Przegląd i porównanie technik instrumentacji procesów w środowisku Linux

Przemysław Samsel

Zachodniopomorski Uniwersytet Technologiczny w Szczecinie, Wydział Informatyki

Studenckie Koło Naukowe Grupa .NET

sp39427@zut.edu.pl

Abstract.

Instrumentacja dynamiczna wykorzystuje tzw. blackbox debuggery, pozwalając w szczegółowy sposób przeanalizować działanie danego programu. W tej pracy zostały przedstawione podstawowe mechanizmów debuggera, takie jak obsługa zdarzeń debugowania, w tym wyjątków m.in. programowych czy sprzętowych (*software/hardware breakpoints*) oraz tych mogących wystąpić w czasie wykonywania programu (*exceptions*). Celem przechwytywania różnych zdarzeń jest śledzenie kontekstu danego procesu – np. odczytanie wartości zmiennych bądź rejestrów procesora w danej chwili wykonania programu**.** Poznanie elementów, z których zbudowany jest debugger umożliwia zastosowanie wielu praktycznych technik np. przechwytywania wywołań z bibliotek współdzielonych bądź systemowych, wstrzykiwania kodu do procesu, czy ukrycia, na wiele sposobów, złośliwego kodu przed próbą analizy. Pozwala to także na lepsze zrozumienie działania popularnych debuggerów, umożliwiając wykorzystanie w pełni ich możliwości.

Dynamic instrumentation uses blackbox debugger to help analyst trace execution of a process. In this work basic debugging mechanisms were discussed, such as handling debug events i.e. software or hardware breakpoints, as well as handling exceptions that may suddenly stop execution of analysed software. The goal of handling debug events is tracing context of a given process – i.e. variables' values, reading registers etc. in given moment of execution. Based on the knowledge of how debuggers work at their lowest level, an analyst is able to make use of several techniques helpful in tracing program execution – be it interception system or library calls, code injection, or hiding/detection presence of a debugger. Armed with this knowledge, the reader may be able to make use of debugger's mechanisms better than ever before.

Keywords:

Blackbox debugger; dynamic instrumentation; debug events; debug registers; computer architecture

Słowa kluczowe:

Debugger; instrumentacja dynamiczna; wyjątki programowe; rejestry debuggera; wstrzykiwanie kodu

Spis Treści

1. Wstęp

Instrumentacja programu jest od początku nierozłączną czynnością programistycznej rzeczywistości [1], pozwalającą prześledzić krok po kroku logikę działania danego kodu i wyłuskać ewentualne błędy. Zarówno debugger pozwalający dynamicznie prześledzić kolejne rozgałęzienia wykonywanego procesu, a także znajomość architektury komputera i asemblera usprawniają pracę programisty, który przy użyciu tzw. whitebox debuggera – czyli narzędzia posiadającego dostęp do kodu źródłowego – może ustalić zasięg oraz implikacje wykonywanego kodu. Analityk złośliwego oprogramowania czy śledczy sądowy przy pomocy tzw. blackbox debuggera – umożliwiającego analizę działania oprogramowania bez dostępu do jego kodu źródłowego, posiadając dostęp jedynie do dezasemblowanego kodu – będzie w stanie zajrzeć „pod maskę" skompilowanego programu, mogąc nieraz uzyskać nawet większą wiedzę na jego temat niż jego twórcy. Głównym celem takiego debuggowania jest określenie obszaru wykonania kodu (ang. code coverage) – czyli pozyskanie informacji o tym jakie bloki programu, skoki były w rzeczywistości wykonane. Komplikować analizę może większa złożoność kodu bądź nietrywialność skoków warunkowych – we wszystkich tych wyzwaniach nieodłącznym narzędziem przy analizie są debuggery [2], [3].

Idąc o krok dalej, czyli rozbierając debugger na części pierwsze, jesteśmy w stanie jeszcze bardziej rozszerzyć jego możliwości i bardziej dostosować do konkretnych potrzeb, tym samym ułatwiając sobie później analizę innych programów przy jego użyciu. W niniejszej pracy poznamy podstawowe mechanizmy tzw. blackbox debuggera, przykładowo ustawianie w śledzonym procesie różnych rodzajów wyjątków, a także obsługa zdarzeń debuggowania. Zrozumienie działania tych mechanizmów na najniższym poziomie umożliwi jeszcze lepszy wgląd w budowę tego rodzaju narzędzia. Pozwoli to na umiejętne stosowanie technik debuggowania w konkretnych sytuacjach, czy to w przypadku instrumentacji statycznej – czyli biernej analizy kodu binarnego, czy instrumentacji dynamicznej (behawioralnej) – w której celem jest zobrazowanie interakcji analizowanego oprogramowania w relacji ze środowiskiem w którym jest wykonywane.

2. Architektura komputera

Nie można wykorzystać w pełni możliwości debuggera bez znajomości środowiska, pod którym pracuje. W tym rozdziale opisana została budowa rejestrów procesora w architekturze o modelu programowym x86-64, a także ogólny model pamięci wraz z jej ważniejszymi elementami np. stosem programowym. Poznanie podstawowych pojęć związanych z architekturą komputera, a także kilka prostych przykładów programów w assemblerze pozwoli czytelnikowi na lepsze zrozumienie zagadnień związanych z działaniem debuggera.

2.1. Rejestry

Rejestry są pojedynczymi komórkami pamięci wmontowanymi bezpośrednio w CPU. Są więc dla procesora najszybszym dostępnym rodzajem pamięci, ponieważ nie używają szyny do komunikacji. Mają one niewielką pojemność, ponieważ ich budowa jest bardziej złożona niż innych rodzajów pamięci i w związku z tym są dużo droższe. Z racji, że są pojedynczymi komórkami pamięci, nie posiadają adresów tak jak komórki pamięci o dostępie swobodnym (RAM). Rejestry oparte są na tranzystorach, podczas gdy główna pamięć zbudowana jest z kondensatorów. Dzięki temu rejestry są dużo szybsze od pamięci RAM czy pamięci podręcznej (cache) procesora, ale są także droższe w budowie [4].

W większości przypadków działanie programu ograniczone jest do małego zbioru danych mieszczącego się wewnątrz rejestrów. Wyniki przetwarzania danych przechowywane są w pamięci operacyjnej, zaś do rejestrów ładowane są kolejne. Duża ilość odczytów z pamięci jest niepożądana, ponieważ dane muszą być najpierw wczytane do rejestrów, a jeśli wszystkie rejestry są zajęte to ich aktualna zawartość musi zostać odłożona na stos – co zwalnia cały proces obliczeń [4]. W związku z tym (gdy szybkość działania jest priorytetem), ważnym parametrem jest tzw. lokalność odniesień (ang. locality of reference) [4] mówiąca o tym, jak bardzo zbliżone w czasie są kolejne odniesienia do tego samego adresu, bądź jaki przedział bajtów będzie dzielił kolejne odniesienia (im mniejsza odległość/offset tym „lepsza” lokalność odniesień).

Procesory o modelu programowym x86-64 posiadają 16 rejestrów nazwanych kolejno r0…r15. Pierwsze 8 rejestrów są to tzw. rejestry ogólnego przeznaczenia. Posiadają one także swoje zwyczajowe nazwy, analogiczne do rejestrów używanych we wcześniejszych procesorach Intela. Ich funkcje oraz nazewnictwo zostaną przedstawione w tabeli poniżej. Pozostałe rejestry r8…r15 są wykorzystywane głownie jako tymczasowe zmienne, bądź przechowują określone flagi CPU. Pozostałe rejestry to przede wszystkim wskaźnik instrukcji (RIP – Instruction Pointer), rejestr przechowujący flagi użyteczne podczas obliczeń (RFLAGS), rejestry wyspecjalizowane w konkretnych typach działań jak FPU czy MMX, czy wreszcie 16-bitowe rejestry segmentowe które używane były w czasach, gdy 16 bitowe procesory były w stanie zaadresować więcej pamięci niż było fizycznie dostępnej (dzisiaj mamy odwrotną sytuację) [5]. Część rejestrów jest w obecnej architekturze tylko ze względu na kompatybilność wsteczną, umożliwiając np. emulację działania wcześniejszych procesorów.

Rejestr	Nazwa zwyczajowa	Funkcja
r0	RAX (Accumulator)	Inaczej zwany akumulatorem. Przechowuje wyniki obliczeń czy wartości zwracane przez funkcje (zawartość tego rejestru może odpowiedzieć na pytanie czy funkcja zakończyła się sukcesem)
r2	RDX (Data)	Rejestr danych. Jest rozszerzeniem akumulatora przechowującym dodatkowe dane, umożliwiając wykonanie złożonych operacji matematycznych
r1	RCX (Counter)	Zwany licznikiem, używany np. przy pętlach. Należy pamiętać, że rejestr ten przy kolejnych iteracjach zawsze odlicza w dół [4]
r6	RSI (Source Index)	Ułatwia operację na dużych ilościach danych w pamięci, przechowując adres danych wejściowych. Używany m.in. do operacji na ciągach znaków
r7	RDI (Destination Index)	Pełni podobną funkcję do rejestru RSI, jednak przechowuje adres danych wyjściowych
r4	RSP (Stack Pointer)	Wskazuje na szczyt stosu w pamięci
r5	RBP (Base Pointer)	Przechowuje adres danych na stosie wywołań
r3	RBX	Wskaźnik na dane w segmencie DS (Data Segment)

Tabela 1. Opis rejestrów ogólnego przeznaczenia

Pierwsza litera w nazwie wszystkich rejestrów jest taka sama i jest znakiem rozpoznawczym dla architektury x86-64 – oznacza to, że każdy rejestr zaczynający się literą „R” posiada pojemność 8 bajtów (64 bity). W przypadku architektury 32-bitowej Intela każdy rejestr ogólnego przeznaczenia zaczynał się literą „E” (np. EAX) i posiadał pojemność 4 bajtów (32 bity), zaś w pierwszych procesorach Intela rejestry 16-bitowe nie posiadały żadnego przedrostka (np. AX). Obecnie jest także możliwość dostania się do mniejszych partii (prawie) każdego rejestru poprzez odpowiednią adresację. Podyktowane jest to m.in. kompatybilnością wsteczną. 8-bajtowe części rejestru są adresowane poprzez zastąpienie litery „X” literami „H” (High) lub „L” (Low) w celu dostania się do odpowiedniej połowy rejestru 16 bitowego. Podsumowanie adresowania rejestrów ogólnego przeznaczenia zostało przedstawione w tabeli poniżej. Cyfry nad tabelą oznaczają numer bitu – należy zwrócić uwagę, że jesteśmy jedynie w stanie zaadresować dolne części rejestrów (znajdujące się po prawej stronie) w przypadku architektury zarówno x86-64 oraz x86.

64	32 31	16 15	8 7	0	**16-bit**	**32-bit**	**64-bit**
		AH	AL		AX	EAX	RAX
		BH	BL		BX	EBX	RBX
		CH	CL		CX	ECX	RCX
		DH	DL		DX	EDX	RDX
		BP				EBP	RBP
		SI				ESI	RSI
		DI				EDI	RDI
		SP				ESP	RSP

Tabela 2. Zakres poszczególnych części rejestrów ogólnego przeznaczenia

Pozostałe rejestry (r8…r15) są używane głównie do przechowywania zmiennych tymczasowych, bądź w konkretnych sytuacjach, np. przy wywołaniach systemowych [6]. One również umożliwiają adresowanie ich mniejszych części, jednak używając nieco innej konwencji. Należy dodać odpowiedni sufiks do nazwy rejestru, wybierając spośród poniższych:

- d (double word) – niższe 32 bity (np. r0d – najniższe 4 bajty rejestru r0)

- w (word) – niższe 16 bitów (np. r0w najniższe 2 bajty rejestru r0)
- b (byte) – niższe 8 bitów

Co ciekawe, wszystkie odczyty z niższych partii rejestrów zachowują się „naturalny” sposób, natomiast wpisy do 32-bitowych części domyślnie wypełniają wyższe 32-bity rejestru z bitami znaku. Na przykład – wyzerowanie EAX wypełni zerami cały RAX zerami, natomiast wpisanie wartości -1 do EAX wypełni górne 32-bity jedynkami [4].

