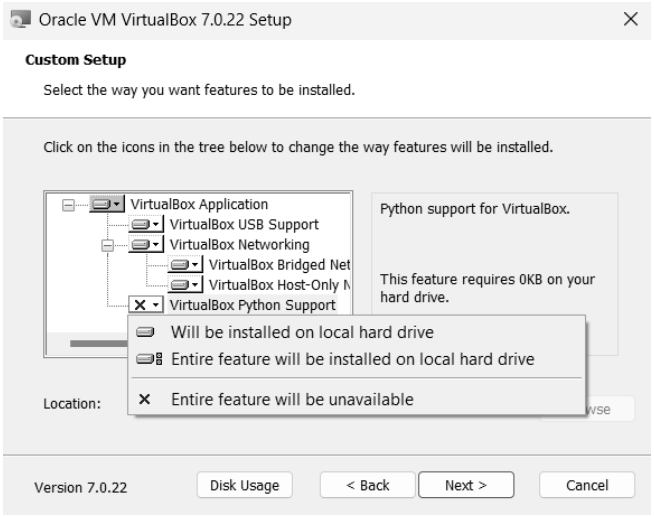


Po ukończeniu pobierania instalator można uruchomić wybierając łącze Otwórz plik.

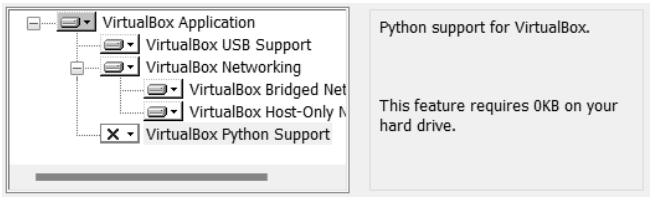


Poniższy rysunek przedstawia kreator instalacji Oracle VM VirtualBox. Aby przejść dalej należy wybrać przycisk Next.

W przeprowadzanym eksperymencie wsparcie dla języka Python nie będzie wymagane. Dlatego należy wybrać opcję VirtualBox Python Support / Entire feature will be unavailable.

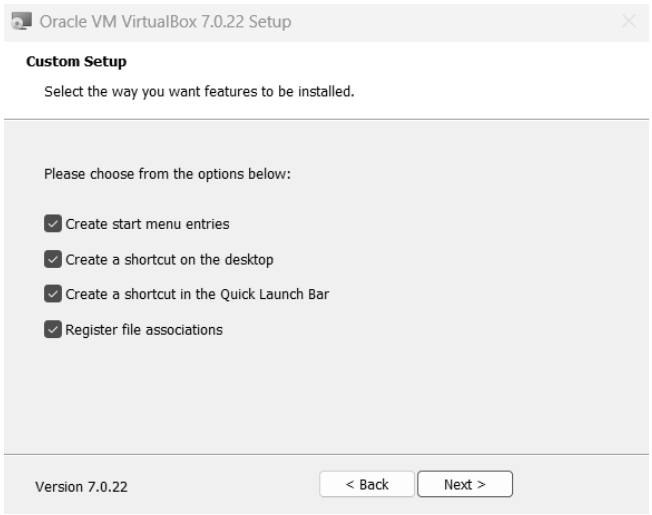


Po wyłączeniu wsparcia dla języka Python obok elementu pojawi się znak X.

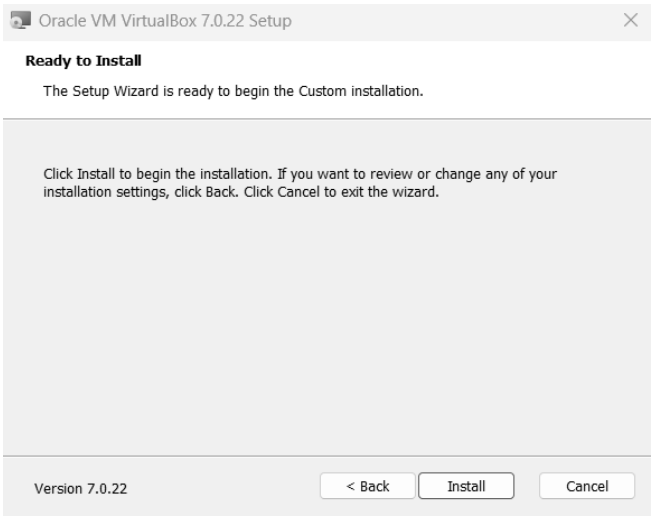


Kolejny krok instalacji to ostrzeżenie o chwilowym odłączeniu od sieci internetowej z powodu instalacji wirtualnej karty sieciowej.

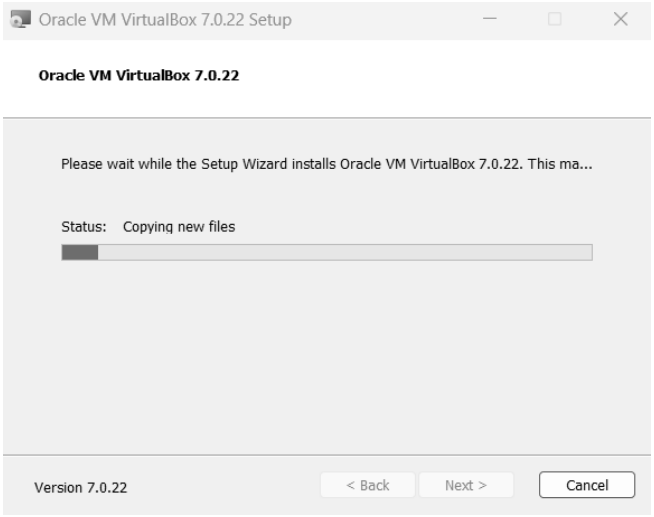
W kolejnym oknie kreatora instalacji można wybrać, aby utworzyć skrót w menu Start, na Pulpicie, na pasku Szybkiego uruchamiania oraz zarejestrować powiązania rozszerzeń plików.



Dalej w celu rozpoczęcia instalacji należy kliknąć przycisk z napisem Install.



Instalacja trwa.



Po zakończeniu instalacji za pomocą pola wyboru można określić czy zainstalowany program ma zostać uruchomiony po kliknięciu Finish.

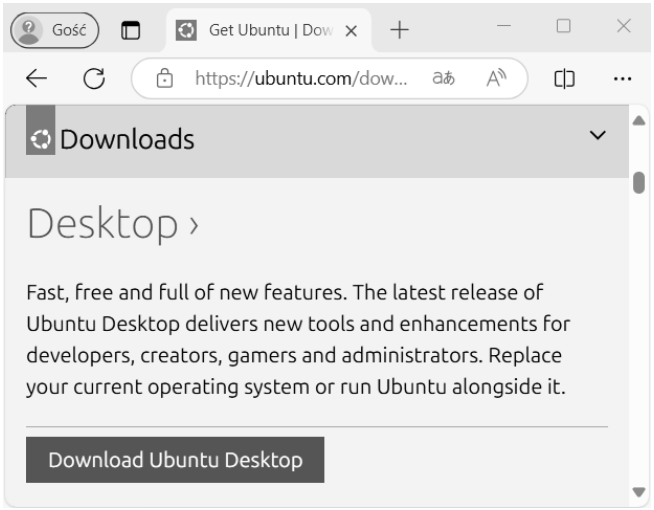
Jeśli wszystko przebiegło pomyślnie, to powinno pojawić się okno główne narzędzia Oracle VM VirtualBox.

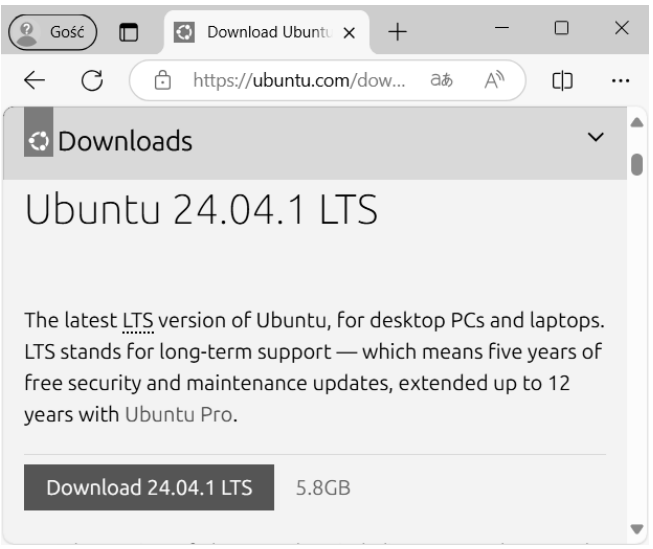
Okno narzędzia Oracle VM VirtualBox można na ten moment zminimalizować wybierając trzecią ikonę od prawej na pasku tytułu.

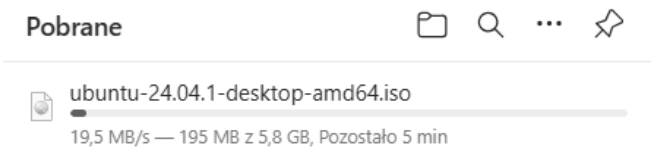
Instalacja Ubuntu Desktop

Kolejny etap to pobranie pliku obrazu systemu operacyjnego Ubuntu Desktop do zainstalowania na wirtualnej maszynie.

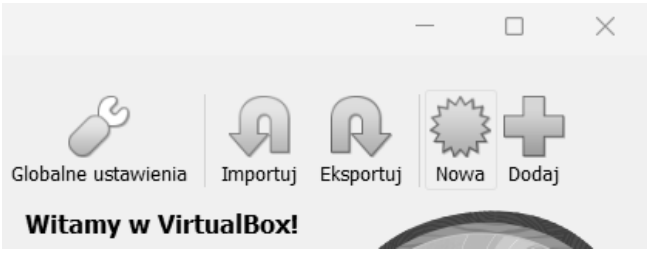
https://ubuntu.com/download



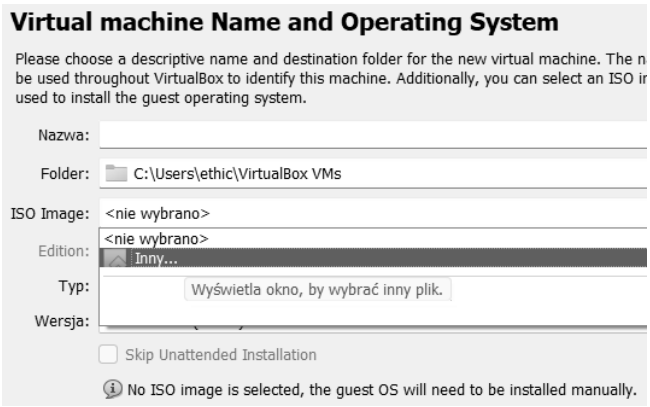


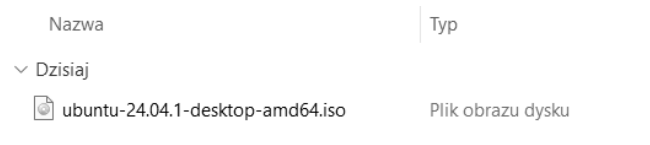


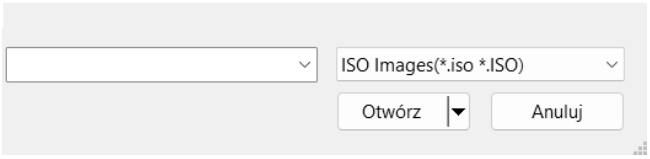
Kolejny krok po pobraniu pliku obrazu systemu Ubuntu Desktop to utworzenie nowej maszyny wirtualnej wybierając przycisk Nowa w oknie narzędzia Oracle VM VirtualBox.



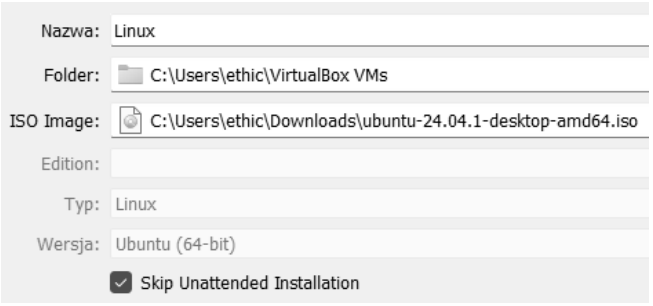
Pierwsze okno dialogowe kreatora tworzenia maszyny wirtualnej pozwala wprowadzić nazwę oraz wybrać plik obrazu (*.iso). W polu ISO Image należy kliknąć listę rozwijaną, wybrać Inny... i otworzyć pobrany wcześniej plik ISO systemu Ubuntu Desktop.



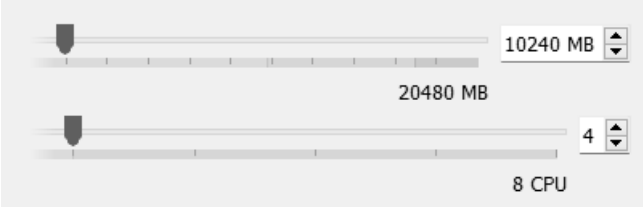




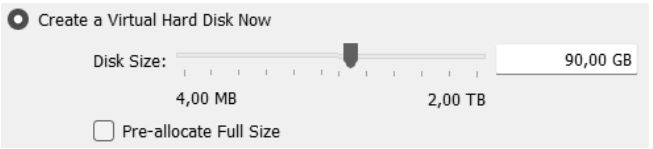
W celu łatwiejszego dostosowania ustawień instalowanego systemu Ubuntu Desktop należy zaznaczyć pole wyboru Skip Unattended Installation. Oznacza to pominięcie instalacji nienadzorowanej.



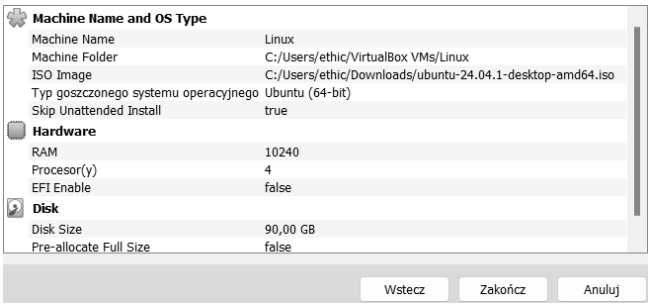
W oknie dialogowym Hardware możliwe jest ustawienie parametrów sprzętowych maszyny wirtualnej. Maszyna gospodarza użyta do eksperymentu posiada 20 GB pamięci operacyjnej, więc dla maszyny wirtualnej przeznaczono 10 GB. Podobnie z rdzeniami procesora.



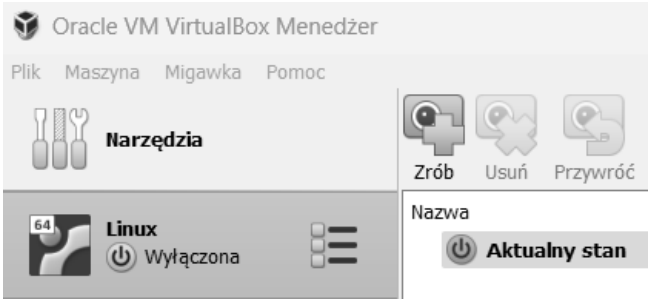
Maszyna gospodarza podczas eksperymentu ma około 120 GB wolnego miejsca na dysku. Z tego powodu na maszynę wirtualną można przeznaczyć przykładowo 90 GB.



W oknie dialogowym Podsumowanie można przejrzeć parametry tworzonej maszyny wirtualnej. Po kliknięciu Zakończ maszyna powinna zostać utworzona.



Po utworzeniu maszyny wirtualnej pierwsze uruchomienie to będzie instalacja systemu Ubuntu Desktop. W celu uruchomienia utworzonej maszyny wirtualnej należy wybrać przycisk Uruchom.

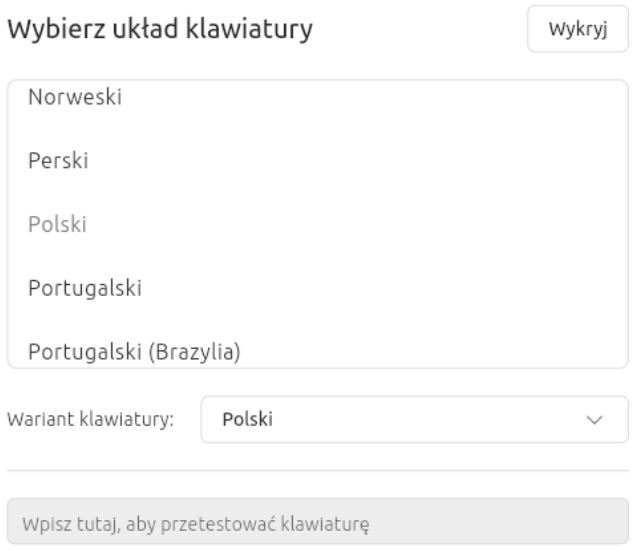


Kolejne okno to program rozruchowy (ang. bootloader) o nazwie GNU GRUB. Z wyświetlonego menu należy wybrać opcję Try or Install Ubuntu (pol. wypróbuj lub zainstaluj Ubuntu).

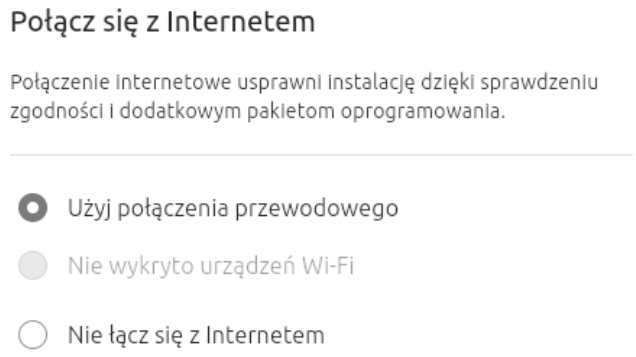
Pierwszy krok instalacji Ubuntu Desktop to wybór języka. Można tutaj wybrać Polski.

Kolejny krok dotyczy Ułatwień dostępu. Pozwala dostosować system dla osób z problemami ze wzrokiem, słuchem lub inną niepełnosprawnością.

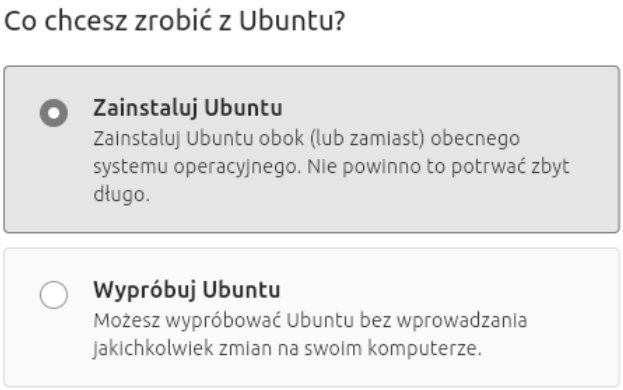
Kolejne okno to wybór układu klawiatury. Można tutaj wybrać Polski.



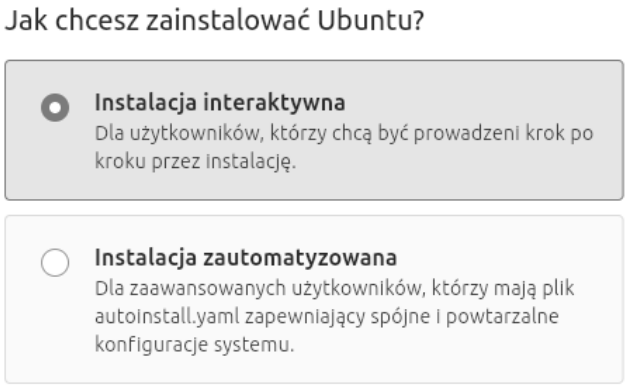
Kolejne okno dotyczy połączenia z internetem. Należy tutaj wybrać Użyj połączenia przewodowego.



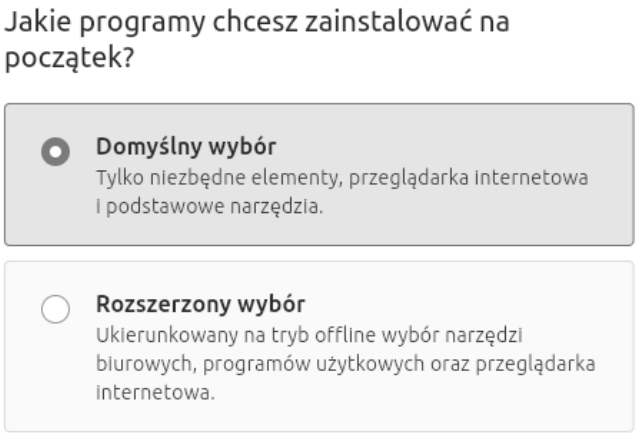
W przypadku chęci zapoznania się z systemem możliwe jest uruchomienie Ubuntu Desktop bez instalacji systemu na dysku. Na potrzeby eksperymentu wygodniej będzie zainstalować system na stałe, dlatego należy tutaj wybrać Zainstaluj Ubuntu.



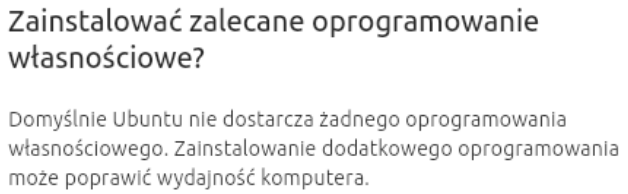
Dostępne sposoby instalacji to instalacja interaktywna oraz instalacja zautomatyzowana. Aby nie utrudniać należy wybrać Instalacja interaktywna w której kreator przeprowadzi użytkownika krok po kroku przez poszczególne etapy.



Kolejny etap to wybór zestawu programów dostępnych zaraz po instalacji systemu. Można tutaj zaznaczyć pole Domyślny wybór.

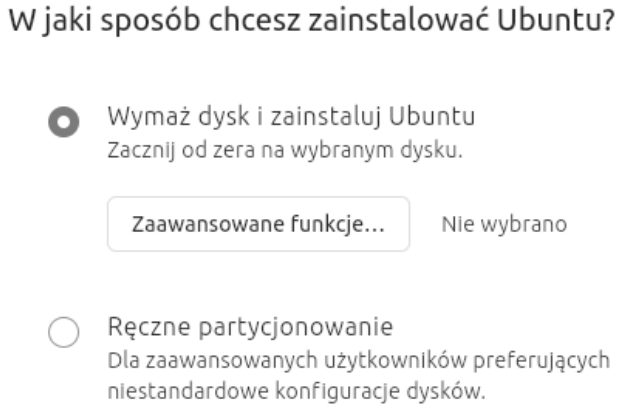


Kolejny krok dotyczy wydajności. Do niniejszego eksperymentu nie jest wymagane oprogramowanie własnościowe.

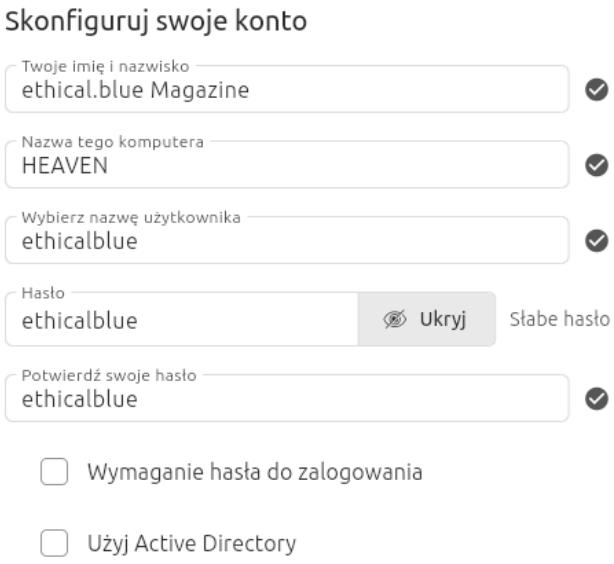


Partycja nazywana też dyskiem logicznym to wydzielony obszar na dysku, któremu można nadać określony format plików. Możliwe jest utworzenie kilku partycji zależnie od potrzeb.

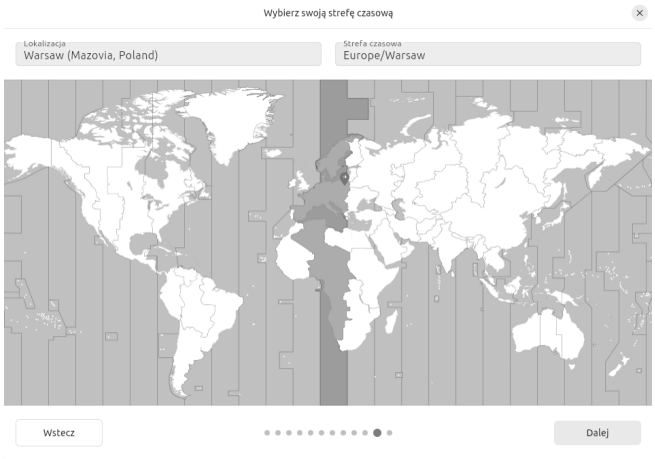
Jednak maszyna wirtualna jest wyłącznie do celów eksperymentalnych, więc dla ułatwienia można wybrać opcję Wymaż dysk i zainstaluj Ubuntu.



Kolejne okno to konfiguracja konta użytkownika w systemie Ubuntu Desktop. Maszyna po eksperymencie najprawdopodobniej zostanie usunięta, więc nie ma sensu jej szczególnie zabezpieczać.

