Wyjątki od kuchni

Wyjątki w C++ od strony programisty wyglądają stosunkowo prosto - ot zamykamy fragment kodu w bloku `try`, definiujemy jeden lub kilka bloków `catch` przechwytujących wyjątki danego typu i na tym nasza robota skończona; od teraz jeśli podczas wykonywania naszego kodu zostanie użyte wyrażenie `throw`, wykonanie programu zostanie "magicznie" przeniesione do odpowiedniego bloku *catch* (jeśli taki istnieje). Patrząc od środka cały proces jest dużo bardziej złożony i warto na niego rzucić okiem, choćby z ciekawości.

Wyjątki C++

Niejako oczywistym stwierdzeniem jest, że język C++ wyposażony jest w mechanizm wyjątków. Konkretniej, w wyjątkowych sytuacjach kod może "rzucić wyjątek" (ang. *throw an exception*), czyli przekazać pewien obiekt (zazwyczaj opisujący ową wyjątkową sytuację) w górę stosu wywołań do procedury przystosowanej do jego obsługi (czyli bloku *catch*). Programista może zdecydować również nie obsługiwać wyjątków danej klasy, co w wypadku zaistnienia takowego prowadzi do zamknięcia procesu z błędem (ang. *abort*) lub innej akcji zdefiniowanej przez programistę przy pomocy funkcji `std::set_terminate` (domyślnie `abort`). Przykładowy kod używający wyjątków, do którego będziemy odnosić się w całym niniejszym artykule, znajduje się poniżej:

```
#include <cstdio>
#include <climits>
#include <exception>
#include <stdexcept>

void get_input(int *a, int *b) {
  printf("Podaj a oraz b: ");
  int ret = scanf("%i %i", a, b);
  if (ret != 2) {
    throw std::invalid_argument("zle wejscie");
  }
}

int do_division(int a, int b ) {
  if (b == 0) {
    throw std::invalid_argument("b == 0");
  }

  if (a == INT_MIN && b == -1) {
    throw std::range_error("INT_MIN/-1 > INT_MAX");
  }
```

```
  return a / b;
}

int main() {
  int a, b, c;
  try {
    get_input(&a, &b);
    c = do_division(a, b);

    printf("Wynik: %i\n", c);
  } catch(std::invalid_argument& ex) {
    printf("Nieprawidlowy argument: %s\n",
           ex.what());
    return 1;
  } catch(std::range_error& ex) {
    printf("Blad zakresu: %s\n",
           ex.what());
    return 2;
  }
}
```

Istotnymi cechami wyjątków w C++ (z punktu widzenia niniejszego artykułu) są:

- **Możliwość rzucenia wyjątku dowolnego typu**. W niektórych innych językach rzucony musi zostać obiekt konkretnego typu lub dziedziczący po konkretnej klasie; przykładowo w języku Python 3 rzucić można jedynie obiekt klasy dziedziczącej po klasie BaseException [4].
- **Możliwość zdefiniowania bloku *catch* dla konkretnego typu**. Niektóre inne mechanizmy wyjątków przekazują procedurze obsługi wyjątków informacje o każdym rzuconym wyjątku, niezależnie od jego typu; dopiero w kodzie takiej procedury następuje decyzja czy wyjątek zostanie obsłużony, czy też należy kontynuować poszukiwania procedury, które go w pełni obsłuży. Przykładem może być SEH (*Structured Exception Handling*) dostępny w kompilatorach z rodziny Microsoft Visual C++ jako rozszerzenie do języków C oraz C++ [1][2][3].
- **Możliwość przekazania wyjątku dalej**. Służy do tego wyrażenie `throw` bez parametru, które powoduje kontynuację poszukiwań kolejnego bloku *catch* potrafiącego obsłużyć dany wyjątek (a technicznie: ponowne rzucenie tego samego wyjątku bez potrzeby jego re-definiowania).
- **Brak możliwości wznowienia wykonania w miejscu rzucenia wyjątku**. Niektóre inne modele wyjątków oferują możliwość wznowienia wykonania, po obsłużeniu wyjątku, od miejsca w którym wyjątek został rzucony. Przykładem mogą być wyjątki wbudowane w system Windows; np. procedura obsługi nieobsłużonych wyjątków (ustawiona za

pomocą funkcji `SetUnhandledExceptionFilter`) umożliwia kontynuację wykonania poprzez zwrócenie wartości `EXCEPTION_CONTINUE_EXECUTION` [5].