2.1.1. Rejestry debuggera

Współcześnie mikroprocesory wyposażone są w tzw. rejestry debuggera, służące do przechowywania informacji o adresach wyjątków sprzętowych, typie tych wyjątków oraz ich zakresie. Począwszy od architektury x86, istnieje 8 rejestrów debuggera o numerach kolejno DR0…DR7 – ang. Debug Register [7]. W procesorach o modelu programowym x86-64 udostępnione są te same rejestry debuggera, jednak o odpowiednio większej (8 bajtów zamiast 4) pojemności. Każdy z rejestrów ma swoje określone zadanie.

Rejestry DR0…DR3 zarezerwowane są na adresy wyjątków. Oznacza to, że jednocześnie można ustawić maksymalnie 4 wyjątki sprzętowe. Rejestry DR4 i DR5 są wyłączone z użytku, gdy tzw. debug extensions są włączone (świadczy o tym ustawiona flaga DE w rejestrze kontrolnym CR4) zaś gdy bit flagi DE nie jest ustawiony, rejestry te są aliasem do zawartości rejestrów DR6 i DR7 [7]. Rejestr DR6 odpowiedzialny jest za przechowywanie typu wyjątków. DR7 pełni dwie funkcje – przede wszystkim ustala czy dany wyjątek o adresie w rejestrach DR0-DR4 jest włączony (on/off switch), ponadto przechowuje on dodatkowe flagi kontrolując sposób interpretowania zawartości rejestrów DR0-DR3 [2]. W zależności od stanu tych flag, rozróżniamy następujące wyjątki sprzętowe:

- Wyjątek, gdy wykonywany jest kod z określonej komórki pamięci
- Wyjątek, gdy nastąpił odczyt z określonej komórki pamięci
- Wyjątek, gdy nastąpił zapis bądź odczyt z określonej komórki pamięci
- Wyjątek, gdy nastąpił zapis bądź odczyt z określonej komórki pamięci poprzez szynę adresową IO [2]

Bardziej szczegółowo, bity 0-7 odpowiadają za włączenie poszczególnych 4 wyjątków. Po 2 bity na każdy wyjątek, ponieważ możliwe jest postawienie wyjątku o lokalnym (L) oraz globalnym (G) zasięgu. Bity 8-15 zapewniają dodatkowe funkcjonalności. W architekturze x86-64 bity 8 oraz 9 nie są w ogóle używane. Bit 13 służy do zabezpieczenia rejestrów debuggera przed jakimkolwiek dostępem (jest automatycznie zwalniany w momencie uruchomienia procedury obsługi wyjątku [7]. Druga połowa dolnej części rejestru (bity 16-31) odpowiada za ustawienie typu oraz długości wyjątku. Należy zwrócić uwagę, że w architekturze x86-64 rejestry kontrolne (DR6 oraz DR7) udostępniają jedynie niższe 32-bity. Górna połowa rejestru jest zawsze wypełniana zerami i jest wyłączona z użytku.

(…)	D	T	D	T	D	T	D	T		G	L	G	L	G	L	G	L
	DR3	DR3	DR2	DR2	DR1	DR1	DR0	DR0		DR3	DR3	DR2	DR2	DR1	DR1	DR0	DR0
	31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	8-15	7	6	5	4	3	2	1	0

Tabela 3. Układ rejestru DR7

W Tabeli 3 kolumny oznaczone literą D określają Długość wyjątku, natomiast literą T – typ. Tabela 4 przedstawia określone wartości jakie mogą się pojawić w tych polach, wraz z ich znaczeniem [7].

Typ	Długość
00 – Wyjątek na wykonanie	00 – 1 bajt
01 – Wyjątek na zapis	01 – 2 bajty (WORD)
11 – Wyjątek na odczyt lub zapis, ale nie wykonanie	11 – 4 bajty (DWORD)
	10 – 8 bajtów

Tabela 4. Długości oraz typ wyjątków w rejestrze DR7

2.2. Pamięć

Pamięć o dostępie swobodnym (RAM) jest fizyczną pamięcią zorganizowaną w komórki o pojemności 8 bitów, z których każda ma przypisany unikalny adres – zwany fizycznym adresem. Zakres fizycznych adresów wynosi $0 – 2^{64}- 1$ i jest to około 16 egzabajtów pamięci. Jest to spora różnica i znacząco przekracza potrzeby użytkowników w nawet najbardziej ekstremalnych przypadkach, w porównaniu do architektury x86, gdzie maksymalna ilość adresowalnej pamięci RAM wynosiła 64 gigabajty. Intel stworzył mechanizmy obsługi pamięci RAM dla systemów operacyjnych, które pozwalają im widzieć pamięć operacyjną jako ciągły, liniowy zakres adresowy [5].

Wcześniejsza architektura Intela wprowadziła także tzw. model segmentowy – w którym pamięć podzielona jest na grupy niezależnych segmentów (jak segment kodu, danych czy stos) które adresowane są niezależnie za pomocą adresu logicznego. Architektura o modelu x86-64 wprowadza tryb 64-bitowy, w którym segmentacja jest częściowo wyłączona, natomiast liniowe adresowanie stało się równoważne efektywnemu adresowi. Podsumowując, wszystkie te modele prowadzą do tego, aby udostępnić systemowi operacyjnemu odpowiedni interfejs do zarządzania pamięcią, dzięki czemu nie musi on posługiwać się fizycznymi adresami, a jedynie zakresem udostępnionym przez architekturę, pod którą pracuje [5] [4].

2.2.1. Stos i konwencje wywołań

Stos sprzętowy, dla odznaczenia różnicy między abstrakcyjną strukturą danych o tej samej nazwie, jest zaimplementowany w pamięci umożliwiając przechowywanie argumentów do funkcji, adresów powrotu czy zmiennych lokalnych. Jest to struktura FILO (First In, Last Out) posiadająca dwie operacje – odkładania (push) oraz pobierania (pop) elementu ze szczytu stosu. Ważnym detalem jego funkcjonalności jest to, że stos w pamięci jest odwróconym stosem tzn. kolejne elementy zajmują komórki o coraz niższych adresach [4]. Gdyby zobrazować pamięć jako płaską listę komórek od adresów niższych (0) do najwyższych (0xFFFFFFFFFFFFFFFF) to stos wyglądałby w ten sposób:

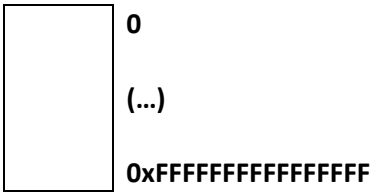


Tabela 5. Uproszczony model pamięci

Ramką stosu nazywany jest fragment stosu odpowiadający za wywołanie funkcji – czyli przechowujący adres powrotu z funkcji oraz wszystkie argumenty i zmienne lokalne należące do danej funkcji. Przykładowo, deklaracja pewnej funkcji w C wyglądałaby następująco:

```
int myfun(int a, int b, int c)
{
        int d;
        // (...)
        // (...)
        return a;
}
```

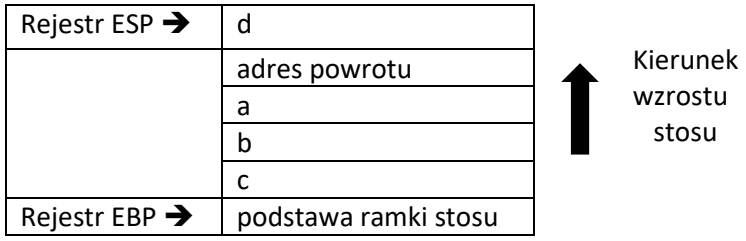
Wycinek kodu 1. Przykład funkcji w C.

Wtedy przygotowanie ramki stosu przed wywołaniem tej funkcji, według konwencji *cdecl* (stosowanej powszechnie m.in. w kompilatorze gcc), wyglądałoby następująco:

push	c
push	b
push	a
call	myfun

Tabela 6. Przykład przygotowania ramki stosu

Ramka stosu dla tej funkcji po jej wywołaniu (i zaalokowaniu pamięci dla zmiennej d) wyglądałaby w ten sposób (w architekturze x86):



Jedną z głównych różnic w x86-64 jest to, że pierwsze 6 parametrów przekazywane jest poprzez rejestry (kolejno RDI, RSI, RDX, RCX, R8, R9) a każdy kolejny parametr przekazywany jest na stosie, analogicznie do *cdecl* w x86. Różnice te są szczegółowo omówione w [8] oraz [9]. Wracając do ramki stosu, przechwycenie jej w celu analizy w debuggerze pozwala zdobyć cenne informacje przy awaryjnym zakończeniu aplikacji w celu np. wykrycia ataków polegających na przepełnieniu stosu (ang. stack-based overflows) [10].

Dodatkowo, kolejność podawania argumentów do funkcji w konwencji cdecl jest od prawej-do-lewej (LTR). Warto wspomnieć, że kolejność ewaluacji argumentów do funkcji w językach C/C++ nie jest zdefiniowana na poziomie specyfikacji języka programowania, lecz na poziomie konkretnej implementacji (np. dla kompilatorów clang/llvm jest odwrotna kolejność).

Należy pamiętać, że z poziomu mikroprocesora pojęcie funkcji nie istnieje. Funkcje są warstwą abstrakcji stworzoną w celu utrzymania pewnej struktury w kodzie, jednak ta sama pamięć, którą programista interpretuje jako strukturę rozgałęzień, dla mikroprocesora ma zupełnie płaską formę. Weźmy dla przykładu poniższy kod z wycinka nr. 2.

```
section     .data
msg   db 'krotka wiadomosc', 10 ;'msg ma 16 znakow dlugosci'
section   .text
global  _start

myfun2:
  mov   RAX, 1       ; numer wywolania systemowego 'write'
  mov RDI, 1       ; deskryptor wyjscia standardowego 'stdout'
  mov   RSI, msg     ; adres ciagu znaków jako parametr do wywolania
  mov RDX, 17      ; dlugosc ciagu znaków + 1 (miejsce na /0)
  syscall
  ret

_start:
  call  myfunc       ; wywolanie funkcji
  mov RAX, 60      ; numer wywolania systemowego 'exit'
  xor RDI, RDI     ; kod wyjscia
  syscall
```

Wycinek kodu 2. Przykład kodu w asemblerze

Do implementacji funkcji służy instrukcja *call*. Odkłada ona obecny adres znajdujący się w rejestrze RIP (przed tym jeszcze przekazując ewentualne parametry), następnie przekierowuje pod adres wskazywany przez etykietę funkcji. Dla mikroprocesora jest to po prostu skok w inne miejsce pamięci – to programiści nadali wartość głębokości dla tego skoku. Na koniec, przyjrzyjmy się jeszcze pojedynczej instrukcji na wycinku nr 3.

```
mov     RAX, 1
```

Wycinek kodu 3. Pojedyncza instrukcja w postaci mnemonicznej

Jest to tzw. postać mnemoniczna. Aby mikroprocesor mógł ją zinterpretować, musi ona zostać przetłumaczona na postać bajtkodu. Jest to możliwe dzięki przypisaniu opkodu (ang. opcode – operation code) do każdej możliwej instrukcji, a także zakodowaniu numerów rejestrów jako część opkodu (zatem długość instrukcji może ulegać zmianie w zależności od rejestru). Postać powyższej instrukcji wyglądałaby następująco:

Adres	Postać bajtkodu instrukcji	Postać mnemoniczna
0:	48 c7 c0 01 00 00 00	mov rax, 0x1

Tabela 7. Postać bajtkodu oraz mnemoniczna instrukcji

Podsumowując, w tym rozdziale przedstawione zostały podstawy architektury komputera, w tym działania poszczególnych rejestrów oraz różnych mechanizmów pamięci. Omówione zostały także konwencje wywołań i różne postaci kodu, widziane z perspektywy człowieka oraz maszyny na różnych poziomach abstrakcji. Informacje te okażą się z pewnością niezbędne do zrozumienia mechanizmów o których działanie oparte są debuggery.