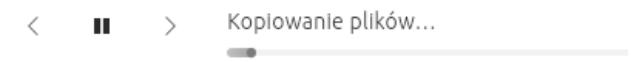


Jeszcze tylko wybór lokalizacji (Warsaw, Mazovia, Poland) oraz strefy czasowej (Europe/Warsaw).



Przegląd ustawień i można kliknąć Instaluj.

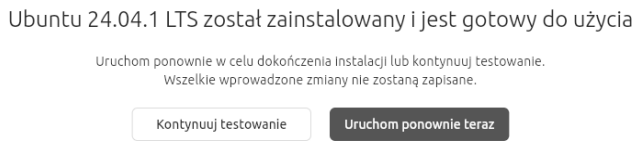
Instalacja Ubuntu Desktop chwilę potrwa.



W międzyczasie można zagrać w ByteZ (Roguelike) dostępne pod adresem:

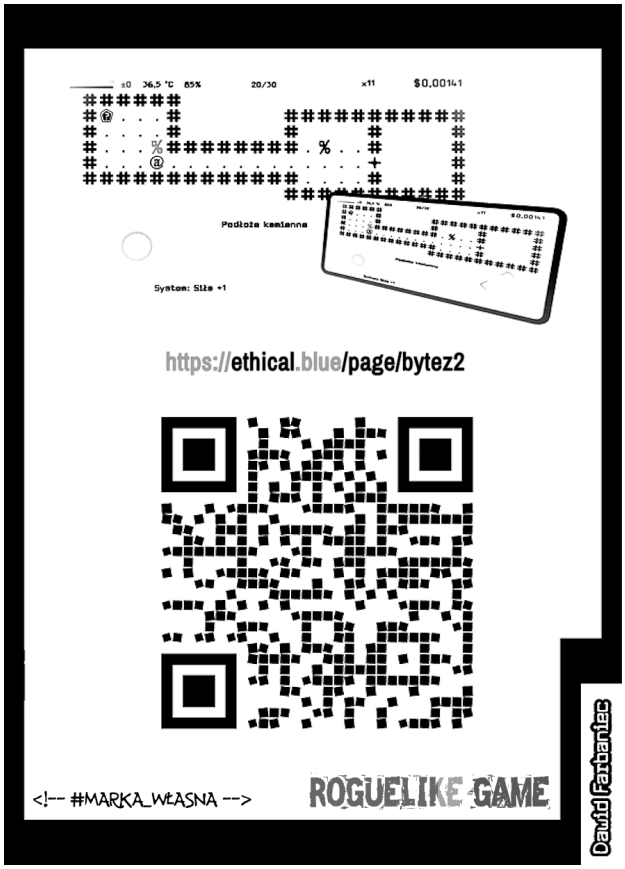
https://ethical.blue/page/bytez2
[dostęp: 2024-11-13 00:03]

Po zakończeniu instalacji można kliknąć przycisk Uruchom ponownie teraz.

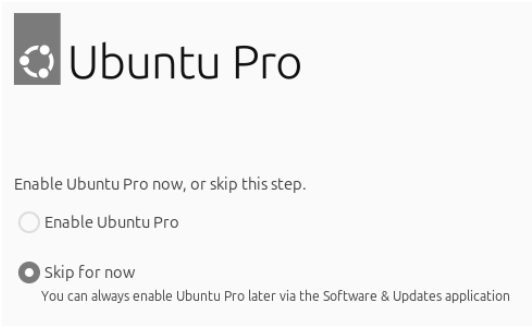


Gdy maszyna wirtualna uruchomi się ponownie należy wcisnąć Enter.

Na ekranie powitalnym w nowym systemie Ubuntu Desktop należy kliknąć przycisk Dalej.

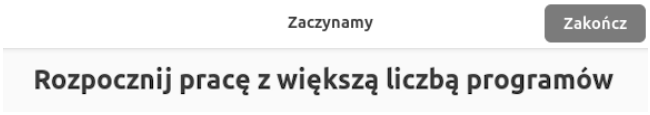


Ofertę Ubuntu Pro można na ten moment pominąć wybierając Skip for now.

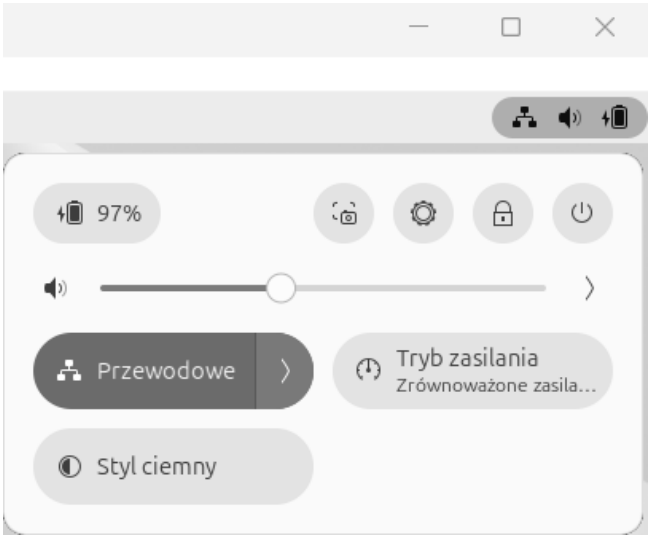


Udostępnianie danych diagnostycznych też można wyłączyć i kliknąć przycisk Dalej.

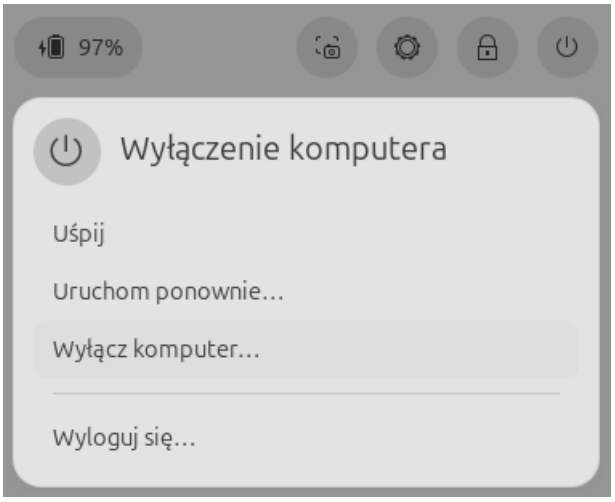
W celu zainstalowania większej liczby programów można przejść do Centrum oprogramowania. Na razie jednak nie jest to potrzebne, więc można kliknąć Zakończ.



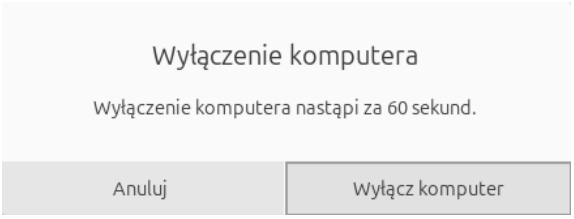
Ostatni etap związany z instalacją to zamknięcie systemu Ubuntu i stworzenie migawki. W tym celu należy kliknąć w ikony zasobnika w prawym górnym rogu i wybrać ikonę zasilania.



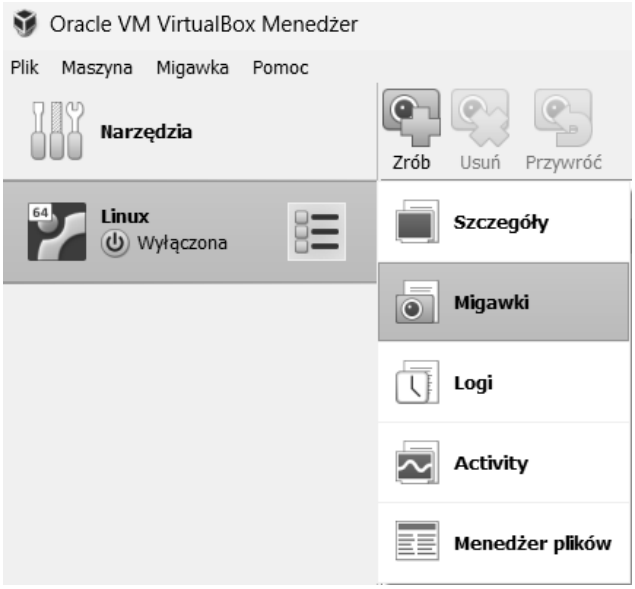
Dalej z rozwijanego menu należy wybrać Wyłącz komputer. Uwaga: wyłączamy tutaj maszynę wirtualną (Ubuntu), a nie system gospodarza (Windows).



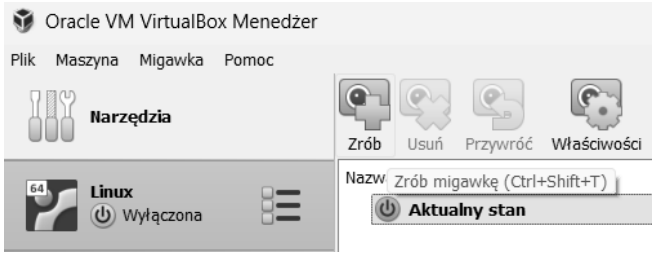
Wybór należy potwierdzić w oknie dialogowym.



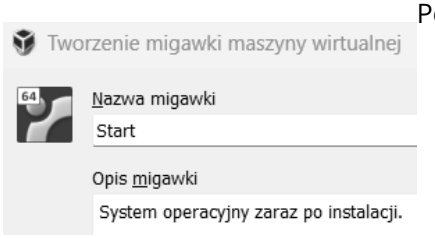
Teraz, aby nie zaginęły zmiany wprowadzone w utworzonej maszynie wirtualnej należy utworzyć migawkę (ang. snapshot). W tym celu należy przejść do Migawki.



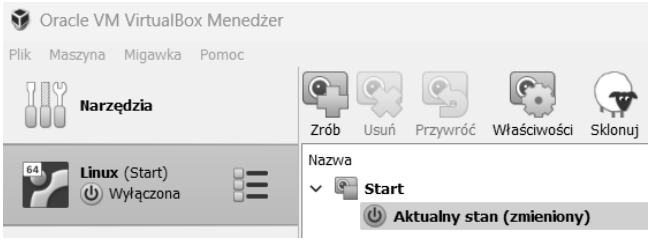
Utworzenie migawki (ang. snapshot) jest możliwe przyciskiem Zrób migawkę (ang. take snapshot).



Pozostało jeszcze

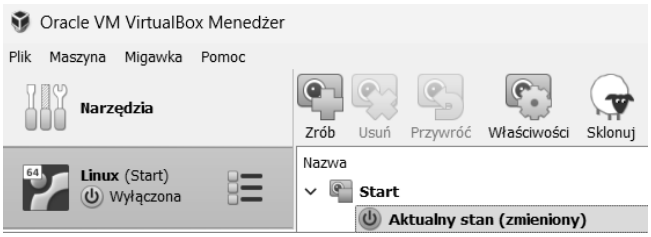


Maszyna wirtualna jest już z zainstalowanym systemem Ubuntu, a jej stan został trwale zachowany przez zrobienie Migawki (ang. snapshot).



Podstawy obsługi terminala systemu Ubuntu z rodziny Linux

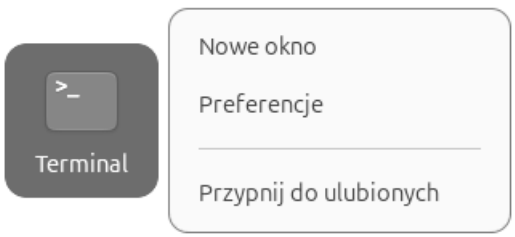
Maszyna wirtualna jest wyłączona.



Należy ją uruchomić klikając przycisk Uruchom.

Dalej należy kliknąć logo systemu w panelu dokowanym, aby przejść do wyboru programów.

Po kliknięciu ikony Terminal i wybraniu Przypnij do ulubionych zostanie dodany skrót w panelu dokowanym do uruchamiania programu Terminal.



Teraz dostęp do narzędzia Terminal jest o wiele łatwiejszy.

Na początek poleceniem **pwd** można sprawdzić bieżący katalog roboczy, a za pomocą **ls** wyświetlić zawartość tego katalogu.

```
ethicalblue@HEAVEN:~$ pwd
/home/ethicalblue
ethicalblue@HEAVEN:~$ ls
Dokumenty  Obrazy   Publiczny  snap     Wideo
Muzyka     Pobrane  Pulpit     Szablony
ethicalblue@HEAVEN:~$
```

Przed instalacją oprogramowania zawsze warto wykonać synchronizację za pomocą polecenia **sudo apt-get update**, które informuje **apt-get**, aby pobrało najnowsze informacje o dostępnych wersjach programów.

Po wcześniejszej synchronizacji można zainstalować narzędzie AFL++ za pomocą polecenia **sudo apt install afl++**.

```
Pobrano 6 283 kB w 8s (806 kB/s)
Czytanie list pakietów... Gotowe
ethicalblue@HEAVEN:~$ sudo apt install afl++
```

W celu kontynuowania instalacji należy wpisać literę **T** i zatwierdzić Enter.

```
  llvm-17-runtime llvm-17-tools llvm-18
  llvm-18-dev llvm-18-linker-tools
  llvm-18-runtime llvm-18-tools
  lto-disabled-list make
0 aktualizowanych, 73 nowo instalowanych, 0 usuw
anych i 62 nieaktualizowanych.
Konieczne pobranie 251 MB archiwów.
Po tej operacji zostanie dodatkowo użyte 1 426 M
B miejsca na dysku.
Kontynuować? [T/n] T
```

W celu weryfikacji instalacji narzędzia AFL++ można uruchomić je bez parametrów poleceniem **afl-fuzz**.

```
ethicalblue@HEAVEN:~$ afl-fuzz
afl-fuzz++4.09c based on afl by Michal
Zalewski and a large online community

afl-fuzz [ options ] -- /path/to/fuzzed
_app [ ... ]

Required parameters:
  -i dir        - input directory with
test cases (or '-' to resume, also see
                  AFL_AUTORESUME)
  -o dir        - output directory for
fuzzer findings
```

Przykładowy program z podatnościami

Za pomocą polecenia **pwd** można sprawdzić bieżący katalog roboczy. Dalej jest utworzenie pliku o nazwie **vuln.cpp** poleceniem **touch**. Po wpisaniu **ls** można zobaczyć, że plik został utworzony. Zawartość pliku można edytować np. za pomocą polecenia **editor ./vuln.cpp**.

```
ethicalblue@HEAVEN:~$ pwd
/home/ethicalblue
ethicalblue@HEAVEN:~$ touch ./vuln.cpp
ethicalblue@HEAVEN:~$ ls
Dokumenty  Obrazy   Publiczny  snap      vuln.cpp
Muzyka     Pobrane  Pulpit     Szablony  Wideo
ethicalblue@HEAVEN:~$ editor ./vuln.cpp
```

Po wprowadzeniu przykładowego kodu źródłowego można zapisać zmiany wciskając CTRL+O i zatwierdzając Enter. Natomiast, aby wyjść z edytora należy wcisnąć CTRL+X.

```
  GNU nano 7.2                ./vuln.cpp *
#include <iostream>
#include <fstream>
#include <sstream>

int main(int argc, char* argv[])
{
    void (*crash)() = reinterpret_cast<void(*)()>(0xDEAD);
    if (argc != 2) return EXIT_SUCCESS;
    std::string path = argv[1];
    std::fstream fileStream(path.c_str(), std::fstream::in);
    fileStream.open(path.c_str());
    std::stringstream textStream;
    textStream << fileStream.rdbuf();
    std::cout << "Payload is ";
    std::cout << textStream.str() << std::endl;
    if(textStream.str().c_str()[0] != 'F') return EXIT_SUCCESS;
    std::cout << "Crash?!" << std::endl;
    crash();
    return EXIT_SUCCESS;
}
^G Pomoc    ^O Zapisz   ^W Wyszukaj ^K Wytnij   ^T Wykonaj
^X Wyjdź    ^R Wczyt.pli^\ Zastąp   ^U Wklej    ^J Wyjustuj
```

Dalsze przygotowania środowiska do eksperymentu to utworzenie katalogów **test_cases** oraz **findings**. Pierwszy katalog będzie zawierał pliki, które posłużą za przypadki testowe, a drugi katalog jest na artefakty znalezione przez narzędzie typu fuzzer. Do celów prostej prezentacji wystarczy jeden przypadek testowy, który spowoduje błąd wcześniej napisanego programu.

Poleceniem **cat > ./test_cases/data.txt** można utworzyć plik o nazwie **data.txt** w katalogu **test_cases** o zawartości **Payload_example**. W konsoli po wpisaniu zawartości należy wcisnąć CTRL+D, aby wyjść z trybu edycji.

```
ethicalblue@HEAVEN:~$ mkdir ./test_cases
ethicalblue@HEAVEN:~$ mkdir ./findings
ethicalblue@HEAVEN:~$ cat > ./test_cases/data.txt
Payload_example
ethicalblue@HEAVEN:~$
```

Należy jeszcze wyłączyć wysyłanie informacji o zrzucie w przypadku błędów poleceniami:

```
sudo su
echo core >/proc/sys/kernel/core_pattern
su ethicalblue
```

Polecenie **sudo su** to zalogowanie na bardziej uprzywilejowane konto **root**. Dalej jest polecenie sugerowane przez narzędzie AFL++ i powrót na konto zwykłego użytkownika za pomocą polecenia **su <nazwa-użytkownika>**.

Kolejny krok to utworzenie pustego pliku wykonywalnego.

```
touch ./vuln.elf
```

Niniejszym poleceniem możliwe jest ustawienie atrybutów **rwx** (ang. read, write and execute) dla bieżącego użytkownika (**u**) dla pliku **vuln.elf** w bieżącym katalogu (**./**).

```
chmod u+rwx ./vuln.elf
```

W celu zbudowania programu wraz z instrumentacją należy użyć polecenia:

```
afl-clang-fast++ ./vuln.cpp -o
./vuln.elf
```

Jeśli polecenia się powiodły, to na konsoli powinna pojawić się informacja o dodaniu instrumentacji.

```
ethicalblue@HEAVEN:~$ touch ./vuln.elf
ethicalblue@HEAVEN:~$ chmod u+rwx ./vuln.elf
ethicalblue@HEAVEN:~$ afl-clang-fast++ ./vuln.cpp -o ./vuln.elf
afl-cc++4.09c by Michal Zalewski, Laszlo Szekeres, Marc Heuse -
 mode: LLVM-PCGUARD
SanitizerCoveragePCGUARD++4.09c
[+] Instrumented 66 locations with no collisions (non-hardened
mode) of which are 4 handled and 0 unhandled selects.
ethicalblue@HEAVEN:~$
```

Poniższe polecenie uruchamia fuzzer AFL++ wskazując katalog **./test_cases** jako wejście oraz katalog **./findings** na znalezione artefakty. Znaki **--** oddzielają wywołanie narzędzia fuzzer od pliku testowanego programu, którym to plikiem jest tutaj **vuln.elf**. Znaki **@@** oznaczają miejsce, gdzie ma być wstawiony ładunek (ang. payload), czyli plik wczytywany przez program, którego odporność jest testowana.

```
afl-fuzz -i ./test_cases -o ./findings
-- ./vuln.elf @@
```

W celu wyświetlenia znalezionych podatności (ang. vulnerability) z folderu **./findings/crashes** można zastosować polecenie:

```
for f in ./findings/default/crashes/*:*;
do xxd "$f"; echo ""; done;
```

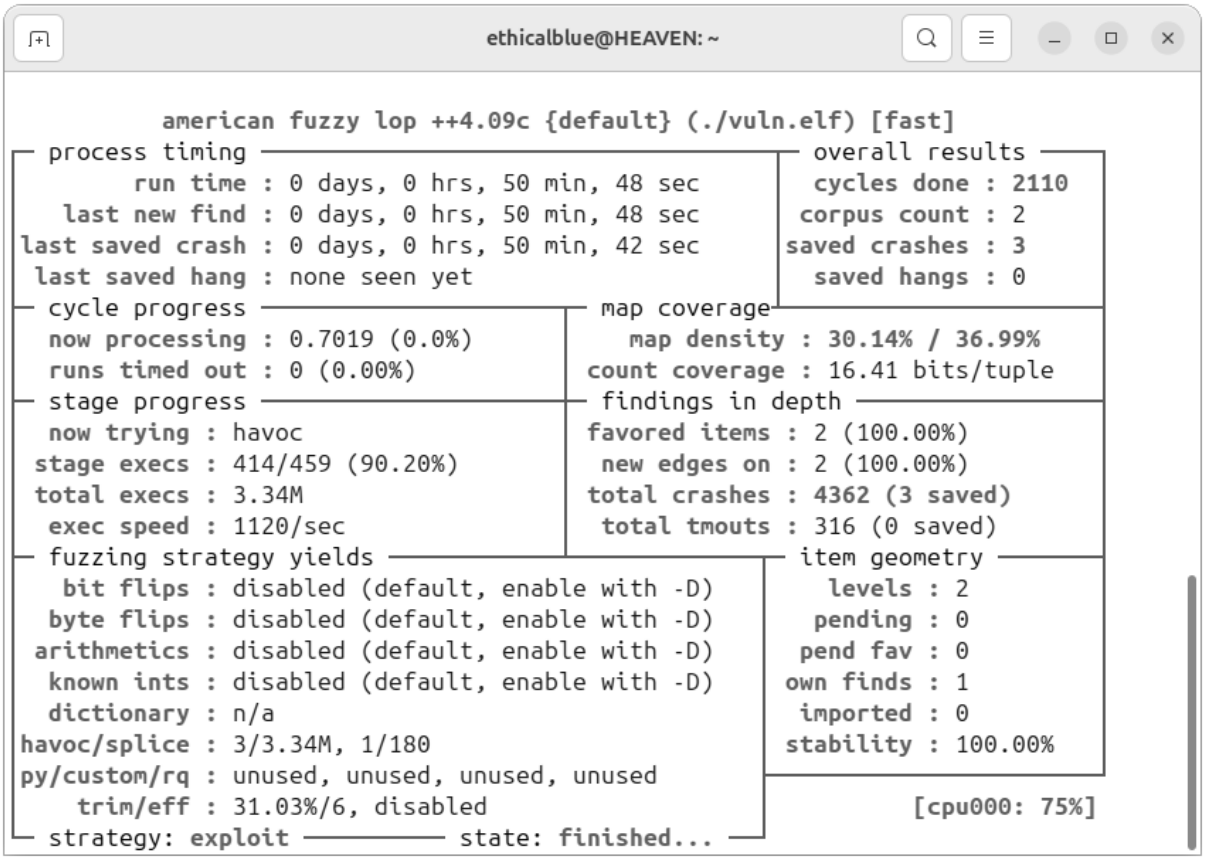
```
ethicalblue@HEAVEN:~$ for f in ./findings/default/crashes/*:*; do xxd "$f"; echo ""; done;
00000000: 4661 796c 6f6f 6f6f 6f6f 6f6f 6f6f  Faylooooooooo

00000000: 4661 6161 6177 6161 6161 7761 6161 616c  Faaaawaaaawaaaal
00000010: 6c6c 6c6c 650a                           llllle.

00000000: 467e 796c c754 6401 4d78 006d 706c 650a  F~yl.Td.Mx.mple.

ethicalblue@HEAVEN:~$
```

vuln.cpp

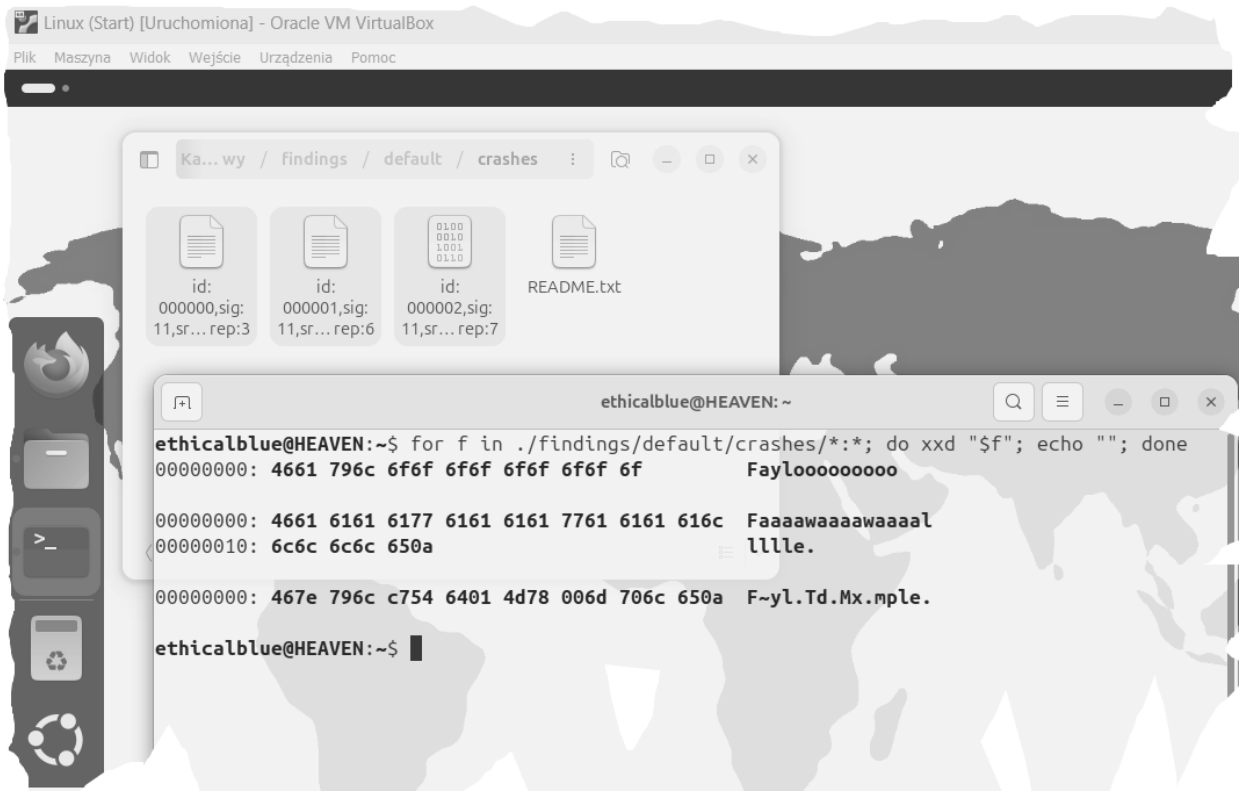
```
/* Przykładowy program z podatnościami (ang. vulnerability) */

#include <iostream>
#include <fstream>
#include <sstream>

int main(int argc, char* argv[])
{
    void (*crash)() = reinterpret_cast<void(*)()>(0xDEAD);
    if (argc != 2) return EXIT_SUCCESS;
    std::string path = argv[1];
    std::fstream fileStream(path.c_str(), std::fstream::in);
    fileStream.open(path.c_str());
    std::stringstream textStream;
    textStream << fileStream.rdbuf();
    std::cout << "Payload is ";
    std::cout << textStream.str() << std::endl;
    if(textStream.str().c_str()[0] != 'F') return EXIT_SUCCESS;
    std::cout << "Crash?!" << std::endl;
    crash();
    return EXIT_SUCCESS;
}
```

```
  GNU nano 7.2                  ./vuln.cpp
#include <iostream>
#include <fstream>
#include <sstream>

int main(int argc, char* argv[])
{
    void (*crash)() = reinterpret_cast<void(*)()>(0xDEAD);
    if (argc != 2) return EXIT_SUCCESS;
    std::string path = argv[1];
    std::fstream fileStream(path.c_str(), std::fstream::in);
    fileStream.open(path.c_str());
    std::stringstream textStream;
    textStream << fileStream.rdbuf();
    std::cout << "Payload is ";
    std::cout << textStream.str() << std::endl;
    if(textStream.str().c_str()[0] != 'F') return EXIT_SUCCESS;
    std::cout << "Crash?!" << std::endl;
    crash();
    return EXIT_SUCCESS;
}
```

Narzędzie AFL++ (American Fuzzy Lop) wyświetla 4362 awarie (w tym 3 warte zapisania).