Warto dodać, że implementacja obsługi wyjątków C++ może, ale nie musi wspierać się niskopoziomowymi mechanizmami wyjątków udostępnianymi przez system operacyjny (lub w szczególnych przypadkach - przez sam procesor). Zasadniczo wynika to z faktu, iż opisywany model wyjątków różni się od modeli wyjątków poszczególnych innych systemów wyjątków typu Windowsowego SEH/VEH, sygnałów (ang. *signal*) znanych z systemów UNIXo-podobnych, czy przerwań/pułapek (ang. *interrupt*/*trap*) obecnych na poziomie procesora.

Implementacja

W zależności od kompilatora (w niniejszym artykule pisząc "kompilator" mamy na myśli również towarzyszące mu środowisko uruchomieniowe), systemu operacyjnego oraz architektury używane są różne implementacje wyjątków C++. Wymienić można dwie główne kategorie:

- **Implementacje aktywne**, których działanie opiera się o aktywne zapamiętanie stanu procesora w momencie wejścia do bloku *try*. Przykładem może być SJLJ (ang. *Set Jump Long Jump*) - mechanizm oparty o znane z języka C funkcje `setjmp` i `longjmp` [6].
- Oraz tzw. systemy **z zerowym kosztem**, których działanie opiera się o dodatkowe metadane pozwalające w ograniczonym zakresie odtworzyć stan jednostki wykonującej (zazwyczaj procesora i pamięci) z dowolnego wcześniejszego miejsca stosu wywołań (ang. *callstack*).

Warto dodać, że różne mechanizmy zazwyczaj nie są ze sobą kompatybilne, stąd też autorzy bibliotek wykorzystujących wyjątki C++, dostarczanych w formie skompilowanej, są zmuszeni do dystrybucji kilku ich wersji. Przykładem może być biblioteka SFML w wersji 2.1 (http://sfml-dev.org/download/sfml/2.1/) dostępna m.in. w wersjach SLJL, DW2, oraz innych.

DW2

Przykładem implementacji z zerowym kosztem, na którym się skupimy w niniejszym artykule, jest mechanizm używany przez kompilatory z rodziny GCC, potocznie zwany DWARF-2 EH (lub w skrócie DW2), używany do kompilacji aplikacji działających pod kontrolą systemów opartych o jądro Linux (choć nie tylko). DW2 nie jest jednak prawidłową nazwą tego mechanizmu, która według niektórych źródeł [7] powinna brzmieć raczej "obsługa wyjątków C++ korzystająca z DWARF Call Frame Information z rozszerzeniami", jednak nazwa DW2 jest na tyle rozpopularyzowana, że i my z niej będziemy korzystać.

Warto dodać, że system ten oparty jest o specyfikację "Itanium C++ ABI for IA-64: Exception Handling" [8], oczywiście odpowiednio dostosowaną do innych platform (np. x86-64 której użyjemy jako przykładu).

Z punktu widzenia wysokopoziomowego DW2 składa się z następujących kompontentów:

- W środowisku uruchomieniowym (ang. *C++ runtime*):
  - C++ ABI pozwalające rzucać wyjątki oraz je obsługiwać (dokładny opis ABI wykracza jednak poza zakres niniejszego artykułu).
  - Niezależna od języka implementacja systemu propagacji wyjątków i wyszukiwania procedur obsługujących dany wyjątek, zawierająca m.in. wirtualną maszynę (sic!) wykonującą DWARF *bytecode*.
  - A także tzw. *personality routine*, czyli procedurę która rozumie potrzeby systemu wyjątków C++.
- W danym pliku wykonywalnym lub bibliotece dynamicznej:
  - Metadane i *bytecode* pozwalające przywrócić częściowy stan procesora (oraz opcjonalnie pamięci) z wcześniejszych ramek stosu wywołań (w języku angielskim czynność ta określana jest pojedynczym słowem *unwind*).
  - Spis bloków *try*, *catch* oraz typów przez nie obsługiwanych.