3. Mechanizmy debuggera

Debugger jest procesem działającym w nieskończonej pętli, oczekującym na zdarzenia debuggowania (ang. debug events) [11]. Gdy takie zdarzenie nastąpi, debugger wywołuje funkcję odpowiedzialną za obsługę zdarzenia (ang. event handler). W systemach linuksowych funkcjonalności np. obsługi zdarzeń, czy przechwytywania wykonania procesu zaimplementowana są przy użyciu systemowego wywołania *ptrace* [12], [13]. Jest to bardzo wszechstronne narzędzie, pozwalające na przechwytywanie różnego typu zdarzeń jak wywołania systemowe, sygnały, czy np. uruchomienie nowego programu z przestrzeni danego procesu (*execve*). Pozwala także na odczytywanie oraz modyfikację dowolnych obszarów pamięci w pamięci śledzonego procesu oraz rejestrów CPU [14], [15].

```
def debugger(pid):
    # Oczekuj aż proces wykona pierwszą instrukcję
    status = os.wait()

    # Nastąpiło zatrzymanie procesu
    while (os.WIFSTOPPED(status[1])):
        obsluga_zdarzenia()
        status = os.wait()
```

Wycinek kodu 4. Przykład głównej pętli debuggera

Przykładem zdarzenia istotnego z punktu widzenia debuggera, mogącego wystąpić w trakcie wykonywania debugowanego programu, jest napotkanie wyjątku. Może to być wyjątek celowo umieszczony w kodzie przez programistę w celu inspekcji konkretnego elementu analizowanego oprogramowania bądź wykonanie niedozwolonej operacji, w której wyniku np. program próbuje uzyskać dostęp do pamięci, do której nie ma uprawnień [16].

Debugger umożliwia ustawienie punktu przerwania na różne sposoby np. poprzez wstawienie specjalnej instrukcji w kod programu wykonywany w pamięci lub za pomocą specjalnego zestawu rejestrów. Każdy sposób ma swoje konsekwencje, od których uzależnione jest, jakiego warto użyć w danej sytuacji. Porównanie tych sposobów zostanie porównanie w kolejnych podrozdziałach.

W momencie wywołania funkcji obsługującej wystąpienie zdarzenia, głównym zadaniem analityka jest zebrać jak najwięcej informacji na temat aktualnego stanu wykonywanego programu – na przykład jakie pliki modyfikuje program, gdy przechwycone mają zostać wszystkie operacje wejścia/wyjścia. Umiejętne zastosowanie skryptów w debuggerze pozwoli znacznie przyspieszyć cały proces analizy poprzez automatyzację zbierania interesujących informacji z pamięci.

3.1. Tryby pracy debuggera

Pierwszym „trybem” pracy w debuggera w relacji z analizowanym procesem jest jego uruchomienie przez debugger (stworzenie potomka przy użyciu funkcji *fork()* [17], następnie wewnątrz potomka wykonanie kodu programu, którego nazwa została wskazana np. przez argument) . W tym przypadku debugger posiada całkowitą kontrolę procesu od momentu wykonania pierwszej instrukcji. Może to być przydatne np. w analizie próbki potencjalnie złośliwego oprogramowania. Warto w takiej sytuacji dodatkowo odizolować się od głównego środowiska wykonania np. przy użyciu piaskownicy (ang.

sandbox) [18] czy wirtualizacji, w celu minimalizacji potencjalnych szkód, jakie analizowany program mógłby wyrządzić w systemie. Jednakże należy dodać, że w systemach linuksowych możliwości śledzenia procesu są bardzo ograniczone. Aby debugger mógł przechwytywać sygnały analizowanego potomka, musi uzyskać jego zgodę.

```
if __name__ == "__main__":
    pid = fork()

    # Debugger jako rodzic
    if pid:
        debugger(pid)
    else:
        # Debuggowany potomek
        debugee(argv[1])
#
```

Wycinek kodu 5. Wystartowanie procesu z poziomu debuggera. Przykład start_deb.py z [19]

W Wycinku nr. 5 proces zostaje rozdzielony, rodzic przechodzi do głównej pętli debuggera oczekując na zdarzenia, natomiast do funkcji potomka zostaje przekazana nazwa programu do wykonania jako parametr do skryptu. Ciało funkcji potomka przedstawia Wycinek nr. 6.

```
# Proces debuggowany (potomek)
def debugee(progname):
    # Zezwalaj na debugowanie tego procesu
    ptrace(PTRACE_TRACEME, 0, 0, 0)

    # Uruchom program o nazwie progname
    execv(progname, (progname, str(0)))
#
```

Wycinek kodu 6. Ciało funkcji wykonującej kod potomka. Przykład start_deb.py z [19]

Uruchomienie takiego skryptu z podaniem nazwy istniejącego programu jako parametr spowoduje wykonanie w nim pierwszej instrukcji, następnie przekazanie kontroli debuggerowi. Gdyby jednak usunąć linijkę z *PTRACE_TRACEME* (zezwalającą na debuggowanie), kontrola nigdy nie zostanie przekazana, proces potomny wykona wszystkie swoje instrukcje i zakończy się (niezależnie czy wykonamy ten skrypt z uprawnieniami *roota* czy bez).

W pewnych sytuacjach interesujący jest jedynie fragment wykonania analizowanego programu, wówczas przy użyciu debuggera można przechwycić (ang. attach) [11]wykonywanie innego procesu od pewnego momentu. Należy zwrócić uwagę, że jakiekolwiek śledzenie wykonywania procesu zwalnia jego działanie nawet kilkudziesięciokrotnie. Zatem w sytuacjach, gdy celem analizy jest jedynie fragment czy określona funkcja w złożonym oprogramowaniu bądź gdy nie ma możliwości uruchomienia analizowanego programu, przydatna jest możliwość przechwytywania działającego już programu zamiast uruchamiać go od nowa. W przykładzie Wycinek kodu nr. 7, PID procesu przekazywane jest do funkcji, która następnie (korzystając z wywołania PTRACE_ATTACH) przechwytuje wykonanie wskazanego procesu. Jeśli wszystko pójdzie zgodnie z planem, debuggowany proces otrzyma sygnał SIGSTOP, po którego przechwyceniu możemy przejść do obsługi tego i kolejnych zdarzeń, do momentu zakończenia procesu.

```
def debugger(pid):
    # Przechwytywanie procesu o okreslonym PID
    ptrace(PTRACE_ATTACH, pid, None, None)

    # Funkcja zwraca tuple zawierajaca ID procesu oraz 2-bajtowy status
    status = waitpid(pid, 0)

    # Sprawdz czy udalo sie przechwycic wykonywanie procesu
    # (proces otrzymal sygnal SIGSTOP)
    if WIFSTOPPED(status[1]):
        if WSTOPSIG(status[1]) == SIGSTOP:
            # Dopoki sledzony proces nie zakonczyl pracy
            while (WIFSTOPPED(status[1])):
                obsluga_zdarzenia()
                status = waitpid(pid, 0)
        else:
            print("Otrzymano inny sygnal o nr.: ",    WSTOPSIG(status[1]))
            exit(2)
#
```

Wycinek kodu 7. Przechwytywanie wykonania procesu o podanym PID. Przykład attach_deb.py z [19]

Tutaj ponownie wchodzi linuksowa troska o bezpieczeństwo. Bez uprawnień *roota,* nie jesteśmy w stanie przechwycić żadnego procesu. W poniższym przykładzie, najpierw uruchamiamy program *hello_loop* (po uprzednim skompilowaniu pliku źródłowego *hello_loop.c z* [19]) otrzymując następujące wyjście:

```
$./hello_loop
Petla nr: 0
Petla nr: 1
Petla nr: 2
Petla nr: 3
Petla nr: 4
Petla nr: 5
```

Wycinek kodu 8. Uruchomienie programu hello_loop.c z [19]

Następnie w innej konsoli uruchamiamy debugger, przekazując jako parametr PID śledzonego procesu (odwrotny cudzysłów tzw. backticks służą do wykonania polecenia powłoki):

$ python3 attach_deb.py *`pidof hello_loop`*

Otrzymamy wtedy następujący błąd:

```
Traceback (most recent call last):
  File "attach_deb.py", line 33, in <module>
    debugger(int(argv[1]))
  File "attach_deb.py", line 13, in debugger
    status = waitpid(pid, 0)
ChildProcessError: [Errno 10] No child processes
```

Oznacza to tyle, że nie udało się przejąć kontroli nad wskazanym procesem. Gdyby uruchomić to samo polecenie z uprawnieniami *roota,* takiego problemu już nie będzie. Odpowiada za to moduł

bezpieczeństwa o nazwie YAMA [20]. Aby wyłączyć ten mechanizm by móc śledzić inne procesy o tych samych uprawnieniach, należy zmienić pewną wartość w pliku konfiguracyjnym:

$ echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope

Tym razem ponowne uruchomienie debuggera w celu przechwycenia procesu nie zwróci już żadnego błędu. Ważnym wnioskiem jest, że linux jest dosyć restrykcyjny, jeśli chodzi o śledzenie procesów przez procesy nieuprzywilejowane. Wynika z tego, że niemożliwym jest raczej, aby wykorzystać omawiane tutaj techniki w celu zdobycia wyższych uprawnień (ang. privilege escalation) na atakowanym serwerze. Jednakże, wykorzystując te techniki połączone z odkrytą podatnością w konkretnym oprogramowaniu, możliwe byłoby osiągnięcie dostępu do danej maszyny (ang. initial foothold)

3.2. Szybkość działania

Jak już wspomniano, mechanizm *ptrace* mocno spowalnia działanie analizowanego oprogramowania. Uzależnione to jest rzecz jasna od tego jak często zatrzymujemy analizowany proces, wykonując obsługę konkretnych zdarzeń. Najbardziej obciążającym sposobem analizy jest tzw. wykonanie krokowe (ang. single-stepping). Polega ono na wykonywaniu kodu instrukcja po instrukcji, zwracając kontrolę debuggerowi po każdej z nich.

```
int main()
{
    int somenumber = 0;
    for (int i=0; i < 1000000;i++)
        somenumber += 2;

    return 0;
}
```

Wycinek kodu 1. Przykład prostej pętli w C. Program speed_ex1.c z [19]

Dla przykładowego programu przedstawionego na Wycinku nr. 9 pokazane zostanie jego uruchomienie z użyciem prostego skryptu zliczającego wykonanie instrukcji (Wycinek nr. 10).

```
def debugger(pid):
    # Liczba wykonanych instrukcji
    icounter = 0
    # Oczekuj az potomek wykona pierwsza instrukcje
    status = wait()

    # Oczekuj na sygnaly od potomka do
    # momentu jego zatrzymania
    start = datetime.now()
    while (WIFSTOPPED(status[1])):
        icounter+=1
        # Nakaz potomkowi wykonac kolejna instrukcje
        ptrace(PTRACE_SINGLESTEP, pid, 0, 0)
        status = wait()

    stop = datetime.now()
    print(f"Potomek wykonal {icounter} instrukcji, w czasie {(stop
- start).seconds} sekund")
```

Wycinek kodu 9. Prosty skrypt zliczający wykonane instrukcji śledzonego procesu. Program measure_deb.py z [19]

Po uruchomieniu skryptu, podając jako argument nazwę śledzonego programu, zwraca on następujące wyjście:

```
$ python3 measure_deb.py speed_ex1
Potomek wykonal 4116416 instrukcji, w czasie 97 sekund
```

Wycinek kodu 10. Wyjście ze skryptu measure_deb.py

Skąd wzięło się przeszło 4 miliony instrukcji, skoro wyraźnie pętla w powyższym programie zawiera jedynie 2 miliony dodawań? Odpowiedzią są dołączone biblioteki, które są automatycznie linkowane w czasie uruchomienia programu. Program dynamicznie linkowany zawiera jedynie relatywne adresy konkretnych wywołań systemowych oraz bibliotecznych w swojej tablicy wywołań, które tłumaczone są na adresy bezwzględne w pamięci w momencie jego wykonywania. Zmniejsza to ogólny rozmiar programu, jednak wymaga posiadania bibliotek programistycznych aby umożliwić jego uruchomienie, a także wymaga wspomnianego procesu linkowania. Warto także zwrócić uwagę, że program śledzony wykonał się w czasie około 100 sekund, co jest ponad 100-krotnym przebiciem w stosunku do wykonywania tego samego procesu bez śledzenia z użyciem *ptrace*. Pomiar dokładnego czasu wykonania nieśledzonego programu zostaje dla czytelnika.