Wyświetlenie ładunków (ang. payload), które spowodowały awarię testowanego programu.



Od nas zależy co zrobimy ze znalezionymi błędami.

Podstawy analizy statycznej za pomocą narzędzia Ghidra

Narzędzie nazywane Ghidra jest autorstwa NSA, aby wspierać misję jaką jest cyberbezpieczeństwo.

https://ghidra-sre.org/

W systemie Ubuntu można je zainstalować np. z Canonical Snapcraft.

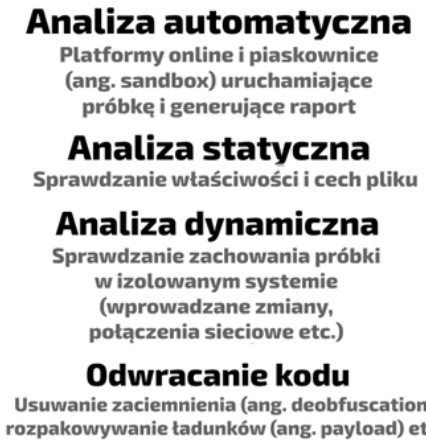
https://snapcraft.io/install/ghidra/ubuntu

Polecenia są następujące:

```
sudo apt update

sudo apt install snapd

sudo snap install ghidra
```

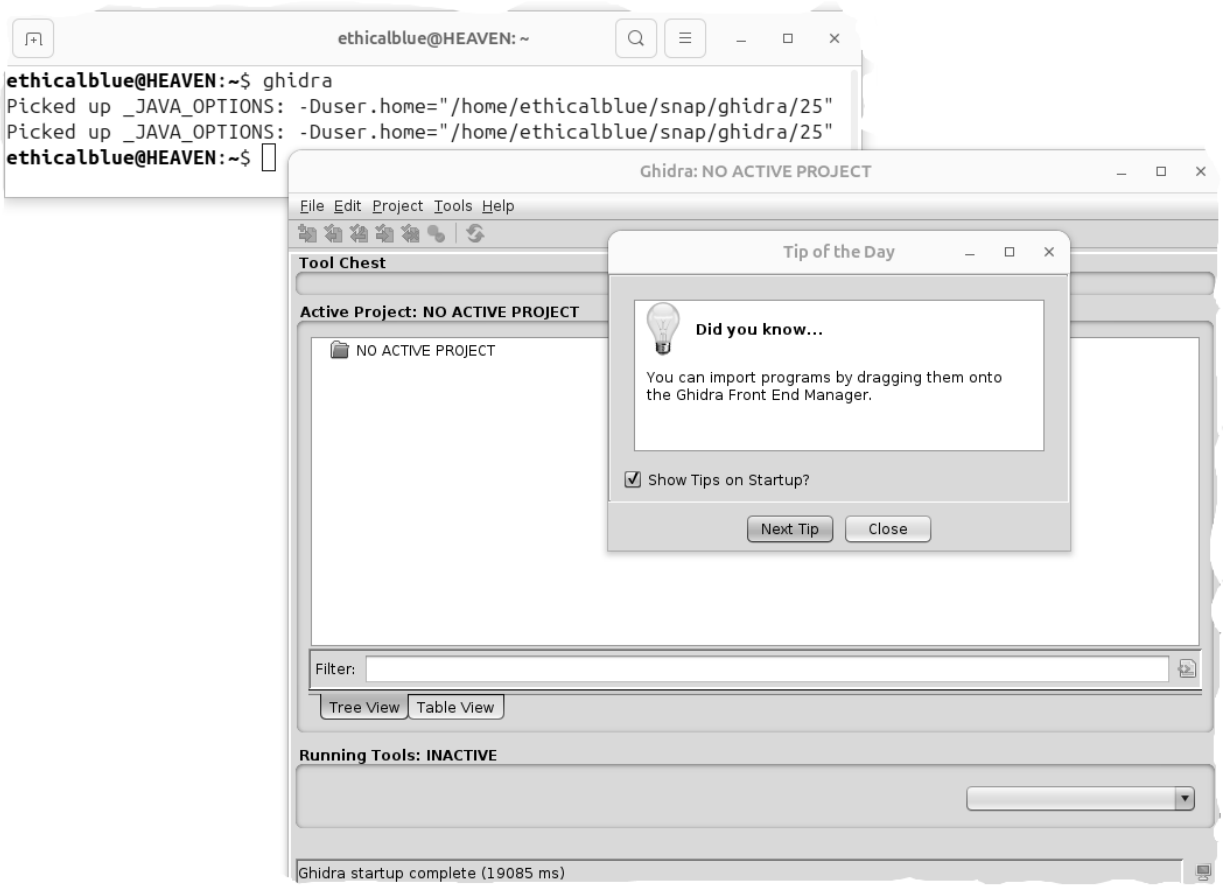
Przykładowe zastosowania inżynierii odwrotnej oprogramowania

+ Analiza złośliwego kodu



+ Audyt plików binarnych aplikacji w poszukiwaniu podatności
+ Analiza poprawek (ang. patch) w celu określenia np. skali możliwego zagrożenia związanego z podatnością

To oczywiście nie są jedyne zastosowania.



Laboratorium

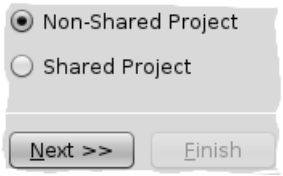
W celu prezentacji podstawowych możliwości narzędzia Ghidra można przeprowadzić prosty eksperyment, który niech polega na znalezieniu w pliku binarnym ELF instrukcji z przykładu **vuln.cpp** powodujących błąd.

```
void (*crash)() =
reinterpret_cast<void(*)()>(0xDEAD);
/* ... */
crash();
```

Narzędzie Ghidra można uruchomić wpisując w Terminal polecenie **ghidra**.

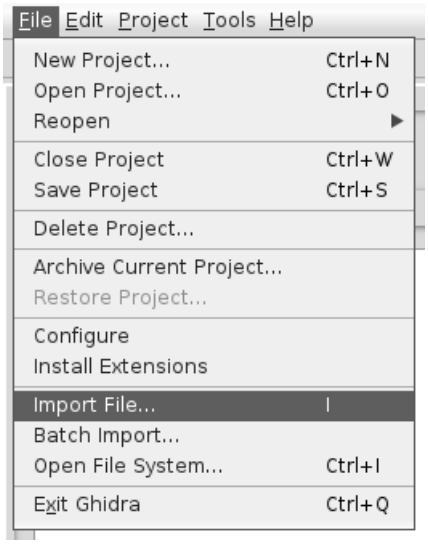
W celu rozpoczęcia pracy należy utworzyć nowy projekt wybierając z górnego menu File / New Project.

Zaznaczyć projekt nieudostępniany i przejść dalej (przycisk Next).

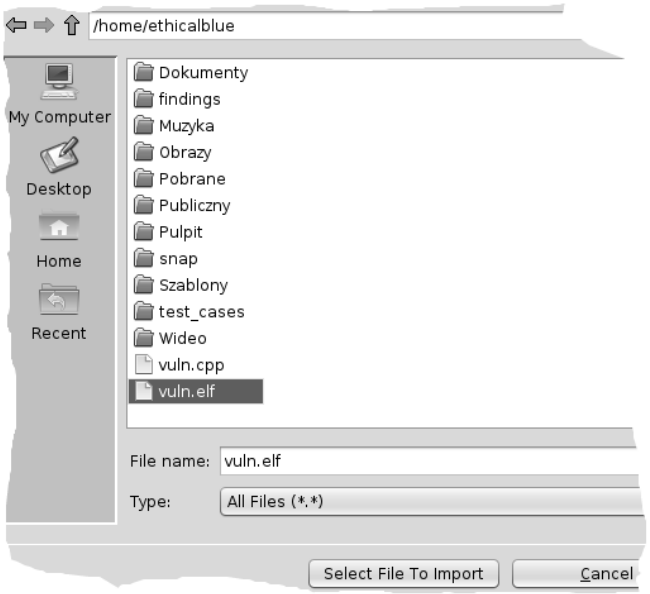


Nazwa projektu np. vuln_example i kliknąć przycisk Finish.

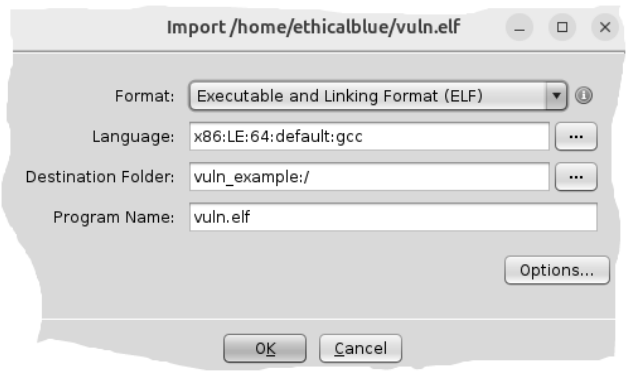
W celu dodania przykładowego pliku ELF do projektu należy wybrać File / Import File.



Teraz należy odnaleźć plik **vuln.elf** z poprzednich eksperymentów i kliknąć Select File To Import.

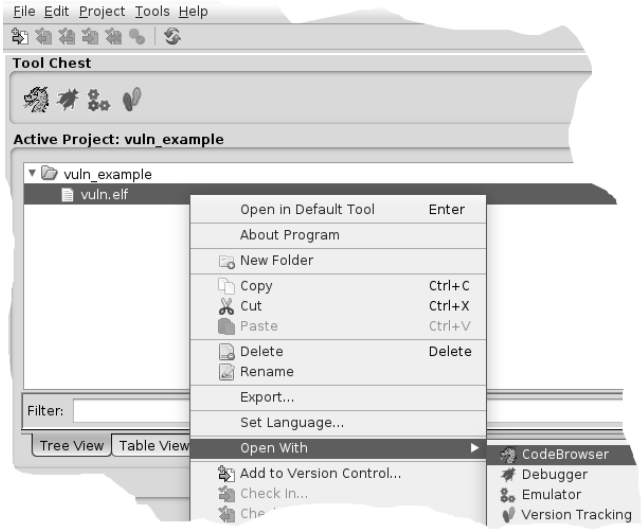


Narzędzie Ghidra rozpoznało rodzaj pliku, czyli OK.

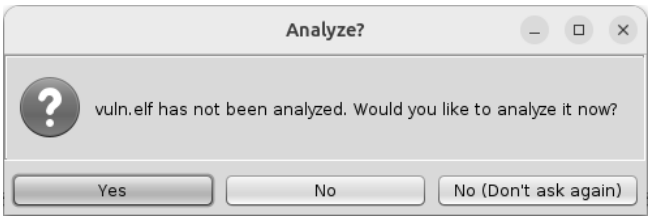


Kolejne okno to rezultat wczytywania pliku. Należy zatwierdzić przyciskiem OK.

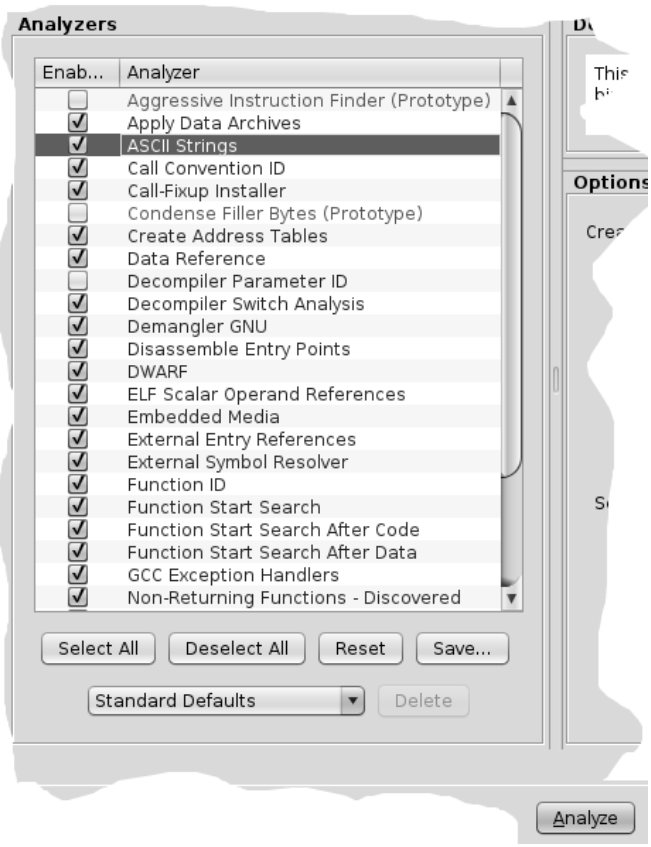
Plik **`vuln.elf`** został zaimportowany do projektu **`vuln_example`**. Klikając prawym przyciskiem na element kontrolki drzewa należy wybrać Open With / CodeBrowser. Pozwoli to na przegląd odzyskanego kodu programu z pliku binarnego.



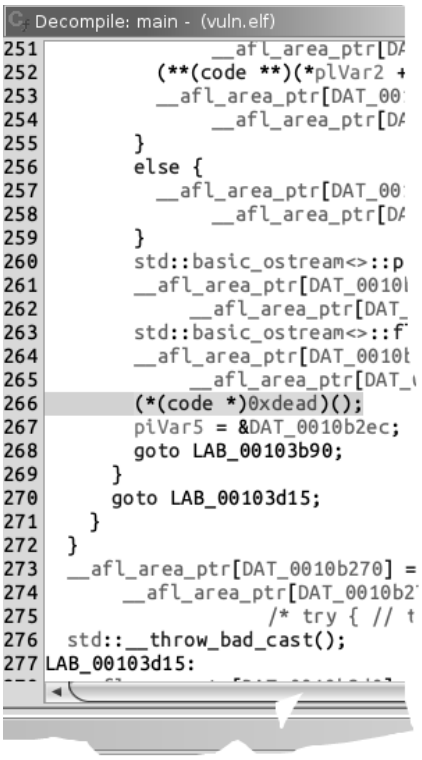
Może pojawić się informacja, że plik nie został jeszcze przeanalizowany, dlatego należy kliknąć Yes (pol. tak).



W kolejnym oknie można zostawić domyślne wybory i po prostu kliknąć przycisk Analyze.



Język C++ i instrumentacja (ang. code instrumentation) nałożona przez fuzzer delikatnie zaciemnia (ang. obfuscate) kod, ale udało znaleźć się charakterystyczną wartość **`0xDEAD`**.



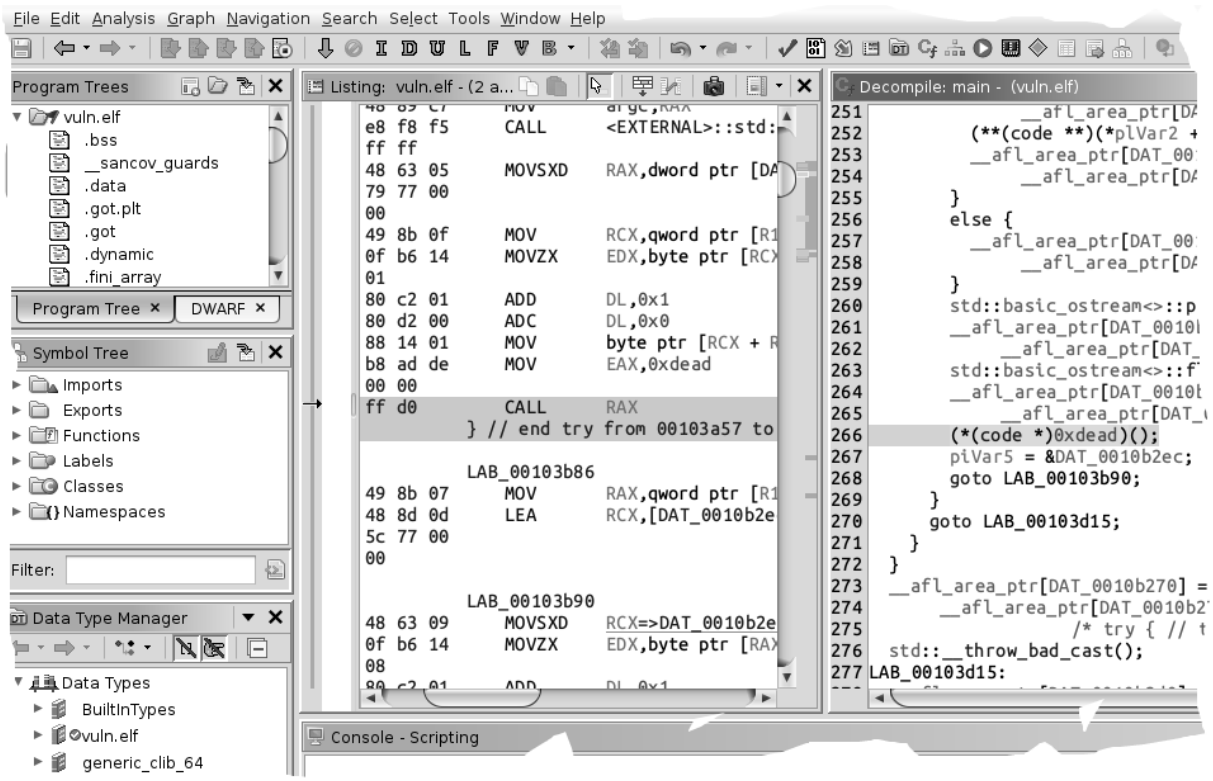
Instrukcja wywołująca awarię analizowanego programu w składni C/C++ to:

```
(*(void *)0xDEAD(); //crash
```

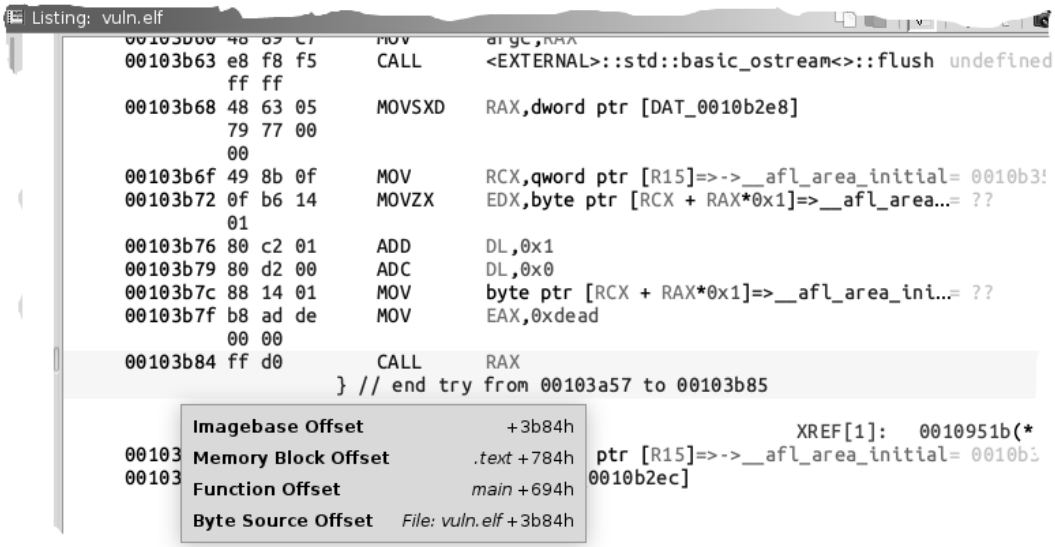
a w języku Asembler to:

```
mov eax, 0xDEAD
call rax ;crash
```

W oknie przeglądania kodu narzędzia Ghidra prezentuje się to jak poniżej.

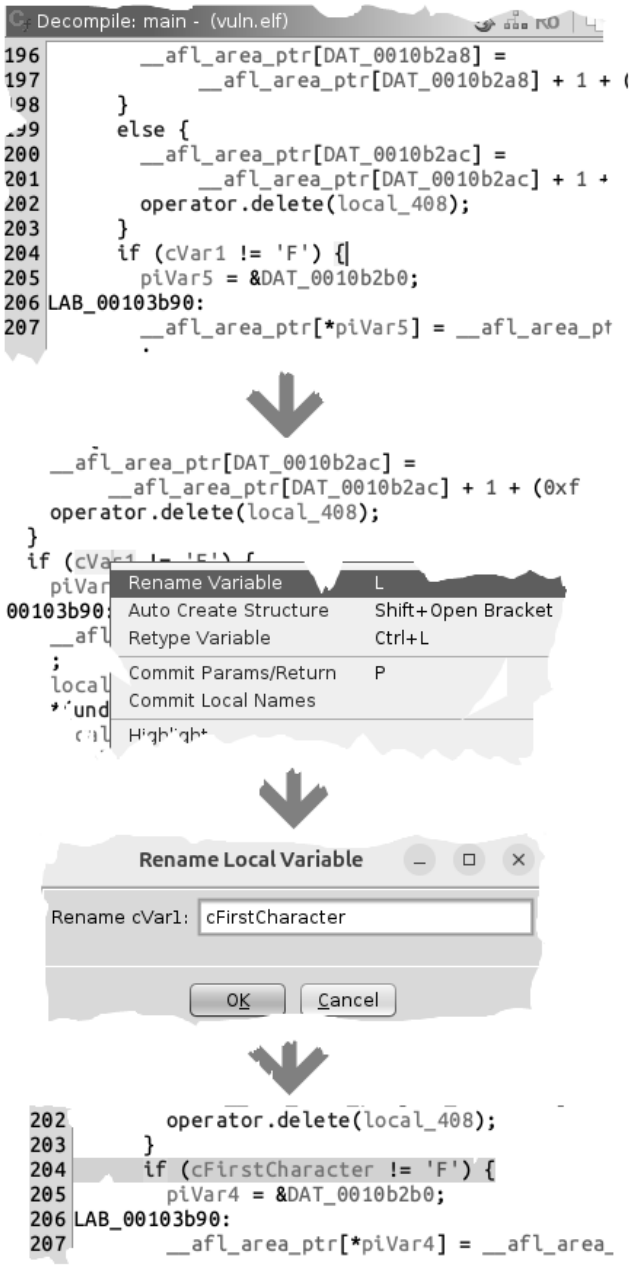


W celu ułatwienia zlokalizowania poszczególnych instrukcji można umieścić kursor myszy nad polem adresu np. 00103b84, a w oknie podpowiedzi (ang. tooltip) pojawi się miejsce instrukcji względem różnych miejsc bazowych.

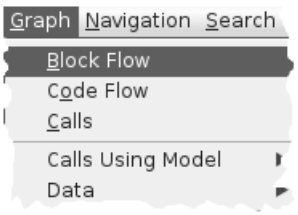


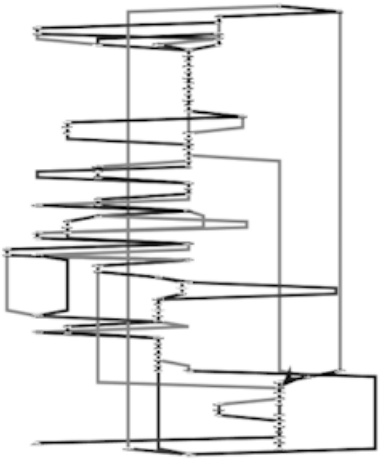
Jedną z podstawowych funkcjonalności narzędzia Ghidra podczas procesu ręcznego odwracania kodu jest możliwość zmiany typów czy też nazw poszczególnych elementów. Tego rodzaju zabieg potrafi bardzo, ale to bardzo zwiększyć przejrzystość i ułatwić analizę.