W dalszej części artykułu prześledzimy integrację każdego z powyższych komponentów z przykładowym programem zaprezentowanym na początku artykułu.

Adresy w obrazie

Ponieważ opisujemy wszystko z niskopoziomowego punktu widzenia warto wcześniej wskazać kilka ważnych miejsc w naszym programie z perspektywy pliku wykonywalnego (można go pobrać spod adresu http://gynvael.coldwind.pl/download.php?f=dw2example.zip), tak aby czytelnik mógł odnieść adresy pojawiające się w dalszej części artykułu do odpowiednich fragmentów obrazu wykonywalnego. W tym miejscu pominiemy wyjaśnienia dlaczego dane adresy są potrzebne - wyniknie to z kontekstu w dalszej części artykułu. Dla ułatwienia wszystke adresy w niniejszym artykule zapisane są w systemie szesnastkowym, chyba, że zaznaczone jest inaczej.

Adresy funkcji (początek, koniec):

- **`main:`** `400ED5 - 400FEC`
- **`get_input:`** `400C7D - 400D69`
- **`do_division:`** `400D69 - 400ED5`

Adresy sekcji (początek):

- **`.eh_frame_hdr:`** `4010F8`
- **`.eh_frame:`** `401140`
- **`.gcc_except_table:`** `4012AC`
- **`.bss:`** `6020C0`

Generowanie tablicy *unwind*

W obrazie pliku wykonywalnego sekcje *.eh_frame* oraz *.eh_frame_hdr* zawierają metadane, które pozwalają na doprowadzenie istotnej części stanu procesora do takiej postaci, jak gdyby nigdy nie doszło do rozpoczęcia wykonania funkcji, w której obecnie znajduje się jednostka wykonująca. Posiadając takie informacje dla każdej funkcji możliwe jest odzyskanie stanu, idąc w górę stosu wywołań, aż do funkcji leżącej na samej górze stosu (z punktu widzenia języka C++ zazwyczaj będzie to funkcja `main`, ale w niektórych przypadkach może to być również konstruktor lub destruktor obiektu globalnego lub statycznego, a także jedna z innych funkcji wykonywanych przed lub po `main`).

Oczywistym pytaniem jest "dlaczego potrzebna jest możliwość odzyskania stanu?". Wynika to przede wszystkim z faktu, iż blok *catch* obsługujący dany wyjątek może znajdować się w funkcji nadrzędnej, w związku z czym konieczne jest:

- uzyskanie informacji która z funkcji wywołała obecnie przetwarzaną funkcję - oczywiście sprowadza się to do pobrania adresu powrotu (ze stosu lub odpowiedniego rejestru w przypadku niektórych innych architektur); aby móc to zrobić należy wiedzieć gdzie konkretnie na stosie znajduje się adres powrotu (warto przypomnieć, że w przypadku niekorzystania ze wskaźnika na ramkę stosu - opcja *-fomit-frame-pointer* - odpowiedź na to pytanie nie jest trywialna),
- oraz przywrócenie istotnej części stanu procesora (tj. wszystkich istotnych rejestrów), tak aby możliwe było kontynuowanie wykonania wcześniejszej funkcji, która oczywiście ma prawo spodziewać się pewnych konkretnych wartości w danych rejestrach procesora.

Metadane dla wszystkich funkcji zapisane są w formacie DWARFv2 [9] (konkretniej, wykorzystane są struktury *Common Information Entry* oraz *Frame Descriptor Entry*). Szczegółowa analiza budowy tych struktur na poziomie bitowym wykracza poza zakres niniejszego artykułu, dlatego skupimy się na ich wysokopoziomowej analizie.