W przypadku skompilowania programu w sposób statyczny (z użyciem flagi *-static* w gcc) uzyskano następujący rezultat:

```
$ python3 measure_deb.py speed_ex1_static
Potomek wykonal 4040885 instrukcji, w czasie 57 sekund
```

Wycinek kodu 11. Wyjście ze skryptu measure_deb.py dla statycznego pliku wykonywalnego

Czas wykonania znacząco uległ poprawie, niewielkiemu zmniejszeniu uległa także całkowita liczba instrukcji. Sam plik wykonywalny natomiast znacząco przybrał na wadze (Tabela 8).

	speed_ex1	**speed_ex1_static**
Rozmiar	17 KB	852KB

Tabela 8. Porównanie wielkości pliku wykonywalnego statycznego oraz dynamicznego

Dzieje się tak dlatego, że wszystkie wywołania standardowej biblioteki z których korzysta program zostały załączone do określonych sekcji bezpośrednio do pliku wykonywalnego ELF (ang. Executable

```
section .text
  global _start ; info for linker

_start:             ;tell linker entry point
  mov rcx, 1000000  ;initialize loop with 1000000 iterations
  mov rax, 0

l1:
  add rax, 2        ;add 2 to rax
  loop  l1          ;loop again

  mov   rax, 60     ;system call for exit
  xor rdi, rdi      ;return value - 0
  syscall
```

Wycinek kodu 12. Przykładowa pętla w assemblerze. Program speed_loop.c z [19]

Linkable Format – format plików wykonywalnych pod linuksem). W tym przypadku jest to już całkowicie samowystarczalny plik, który nie będzie wymagał już określonych bibliotek do pomyślnego uruchomienia. Czy można jednak jeszcze bardziej ograniczyć plik wykonywalny, aby móc policzyć każdą wykonywaną instrukcję? W językach wysokiego poziomu, jak można sobie wyobrazić, nie jest to możliwe. Aby tego dokonać, należy zejść do poziomu assemblera. Spójrzmy na przykład z Wycinka nr. 12. Aby skompilować ten program program, należy użyć:

$ nasm -felf64 speed_loop.asm && ld speed_loop.o -o speed_loop

Następnie debugger zostaje uruchomiony jak poprzednio:

```
$ python3 measure_deb.py speed_loop
Potomek wykonal 2000005 instrukcji, w czasie 32 sekund
```

Wycinek kodu 13. Wyjście ze skryptu measure_deb.py dla prostego programu w assemblerze

Jak widać, tym razem śledzony proces wykonał znacznie mniej instrukcji, które jesteśmy z resztą w stanie „ręcznie" policzyć. Jednak wciąż czas wykonania pozostaje znacznie dłuższy, niż gdyby program wykonywał się bez żadnego nadzoru. Nieco lżejszym obciążeniem dla śledzonego procesu jest technika zwana wykonaniem krokowym dla bloków (ang. block-stepping). Nie jest to jednak mechanizm udostępniony w postaci parametru przekazywanego jako pierwszy argument do *ptrace* (tak jak w przypadku PTRACE_SINGLESTEP). Polega on na tym, że zamiast śledzić każdą kolejną instrukcję, debugger przejmuję kontrolę tylko w przypadku skoku warunkowego, czyli gdy wykonanie programu uległo rozgałęzieniu.

W książce [2] znajduje się dokładne porównanie wpływu różnych technik debuggowania na szybkość wykonywanego kodu. Poniżej znajduje się tabela prezentująca fragment tych badań dla konwersji pewnego pliku pdf na formę tekstową:

Metoda śledzenia	Czas wykonania (rzeczywisty)
Brak śledzenia	0 m 365 s
Wykonanie krokowe (single-stepping)	285 m 9,137 s
Wykonanie krokowe dla bloków (block-stepping)	40 m 40,233 s
Mechanizm Intel PT	0 m 0.435 s

Tabela 9. Pomiar czasu wykonania programu obciążonego różnymi metodami śledzenia

Duży skok wydajności debuggowania przyniósł asynchroniczny mechanizm IPT (Intel Processor Ptrace), w którym mikroprocesor zapisuje w tle w pamięci różne informacje na temat śledzonego procesu np. informację czy skok warunkowy się odbył czy nie, skompresowane wartości rejestru RIP w procesie śledzonym czy informacje na temat różnych zdarzeń (np. wyjątków) które wystąpiły w śledzonym procesie. Dużą wydajność Intel uzyskał m.in. dzięki odpowiedniemu skompresowaniu tych informacji zapisywanych do pamięci RAM, co zwiększyło szybkość zapisu oraz zapotrzebowanie na pamięć operacyjną. Mechanizm IPT używa do działania rejestrów MSR (ang. Model Specific Registers) [5] które pozwalają mikroprocesorowi m.in. na kontrolowanie wydajności działania. Wcześniejszym mechanizmem stworzonym przez Intel był BTS (ang. Intel Branch Trace Store) od którego nowy mechanizm – IPT – jest co najmniej kilkukrotnie szybszy.

3.3. Zdarzenia debugowania

W czasie wykonywania programu może wystąpić wiele sytuacji, w których z różnych powodów wykonanie to zostanie nagle przerwane zwracając błąd, czy też po prostu odnotowując pewne wskazane informacje w postaci logów, gdy program napotka konkretne zdarzenie. Przykładami takich zdarzeń mogą być [11]:

- Wyjątki programowe bądź sprzętowe (ang. software / hardware breakpoints) – zastawiane przez debugger, zatrzymujące program w przypadku natrafienia na pewien obszar pamięci, bądź wykonania określonej funkcji;
- Nieuprawniony dostęp do pamięci (ang. memory violations) – zdarzenia będące odpowiedzią systemu operacyjnego na niewłaściwy dostęp do stron pamięci;
- Niepoprawne operacje (ang. exceptions) – wszelkiego rodzaju błędy, przeoczenia programistów skutkujące tym, że np. program próbuje uzyskać dostęp do zasobu który nie istnieje, bądź do którego nie ma uprawnień;

Listę zdarzeń, które debugger może przechwycić można rozszerzyć, lecz ich wystąpienie zależne jest od konkretnej architektury czy środowiska, na które stworzony został dany program, a także od rodzaju realizowanych przez niego zadań. Przykładowo może to być utworzenie nowego wątku czy załadowanie biblioteki współdzielonej. Innym przykładem zdarzenia może być otrzymywanie sygnału od innego procesu przy użyciu np. narzędzia kill(2), co może zmusić program do wykonania określonego działania np. wypisania na *stdout* statusu transferu przez narzędzie *dd* po otrzymaniu sygnału SIG_USR.

3.3.1. Wyjątki stawiane przez debugger

Zatrzymanie procesu, aby następnie móc uzyskać dostęp do aktualnego stanu rejestrów, ramki stosu danej funkcji czy wartości poszczególnych zmiennych, odbywa się przy użyciu wyjątków (ang. breakpoints). W tym podrozdziale zostaną przedstawione trzy rodzaje wyjątków, z których każdy ma swoje określone zastosowanie i charakterystykę:

- Wyjątki programowe (ang. software breakpoints) – zastąpienie dowolnej instrukcji w kodzie bajtowym specjalną instrukcją, która powoduje natychmiastowe, bezwarunkowe przerwanie wykonywania;
- Wyjątki sprzętowe (ang. hardware breakpoints) – zmiana wartości określonych rejestrów procesora zwanych rejestrami debuggera;
- Wyjątki pamięciowe (ang. memory breakpoints) – manipulacja uprawnieniami określonych stron pamięci;

3.3.1.1. Wyjątki programowe

Najczęściej używany i najbardziej uniwersalny rodzaj wyjątków. Debugger podmienia pierwszy bajt w instrukcji o wskazanym adresie na wartość 0xCC, jednocześnie zapamiętując oryginalną wartość. Kiedy CPU w czasie wykonania programu napotka na tę instrukcję, natychmiast zatrzymuje wykonywanie aktualnego programu i przechodzi do zdefiniowanej funkcji obsługującej to zdarzenie w debuggerze.

Debugger sprawdza, czy wskaźnik instrukcji (rejestr RIP) wskazuje na adres instrukcji znajdujący się na liście zdefiniowanych wyjątków programowych. Jeśli tak, to podmienia bajt na oryginalny – w tym przypadku mamy do czynienia z tzw. wyjątkiem jednorazowym (ang. one-shot breakpoints) – następnie debugger przystępuje do dalszej obsługi zdarzenia np. wypisania na stdout wartości rejestrów.

Przykładowo, instrukcja o adresie:

Adres w pamięci	Instrukcja w postaci mnemonicznej	Opkod
0x8877665544332211	MOV RAX, RBX	48 89 D8

Tabela 10. Przykład instrukcji assemblera

Zostaje zamieniona na instrukcję:

0x8877665544332211	MOV RAX, RBX	48 89 CC

Tabela 11. Przykład ustawionego wyjątku programowego

Tabela 12. obrazuje proces ustawiania wyjątku programowego.

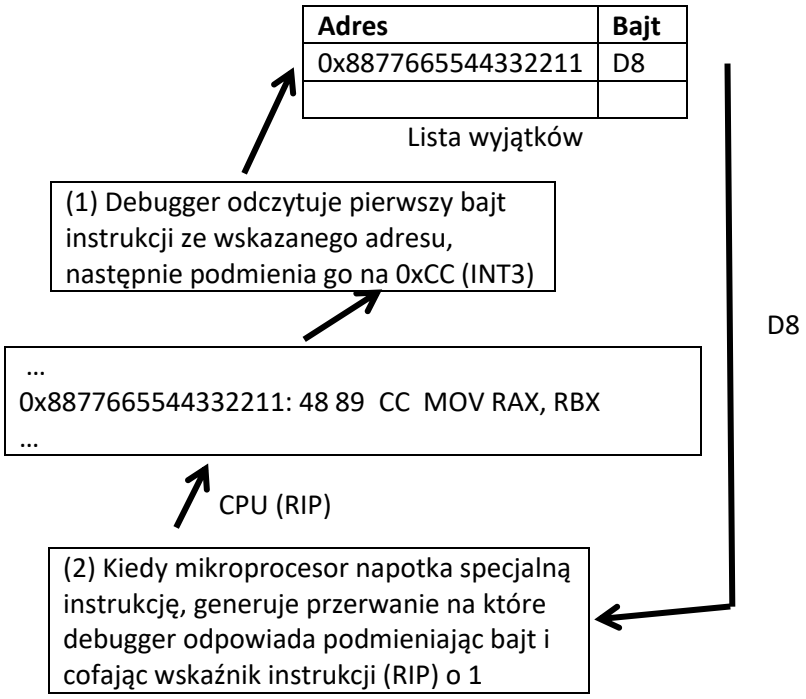


Tabela 12. Ustawianie wyjątku programowego [16]

W kolejnej części rozdziału proces ten zostanie przedstawiony w praktyce. W pierwszej kolejności należy znaleźć pożądany adres. W najprostszym przypadku, mając przykładowy program *hello_loop.asm* [19], przy użyciu narzędzia *objdump* [21] szukamy potencjalnego adresu dla ustawienia wyjątku:

Przede wszystkim należy skompilować przykładowy program:

$ nasm -felf64 hello_loop.asm && ld hello_loop.o -o hello_loop

Następnie wyświetlamy obraz w pamięci wygenerowanego pliku wykonywalnego (Wycinek 14.)

```
$ objdump -d -M Intel hello_loop
hello_loop:     file format elf64-x86-64
(…)
  401012:       b8 01 00 00 00          mov    $0x1, %eax
  401017:       bf 01 00 00 00          mov    $0x1, %edi
  40101c:       48 be 00 20 40 00 00    movabs $0x402000, %rsi
  401023:       00 00 00
  401026:       ba 01 00 00 00          mov    $0x1, %edx
  40102b:       51                      push   %rcx
  40102c:       0f 05                   syscall
  40102e:       48 8b 04 25 00 20 40    mov    0x402000,%rax
  401035:       00
(…)
```

Wycinek kodu 14. Inspekcja kodu programu przy użyciu narzędzia objdump

Opcja -d oznacza wyświetlenie instrukcji assemblera, natomiast -M przekazuje rodzaj użytego do wyświetlenia syntaxu. Powyżej został wyświetlony jedynie interesujący fragment dezasemblowanego programu. Interesująca nas instrukcja znajduje się pod adresem 0x000000000040102e. W skrócie, dany program wyświetla w pętli kolejne cyfry od ‘0’ do ‘9’. Chcemy zatrzymać program tuż przed wypisaniem kolejnych cyfr na stdout. Główna pętla debuggera przyjmie wtedy postać (skrypt *soft_deb.py* z [19]) widoczną na wycinku 15.

```
def debugger(pid):
    breakpoint1_addr = 0x0000000040102c
    status = wait()

    while WIFSTOPPED(status[1]):
        soft_bp(pid, breakpoint1_addr)
        ptrace(PTRACE_SINGLESTEP, pid, 0, 0)
        status = wait()
```

Wycinek kodu 15. Ustawianie wyjątku programowego. Skrypt soft_deb.py z [19]

Czekamy aż śledzony proces wykona pierwszą instrukcję, następnie zatrzymujemy jego wykonanie, ustawiamy punkt przerwania programowego na wskazany adres. Funkcja *soft_bp()* została przedstawiona na Wycinku 16.

Funkcja ta wykonuje następujące operacje – odczytuje wartość rejestru RIP, następnie zakładamy wyjątek programowy pod wskazanym adresem. Kontynuujemy wykonanie programu czekając aż trafi na przerwanie. Gdy potomek otrzymuje sygnał nr. 5 (SIGTRAP), przechwytujemy ponownie jego działanie. Tutaj należałoby zaimplementować obsługę zdarzenia, ale zajmiemy się tym w następnych rozdziałach. Po zakończeniu pracy należy przywrócić oryginalny bajt na jego miejsce w instrukcji, cofnąć licznik instrukcji (RIP) o 1 oraz wznowić wykonanie programu.

```
def soft_bp(pid, instr_addr):
    # Pobierz aktualny stan RIP
    ptrace(PTRACE_GETREGS, pid, 0, byref(regs))
    print("\nPotomek zostal zatrzymany na adresie RIP = 0x%x" % (regs.rip))

    # Odczyt instrukcji ze wskazanej komorki pamieci
    org_instr = ptrace(PTRACE_PEEKTEXT, pid, c_ulonglong(instr_addr), 0)
    print("Oryginalna zawartosc pamieci z 0x%x:  0x%x" % (instr_addr, org_instr))

    # Zapis specjalnej instrukcji int3 pod wskazany adres
    instr_trap = (org_instr & 0xFFFFFFFFFFFFFF00) | 0xCC
    ptrace(PTRACE_POKETEXT, pid, instr_addr, instr_trap)
    zm_instr = ptrace(PTRACE_PEEKTEXT, pid, instr_addr, 0)
    print("Zmieniona zawartosc pamieci z 0x%x:  0x%x" % (instr_addr, zm_instr))

    # kontynuuj wykonanie potomka, zaczekaj az dotrze do wyjatku
    ptrace(PTRACE_CONT, pid, 0, 0)

    # Petla debuggera - oczekujemy na dalsze zdarzenia
    status = wait()
    if (WIFSTOPPED(status[1])):
        print("Potomek otrzymal sygnal: ", WSTOPSIG(status[1]))

    # Pobierz aktualny stan RIP
    ptrace(PTRACE_GETREGS, pid, 0, byref(regs))
    print("Potomek zostal zatrzymany na adresie RIP = 0x%x" % (regs.rip))

    # Przywroc oryginalna instrukcje, cofnij wskaznik instrukcji (RIP)
    ptrace(PTRACE_POKETEXT, pid, instr_addr, org_instr)
    regs.rip -= 1
    ptrace(PTRACE_SETREGS, pid, 0, byref(regs))

    input()
#
```

Wycinek kodu 16. Wnętrze funkcji do ustawiania wyjątku programowego. Skrypt soft_deb.py

W ramach przykładu po wyjściu z funkcji *soft_bp()* nakazujemy potomkowi wykonanie pojedynczej instrukcji, następnie w pętli ponownie ustawiamy przerwanie itd. Widoczne jest to jako wypisywanie kolejnych cyfr debugowanego programu (po przejściu instrukcji *input()* w skrypcie debuggera). Na wyjściu debuggera generowane zostają kolejne cyfry wypisywane przez śledzony program i stan rejestru RIP:

```
$ python3 soft_deb.py hello_loop
(...)
0
Potomek zostal zatrzymany na adresie RIP = 0x40102e
Oryginalna zawartosc pamieci z 0x40102c:  0x200025048b48050f
Zmieniona zawartosc pamieci z 0x40102c:  0x200025048b4805cc
Potomek otrzymal sygnal:  5
Potomek zostal zatrzymany na adresie RIP = 0x40102d
(...)
```

Wycinek kodu 17. Ustawianie kolejnych wyjątków w pętli śledzonego procesu

Po kolejnych naciśnięciach klawisza enter na ekranie ukazują się kolejne cyfry generowane przez program. W przypadku programu w języku wysokiego poziomu (np. C) znalezienie odpowiedniego adresu jest nieco bardziej kłopotliwe. Dzieje się tak z uwagi na dynamiczne linkowanie bibliotek których adresy bezwzględne są obliczane w trakcie uruchamiania programu. Mając przykładowy program *hello_loop.c* [19] skompilowany w standardowy sposób:

$ gcc -o hello_loop hello_loop.c

Gdy przyjrzymy się wyjściu narzędzia *objdump*, widać że adresy kolejnych instrukcji mają inną strukturę niż w poprzednim przykładzie (i.e. są dużo mniejsze):

```
0000000000001169 <printer>:
    1169:              f3 0f 1e fa           endbr64
    116d:              55                    push   %rbp
    116e:              48 89 e5              mov    %rsp,%rbp
    1171:              48 83 ec 10           sub    $0x10,%rsp
    1175:              89 7d fc              mov    %edi,-0x4(%rbp)
    1178:              8b 45 fc              mov    -0x4(%rbp),%eax
    117b:              89 c6                 mov    %eax,%esi
    117d:              48 8d 3d 80 0e 00 00  lea    0xe80(%rip),%rdi #
2004 <_IO_stdin_used+0x4>
    1184:              b8 00 00 00 00        mov    $0x0,%eax
    1189:              e8 d2 fe ff ff        callq  1060 <printf@plt>
    118e:              90                    nop
    118f:              c9                    leaveq
    1190:              c3                    retq
```

Wycinek kodu 18. Wyjście z narzędzia objdump dla dynamicznego pliku wykonywalnego

Adresy podane tutaj są jedynie offsetami, które obliczane są (przy pomocy sekcji GOT/PLT) na rzeczywiste adresy w pamięci w czasie wykonania programu przez linker dynamiczny. Istnieje możliwość obliczenia ich z poziomu debuggera przed uruchomieniem programu, jednak proces ten jest nieco bardziej złożony i wrócimy do niego w późniejszych rozdziałach. W tym momencie, aby nieco ułatwić zadanie, wystarczy że ponownie skompilujemy program w gcc z użyciem flagi *-static* a następnie wyświetlimy jego zawartość w *objdump*:

$ gcc -o hello_loop_static hello_loop.c -static && objdump -d -M Intel hello_loop_static | more

Po wyszukaniu interesującej funkcji *printer()* otrzymamy wyjście przedstawione na Wycinku 18. Teraz już jesteśmy w stanie ustawić przerwanie w ustalonym miejscu, podając (analogicznie do poprzedniego przykładu) adres wywołania funkcji z biblioteki standardowej *printf*(): 0x0000000000401ce5. Przetestowanie tego przykładu zostaje dla czytelnika. Podsumowując, ustawianie wyjątków programowych jest stosunkowo prostym zadaniem, ponadto dzięki charakterystyce tych wyjątków możemy ustawić ich nieograniczoną ilość. Z drugiej jednak strony, są to wyjątki ustawiane na pojedynczy bajt (0xCC), zatem wymagają precyzji w określeniu miejsca docelowego takiego wyjątku. Warto zwrócić także uwagę, że wyjątki te modyfikują pamięć procesu, a zatem są łatwo wykrywalne z poziomu analizowanego procesu. W przypadku gdyby taki program chciał uniknąć śledzenia, może zostać zabezpieczony okresowo sprawdzając integralność zajmowanej przez niego pamięci. W przypadku wykrycia jakichkolwiek zmian, może na przykład ukryć szkodliwe działanie, bądź zakończyć awaryjnie wykonanie, utrudniając tym samym analizę [11].

```
0000000000401ce5 <printer>:
(…)
  401cf9:       48 8d 3d 04 33 09 00   lea
0x93304(%rip),%rdi        # 495004 <_IO_stdin_used+0x4>
  401d00:       b8 00 00 00 00         mov    $0x0,%eax
  401d05:       e8 b6 ee 00 00         callq  410bc0
<_IO_printf>
  401d0a:       90                     nop
  401d0b:       c9                     leaveq
  401d0c:       c3                     retq
```

Wycinek kodu 19. Obraz kodu statycznego przedstawiony przy użyciu narzędzia objdump

3.3.1.2. Wyjątki sprzętowe

Kolejnym rodzajem wyjątków w arsenale debuggera są wyjątki sprzętowe. Do ich implementacji używane są rejestry debuggera, opisane we wcześniejszych rozdziałach. Ilość wyjątków sprzętowych które debugger jest w stanie ustawić jednocześnie jest ograniczona przez liczbę rejestrów i w wynosi 4 architekturze x86-64.

Przykładowo, wyjątek ma zostać założony pod adresem 0x8877665544332211, jest to wyjątek na odczyt/zapis o długości 2 bajtów. Jak wspomniano w rozdziale dotyczącym wyjątków debuggera, górne 32 bity rejestru kontrolnego DR7 są wyłączone z użytku. Zatem stan tego rejestru wyglądałby następująco w naszym przypadku (Tabela 13).