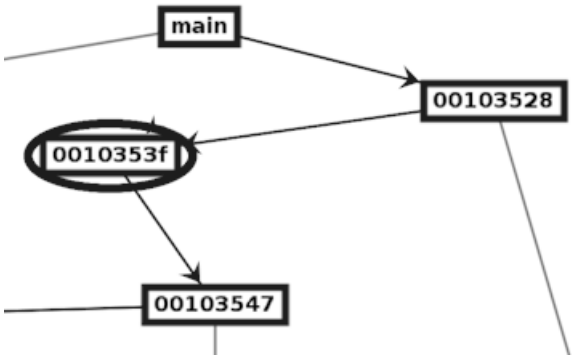
Na rysunku poniżej zaprezentowano zmianę nazwy zmiennej z mało znaczącego **`cVar1`** na **`cFirstCharacter`**.



Inna podstawowa funkcjonalność to grafy przepływu sterowania. Pozwolą ogólnie przyjrzeć się zależnością pomiędzy wywołaniami różnych fragmentów kodu, a także potrafią ochronić przed zagubieniem się i traceniem czasu na analizę mało istotnych fragmentów.







Grafy można przybliżać, a nawet układać według własnego uznania poszczególne bloki, aby ułatwić sobie analizę.

Niewłaściwa weryfikacja danych wejściowych

Autor: Dawid Farbaniec

Brak odpowiedniej weryfikacji danych wejściowych przeważnie prowadzi do błędów bezpieczeństwa w programach.

Wszystkie dane wprowadzane do kodu powinny zostać sprawdzone m.in. pod kątem:

+ rozmiaru (szczególnie wymaganej długości, a rzeczywistej długości),

+ miejsca do którego dane są zapisywane,

+ zgodności z wymaganym typem,

+ zgodności z wymaganą składnią,

+ zgodności z logiką aplikacji,

+ autentyczności np. poprzez sprawdzenie źródła danych czy podpisu,

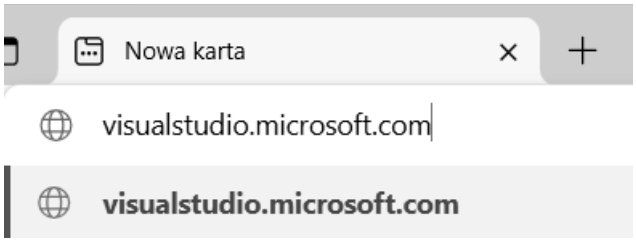
+ etc.

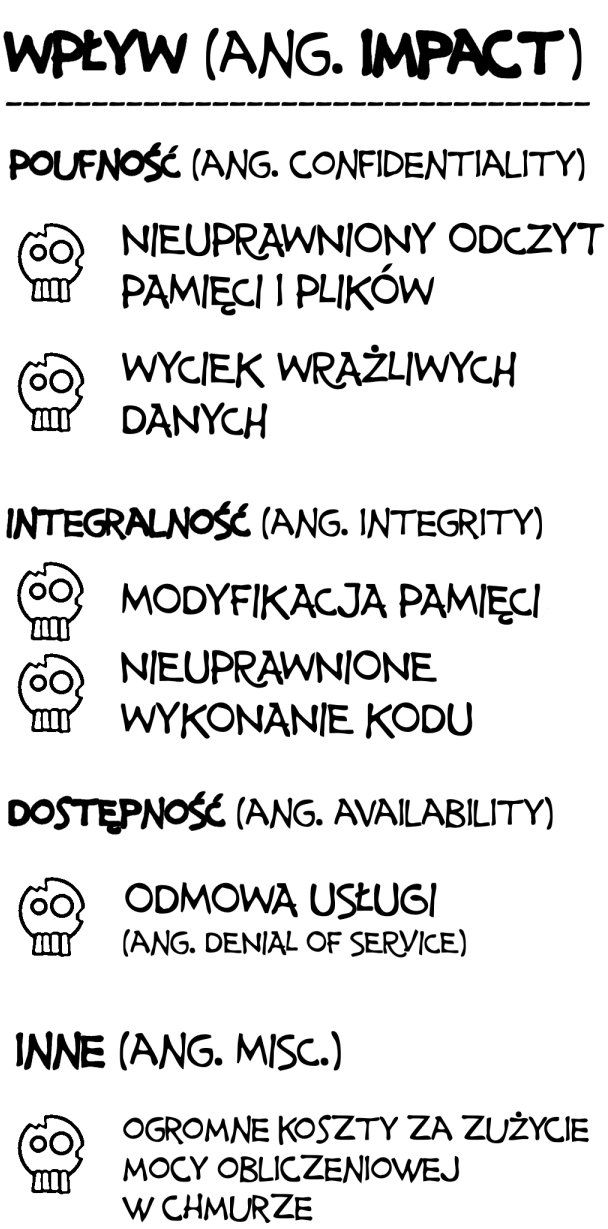
Dalej w celach eksperymentalnych umyślnie uszkodzimy podatny program w języku C++ dla systemu Windows.

Instalacja Microsoft Visual Studio

Środowisko programistyczne Visual Studio firmy Microsoft można pobrać w bezpłatnej wersji dla społeczności (ang. community) z witryny:

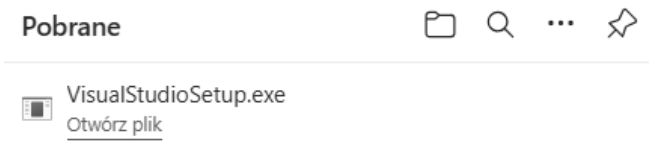
https://visualstudio.microsoft.com/



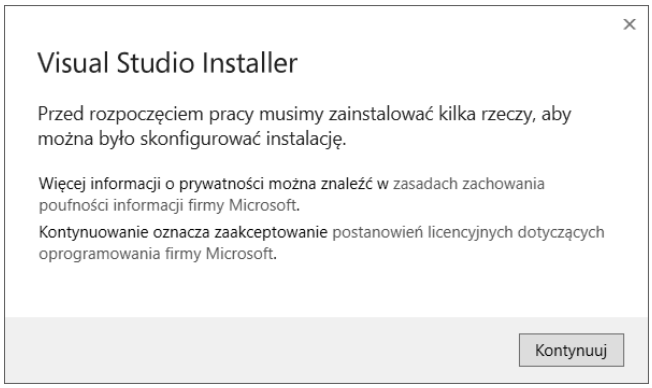


Do zastosowań indywidualnych można wybrać bezpłatną wersję Community.

Kolejny krok to uruchomienie pliku instalacyjnego.

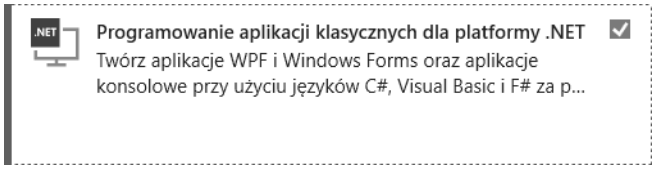


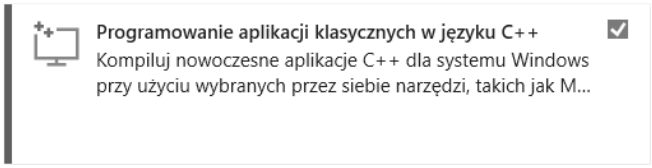
Należy zaakceptować postanowienia licencyjne.



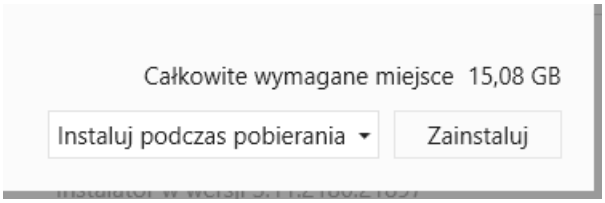
Do przyszłych zastosowań można zaznaczyć komponenty związane z technologią ASP.NET.

Warto też dodać składniki związane z aplikacjami klasycznymi dla platformy .NET oraz C++.

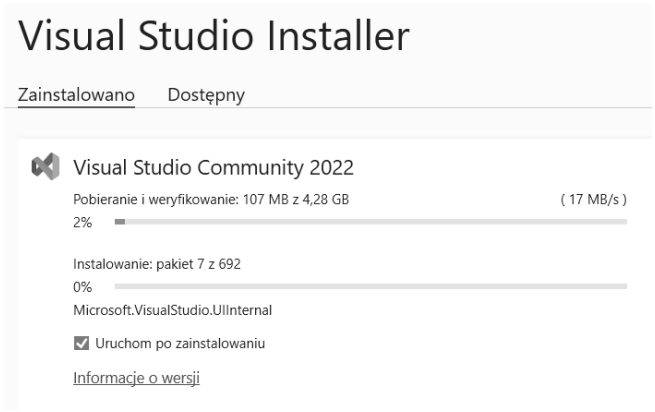


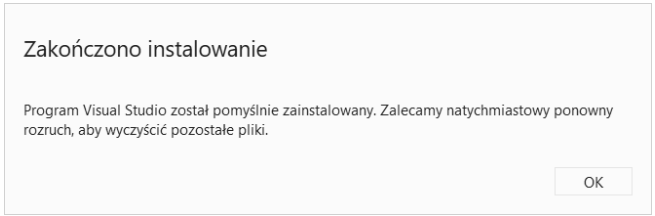


Przycisk Zainstaluj rozpoczyna instalację zaznaczonych wcześniej składników.



Instalacja środowiska Microsoft Visual Studio.

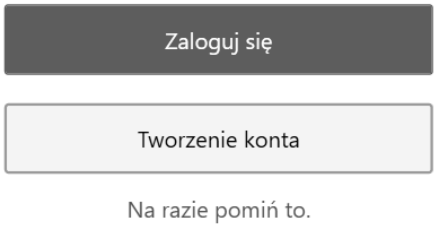




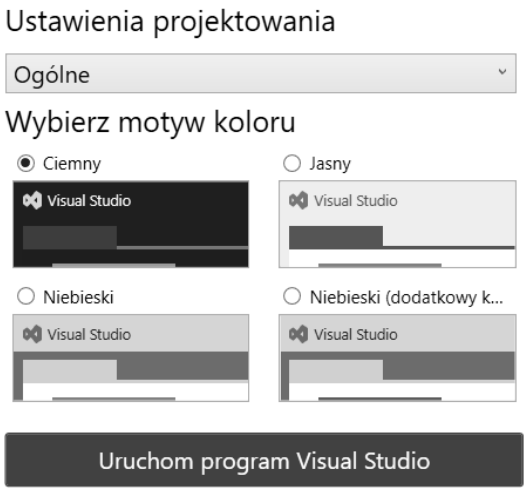
Teraz można ponownie uruchomić urządzenie, aby wyczyścić pliki pozostałe po instalacji.

Tworzenie projektu dla Visual C++

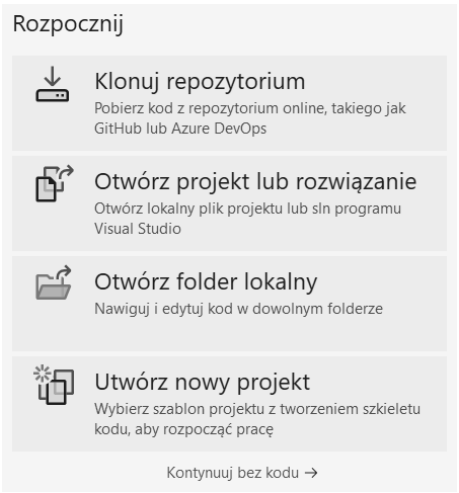
Środowisko programistyczne podczas uruchomienia zachęca do założenia konta, jednak na ten moment można wybrać opcję Na razie pomiń to.



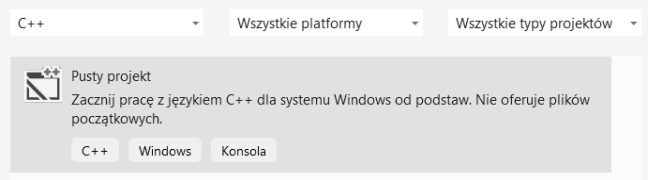
Ustawienia projektowania pozwalają dostosować wygląd środowiska do własnych preferencji oraz lepiej dopasować do używanych technologii.



Tutaj należy wybrać Utwórz nowy projekt.

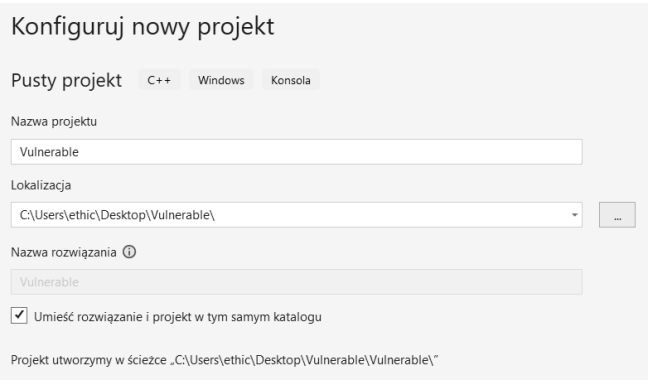


Dalej jako szablon projektu należy wybrać Pusty projekt (C++).



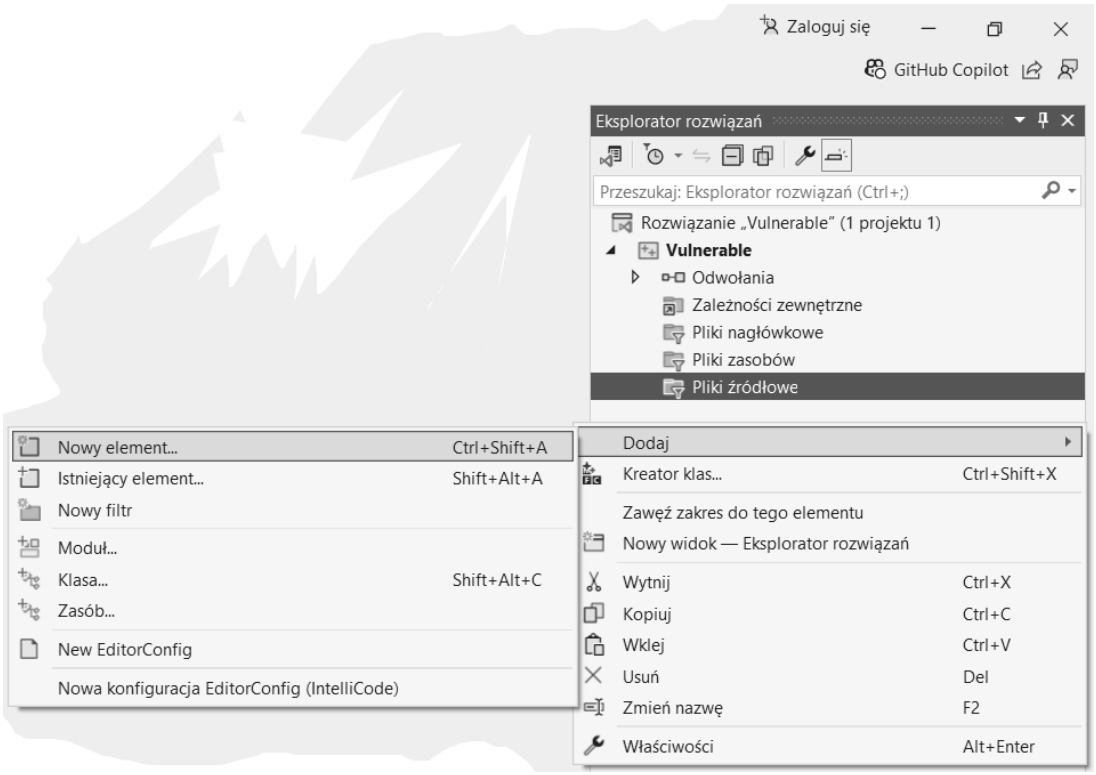
Można przejść dalej.

W kolejnym oknie należy wpisać nazwę projektu i wybrać folder w którym zostaną zapisane pliki rozwiązania.



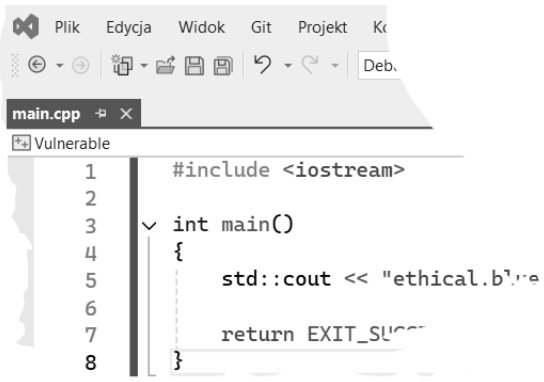
Teraz należy kliknąć przycisk Utwórz.

W celu dodania pliku z kodem źródłowym należy wybrać w oknie Eksplorator rozwiązań (ang. solution explorer) element Pliki źródłowe, kliknąć prawym przyciskiem myszy i z menu podręcznego wybrać Dodaj / Nowy element.



Zgodnie z przyjętymi zasadami główny plik źródłowy programu nazywa się słowem main (pol. główny).

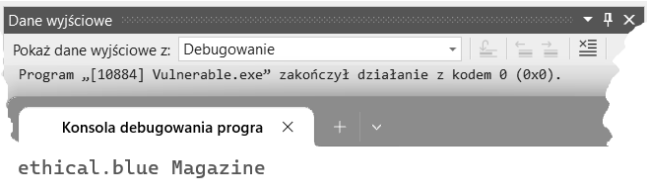
W celu weryfikacji ustawień można uruchomić prosty fragment kodu.




```
#include <iostream>

int main()
{
    std::cout << "ethical.blue Magazine"
        << std::endl;

    return EXIT_SUCCESS;
}
```

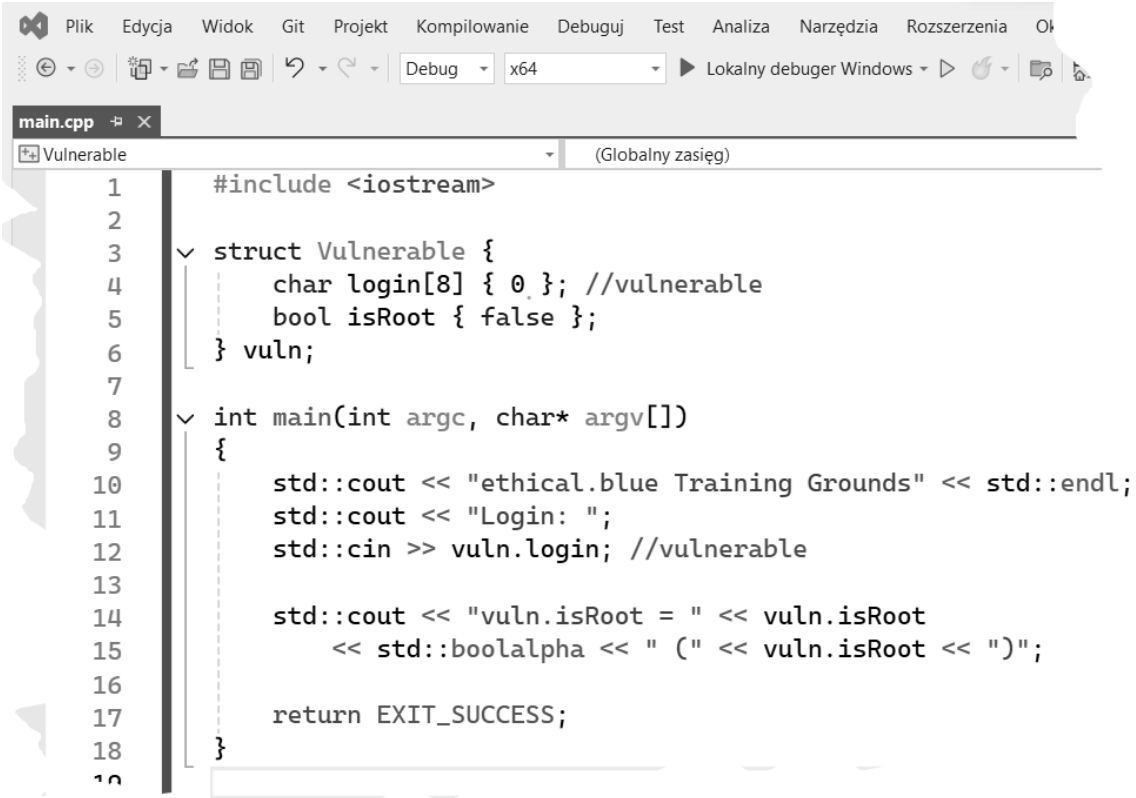
W celu uruchomienia programu należy kliknąć Lokalny debuger Windows.

Jeśli wszystko poszło dobrze, to na konsoli pojawi się tekst ethical.blue Magazine.



Przykładowy program z podatnościami

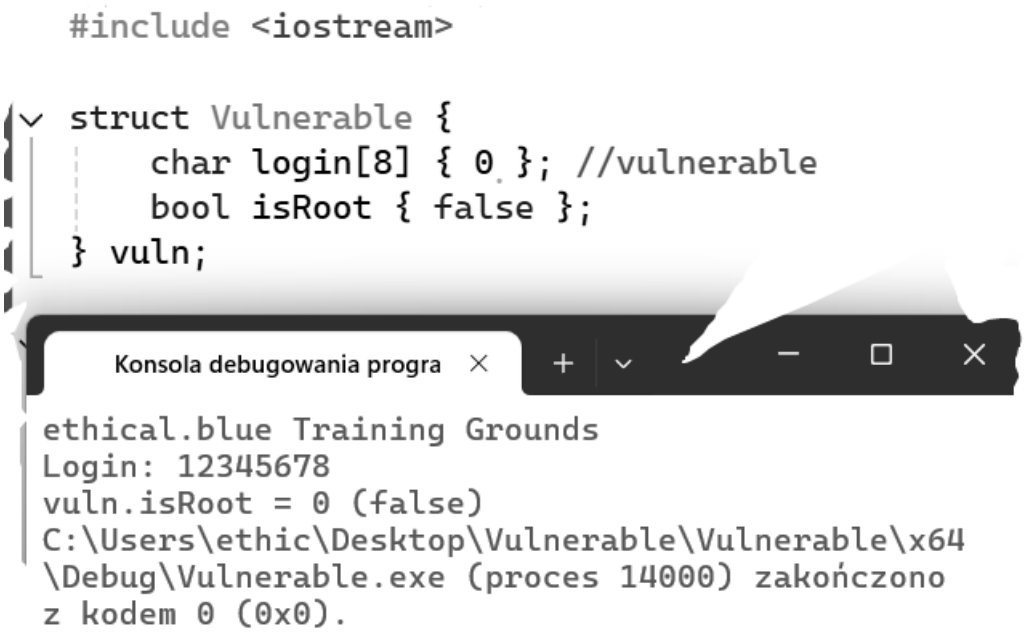
W celu zaprezentowania jak niebezpieczne potrafią być operacje na pamięci w języku C++ stworzono program z umyślnie popełnionym błędem bezpieczeństwa.



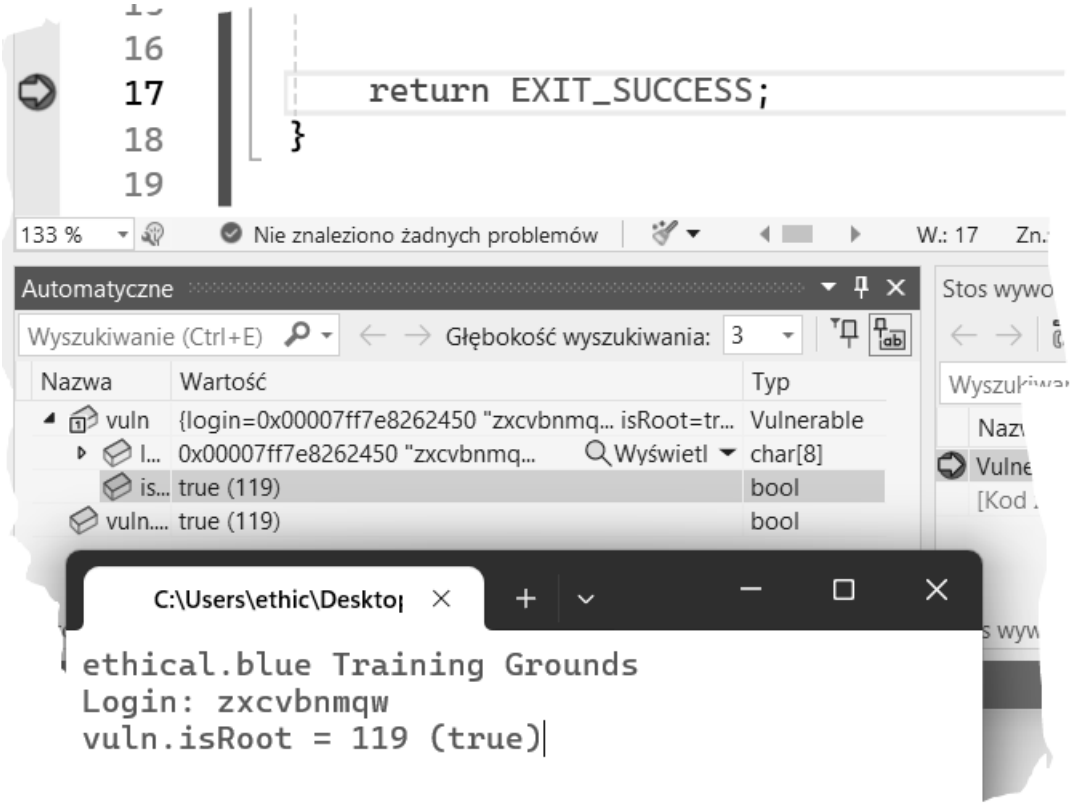
Program z podatnością (ang. vulnerability) pyta użytkownika o nazwę, którą zapisuje w zmiennej **`login`** w strukturze **`Vulnerable`**. Jednak zmienna **`login`** ma rozmiar osiem znaków (8 bajtów), a program nie posiada funkcji sprawdzającej ile bajtów jest zapisywane pod to miejsce w pamięci.

```
#include <iostream>

struct Vulnerable {
    char login[8] { 0 }; //vulnerable
    bool isRoot { false };
} vuln;

int main(int argc, char* argv[])
{
    std::cout << "ethical.blue Training Grounds" << std::endl;
    std::cout << "Login: ";
    std::cin >> vuln.login; //vulnerable

    std::cout << "vuln.isRoot = " << vuln.isRoot
        << std::boolalpha << " (" << vuln.isRoot << ")";

    return EXIT_SUCCESS;
}
```

Wszystko wydaje się być w porządku, gdy wprowadzi się osiem znaków lub mniej.



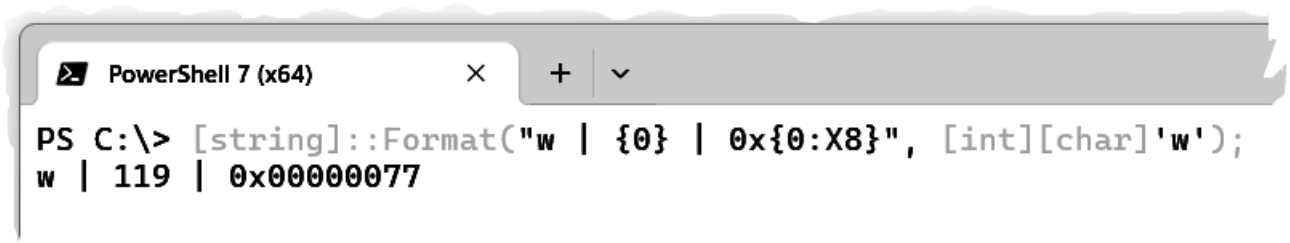
Jednak wprowadzenie dziewięciu znaków powoduje nadpisanie wartości znajdującej się za polem **`login`** struktury **`vuln`** (**`Vulnerable`**), a jest to zmienna typu logicznego, która decyduje o uprawnieniach (tylko przykładowo).



Dziewiąty znak, czyli ten, który został zapisany poza zmienną **login** to litera **w** o kodzie ASCII równym 119. Można to łatwo i szybko sprawdzić np. za pomocą takiej linijki w PowerShell:

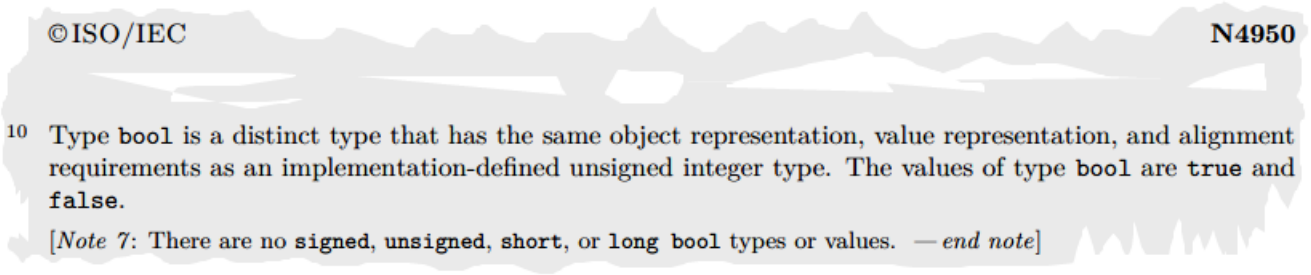
```
[string]::Format("w | {0} | 0x{0:X8}", [int][char]'w');
```

Powyższy fragment kodu to wywołanie metody **Format** z klasy **System.String**, gdzie jako parametry podano ciąg formatujący oraz literę **'w'** przekonwertowaną najpierw na typ **char**, a później **int**.



W myślach może jednak pojawić się pytanie w jaki sposób typ logiczny **bool** przyjął wartość 119. Przecież w kodzie źródłowym dla typu logicznego **bool** stosuje się jedną z dwóch wartości: **true** lub **false**.

Odpowiedź można znaleźć w standardzie języka C++, który informuje, że typ **bool** jest wewnętrznie reprezentowany przez typ całkowity bez znaku (ang. unsigned integer). Z tego też wynika, że zero to **false**, a każda inna wartość to **true**.



©ISO/IEC N4950

10 Type `bool` is a distinct type that has the same object representation, value representation, and alignment requirements as an implementation-defined unsigned integer type. The values of type `bool` are `true` and `false`.

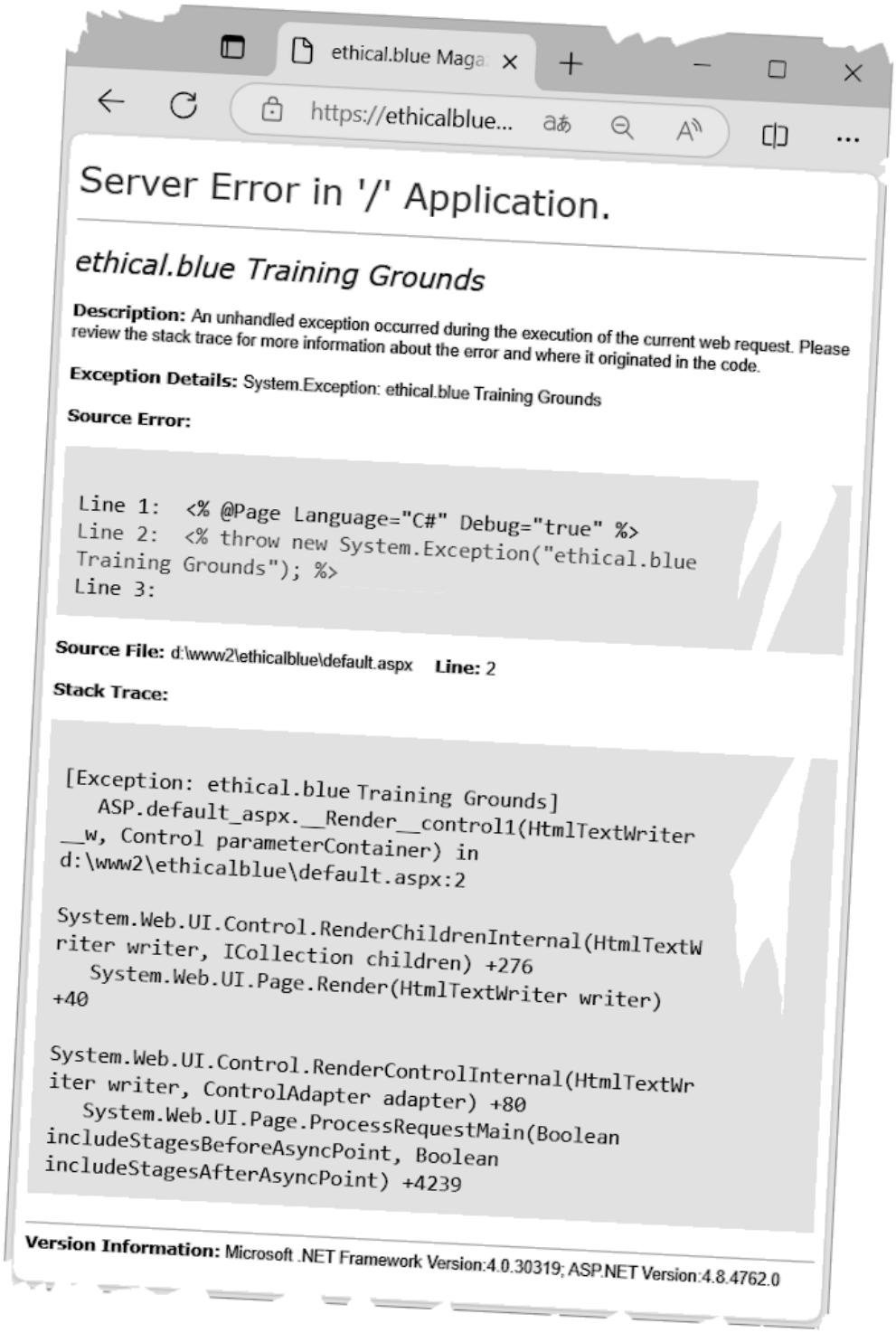
[*Note 7*: There are no **signed**, **unsigned**, **short**, or **long** **bool** types or values. — *end note*]

Working Draft, Standard for Programming Language C++ (Document Number: N4950)

Plik binarny programu w konfiguracji DEBUG i szczegółowe strony błędów

Autor: Dawid Farbaniec

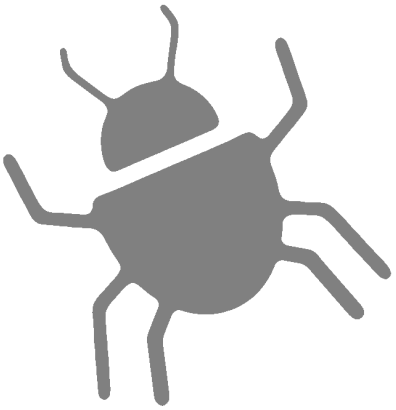
Szczegółowe komunikaty o błędach lub pozostawione domyślne strony błędów pozwalają podmiotom zagrażającym (ang. threat actor) przeprowadzić skuteczne rozpoznanie celu. Należy unikać publikowania aplikacji w trybie DEBUG oraz wyświetlania szczegółowych informacji na temat działającego systemu.



Ekspozycja krytycznego parametru na zewnątrz poza środowisko

Autor: Dawid Farbaniec

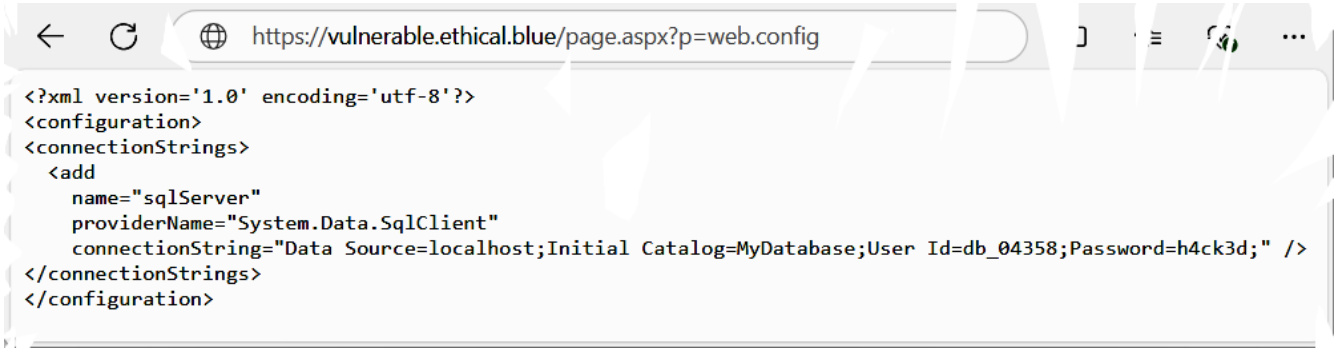
Dane z zewnątrz wprowadzane do kodu powinny być zawsze traktowane jako niezaufane (ang. untrusted). Nie powinno się też pozwalać na zmianę ustawień środowiska bez odpowiedniej weryfikacji wprowadzanych zmian.

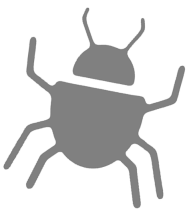


```
#include <iostream>
#include <Windows.h>

int main(int argc, char* argv[])
{
	// ...

	SetComputerNameA(argv[1]); //vulnerable

	// ...

	return EXIT_SUCCESS;
}
```

Powyższy przykład prezentuje przykładowy błąd w którym argument wiersza polecenia wywoływanego programu jest bez weryfikacji przekazywany funkcji zmieniającej nazwę NetBIOS maszyny.

Innym przykładem może być bezpośrednie połączenie funkcji odczytującej plik z parametrem aplikacji internetowej. Jeśli **`page.aspx`** bez odpowiedniego filtrowania odczytuje plik z bieżącego katalogu o nazwie podanej w parametrze **`p`**, to podmioty zagrażające (ang. threat actor) mogą uzyskać dostęp też do innych plików.





Niezamierzone przejście między folderami (ang. path traversal)

Autor: Dawid Farbaniec

```
[System.Text.Encoding]::Default.GetString([System.Convert]::FromBase64String("LS09PSBldGhpY2FsLmJsdWUgVHJhaW5pbmcgR3JvdW5kcyA9PS0t"));
--== ethical.blue Training Grounds ==--
```

Ścieżka do pliku czy folderu może być w formie bezpośredniej np. **C:\WINDOWS**, ale może też przyjąć formę względem bieżącego katalogu (ang. relative path). W przypadku ścieżek względnych należy wspomnieć o dwóch specjalnych nazwach folderów. Pojedyncza kropka (**.**) oznacza katalog bieżący, natomiast dwie kropki (**..**) oznaczają katalog wyżej względem folderu bieżącego. Przykładowe polecenia PowerShell poniżej.

```
PS C:\> [System.IO.Path]::GetFullPath(".");
C:\WINDOWS\system32

PS C:\> [System.IO.Path]::GetFullPath("..");
C:\WINDOWS

PS C:\> [System.IO.Path]::GetFullPath("..\..\");
C:\

PS C:\> [System.IO.Path]::GetFullPath("..\TEMP");
C:\WINDOWS\TEMP

PS C:\> [System.IO.Path]::DirectorySeparatorChar
\

PS C:\> [System.IO.Path]::AltDirectorySeparatorChar
/
```

Warto też wspomnieć o ścieżkach w przypadku zasobów sieciowych (ang. Universal Naming Convention), które rozpoczynają się dwoma znakami ukośnika (ang. backslash) i nazwą maszyny, czyli np. **\\HEAVEN\C:** wskazuje na dysk oznaczony literą **C** na maszynie o nazwie **HEAVEN**.

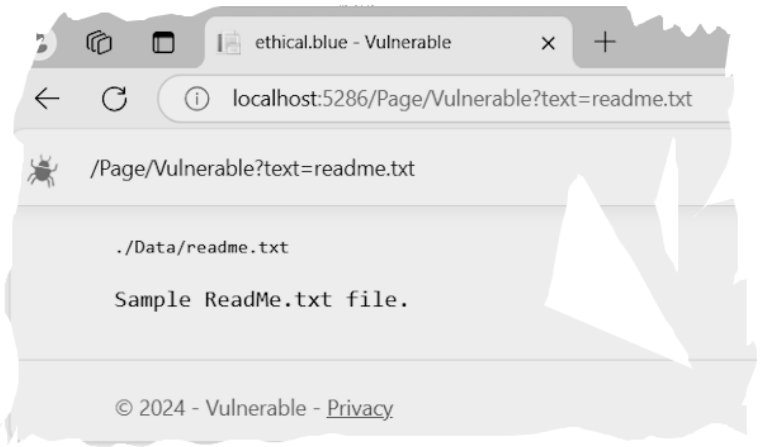
```
PS C:\> [System.IO.Path]::GetFullPath("\\HEAVEN");
\\HEAVEN

PS C:\> [System.IO.Path]::GetFullPath("\\HEAVEN\C:\");
\\HEAVEN\C:\

PS C:\> [System.IO.Path]::GetFullPath("\\HEAVEN\C:\Windows\..");
\\HEAVEN\C:

PS C:\> [System.IO.Path]::GetFullPath("\\HEAVEN\C:\Windows\system32\..");
\\HEAVEN\C:\Windows

PS C:\>
```

Podatności (ang. vulnerability) związane z nieprawidłowym przetwarzaniem ścieżek nazywane są w języku angielskim **path traversal** lub **directory traversal**.

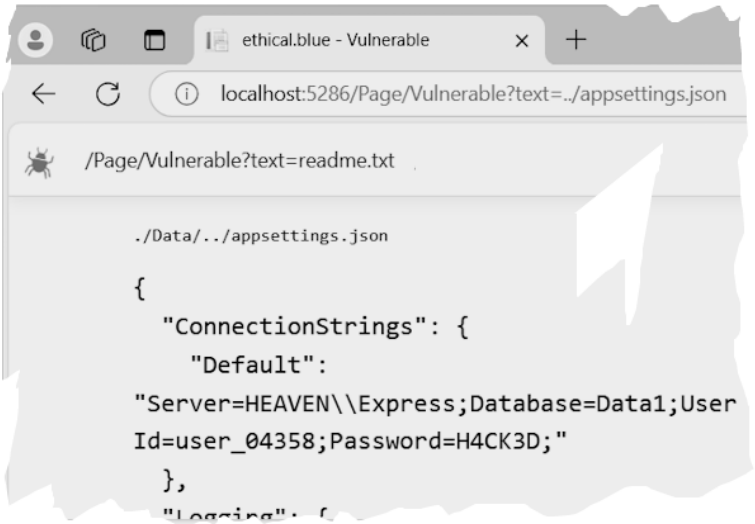
Poniższy fragment kodu prezentuje akcję **Vulnerable** kontrolera **Page** w przykładowej aplikacji internetowej w technologii ASP.NET. Parametr **text** określa, który plik z katalogu z danymi (**Data**) program ma odczytać. Rozwiązanie tutaj zastosowane zawiera umyślne podatności, aby pokazać jak bardzo tego rodzaju fragment kodu może być niebezpieczny.

```
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;
using Vulnerable.Models;

namespace Vulnerable.Controllers;

public class PageController : Controller
{
    /* ... */

    public IActionResult Vulnerable(string text = "readme.txt")
    {
        if(System.IO.File.Exists($"./Data/{text}"))
            ViewBag.Text = System.IO.File.ReadAllText($"./Data/{text}");

        ViewBag.Debug = $"./Data/{text}";

        return View();
    }

    /* ... */
```

Wystarczy, że podmiot zagrażający (ang. threat actor) zastosuje sekwencję dwóch kropek (**../**), aby odczytywać inne pliki z katalogu programu.



Obok można zobaczyć, że wprowadzenie wartości **../appsettings.json** dla parametru **text** pozwoliło na odczytanie pliku konfiguracyjnego.

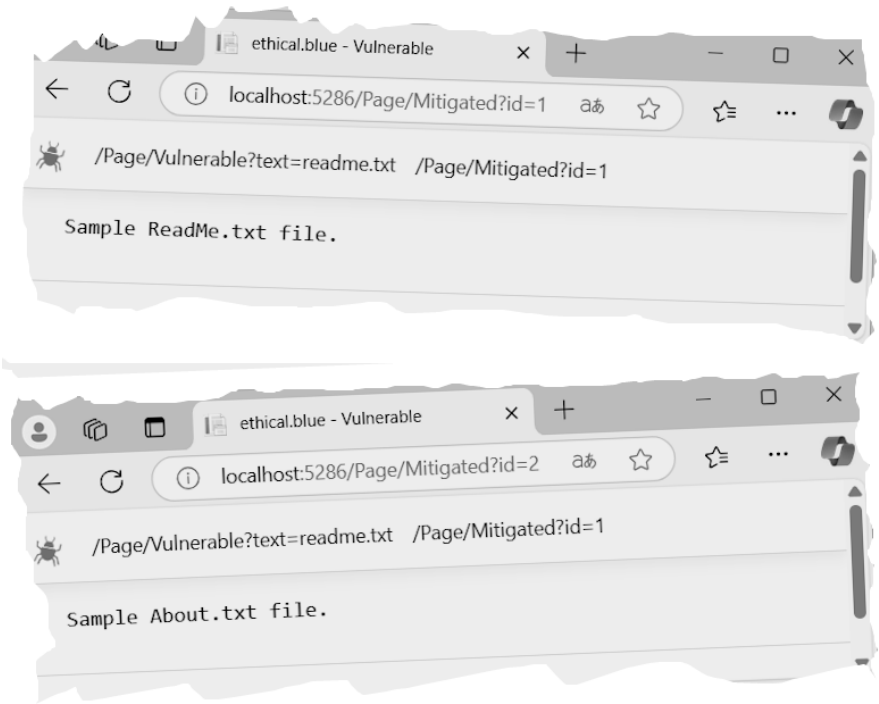


Dość skutecznym rozwiązaniem pozwalającym na złagodzenie (ang. mitigation) podatności typu **path traversal** jest wprowadzenie dodatkowej warstwy izolującej podobnie jak zaprezentowano poniżej.

```
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;
using Vulnerable.Models;

namespace Vulnerable.Controllers;

public class PageController : Controller
{
    /* ... */

    public IActionResult Mitigated(int id = 1)
    {
        if (id == 1)
            ViewBag.Text = System.IO.File.ReadAllText("./Data/readme.txt");
        else if (id == 2)
            ViewBag.Text = System.IO.File.ReadAllText("./Data/about.txt");
        else
            ViewBag.Text = string.Empty;

        return View();
    }

    /* ... */
```

W zaprezentowanym obok rozwiązaniu można zauważyć, że parametr odpowiedzialny za nazwę pliku do wczytania nie jest eksponowany na zewnątrz.

Nazwa pliku została zastąpiona identyfikatorem typu **`int`** na podstawie którego wczytywane są pliki z katalogu z danymi (**`Data`**).



Warstwa izolacyjna tego rodzaju jest skuteczniejsza od filtrowania znaków kropki (**`../`**) w ścieżce.

Nieprawidłowa neutralizacja elementów specjalnych pozwalająca na wstrzyknięcie poleceń

Autor: Dawid Farbaniec

W celu prawidłowej neutralizacji elementów specjalnych należy korzystać z bezpiecznych funkcji oferowanych przez środowisko oraz przy wprowadzaniu danych do kodu pomyśleć o technologii na której oparty jest system i co w związku z tym może zostać źle zinterpretowane.

Podstawowe ataki polegające na wstrzyknięciu poleceń to m.in.

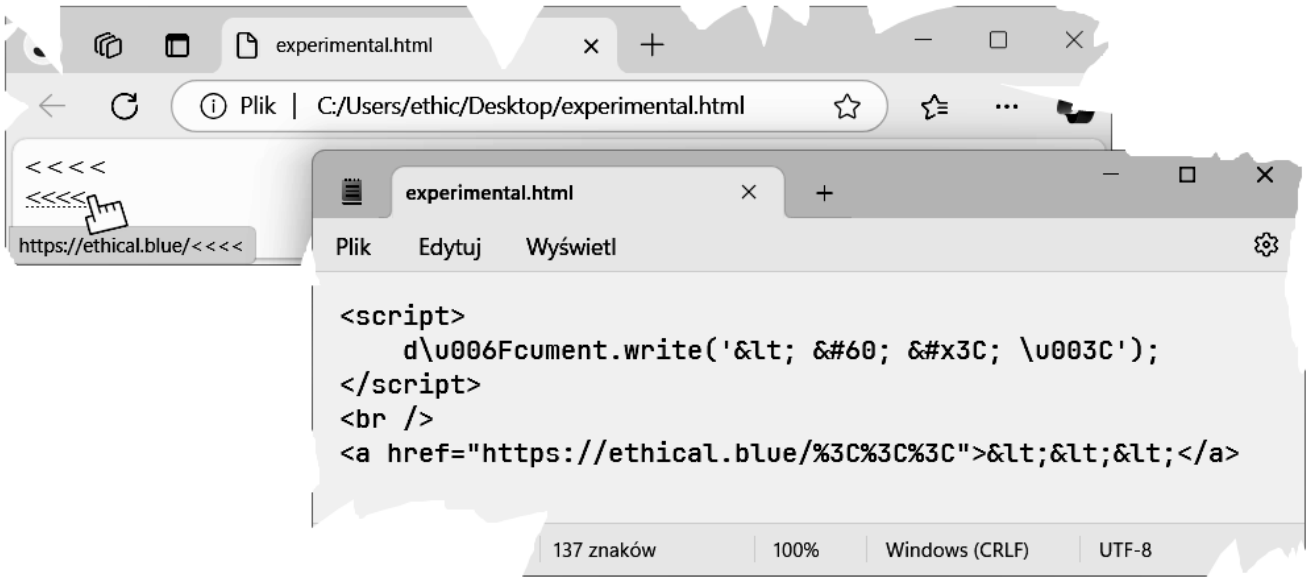
- XML Injection (pol. wstrzyknięcie do znaczników XML),
- CRLF Injection (pol. wstrzyknięcie znaku końca linii),
- Poison Null Byte (pol. zatrucie bajtem zerowym, czyli wstrzyknięcie bajtu zerowego powodującego urwanie interpretacji sekwencji w określonym miejscu),
- File Include (pol. wstrzyknięcie nazwy pliku i nieautoryzowane dołączenie zawartości pliku w danym miejscu w kodzie),
- SQL Injection (pol. wstrzyknięcie poleceń do zapytania w języku SQL),
- CSV Injection (pol. wstrzyknięcie do danych w formacie Comma Separated Values),
- Zip Slip (pol. wstrzyknięcie do archiwum relatywnej ścieżki np. `../../evil.elf` co w przypadku braku izolacji może spowodować wypakowanie pliku w niedozwolone miejsce),
- ...i inne.