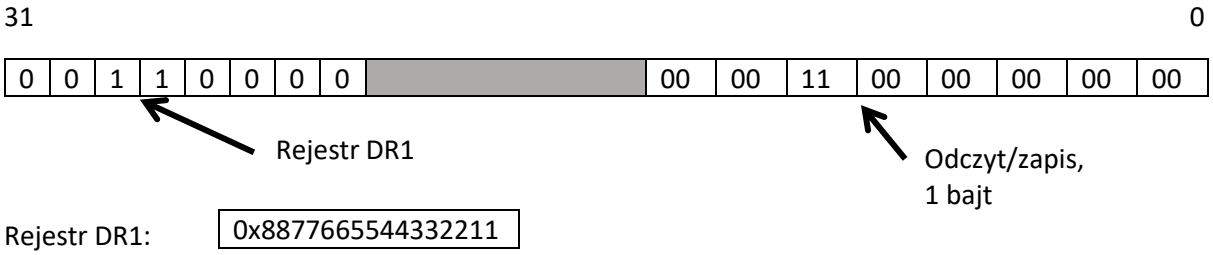


Tabela 13. Przykład zawartości rejestru DR7 dla ustawionego wyjątku sprzętowego

Wyjątki sprzętowe używają przerwania o numerze 1 (INT1), który zarezerwowany jest dla wyjątków sprzętowych oraz trybu krokowego wykonania (single-step events). Są one obsługiwane przez system operacyjny i debugger w analogiczny sposób to wyjątków programowych, natomiast sam mechanizm schodzi o krok niżej – zanim CPU wykona jakąkolwiek instrukcję, najpierw sprawdza czy jej adres nie został oznaczony w rejestrach debuggera i w odpowiednim przypadku rzuca przerwanie INT1. Wyjątki sprzętowe z jednej strony pozwalają na działanie debuggera bez konieczności modyfikowania analizowanego kodu, jednak posiadają także pewne ograniczenia [11] – z uwagi na budowę rejestrów debuggera można ustawić jednocześnie do 4 wyjątków, a każdy z nich może dotyczyć do 8 bajtów pamięci. W związku z tym śledzenie większych obszarów pamięci jest tutaj utrudnione. Ponadto, linux nie umożliwia bezpośredniego dostępu do rejestrów debuggera z poziomu przestrzeni użytkownika. Jedyna możliwość odczytu i zapisu tych rejestrów możliwa jest przez kernel (ring 0) [22], po odczytaniu kontekstu konkretnego wątku [23]. Jest to kolejne utrudnienie ze strony linuksa mające podnieść ogólne bezpieczeństwo. Rejestry debuggera są często wykorzystywane przez wszelkiego rodzaju malware czy programy do crackowania legalnego oprogramowania [24] z uwagi na ich jedną bardzo ważną właściwość – mianowicie pozwalają na ustawianie wyjątków bez ingerencji w przestrzeń analizowanego programu. Przez to trudniej jest je wykryć, a nawet w przypadku gdy oprogramowanie chcące ustrzec się przed możliwością bycia debuggowanym z użyciem wyjątków sprzętowych, istnieją specjalne funkcje w kernelu pozwalające zwrócić fałszywy stan rejestrów debuggera, maskując ustawione przez złośliwe oprogramowanie wyjątki [25]. Ma to zatem sens, w kwestii bezpieczeństwa, aby jak najbardziej utrudnić możliwość ustawienia tych wyjątków. Co ciekawe, podobne mechanizmy na platformie Windows funkcjonują w dużo prostszym ujęciu [11].

3.3.2. Nieuprawniony dostęp do obszarów pamięci (ang. memory violations)

Nie są to właściwie wyjątki w tym samym znaczeniu co wyjątki sprzętowe bądź programowe. Są po prostu mechanizmem udostępnionym przez systemy operacyjne do zarządzania dostępem do pamięci [11].

Strona [26] jest najmniejszą jednostką pamięci widoczną dla systemu operacyjnego. Po zaalokowaniu danej strony, system operacyjny ustawia dla niej konkretne uprawnienia dotyczące dostępu do jej danych. Wśród tych uprawnień są [11]:

- Dostęp na wykonanie – pozwala wykonać kod z danej strony, natomiast wyrzuca błąd dostępu (ang. access violation) jeśli proces próbuje zapisać lub czytać ze strony;
- Dostęp na odczyt – Pozwala procesowi czytać z danej strony, natomiast jakikolwiek zapis lub próba wykonania powoduje błąd dostępu;
- Dostęp na zapis – analogicznie, pozwala jedynie na zapis (w systemach o architekturze programowej x86-64 nie ma możliwości ustawienia dostępu jedynie na zapis – automatycznie przyznawany jest także dostęp na odczyt [27]);
- tzw. Guard Page – jakakolwiek próba dostępu powoduje rzucenie jednorazowego wyjątku, zaś zawartość strony jest przywrócona

Większość systemów operacyjnych pozwala łączyć ze sobą te uprawnienia – np. zezwolenie na odczyt oraz zapis, ale nie wykonanie. Każdy system operacyjny posiada także mechanizmy pozwalające manualnie ustawić uprawnienia dla danego obszaru pamięci bądź zdobyć informację o aktualnych uprawnieniach. I tutaj wchodzą wyjątki pamięciowe – które w zasadzie nie są wyjątkami, ale z uwagi na funkcjonalność uprawnień do stron pamięci pełnią rolę analogiczną do przedstawionych wcześniej rodzajów wyjątków. Najbardziej interesującym uprawnieniem jest tutaj *Guard Page,* używane przez OS aby rozdzielić stos od sterty czy uniemożliwić danym przekroczenie pewnych ram pamięci.

Wyjątki pamięciowe mogą działać w następujący sposób – analizowany przez nas program wykonuje operacje, której wynik przechowuje w znanym obszarze pamięci – więc aby móc przechwycić wynik tej operacji przez debugger, możliwe jest ustawienie uprawnienia Guard Page na danym obszarze pamięci, a następnie przechwycenie i obsługę rzuconego wyjątku przez debugger. Dodatkowo, wyjątki pamięciowe mają jedną przewagę wobec wyjątków programowych – nie modyfikują wykonywanego przez program kodu, więc są trudniejsze do wykrycia.

W systemach linuksowych do zarządzania uprawnieniami do pamięci służy narzędzie *mprotect(2)* [28]. W celu ustawienia uprawnień do konkretnej strony pamięci potrzebuje ono adresu początku strony [29] oraz ustawionej flagi mówiącej o tym jakie uprawnienia mają zostać jej nadane (np. PROT_NONE służy jako wcześniej przedstawiony mechanizm GUARD PAGE). Ważną cechą wyjątków pamięciowych jest to, że nie pozwalają one na kontrolę dostępu do pojedynczej instrukcji, ale na całe strony pamięci w których dana instrukcja, bądź interesujące nas dane, mogą się znajdować. Jest to zatem mechanizm mniej precyzyjny, jednakże pozwalający śledzić znacznie większe obszary pamięci – w przypadku gdy albo interesuje nas szerszy kontekst wykonania programu, albo np. nie znamy dokładnego adresu docelowej funkcji. Domyślna wielkość strony w systemie o architekturze x86-64 wynosi około 4KB.

Ważnym ograniczeniem systemów linuksowych, w przypadku manipulacji tymi uprawnieniami jest to, że każdy proces może modyfikować jedynie uprawnienia do pamięci przydzielonej mu przez system [27] . Nie ma możliwości zmiany uprawnień do pamięci innych procesów. Jest to spowodowane tym, że każdy proces posiada własny obszar pamięci wirtualnej przydzielonej przez kernel. Żaden proces

nie ma zatem dostępu do wybranego (ang. arbitrary) adresu pamięci, gdyż domyślnie widzi jedynie swój przydział. Jednym ze sposobów obejścia tego problemu jest wstrzyknięcie kodu który ustawi pożądane uprawnienia z kontekstu analizowanego procesu [27]. Proces ten nie jest trudny, jednak zawiera kilka kroków:

1. Przechwycenie procesu z *PTRACE_ATTACH* (następuje zatrzymanie procesu)
2. Podmiana aktualnej instrukcji wskazywanej przez RIP na *syscall*
3. Zmiana wartości rejestrów tak, aby odpowiadały wywołaniu *mprotect* (opisane dokładnie w [27])
4. Wznowienie procesu aż do przejścia przez *syscall*, następnie cofnięcie licznika RIP o 1 oraz przywrócenie oryginalnych zawartości rejestrów

Powyższe kroki umożliwią „wstrzyknięcie" wywołania *mprotect* w kontekst analizowanego procesu, tym samym ustawiając wyjątek pamięciowy na wskazany adres.

3.3.3. Podsumowanie różnic między różnymi kategoriami wyjątków

Po zapoznaniu się z najczęściej używanymi mechanizmami generowania wyjątków przez debuggery, można dojść do wniosku że żadne z tych rozwiązań nie jest idealne. Linuks dodatkowo nie ułatwia jeszcze pracy w wielu aspektach, co jednak stosunkowo podnosi bezpieczeństwo tego systemu. Najłatwiejsze do implementacji okazały się wyjątki programowe, jednak ich charakterystyka sprawia że nie zawsze są rozwiązaniem idealnym, gdyż np. mogą zostać łatwo wykryte przez analizowany proces. Wyjątki sprzętowe nie są łatwe do implementacji, gdyż wymagają bezpośredniej interakcji z kernelem, jednak umożliwiają większą elastyczność i trudniej jest je wykryć. Ich znaczącą wadą jest, że jesteśmy w stanie ustawić jedynie 4 na raz. Wyjątki pamięciowe zaś są dużo mniej precyzyjnym mechanizmem, ale dzięki temu umożliwiają śledzenie większych obszarów pamięci, ponadto są także trudne do wykrycia przez śledzony program (aczkolwiek należy zaznaczyć, że metoda zakładania wyjątków pamięciowych przedstawiona w poprzednim rozdziale w sposób oczywisty modyfikuje pamięć procesu, stąd też działa analogicznie do wyjątków programowych – a zatem jest łatwa do wykrycia. Podsumowanie wiadomości o wyjątkach przedstawione zostało w Tabeli 14.

Rodzaj wyjątków	Charakterystyka	Dostępność API	Precyzja	Uwagi
Programowe	Podmiana dowolnej instrukcji na pojedynczy bajt (0xCC) generujący przerwanie. Możliwość ustawienia dowolnej liczby przerwań	Mechanizmy ptrace	Pojedyncza instrukcja	Użycie modyfikuje pamięć procesu (wykrywalność)
Sprzętowe	Użycie rejestrów debuggera	Funkcje kernela	Od 1 do 8 bajtów pamięci	Maksymalnie 4 wyjątki równocześnie (ograniczona liczba rejestrów)
Pamięciowe	Użycie funkcji *mprotect*, jednak pozwala ona tylko na modyfikację przestrzeni adresowej danego procesu	Mechanizmy mprotect	Całe strony pamięci (4KB)	Do implementacji potrzebne są specjalne funkcje kernela, bądź funkcja mprotect musi zostać wykonana z kontekstu analizowanego procesu

Tabela 14. Różnice między typami wyjątków stosowanymi przez debuggery

3.3.4. Rozszerzona obsługa zdarzeń

W momencie zatrzymania analizowanego przez debugger programu na określonym zdarzeniu debuggowania, ważne jest poznanie kontekstu procesu w celu odpowiedzi na pytanie – jak się tu znaleźliśmy? – czy – co spowodowało dane zdarzenie? Przypomina to trochę udział w śledztwie w sprawie morderstwa (w którym my jesteśmy zabójcą). Odpowiedzi na nurtujące nas pytania może udzielić zawartość rejestrów w danym momencie wykonywania programu, a następnie na podstawie zawartości rejestrów przegląd np. aktualnej ramki stosu w celu zidentyfikowania zmiennych lokalnych. Przykład został zawarty w skrypcie *event_deb.py*, którego fragment przedstawia Wycinek 19.

```
def obsluga_zdarzenia(pid):
    # Wyswietl aktualny stan rejestrow oraz kontekst procesu
    ptrace(PTRACE_GETREGS, pid, 0, byref(regs))
    print("==================== DUMP ==========================")
    print("RIP = 0x%x" % (regs.rip))
    print("RAX = 0x%x" % (regs.rax))
    print("RBX = 0x%x" % (regs.rbx))
    print("RCX = 0x%x" % (regs.rcx))
    print("RDX = 0x%x" % (regs.rdx))
    print("RSI = 0x%x" % (regs.rsi))
    print("RDI = 0x%x" % (regs.rdi))
    print("RSP = 0x%x" % (regs.rsp))
    print("RBP = 0x%x" % (regs.rbp))
    print("EFLAGS = 0x%x" % (regs.eflags))
    print("==================== DUMP ==========================")
    print()
```

Wycinek kodu 20. Przegląd stanu rejestrów. Skrypt event_deb.py z [19]

Używając narzędzia *objdump* znaleźliśmy dogodne miejsce dla ustawienia przerwania (Wycinek 20).

```
401042: e2 c6                   loop   40100a <l1>
```

Wycinek kodu 21. Adres instrukcji loop w programie hello_loop.asm [19]

Instrukcja ta odejmuje 1 od licznika pętli (RCX), następnie przejdzie do kodu znajdującego się pod adresem 40100a (etykieta *l1*) . Gdy w tym momencie uruchomiony zostanie skrypt, otrzymamy wyjście przedstawione na Wycinku 21.

```
$ p3 event_deb.py hello_loop_asm
(…)
3
Potomek otrzymal sygnal:  SIGTRAP
==================== DUMP ==========================
RIP = 0x401043
RAX = 0x34
RBX = 0x0
RCX = 0x7
RDX = 0x1
RSI = 0x402000
RDI = 0x1
RSP = 0x7fffffffdf00
RBP = 0x0
EFLAGS = 0x202
==================== DUMP ==========================
```

Wycinek kodu 22. Zrzut rejestrów w momencie wykonywania hello_loop_asm [19]

Jak widać, po 3 iteracjach w rejestrze RAX znajduję się wartość 0x34 („4”), natomiast w liczniku pętli 0x7. Skrypt przedstawia kilka ważniejszych rejestrów w kontekście analizowanego programu. Dodatkowo można by umieścić zawartość ramki stosu w dumpie, odczytaną z adresu wskazywanego przez RSP. Na koniec należy dodać, że obsługa wydarzeń nie musi mieć wartości czysto informacyjnej. Możemy np. stworzyć taki skrypt, który wykonywałby automatyczne akcje na śledzonym procesie w zależności od zawartości jego określonych obszarów pamięci.