Cross-Site Scripting (XSS) — podatność polegająca na błędnym zinterpretowaniu zwykłych danych jako skrypt lub znaczniki HTML. Powodem tego rodzaju błędów jest brak neutralizacji specjalnych elementów przez nieprawidłowe zakodowanie. Np. znak mniejszości otwierający znacznik HTML może być zakodowany na różne sposoby m.in.

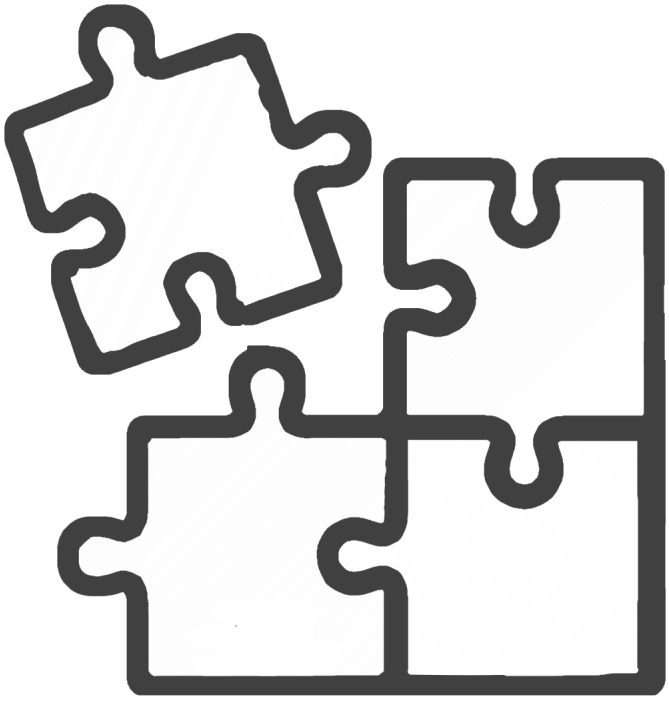
- `<`
- `<`
- `&#60;`
- `&#x3C;`
- `\u003C`
- `%3C`

Trwałe (ang. persistent, stored) — złośliwy skrypt jest zapisany trwale np. w bazie danych aplikacji internetowej i wykonywany przy każdym wczytaniu witryny.
Nietrwałe (ang. reflected, non-persistent) — złośliwy skrypt przeważnie jest podawany np. jako wartość parametru aplikacji internetowej.

Oparte o modyfikacje DOM (ang. Document Object Model) — złośliwy skrypt modyfikuje elementy DOM takie jak na przykład `document.body.innerHtml` czy `document.forms...` etc.



ŁAMIGŁÓWKA LOGICZNA



Różne ścieżki wskazujące na ten sam plik lub folder (ang. path equivalence)

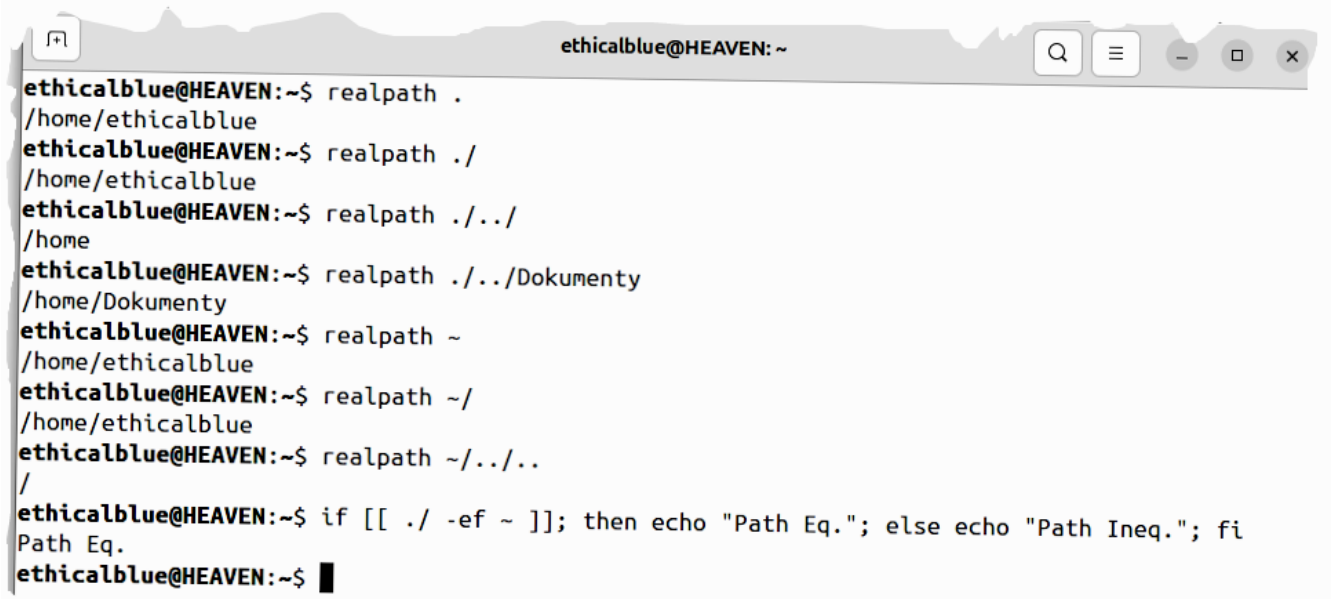
Autor: Dawid Farbaniec

Zależnie od środowiska niektóre ścieżki pomimo różnego zapisu wskazują na ten sam plik lub katalog. Ma to szczególne znaczenie m.in. podczas niewłaściwej próby umieszczenia pliku czy katalogu na liście elementów zabronionych.

Przykładowe ścieżki w systemie Windows.

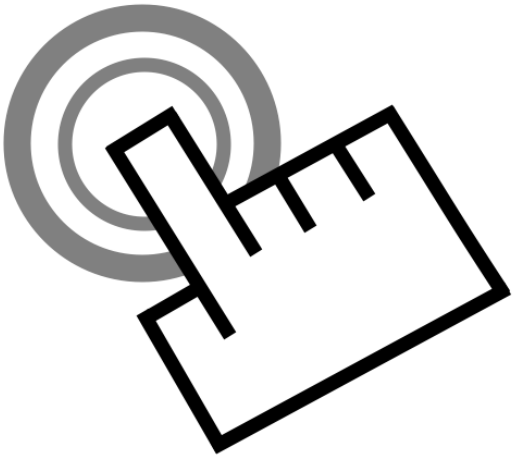
```
C:\WINDOWS                 ==      C:\WINDOWS\.              (Path Equivalence)
C:\WINDOWS                 ==      C:\WINDOWS\....           (Path Equivalence)
C:\WINDOWS                 ==      C:\WINDOWS                (Path Equivalence)
C:\WINDOWS                 ==      C:\WINDOWS/               (Path Equivalence)
C:\WINDOWS                 ==      C:\\\WINDOWS              (Path Equivalence)
C:\WINDOWS                 ==      C:\WINDOWS///             (Path Equivalence)
C:\WINDOWS\\\              ==      C:\WINDOWS                (Path Equivalence)
C:\                        ==      C:\WINDOWS\..             (Path Equivalence)
C:\WINDOWS\notepad.exe     !=      C:\WINDOWS\write.exe      (Path Inequivalence)
//...and more.
```

Przykładowe ścieżki w systemie Linux.



Przykładowy program w języku C#, który weryfikuje czy ścieżki wskazują na ten sam plik lub katalog.

```
internal class Program
{
    /// <summary>
    /// --== ethical.blue Training Grounds ==--
    /// </summary>
    private static void Main(string[] args)
    {
        List<(string a, string b)> p = [
            new(@"C:\WINDOWS", @"C:\WINDOWS\."),
            new(@"C:\WINDOWS", @"C:\WINDOWS\...."),
            new(@"C:\WINDOWS", @"C:\WINDOWS "),
            new(@"C:\WINDOWS", @"C:\WINDOWS/"),
            new(@"C:\WINDOWS", @"C:\\\WINDOWS"),
            new(@"C:\WINDOWS", @"C:\WINDOWS///"),
            new(@"C:\WINDOWS\\\", @"C:\WINDOWS"),
            new(@"C:\", @"C:\WINDOWS\.."),
            new(@"C:\WINDOWS\notepad.exe", @"C:\WINDOWS\write.exe")
            //...
        ];

        foreach (var item in p)
        {
            string a = Path.GetFullPath(item.a)
                .TrimEnd(Path.DirectorySeparatorChar);
            string b = Path.GetFullPath(item.b)
                .TrimEnd(Path.DirectorySeparatorChar);

            if (string.Equals(a, b))
                Console.WriteLine($"{item.a} == {item.b} (Path Equivalence)");
            else
                Console.WriteLine($"{item.a} != {item.b} (Path Inequivalence)");
        }
    }
}
```

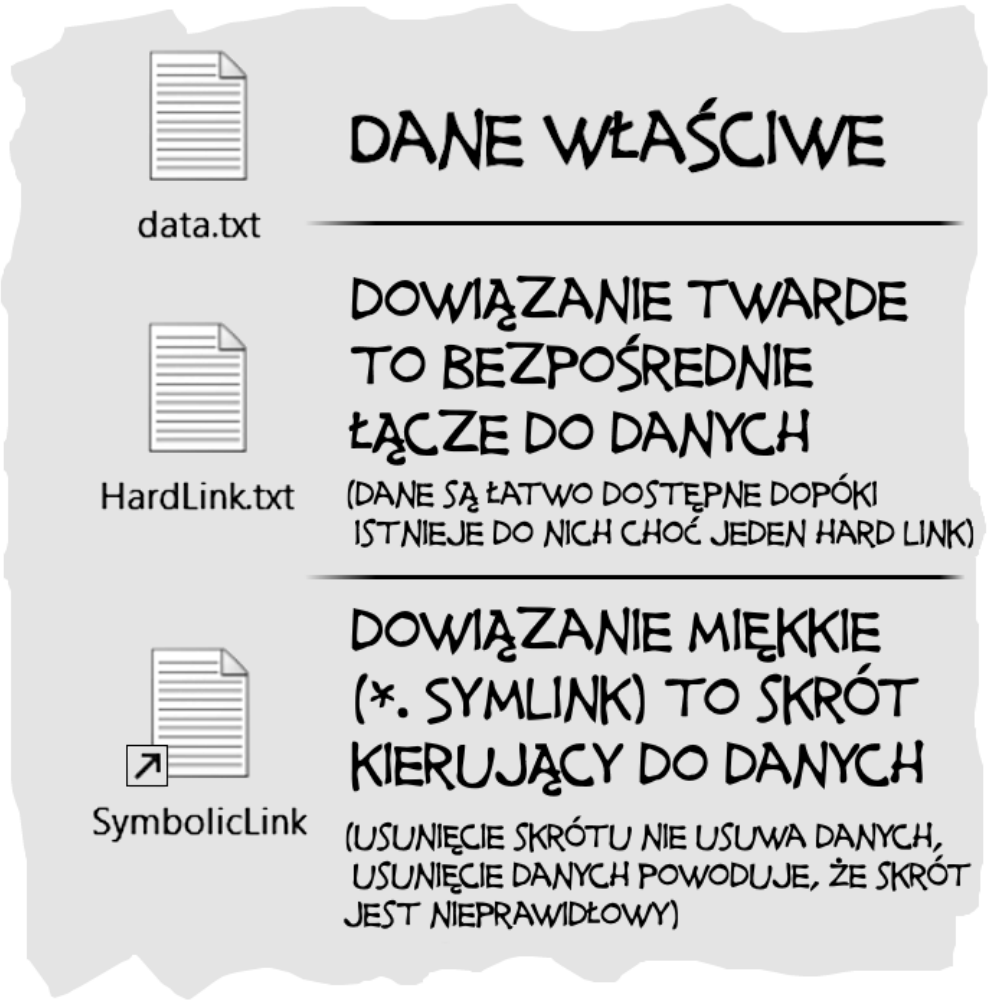


Dowiązania symboliczne (ang. symbolic link) oraz pliki skrótu (.LNK)

Autor: Dawid Farbaniec

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 odczytać, a nawet modyfikować.

```
struct WindowsShortcut {
   struct ShellLinkHeader sShellLinkHeader;
   struct LinkTargetIDList sLinkTargetIDList;
   struct LinkInfo sLinkInfo;
   struct StringData sStringData;
   struct ExtraData sExtraData;
};
```

Dowiązania symboliczne (*.SYMLINK) oraz pliki skrótu (*.LNK) mogą powodować wystąpienie różnych podatności w systemach przetwarzających dane. Nieprawidłowa obsługa dowiązań symbolicznych oraz skrótów .LNK w aplikacji może dać nieautoryzowany dostęp do plików spoza katalogu programu. Na potrzeby dalszych eksperymentów poniżej dołączono materiały potrzebne do tworzenia dowiązań symbolicznych.

Przykładowy program w języku C#, który pozwala tworzyć dowiązania symboliczne w systemie Windows.

```
using System.Runtime.InteropServices;

internal partial class Program
{
    [LibraryImport("Kernel32.dll",
        EntryPoint = "CreateHardLinkW",
        StringMarshalling = StringMarshalling.Utf16)]
    [return: MarshalAs(UnmanagedType.Bool)]
    internal static partial bool CreateHardLinkW(
      string lpFileName, string lpExistingFileName, IntPtr lpSecurityAttributes
    );

    /// <summary>
    /// --== ethical.blue Training Grounds ==--
    /// </summary>
    private static void Main(string[] args)
    {
        string desktopFolder = Environment.GetFolderPath(
            Environment.SpecialFolder.Desktop)
            .TrimEnd(Path.DirectorySeparatorChar);

        File.CreateSymbolicLink(@$"{desktopFolder}\SymbolicLink",
            @$"{desktopFolder}\data.txt");

        CreateHardLinkW(@$"{desktopFolder}\HardLink.txt",
            @$"{desktopFolder}\data.txt", IntPtr.Zero);
    }
}
```

Natomiast dowiązania symboliczne w Linuxie można utworzyć np. poleceniem **`ln --symbolic ...`** jak zaprezentowano poniżej.

Niekontrolowane wysłanie pliku niebezpiecznego typu

Ryzyko przyjęcia przez aplikację internetową pliku *.SYMLINK, *.LNK czy jeszcze innego i przetworzenia w nieprawidłowy sposób można łagodzić (ang. mitigate) m.in. przez:

- Niekorzystanie z nazwy pliku wysyłanego przez formularz aplikacji internetowej.

```
//var thisIsUntrustedFileName = formFile.FileName; //vulnerable
var thisIsRandomFileName = Path.Combine("./Data", $"{Guid.NewGuid()}.txt");
```

- Zweryfikowanie rozmiaru pliku, aby uniknąć ataków odmowy usługi (ang. denial of service).
- Sprawdzanie właściwości pliku server-side, gdyż zabezpieczenia client-side łatwiej ominąć.
- Nieprzechowywanie wysyłanych plików w katalogach aplikacji internetowej.
- Wyłączenie przyjmowanym z zewnątrz plikom atrybutu wykonywalny (ang. executable).
- Zweryfikowanie czy plik nie zawiera niedozwolonego rozszerzenia lub zawartości.
- Warto też sprawdzić sygnaturę pliku (bajty nagłówka) w celu rozpoznania typu danych.

Alternatywny strumień danych w systemie plików NTFS

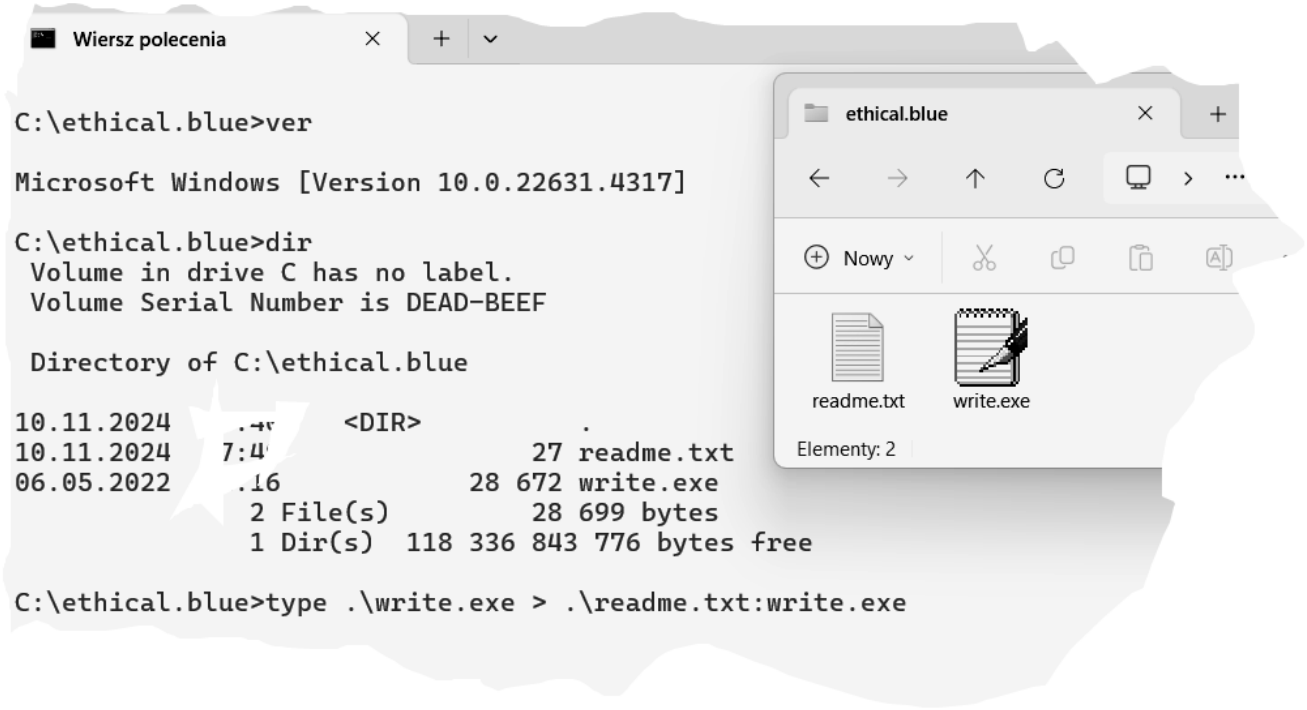
Autor: Dawid Farbaniec

Domyślnie dane związane z plikiem są zapisywane w głównym strumieniu. Jednak system plików NTFS obsługuje też dodatkowe, alternatywne strumienie danych powiązanych z określonym plikiem.

Podmioty zagrażające (ang. threat actor) mogą używać alternatywnych strumieni danych do ukrywania różnego rodzaju artefaktów, nawet w postaci całych plików binarnych.

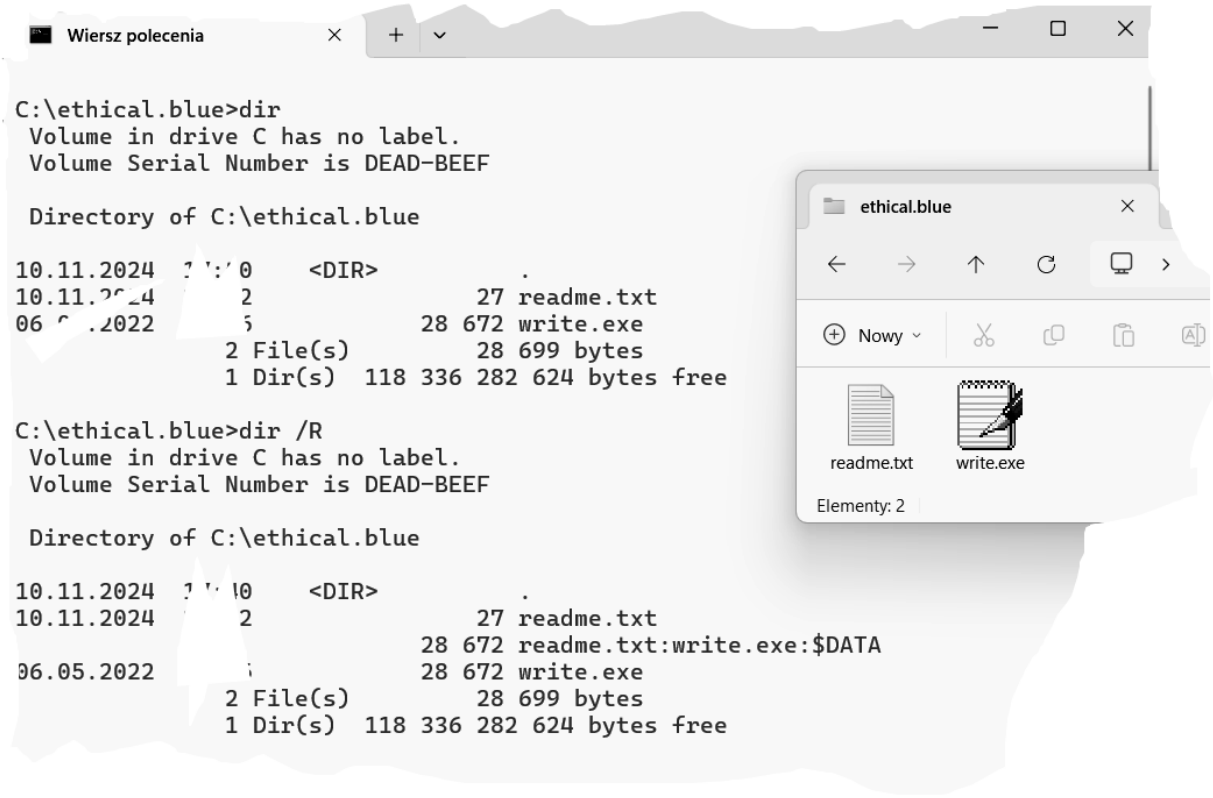
Eksperyment rozpoczyna się w folderze **C:\ethical.blue** w którym są dwa przykładowe pliki. Plik tekstowy readme.txt oraz program WordPad z systemu Windows (write.exe). Zawartość folderu można zweryfikować w domyślny sposób za pomocą polecenia **dir** w oknie Wiersza polecenia.

Dalej za pomocą polecenia **type .\write.exe > .\readme.txt:write.exe** następuje skopiowanie bajtów pliku write.exe do alternatywnego strumienia danych pliku readme.txt. Nazwa strumienia jest podana po dwuropku.



Jednym z prostszych sposobów na wyświetlenie alternatywnych strumieni danych jest polecenie **dir** z parametrem **/R**, czyli **dir /R**. Poniżej można też zauważyć, że pliki zachowane jako alternatywne strumienie nie są widoczne w oknie widoku folderu.

```
C:\>dir /?
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] ... [/R] ...

  /R    Display alternate data streams of the file.
```



Poniżej zaprezentowano, że po usunięciu oryginalnego pliku (**Remove-Item write.exe**), bajty z alternatywnego strumienia można uruchomić. Dowodem na to jest wywołanie, które otwiera okno WordPad (write.exe).

```
[System.Diagnostics.Process]::Start("C:\ethical.blue\readme.txt:write.exe");
```

Warto przeczytać

+ https://attack.mitre.org/techniques/T1564/004/
[dostęp: 2024-11-13 00:03]
+ https://learn.microsoft.com/en-us/sysinternals/downloads/streams
[dostęp: 2024-11-13 00:03]

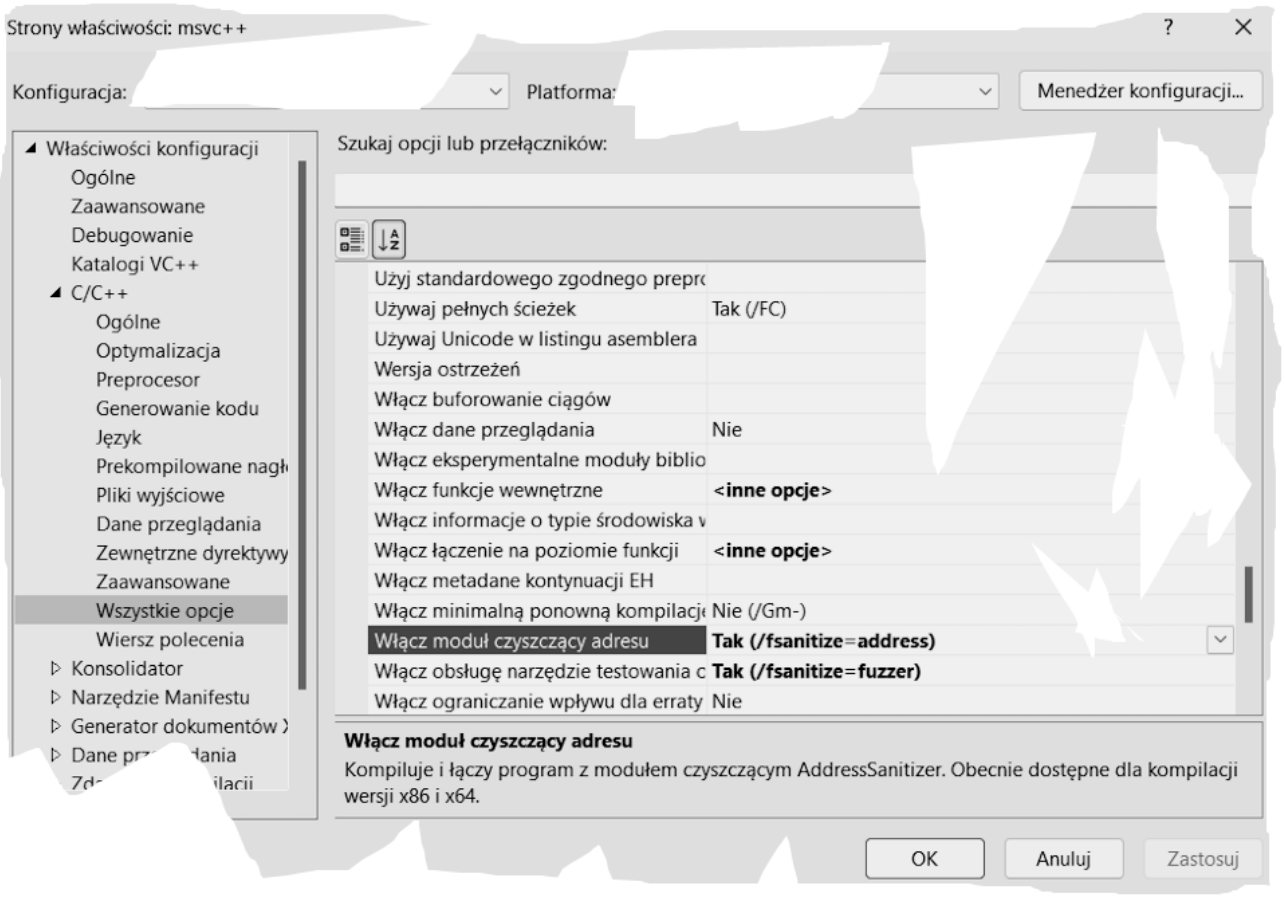
Błędy niewłaściwego korzystania z pamięci

Autor: Dawid Farbaniec

Język C++ cierpi na problemy z niewłaściwym korzystaniem z pamięci przez programistów, mimo że jest potężny oraz w niektórych sytuacjach niezastąpiony.