3.3.5. Wyjątki / błędy w czasie wykonywania

W czasie wykonywania programu może wystąpić wiele zdarzeń które nie zostały przewidziane przez programistów. Może to być np. dostęp do pliku który nie istnieje, albo wykonanie niepoprawnej operacji matematycznej. We wszystkich takich przypadkach, gdy mikroprocesor napotka na błąd w czasie wykonywania programu, generowane jest przerwanie a wykonanie programu zostaje awaryjnie zakończone (np. gdy potomek otrzyma sygnał od kernela o nazwie SIGSEGV, gdy próbuje uzyskać dostęp do nieprzydzielonego mu obszaru pamięci).

Gdy takie nieprzewidziane zdarzenie nastąpi, kontrola może zostać automatycznie przekazana debuggerowi gdy przechwyci on sygnał wysłany do procesu. Można wtedy dokonać inspekcji np. pamięci procesu w celu znalezienia przyczyny problemu. W językach wysokiego poziomu istnieje możliwość przechwycenia pewnych zdarzeń natywnie, bez pomocy zewnętrznego debuggera, przy użyciu instrukcji try (…) catch, a każdy z tych języków posiada określoną strukturę klas wyjątków które mogą się pojawić, np. w języku Python struktura ta została zobrazowana pod linkiem [30].

4. Praktyczne aspekty

Po przeanalizowaniu popularnych technik śledzenia procesów wykorzystywanych przez debuggery, przyszedł czas na krótkie podsumowanie dotyczące praktycznych możliwości stworzonych przez te mechanizmy. Żaden omawianych tutaj przykładów nie jest jednoznacznie zły (tzn. wykorzystywany tylko przez złośliwe oprogramowanie) bądź dobry. Cel implementacji tych technik uzależniony jest jedynie od intencji programisty. Poniższy rozdział przedstawi kilka dodatkowych zagadnień związanych z możliwościami debuggera, wraz z towarzyszącymi przykładami ich użycia.

4.1. Wykrywanie debuggera

Mechanizm *ptrace* pozwala na wykrycie z poziomu danego procesu czy jest on śledzony przez inny proces. Może to zostać użyte np. przez złośliwe oprogramowanie, by ukryć część kodu przed próbą analizy i utrudnić tym samym ustalenie jakie potencjalne szkody może wyrządzić w systemie. Z drugiej strony, może zostać także użyte przez np. oprogramowanie do zabezpieczania wszelkiej cyfrowej własności intelektualnej, utrudniając tym samym próby uruchamiania oprogramowania bez posiadania ważnej licencji. W przypadku *ptrace* wystarczy jedynie wywołać (Wycinek 23) [31]:

```
if ptrace(PTRACE_TRACEME, 0, NULL, 0) == -1:
    print("traced!")
```

Wycinek kodu 23. Wykrycie działania debuggera przy użyciu mechanizmu ptrace

Innym sposobem na wykrycie działania debuggera jest iterowanie listy działających procesów w celu znalezienia programu będącego popularnym debuggerem np. gdb. Jest to jednak bardzo mało precyzyjny sposób gdyż samo działanie debuggera w tle nie gwarantuje, że akurat konkretny proces jest śledzony, ponadto może być łatwo oszukane przez zmianę nazwy debuggera aby wyświetlał się na liście działających procesów jako potencjalnie „nieszkodliwe" oprogramowanie.

Jak zostało wcześniej wspomniane, można także łatwo wykryć obecność wyjątków programowych poprzez analizę zmian obrazu programu wykonywalnego w pamięci. W przypadku wyjątków sprzętowych, pomocne może być sprawdzenie zawartości rejestrów debuggera, jednak metoda ta nie gwarantuje, że wyjątki takie nie zostały ustawione, z uwagi na możliwość przechwycenia funkcji systemowej odczytującej zawartość tych rejestrów. Każda z przedstawionych technik posiada zatem swoje ograniczenia [32].

4.2. Wstrzykiwanie kodu

W tej sekcji omówimy podstawy wstrzykiwania kodu do śledzonego procesu. Jedną z technik wykonania swojego kodu w dowolnym procesie jest nadpisanie adresu powrotu z funkcji. Jak już wcześniej wspomniano, w momencie wywołania funkcji na stos odkładane są parametry przekazywane do funkcji (w konwencji x86, natomiast w konwencji x86-64 przekazywane są poprzez określony zestaw rejestrów), a następnie adres z którego została wywołana funkcja, do którego wykonanie powróci po jej zakończeniu.

Znając ten adres, możemy nadpisać go adresem w pamięci, w którym znajduje się kod wstrzykiwanej funkcji. Po jej wykonaniu możemy w naturalny sposób powrócić do wcześniej wykonywanego kodu, zapamiętując i przywracając oryginalny adres. Znając ograniczenia procesów pod linuksem, wstrzykiwany kod musi znajdować się w przestrzeni adresowej procesu. Można jednak łatwo znaleźć dostępne miejsce w pamięci przyglądając się zakresom adresów z */proc/< PID procesu>/maps* [33] [32]. Następnie alokujemy pożądany zakres pamięci przy pomocy funkcji *mmap()* i wstrzykujemy w to miejsce kod. Jednym ze sposobów byłoby użycie trybu *PTRACE_POKEDATA* mechanizmu *ptrace*, jednak posiada ono jedno ważne ograniczenie – jesteśmy w stanie zapisać pod dany adres jedynie 8 bajtów (1 komórkę pamięci) jednocześnie. Innym rozwiązaniem byłoby użycie funkcji *proces_vm_writev(2)* [34]. Podejście to zostało szczegółowo opisane w artykule [35]. We wspomnianym przykładzie do nadpisania użyto funkcji wywoływanej z biblioteki standardowej, a nie funkcji należącej bezpośrednio do analizowanego procesu, jednak procedura w tym przypadku jest bardzo podobna. Po prostu zamiast szukać adresu bazowego segmentu tekstowego programu, szukamy adresu bazowego określonej biblioteki współdzielonej, a następnie offsetu interesującej nas funkcji w danej bibliotece.

W przykładzie posłużymy się nieco prostszą techniką – zamiast wstrzykiwać kod, podmienimy jedynie argument przekazywany do wywoływanej funkcji. Bardziej interesujące z perspektywy tego przykładu jest jednak odnalezienie interesującego nas adresu w dynamicznie linkowanym pliku wykonywalnym, proces ten zostanie pokazany krok po kroku.

W pliku *hello_loop.c* [19] znajduje się nieskończona pętla przekazująca kolejne wartości i=0,1,2… do funkcji *printer()*, która wypisuje na stdout numer każdej kolejnej iteracji, następnie przy pomocy funkcji *sleep()* program odczekuje sekundę, zwiększa licznik pętli o 1 itd. Program został skompilowany przy użyciu gcc:

```
$ gcc -o hello_loop hello_loop.c
```

Następnie przy użyciu *objdump* przyglądamy się adresom instrukcji (Wycinek 22).

```
$ objdump -d -M Intel hello_loop

hello_loop:     file format elf64-x86-64

Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp

(…)

0000000000001191 <main>:
    1191:       f3 0f 1e fa             endbr64
    1195:       55                      push   %rbp
    1196:       48 89 e5                mov    %rsp,%rbp
    1199:       48 83 ec 10             sub    $0x10,%rsp
    119d:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    11a4:       8b 45 fc                mov    -0x4(%rbp),%eax
    11a7:       89 c7                   mov    %eax,%edi
    11a9:       e8 bb ff ff ff          callq  1169 <printer>

(…)
```

Wycinek kodu 24. Analiza pliku wykonywalnego hello_loop [19] narzędziem objdump

Innym sposobem znalezienia interesującej nas funkcji jest samodzielne parsowanie symboli znajdujących się w tablicach .symtab oraz .dynsym w sekcji .dynamic. Można do tego użyć narzędzia [36]. Podejście to zostało szerzej opisane w [37] oraz [38].

Wywołanie funkcji *printer()* znajduje się pod offsetem 0x11a9. Główny punkt startowy programu znajduje się pod offsetem 0x1000. Będzie to przydatne później, przy obliczaniu rzeczywistego adresu w pamięci. Uruchamiamy program, następnie w drugim okienku konsoli sprawdzamy na jakie adresy pamięci zostały zmapowane jego określone sekcje (analogiczne działanie można uzyskać używając narzędzia *pmap*, np. *$ pmap -x `pidof (…)`*) w sposób przedstawiony na Wycinku 23.

Kolejne wiersze posiadają określoną strukturę kolumn:

1. Adres początkowy – końcowy
2. Uprawnienia
3. Offset
4. ID urządzenia
5. Inode
6. Nazwa (ścieżka) pliku wykonywalnego

```
$ cat /proc/`pidof hello_loop`/maps
564f4f0d6000-564f4f0d7000 r--p 00000000 00:32 1222   <sciezka do>/hello_loop
564f4f0d7000-564f4f0d8000 r-xp 00001000 00:32 1222   <sciezka do>/hello_loop
564f4f0d8000-564f4f0d9000 r--p 00002000 00:32 1222   <sciezka do>/hello_loop
564f4f0d9000-564f4f0da000 r--p 00002000 00:32 1222   <sciezka do>/hello_loop
564f4f0da000-564f4f0db000 rw-p 00003000 00:32 1222   <sciezka do>/hello_loop
564f50c88000-564f50ca9000 rw-p 00000000 00:00 0                          [heap]

(…)
```

Wycinek kodu 25. Przegląd map pamięci analizowanego programu

Jedna z pierwszych sekcji posiada flagę *x* (ang. executable), więc jest to najprawdopodobniej obszar pamięci w którym zmapowana została sekcja tekstowa pliku wykonywalnego (przechowująca kod programu). Jedną rzecz, na którą należy zwrócić uwagę jest mechanizm ASLR. Został on szerzej przedstawiony w [39], jednak w dużym skrócie jest to zabezpieczenie polegające na mapowaniu sekcji pliku wykonywalnego na losowe adresy w pamięci, utrudniając tym samym wiele ataków. Mechanizm ten można wyłączyć za pomocą (Wycinek 24).

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
[sudo] password (…):
0
```

Wycinek kodu 26. Wyłączenie losowości układu przestrzeni adresowej (ASLR) w systemie linuks

Wtedy adresy przypominałyby bardziej:

```
$ cat /proc/`pidof hello_loop`/maps
555555554000-555555555000 r--p 00000000 00:32 1222   <sciezka do>/hello_loop
555555555000-555555556000 r-xp 00001000 00:32 1222   <sciezka do>/hello_loop
555555556000-555555557000 r--p 00002000 00:32 1222   <sciezka do>/hello_loop

(…)
```

Wycinek kodu 27. Mapy pamięci analizowanego programu po wyłączeniu ASLR

Teraz można już za pomocą prostej matematyki wyznaczyć adres przerwania programowego. Użyjemy do tego skryptu *interception.py* [19]. Obliczenie adresu mogłoby wyglądać w ten sposób przedstawiony na wycinku 26 (fragment funkcji *debugger())*.