Moduł czyszczący adresu w Microsoft Visual C++ (AddressSanitizer)

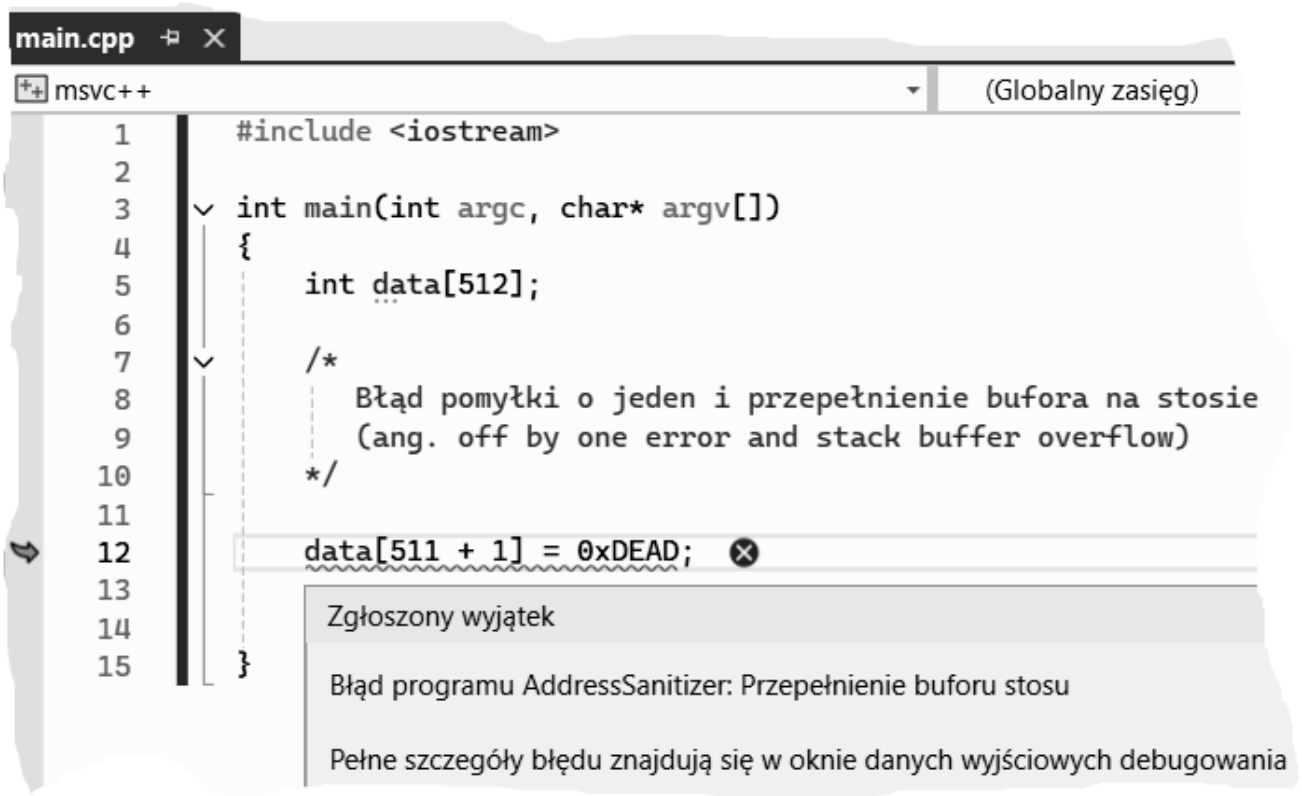
Włączenie modułu AddressSanitizer pozwoli uniknąć wielu błędów w kodzie tworzonych programów. Można to zrobić wybierając z górnego menu element Projekt / Właściwości.



Dalej przedstawiono podstawowe rodzaje błędów, które można popełnić, a w wykryciu których pomocny będzie moduł czyszczący adresu (AddressSanitizer).

Przepełnienie bufora na stosie z punktu widzenia piszącego kod programu

Błąd przepełnienia bufora na stosie to wpisanie do zmiennej większej ilości danych, niż przewidziano. Przykład przedstawiony poniżej to błąd pomyłki o jeden. Bufor o nazwie **`data`** spodziewa się maksymalnie pięćset dwanaście elementów, ale o indeksach od zera do pięćset jedenaście. Próba zapisu do elementu o indeksie pięćset dwanaście to wyjście poza zakres.



Warto przeczytać

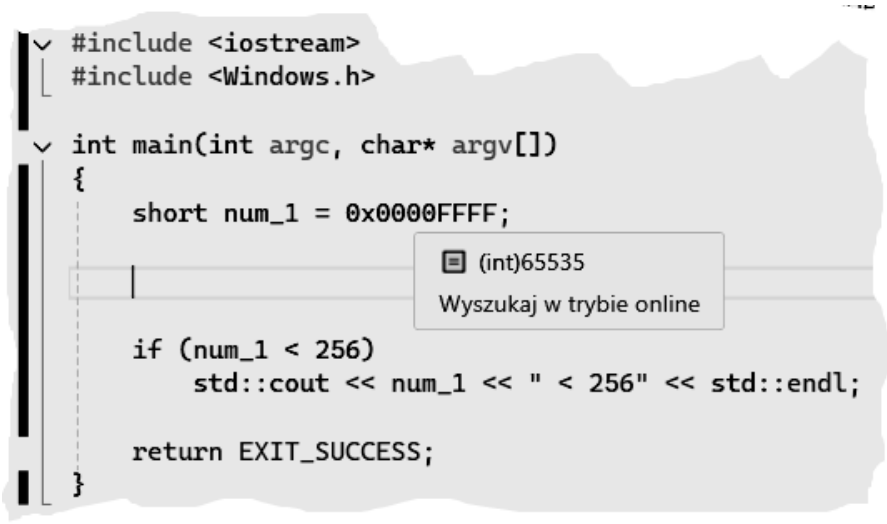
+ https://learn.microsoft.com/en-us/cpp/sanitizers/asan
[dostęp: 2024-11-13 00:03]

+ https://learn.microsoft.com/en-us/defender-endpoint/exploit-protection-reference
[dostęp: 2024-11-13 00:03]

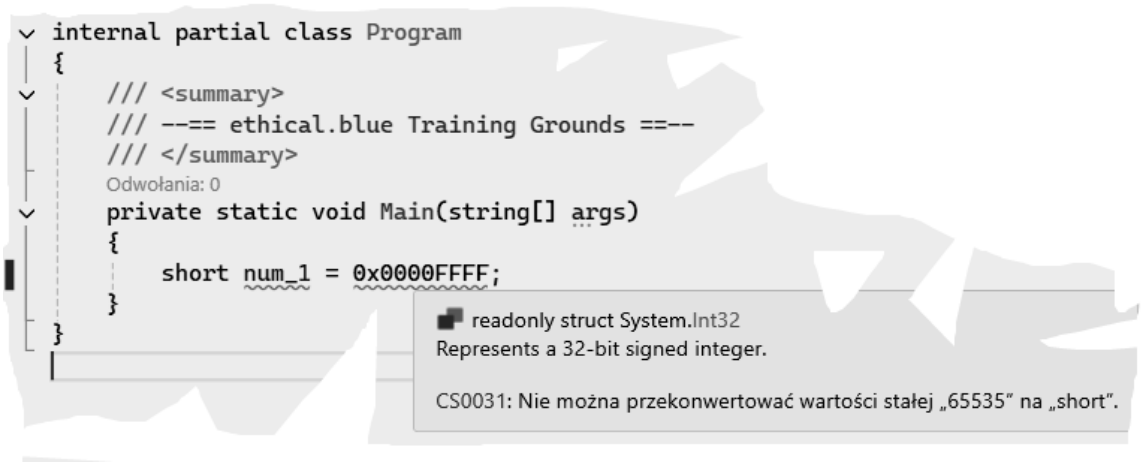
Błędy związane z konwersją typów prostych (ang. coercion, signed/unsigned)

Autor: Dawid Farbaniec

W języku C++ przypisanie stałej **0x0000FFFF** (**65535** lub **-1**) do zmiennej typu **short** może spowodować, że dalsze porównanie w instrukcji **if** to przyrównanie **-1 < 256**, a nie **65535 < 256**. Na szczęście środowiska programistyczne rozpoznają takie zapisy i przeważnie wyświetlą ostrzeżenie. Nie jest to jednak błąd kompilacji. Program się buduje, a błąd występuje podczas działania. Szczególnie groźne są przypadki, gdy nieprawidłowa wartość trafi do funkcji przeprowadzającej operacje na surowej pamięci.



O wiele bezpieczniej zachowuje się w takich sytuacjach język C#. Nie pozwoli na przypisanie takiej stałej, ani tym bardziej na zbudowanie programu.



Warto przeczytać

+ https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/checked-and-unchecked
[dostęp: 2024-11-13 00:03]

Użycie zwolnionej pamięci (ang. use after free)

Autor: Dawid Farbaniec

Podatność polegająca na ponownym użyciu zwolnionej pamięci niesie ze sobą szereg niebezpieczeństw. W tym eksperymencie zostanie zaprezentowane umyślne nadpisanie oraz odczytanie niedozwolonej wartości spod adresu w pamięci.

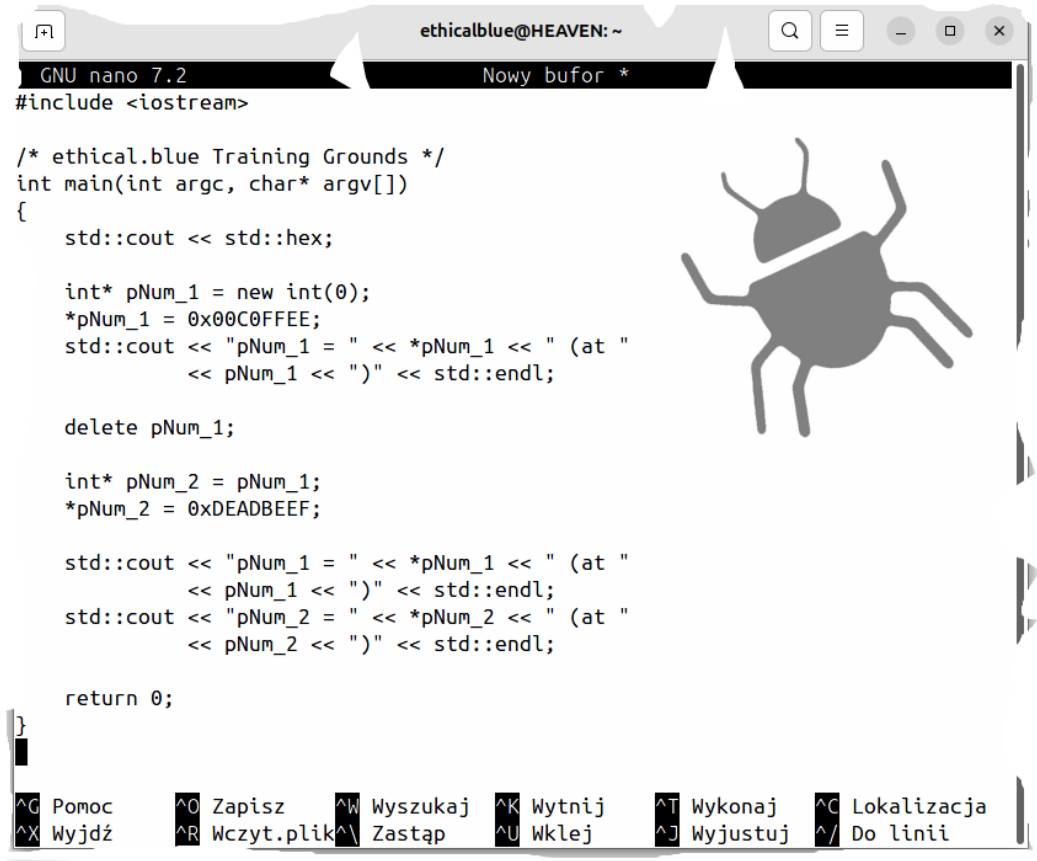
Jako laboratorium zostanie użyty system Ubuntu Desktop zainstalowany w postaci maszyny wirtualnej.

Edytor tekstu GNU nano można uruchomić wpisując w okno terminala polecenie **`editor`**.

Przykładowy program z podatnościami zaprezentowany na rysunku poniżej zawiera dwa wskaźniki na liczbę całkowitą (typu **`int`**) o nazwach **`pNum_1`** oraz **`pNum_2`**. Do pamięci wskazywanej przez wskaźnik **`pNum_1`** wpisywana jest przykładowa wartość w systemie szesnastkowym **`0x00C0FFEE`**. Po wyświetleniu wartości przez **`std::cout`** wskaźnik jest usuwany za pomocą operatora **`delete`**.

Teraz usunięty wskaźnik jest przypisywany do zmiennej **`pNum_2`** i pod wskazywane miejsce w pamięci jest wpisywana wartość **`0xDEADBEEF`**.

Jednak po użyciu wcześniej operatora **`delete`** pamięć została zwolniona i pod ten adres może trafić inna wartość, która może zostać błędnie nadpisana lub odczytana powodując wyciek informacji.



uaf.cpp (Use After Free, CWE-416)

Przykładowy kod dla podatności związanej z użyciem zwolnionej pamięci.

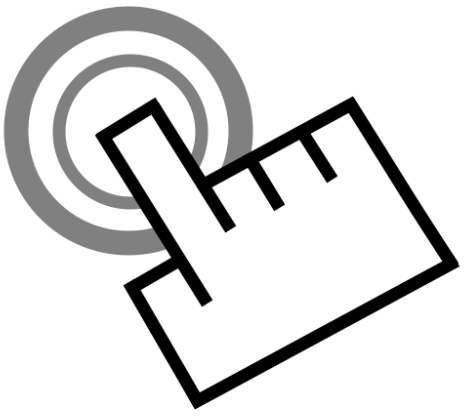
```
#include <iostream>

/* ethical.blue Training Grounds */
int main(int argc, char* argv[])
{
    std::cout << std::hex;

    int* pNum_1 = new int(0);
    *pNum_1 = 0x00C0FFEE;
    std::cout << "pNum_1 = " << *pNum_1 << " (at "
              << pNum_1 << ")" << std::endl;

    delete pNum_1;

    int* pNum_2 = pNum_1; //CWE-416
    *pNum_2 = 0xDEADBEEF; //Use After Free

    std::cout << "pNum_1 = " << *pNum_1 << " (at "
              << pNum_1 << ")" << std::endl;
    std::cout << "pNum_2 = " << *pNum_2 << " (at "
              << pNum_2 << ")" << std::endl;

    return 0;
}
```

Polecenie budowania

```
clang++ ./uaf.cpp -o ./uaf.elf
```

Polecenie uruchomienia

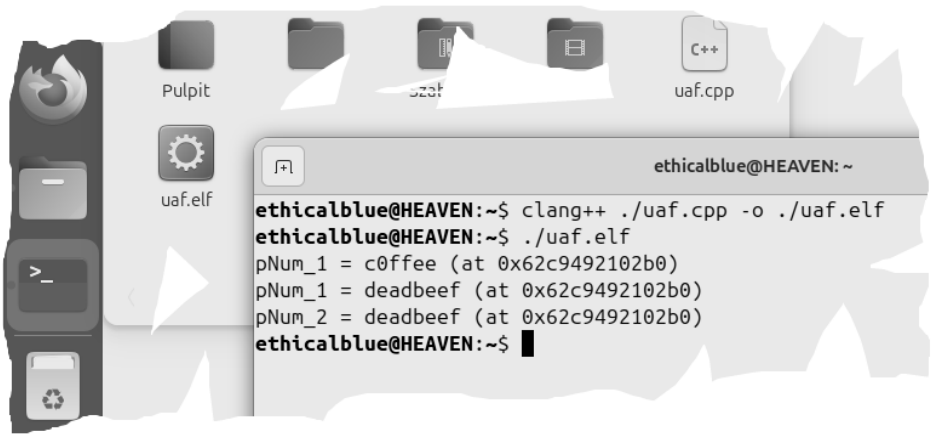
```
./uaf.elf
```



Warto przeczytać

+ https://cwe.mitre.org/data/definitions/416.html
[dostęp: 2024-11-13 00:03]

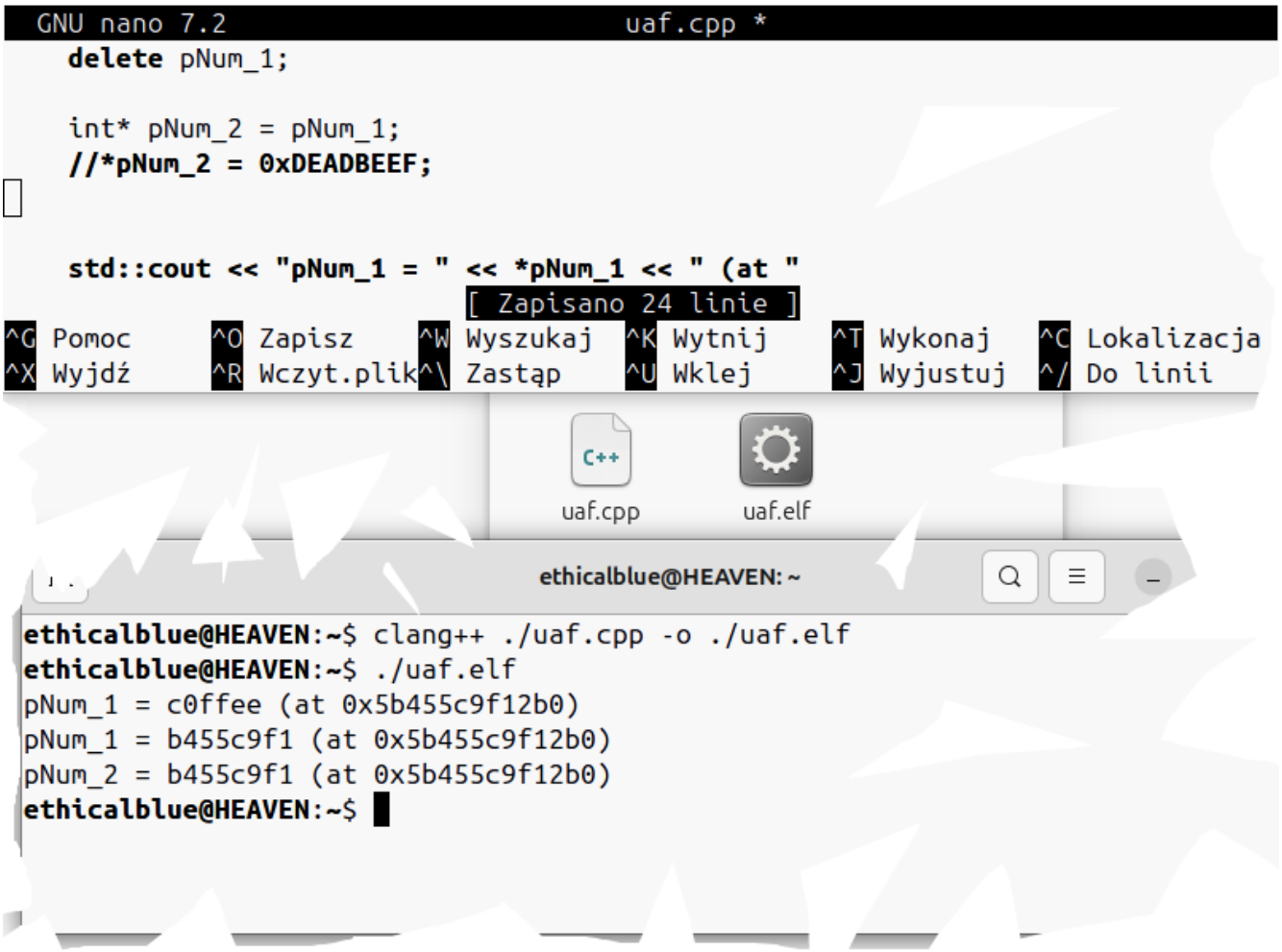
Na rysunku obok można zobaczyć nadpisanie wartości pod adresem `0x62c9492102b0` wartością `0xDEADBEEF`. Jak już wcześniej wspomniano kłopotem jest, że wartość została oznaczona jako zwolniona, więc może tam trafić inna wartość i zostać nadpisana.



W celu lepszego zobrazowania problemu można wykonać kolejny eksperyment i zakomentować linijkę:

```
//*pNum_2 = 0xDEADBEEF;
```

Teraz na rysunku poniżej można zobaczyć, że program odczytał wartość spod adresu w pamięci `0x5b455c9f12b0`. Jako, że pamięć została wcześniej zwolniona, to pod tym adresem może być coś innego.



Poza wykryciem awarii programu przeważnie potrzebna jest analiza kodu aplikacji, aby określić czy podatność jest łatwa w wykorzystaniu (ang. exploitable) przez podmioty zagrażające (ang. threat actor).

Wczytanie modułu z kodem bez weryfikacji integralności

Autor: Dawid Farbaniec

Dołączone dodatkowe moduły z kodem i danymi nazywane też zależnościami cechują większość bardziej rozbudowanych aplikacji.

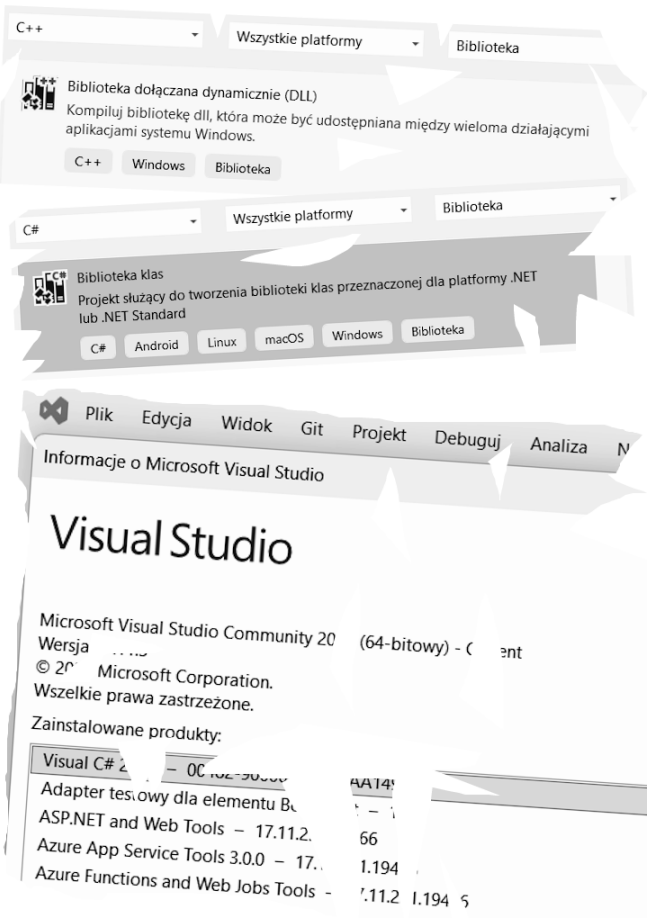
Rozdzielenie programu na komponenty pozwala m.in. na wielokrotne korzystanie z danego modułu przez różne aplikacje.

Biblioteki .DLL i .NET Assembly

Biblioteki klas dla platformy .NET (ang. .NET Assembly) różnią się od natywnych bibliotek DLL stworzonych np. w języku C++.

Natywna biblioteka .DLL zawiera kod, który może zostać bezpośrednio wykonany dlatego nazywany kodem niezarządzanym (ang. unmanaged, unsafe).

Natomiast biblioteka klas .NET zawiera kod w języku pośrednim (ang. Intermediate Language, IL) wykonywanym przez infrastrukturę CLI (ang. Common Language Infrastructure). Z tego powodu nazywany jest kodem zarządzanym (ang. managed).