```
# Obliczanie adresu przerwania
instr_offset = 0x11a9 - 0x1000

maps = load_maps(pid)

text_segment = list(filter(
    lambda x: x.map_name == f"{getcwd()}/hello_loop" and
x.permissions == 'r-xp', maps))[0]

instr_addr = text_segment['addr_start'] + instr_offset
```

Wycinek kodu 28. Obliczanie adresu dla punktu przerwania. Skrypt interception.py, fragment funkcji debugger() [19]

Następnie w pętli debuggera wywołujemy funkcję *modify()*, która jest w większości kopią funkcji *soft_bp()* z pliku *soft_deb.py* [19]. Zmiana polega na podmianie wartości rejestru RDI, w którym przekazywany jest jedyny argument do funkcji *printer()* (Wycinek 27).

```
# Pobierz aktualny stan RIP
ptrace(PTRACE_GETREGS, pid, 0, byref(regs))
print("RIP = 0x%x" % (regs.rip))
print("Orig. RDI = 0x%x : %d" % (regs.rdi, regs.rdi))

# Przywroc oryginalna instrukcje, cofnij wskaznik instrukcji (RIP)
ptrace(PTRACE_POKETEXT, pid, instr_addr, org_instr)
regs.rip -= 1
regs.rdi = randint(0, 100)
ptrace(PTRACE_SETREGS, pid, 0, byref(regs))
```

Wycinek kodu 29. Fragment funkcji modify() ze skryptu interception.py [19]

Po uruchomieniu programu *hello_loop* w osobnym terminalu, uruchamiamy skrypt (Wycinek 28)

```
$ sudo p3 interception.py `pidof hello_loop`
Proces otrzymal sygnal:  SIGTRAP
RIP = 0x5555555551aa
Orig. RDI = 0x3 : 3

Proces otrzymal sygnal:  SIGTRAP
RIP = 0x5555555551aa
Orig. RDI = 0x4 : 4
```

Wycinek kodu 30. Wyjście ze skryptu interception.py

Wyjście z programu *hello_loop* mogłoby wyglądać następująco (Wycinek 29).

```
$ ./hello_loop
Petla nr: 0
Petla nr: 1
Petla nr: 2
Petla nr: 69
Petla nr: 89
Petla nr: 67
(…)

Trace/breakpoint trap (core dumped)
```

Wycinek kodu 31. Wyjście programu hello_loop po podmianie wartości przekazywanej do funkcji

Jak widać, udało się wstrzyknąć losową wartość w argument przekazywany do funkcji *printer()*. Oryginalna wartość przekazywanego argumentu jest na bieżąco wyświetlana w skrypcie. Przykład ten pokazał w jak prosty sposób, przy pomocy debuggera i odpowiedniej wiedzy, można znaleźć rzeczywiste miejsce w pamięci w którym przebywa analizowany program. Na koniec należy dodać, że program zakończył swe działanie awaryjnie z uwagi zakończenie skryptu debuggera sygnałem SIGINT (CTRL+C), przez co prawdopodobnie wykonywanie skryptu zostało przerwane w trakcie zakładania kolejnego przerwania w pętli debuggera. Na potrzeby przykładu to wystarczy, natomiast żeby uniknąć awaryjnego wyjścia z analizowanego programu, należałoby tak napisać pętlę debuggera, aby po wciśnięciu CTRL+C skrypt usunął wyjątek z pamięci procesu, przywrócił rejestry do wartości oryginalnych, następnie odłączył się (*PTRACE_DETACH)* od analizowanego procesu. Wtedy dalsze wykonanie nastąpiłoby bez żadnych problemów.

4.3. Przechwytywanie wywołań (ang. Hooking)

Kolejną techniką którą umożliwia użycie debuggera jest przechwytywanie wywołań określonych funkcji wykonywanych przez analizowany proces. Zasada jest analogiczna do wstrzykiwania kodu, przedstawionego wyżej. Różnica polega na tym, że nie interesuje nas wykonanie własnego kodu w danym procesie, a przechwycenie (potencjalnie wrażliwych) danych na których dany proces operuje. Przykładowo, moglibyśmy uzyskać dostęp do niezaszyfrowanych danych logowania użytkownika poprzez przechwycenie wywołania funkcji *nspr4.PR_Write()* w przeglądarce internetowej Firefox. Funkcja ta przekazuje dane dalej do zaszyfrowania, więc jeśli udałoby się przechwycić te dane wcześniej, możemy uzyskać dostęp do potencjalnie wrażliwych informacji. Przykład ten został opisany

w [40]. Przechwytywanie tego rodzaju danych to nie jedyne zastosowanie tej techniki. Mogłaby ona także zostać użyta w celu np. ominięcia zabezpieczeń w programie wymagającym od użytkownika licencji, bądź uzyskania przewagi nad innymi graczami w grze komputerowej, odnajdując w kodzie funkcje które służą do modyfikacji atrybutów / zasobów danej postaci.

Podsumowując, należy zwrócić uwagę że użycie którejkolwiek z tych technik wymaga posiadania uprawnień *roota* w systemach linuksowych, bądź zdjęcia zabezpieczeń które zawsze domyślnie są włączone. Przedstawione techniki zatem ciężko jest wykorzystać w celu A w przypadku gdy atakujący uzyska takie uprawnienia, prawdopodobnie ostatnim zmartwieniem użytkownika będzie to czy uda mu się odczytać kilka haseł.

5. Podsumowanie

Celem tej pracy było poznanie mechanizmów działania tzw. blackbox debuggera, w celu ułatwienia, przyspieszenia, i wykorzystania w 100% możliwości debuggerów jako nieodłącznych narzędzi instrumentacji dynamicznej oprogramowania. Przedstawione zostały rodzaje zdarzeń debuggowania mogących wystąpić w trakcie wykonywania programu, które debugger jest w stanie przechwycić i poddać zautomatyzowanej analizie. Dotyczy to zarówno zdarzeń spowodowanych ustawieniem przerwania określonego rodzaju w danym miejscu pamięci, jak również nieumyślnych błędów twórców oprogramowania które prowadzą do awaryjnego przerwania wykonania programu, a także wszystkich innych zdarzeń które mogą się pojawić w trakcie działania procesu. Omówione techniki zostały porównane pod względem przydatności w określonych sytuacjach. Innym poruszonym tematem był także wpływ działania debuggera na szybkość wykonania analizowanego programu. Tematy te zostały podsumowane praktycznym wykorzystaniem omówionych technik, posługując się przechwytywaniem wywołań systemowych, bibliotecznych w analizowanym procesie, wstrzykiwania własnego kodu do przechwyconych funkcji, czy ukrywania działania debuggera w systemie a także, z drugiej strony, wykrywania go. W celu lepszego zrozumienia omawianych mechanizmów, jedne z pierwszych rozdziałów traktowały o podstawach architektury x86-64, omawiając podstawy budowy rejestrów, funkcjonowania pamięci, czy wykonywania kodu. Wszystko to miało na celu pozwolić czytelnikowi zrozumieć, jak debuggery funkcjonują, jakie oferują możliwości, a dzięki odpowiedniemu zrozumieniu mechanizmów leżących i ich podstaw, osiągnąć znacznie więcej i wygodniej, automatyzując cały proces instrumentacji.

Bibliografia

[1] C. Walls, “Debugging with printf() or not …,” 2013. [Online].

[2] R. Święcki, “10. Śledzenie ścieżki wykonania procesu w systemie Linux,” in *Praktyczna Inżynieria Wsteczna - Metody, Techniki, Narzędzia*, 2018.

[3] M. D. S. Panchal, “Program, Code Instrumentation using Dynamic,” *IJERT,* 2014.

[4] I. Zhirkow, “Basic Computer Architecture,” in *Low-Level Programming*, 2017.

[5] Intel, “Basic Program execution registers,” in *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2020.

[6] I. Zhirkov, “Interrupts and System Calls,” in *Low-Level Programming*, 2017.

[7] Intel, “17.2 Debug Registers,” in *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2020.

[8] T. Bukowski, “Funkcje, struktury, klasy i obiekty na niskim poziomie,” in *Praktyczna Inżynieria Wsteczna - metody, techniki i narzędzia*, 2018.

[9] E. Bendersky, “stack-frame-layout-on-x86-64,” 06 09 2011. [Online]. Available: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64.

[10] I. Zhirkov, Low-Level Programming, 2017.

[11] J. Seitz, “2. Debuggers and debugger design,” in *Gray Hat Python*, 2009.

[12] [Online]. Available: https://www.man7.org/linux/man-pages/man2/ptrace.2.html.

[13] P. Padala, “Playing with ptrace, Part I,” 2002. [Online]. Available: https://www.linuxjournal.com/article/6100.

[14] O. N. Leija, “Ptrace and You,” 2018. [Online]. Available: https://gatowololo.github.io/blog/ptraceintro.

[15] E. Bendersky, “How debuggers work: Part 1 - Basics,” 23 01 2011. [Online]. Available: https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1.

[16] J. Seitz, “Building a Windows Debugger,” in *Gray Hat Python*, 2009.

[17] https://man7.org/linux/man-pages/man2/fork.2.html.

[18] S. Mandal, “How to Analyze Malware Dynamically Using Cuckoo,” [Online].

[19] P. Samsel, “LinuxPYDebug,” 2020. [Online]. Available: https://github.com/PrzemyslawSamsel/LinuxPYDebug.

[20] NSA, “LIMITING PTRACE ON PRODUCTION LINUX SYSTEMS,” 2019.

[21] [Online]. Available: https://linux.die.net/man/1/objdump.

[22] [Online]. Available: stackoverflow.com/questions/33873779/reading-debug-registers-on-linux.

[23] [Online]. Available: stackoverflow.com/questions/2604439/how-do-i-write-x86-debug-registers-from-user-space-on-osx/2604449#2604449.

[24] Ling, “Hardware breakpoints and exceptions on Windows,” 2020. [Online]. Available: https://ling.re/hardware-breakpoints/#fn:1.

[25] Ling, “Hardware breakpoints and exceptions on Windows,” 2020. [Online]. Available: https://ling.re/hardware-breakpoints/.

[26] I. Zhirkov, “Virtual Memory,” in *Low-Level Programming*, 2017.

[27] R. Hass, 2018. [Online]. Available: https://perception-point.io/changing-memory-protection-in-an-arbitrary-process/.

[28] [Online]. Available: man7.org/linux/man-pages/man2/mprotect.2.html.

[29] [Online]. Available: https://reverseengineering.stackexchange.com/questions/19598/find-base-address-and-memory-size-of-program-debugged-in-gdb.

[30] [Online]. Available: https://overiq.com/python-101/exception-handling-in-python/ .

[31] [Online]. Available: https://stackoverflow.com/questions/3596781/how-to-detect-if-the-current-process-is-being-run-by-gdb.

[32] R. ". O'Neill, “Linux Process Tracing,” in *Learning Linux Binary Analysis*, 2016.

[33] J. Evans, “Profiler adventures: resolving symbol addresses is hard!,” 09 01 2018. [Online]. Available: https://jvns.ca/blog/2018/01/09/resolving-symbol-addresses/ .

[34] [Online]. Available: https://linux.die.net/man/2/process_vm_writev.

[35] ancat, “Injecting Code in Running Processes with Python and Ptrace,” 01 01 2019. [Online]. Available: https://ancat.github.io/python/2019/01/01/python-ptrace.html.

[36] [Online]. Available: https://pypi.org/project/pyelftools/.

[37] T. Schoranov, “Finding Function's Load Address,” 02 04 2016. [Online]. Available: https://uaf.io/exploitation/misc/2016/04/02/Finding-Functions.html.

[38] [Online]. Available: https://github.com/eklitzke/parse-elf.

[39] T. Kwiecień, “Mechanizmy Ochrony Aplikacji,” in *Praktyczna Inżynieria Wsteczna - metody, techniki, narzędzia*, 2018.

[40] J. Seitz, “Hooking,” in *Gray Hat Python*, 2009.