Podpisanie kodu zaufanym certyfikatem

W celu zapewnienia weryfikacji integralności programów stosuje się certyfikaty dostarczane przez jednostki nazywane CA (Certificate Authority), które zajmują się weryfikacją podmiotów chcących podpisywać swoje programy.

https://www.digicert.com/signing/code-signing-certificates
[dostęp: 2024-11-13 00:03]

Rozwiązanie to jest kosztowne (828 USD rocznie na dzień 13.11.2024 r.) co może być kłopotem dla twórców mniejszych, niezależnych aplikacji.

Brak weryfikacji integralności

Bez sprawdzenia integralności zależności z których korzysta program staje się on podatny m.in. na

https://attack.mitre.org/techniques/T1574/002/
[dostęp: 2024-11-13 00:03]

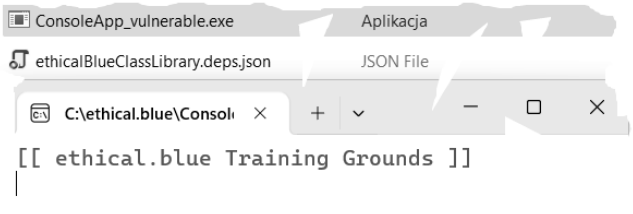
W celu zaprezentowania tego rodzaju podatności stworzono umyślnie niezabezpieczoną aplikację wyłącznie do celów demonstracyjnych.

Program z podatnością wczytuje bez weryfikacji integralności zewnętrzną bibliotekę **`ethicalBlueClassLibrary.dll`** w ten sposób:

```
var module =
System.Reflection.Assembly.LoadFrom(@".
\ethicalBlueClassLibrary.dll");
var type =
module.GetType("ethicalBlueClassLibrary.
ethicalBlueClass");
var method =
type?.GetMethod("ethicalBlueMethod");
_ = method?.Invoke(null, null);
```

W kodzie powyżej następuje dynamiczne wczytanie biblioteki klas o nazwie pliku **`ethicalBlueClassLibrary.dll`** z bieżącego katalogu **`.\`**, aby później wczytać klasę **`ethicalBlueClass`** z przestrzeni nazw **`ethicalBlueClassLibrary`**. Dalej następuje wczytanie metody o nazwie **`ethicalBlueMethod`** i wywołanie za pomocą **`.Invoke(null, null);`**.

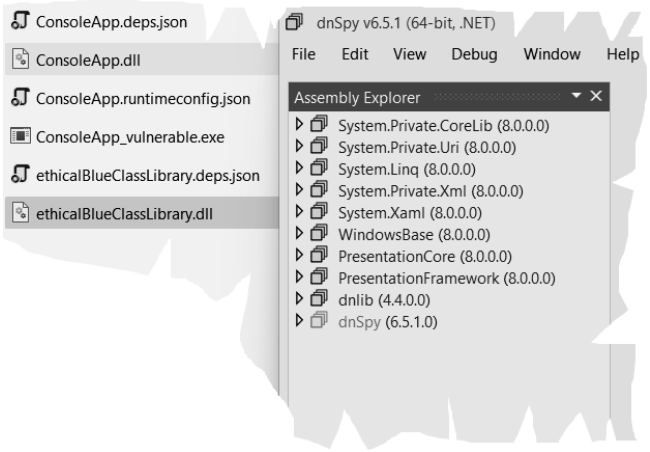
Przykładowa metoda wyświetla na konsoli tekst
`[[ ethical.blue Training Grounds ]]`



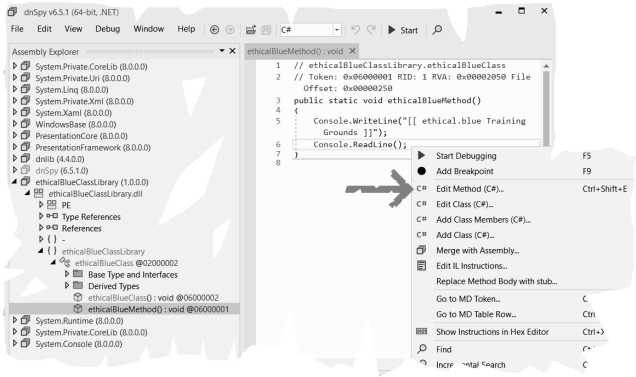
Nieautoryzowana modyfikacja kodu

Przykładowy program z podatnością podczas wczytywania biblioteki nie sprawdza integralności komponentu zewnętrznego.

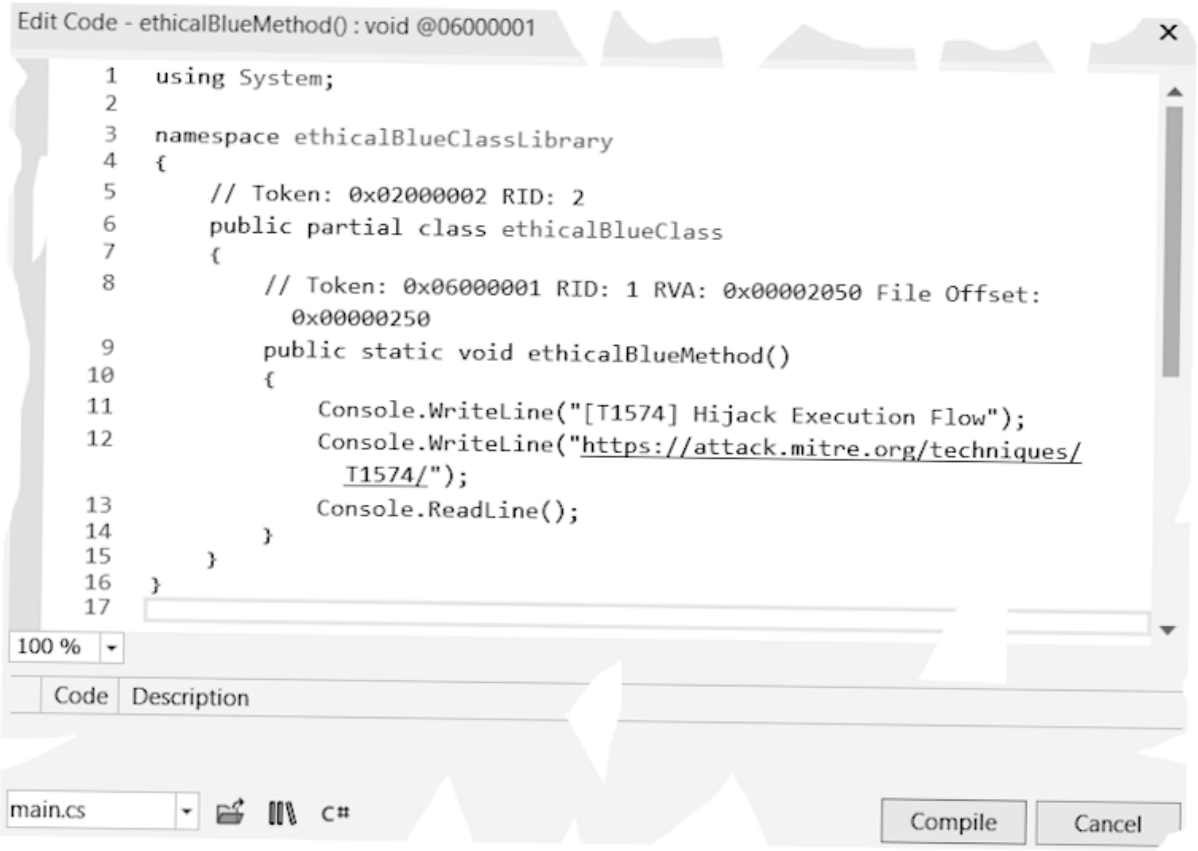
Biblioteka klas **`ethicalBlueClassLibrary.dll`** powstała w technologii .NET, więc jej kod można zmodyfikować np. narzędziem **dnSpy**.



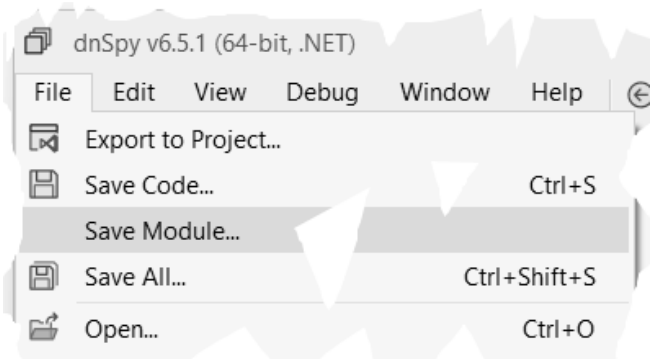
Wystarczy w kontrolce widoku drzewa odnaleźć metodę **`ethicalBlueMethod`**, aby dekompilator wyświetlił jej odzyskany kod źródłowy. Dalej można kliknąć prawym przyciskiem myszy i w menu podręcznym wybrać element Edit Method (C#).



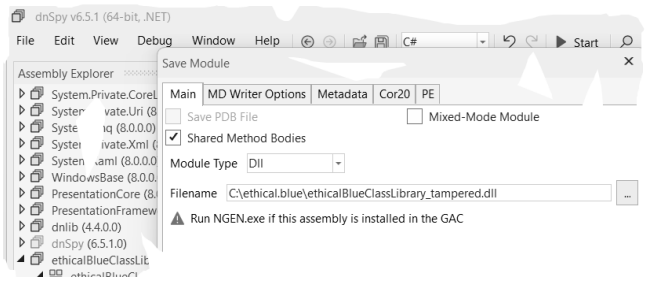
W oknie o nazwie Edit Code narzędzia dnSpy można wykonać modyfikację kodu biblioteki.



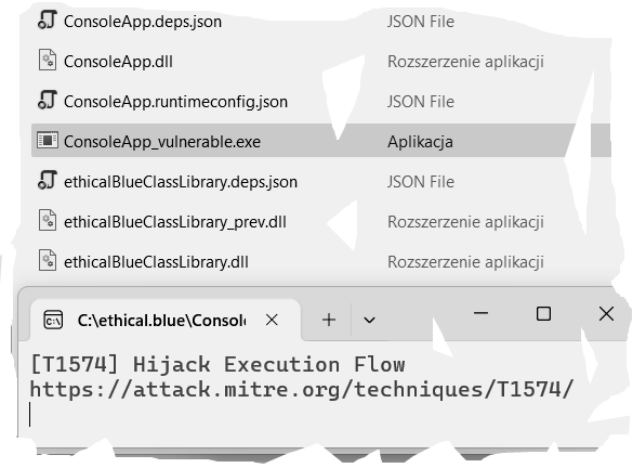
Zmodyfikowaną bibliotekę klas można zapisać wybierając z górnego menu File / Save Module.



W oknie zapisu wystarczy wpisać nazwę i ścieżkę.



Przykładowy program bez weryfikacji integralności zewnętrznej biblioteki klas wczytuje zmodyfikowany kod i wykonuje.



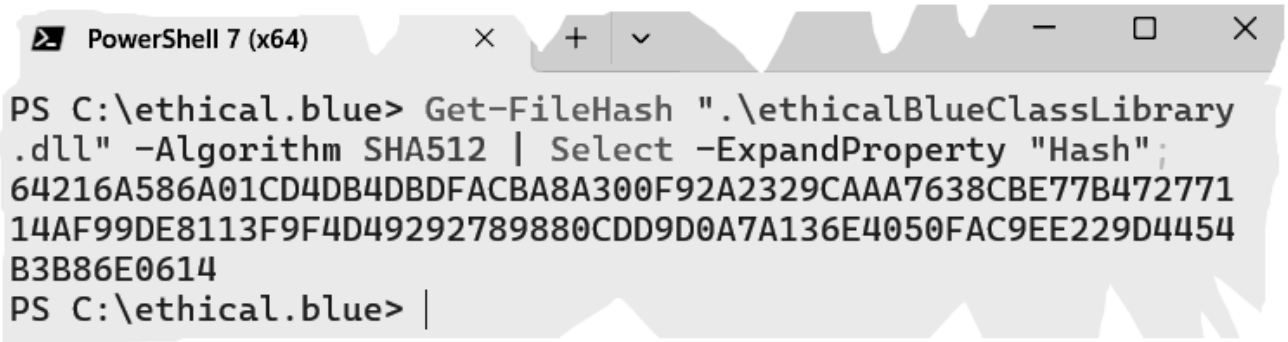
Dalej przedstawiono sposób jak złagodzić (ang. mitigate) tego rodzaju zagrożenie bez kupowania drogiego Code Signing certificate od jednostki CA (Certificate Authority).

W myśl zasady tworzenia bezpieczniejszego kodu, która brzmi **trust only known good,** czyli ufaj tylko temu co znasz i co jest dobre, poniżej przedstawiono przykład, który weryfikuje integralność zewnętrznej zależności przez obliczenie sumy kontrolnej SHA512.

```
using System.Security.Cryptography;

internal partial class Program
{
    /// <summary>
    /// --== ethical.blue Training Grounds ==--
    /// </summary>
    private static void Main(string[] args)
    {
        var plugin = @".\ethicalBlueClassLibrary.dll";
        var sha512 = "64216A586A01CD4DB4DBDFACBA8A300F" +
                     "92A2329CAAA7638CBE77B47277114AF9" +
                     "9DE8113F9F4D49292789880CDD9D0A7A" +
                     "136E4050FAC9EE229D4454B3B86E0614";
        var sha512_2 = Convert.ToHexString(
                       SHA512.HashData(File.ReadAllBytes(plugin)));

        if (string.Compare(sha512, sha512_2) != 0)
        {
            Console.WriteLine($"{plugin} has been tampered.");
            Console.ReadLine();
            return;
        }
    // ...
```

W przypadku zabezpieczonej integralności zewnętrznych zależności pozostaje chronić przed nieautoryzowanymi zmianami główny plik wykonywalny aplikacji.

Jeśli autor programu nie ma możliwości kupienia drogiego certyfikatu do podpisywania kodu od jednostki CA, to warto stworzyć oficjalną witrynę aplikacji wraz z opublikowaną sumą kontrolną np. SHA512, aby użytkownicy mogli sami sprawdzić czy nie dokonano nieautoryzowanej modyfikacji na oryginalnym pliku.

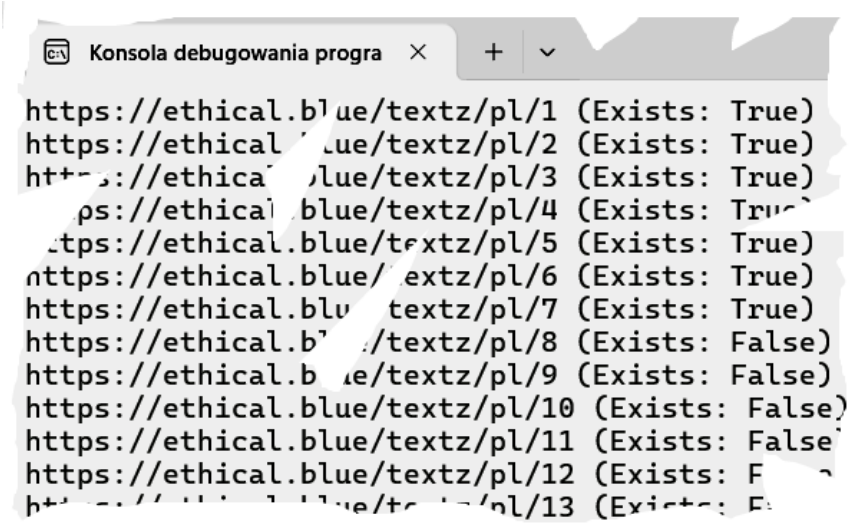


Bezpośrednie zapytania (ang. forced browsing)

Autor: Dawid Farbaniec

Bezpośrednie zapytania mogą zostać użyte w celu identyfikacji nieprawidłowo ukrytych plików, folderów i innych zasobów. Niegroźny przykład poniżej weryfikuje pod jakimi identyfikatorami są teksty w ethical.blue Magazine.

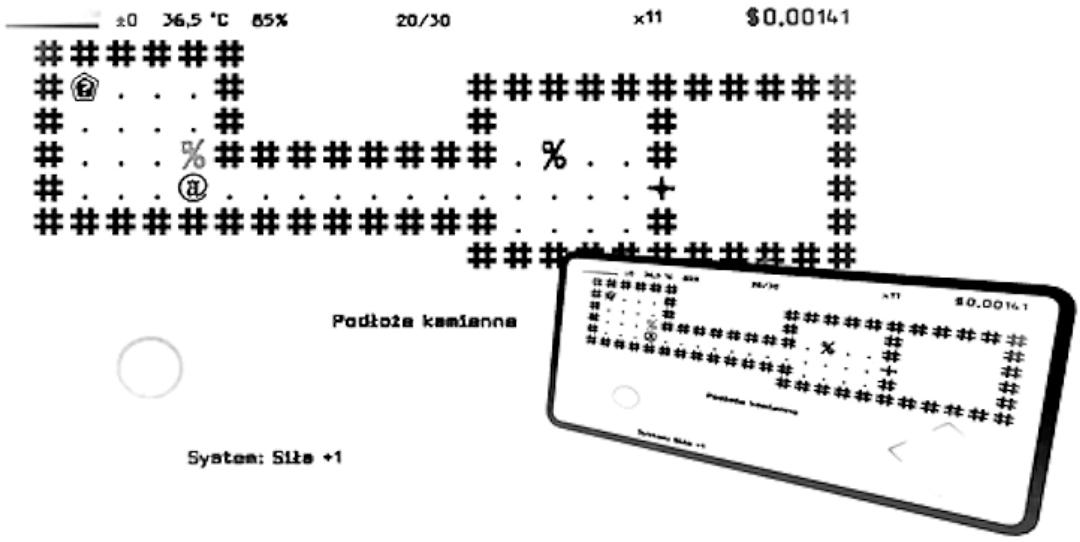
```
internal partial class Program
{
    /// <summary>
    /// --== ethical.blue Training Grounds ==--
    /// </summary>
    private static async Task Main(string[] args)
    {
        string target = "https://ethical.blue/textz/pl/";

        try
        {
            for (int i = 1; i < 14; i++)
            {
                using HttpResponseMessage response = await
                        httpClient.GetAsync($"{target}{i}");
                response.EnsureSuccessStatusCode();
                string text = await response.Content.ReadAsStringAsync();
                bool articleExists = text.Contains("<!-- Tytuł tekstu -->");
                Console.WriteLine($"{target}{i} (Exists: {articleExists})");
            }
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

    }
    private static readonly HttpClient httpClient = new();
}
```



Dodatek. 29 porad zwiększających świadomość zagrożeń

1. Nie przesyłaj danych przez sieć bez szyfrowania i uwierzytelnienia obu stron (najlepiej certyfikatem).
2. Unikaj szczegółowych komunikatów o błędzie, które mogą zdradzić szczegóły związane z bezpieczeństwem systemu.
3. Nie pozwalaj na dostęp do krytycznych parametrów aplikacji z zewnątrz.
4. Jeśli aplikacja przyjmuje pliki, to zwróć uwagę jak reaguje na pliki skrótu (*.LNK) i dowiązania symboliczne (*.SYMLINK).
5. Nie pozwalaj na bezpośrednią możliwość wprowadzenia nazwy pliku do odczytania np. w parametrze widocznym na zewnątrz.
6. Jeśli nie potrafisz odpowiednio zneutralizować niebezpiecznych sekwencji, to utwórz ścisłą listę jakie dane wejściowe są dozwolone, a unikniesz wstrzyknięcia złośliwego kodu.
7. Staraj się unikać tworzenia własnych filtrów na niebezpieczne dane. Stosuj sprawdzone rozwiązania. Pamiętaj, że takie same bajty mogą być zapisane na różne sposoby. W prostych słowach: liczba to bajty, tekst to bajty i grafika czy emotikona to też bajty.
8. Jeśli w raportach pojawiają się dane, które spowodowały awarię, to pamiętaj, aby odpowiednio zneutralizować elementy specjalne, a unikniesz wstrzyknięcia złośliwego kodu.
9. Wartości liczbowe np. o rozmiarze 16 bit mogą przyjąć wartości od 0x0000 do 0xFFFFF (heksadecymalnie). Pamiętaj, aby sprawdzić jak logika aplikacji zareaguje na minimalną oraz maksymalną wartość oraz jak je zinterpretuje (0xFFFF to dziesiętnie -1 ze znakiem, albo 65535 bez znaku).
10. Wartości liczbowe o określonym rozmiarze w bajtach mają dolną i górną granicę. Pamiętaj, aby sprawdzić jak logika aplikacji reaguje na wyjście poza te wartości. Np. czy dane się nasycą i przyjmą minimalną lub maksymalną wartość czy też może nastąpi przekręcenie jak w liczniku w starym samochodzie.
11. W przypadku obliczania rozmiaru ciągu znaków w bajtach zwróć uwagę jaki rozmiar ma jeden znak.
12. Wyrażenia regularne potrafią być bardzo kosztowne obliczeniowo. Ma to szczególne znaczenie w przypadku chmury. Nieprawidłowe wyrażenia mogą przyjąć niebezpieczne dane i przekazać je dalej, albo wywołać awarię typu odmowa usługi (ang. denial of service).
13. Nie polegaj na tym, że elementy w pamięci operacyjnej są ułożone w dany sposób, gdyż mogą zostać przemieszczone w najmniej spodziewanym momencie.
14. Indeksy elementów są przeważnie numerowane od zera, czyli `int a[10]; a[10] = 0xDEAD;` to błąd, gdyż ostatni element ma tutaj indeks dziewięć.
15. Dane wrażliwe nie powinny być przechowywane w katalogu głównym aplikacji internetowej, nawet pod ścieżką w stylu /0b596f4a3cebea1ba38-801e59586fefaf2-592c5c3f0b06dd680-0aead3f5fc466-4827a81d3e4-6f2655cc8526a50-a7e5a5f82f3434d3-f06affdc576804a5-68bbfe/secret.txt.
16. Niesprawdzanie wartości zwracanych z funkcji czy metod oraz niezłapane wyjątki mogą narazić aplikację na awarię.
17. Nigdy nie przechowuj haseł w postaci czystego tekstu lub w możliwym do odzyskania formacie. Stosuj sprawdzone algorytmy kryptograficzne, nie wymyślaj swoich.
18. Nie polegaj na adresie protokołu internetowego (adresie IP) w kwestii uwierzytelniania. Lokalizacja sieciowa może być przedstawiona na wiele sposobów, a maszyna obsługująca DNS (Domain Name System) może zwrócić inny adres IP dla tej samej nazwy hosta. Zależy przez kogo jest kontrolowana.
19. Przy dostępie do bardzo krytycznych funkcji powinno być ponowne uwierzytelnienie.
20. Uwierzytelnianie nie powinno być oparte na jednym składniku np. tylko hasło.
21. Wartości losowe powinny mieć odpowiedni zakres, aby nie były łatwe do przewidzenia czy zgadnięcia.
22. Nie wykonuj deserializacji z niezaufanego źródła bez weryfikacji czy rezultat operacji zwróci poprawne dane.
23. Nie umieszczaj kodu uwierzytelniającego client-side i nie polegaj na zaciemnieniu (ang. obfuscation) jako podstawowej metodze zapewniającej bezpieczeństwo.
24. Nie umieszczaj danych uwierzytelniających na stałe w kodzie (ang. hardcoded).
25. Zarezerwowane bity czy parametry aplikacji na przyszłość powinny być wyłączone i nie mieć wpływu na logikę systemu.
26. Zapewnij odpowiednie bezpieczeństwo fizyczne dla danych.
27. Rejestry przechowujące ustawienia związane z bezpieczeństwem powinny być odpowiednio zerowane przy resecie i nie pozostawać niezainicjalizowane.
28. Pamiętaj, że istnieje tak zwane race condition. Przykład. Plik jest usuwany z katalogu po odczytaniu. Jednak funkcja odczytująca zablokowała plik trochę dłużej i procedura czyszcząca go nie usunęła.
29. To nie wszystko. Te porady nie wystarczą. Dlatego staraj się wciąż rozwijać i dużo czytać.


/--\
| dotLNK_Console is Windows Shortcut (.lnk) Analysis Tool |
| by ethical.blue Magazine // Cybersecurity clarified. |
\--/
FileName: C:\DATA\msgbox.lnk
--== ShellLinkHeader ==--
HeaderSize: 0x0000004C
HeaderSize.IsValid: True
LinkCLSID: 00021401-0000-0000-c000-000000000046
LinkCLSID.IsValid: True
LinkFlags: 0x000802FB [HasLinkTargetIDList, HasLinkInfo, HasRelativePath, HasWorking Dir, HasArguments, HasIconLocation, IsUnicode, HasExpString, EnableTargetMetadata]
FileAttributesFlags: 0x00000020 [FILE_ATTRIBUTE_ARCHIVE]
Static Analysis
About
dotLNK is Windows Shortcut (.lnk) Analysis Tool
by ethical.blue Magazine // Cybersecurity clarified.
About Sample
FileName: C:\DATA\msgbox.lnk
FileSize: 3278 bytes (3 KB)
Entropy: 3,81
SHA-256:
SHA-512:
ShellLinkHeader
HeaderSize: 0x0000004C
HeaderSize.IsValid: True
Open
Exit
<!-- #MARKA_WŁASNA -->
dotLNK
Analiza plików skrótu (.LNK)
w systemie Windows
Dawid Farbaniec



https://ethical.blue/page/bytez2

<!-- #MARKA_WŁASNA -->

ROGUELIKE GAME

Dawid Farbaniec