Common Information Entry

Narzędzie *readelf* (wywołane z parametrem *-w*) pozwala wypisać CIE oraz FDE w formie tekstowej. Poniżej znajduje się CIE oraz FDE funkcji do_division, a także fragment tejże funkcji w asemblerze (*objdump -S dw2example -M intel*), wraz z objaśnieniami.

```
00000070 000000000000001c 00000000 CIE
  Version:               1
  Augmentation:          "zPLR"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 16
```

```
Augmentation data:     03 60 0b 40 00 03 1b

DW_CFA_def_cfa: r7 (rsp) ofs 8
DW_CFA_offset: r16 (rip) at cfa-8
DW_CFA_nop
DW_CFA_nop
```

CIE to niejako nadrzędne informacje, które mogą być wspólne dla wielu FDE. W tym wypadku interesują nas dwie rzeczy: zaznaczone na czerwono *Augmentation data*, oraz sam *bytecode* (w postaci zdisasemblowanej) definiujący początkowe warunki w tabeli *unwind* (zdefiniowanej w FDE).

Augmentation data (dalej AD) należy interpretować w kolejności zgodnej z tagami określonymi w polu *Augmentation* - czyli "zPLR" [10]. W tym wypadku będzie to kolejno:

- "z" oznacza, że AD jest niepuste.
- "P" mówi, że kolejne bajty w Augmentation data należy odczytać jako kodowanie adresu oraz sam adres *personality routine*; w tym wypadku adresem będzie 400B60. Zaglądając pod ten adres w disasemblerze znajdziemy następujący kod:
  ```
  .plt:400B60 ___gxx_personality_v0:
  .plt:400B60   jmp     cs:off_602098
  ```
  Jest to oczywiście skok do *personality routine* języka C++ w środowisku uruchomieniowym kompilatora GCC.
- "L" mówi, że w AD FDE zależnych od tego CIE będzie występować adres LSDA (*Language Specific Data Area*, który jest opisany w dalszej części artykułu), oraz o kodowaniu tego adresu.
- "R" określa kodowanie pozostałych adresów używanych w FDE.

Przejdźmy do samego skryptu (dokładny opis instrukcji można znaleźć w specyfikacji DWARF [9]):

- **DW_CFA_def_cfa** definiuje adres CFA (ang. *Canonical Frame Address*), czyli po prostu relatywny adres ramki stosu, względem którego definiowane są wszystkie kolejne informacje. W tym wypadku jest on określony na RSP+8 (czyli adres który znajduje się w danym momencie w rejestrze stosu plus 8).
- **DW_CFA_offset** definiuje gdzie na stosie, względem CFA, można znaleźć zapamiętaną wartość danego rejestru. W tym wypadku definiowane jest, że rejestr RIP (czyli adres powrotu z funkcji) został zapamiętany pod adresem CFA-8.
- **DW_CFA_nop** to po prostu *nop* (ang. *no operation*), czyli instrukcja, która nic nie robi i zazwyczaj służy jako dopełnienie.

Po wykonaniu powyższego *bytecode* otrzymujemy zastane warunki w momencie rozpoczęcia wykonywania skryptu z FDE; w tym wypadku wiemy, że CFA jest równe RSP+8, a adres

powrotu leży pod adresem CFA-8 (czyli de facto RSP na niego wskazuje, co oczywiście zgadza się z typowym zastanym stosem w momencie wejścia do funkcji na opisywanej platformie).

Frame Descriptor Entry

Budowa FDE jest niemal identyczna jak budowa CIE (z wyjątkiem braku większości nagłówków, z których pozostał jedynie AD).

```
000000b8 0000000000000024 0000004c FDE cie=00000070
pc=0000000000400d69..0000000000400ed5
  Augmentation data:     c9 12 40 00

  DW_CFA_advance_loc: 1 to 0000000000400d6a
  DW_CFA_def_cfa_offset: 16
  DW_CFA_offset: r6 (rbp) at cfa-16
  DW_CFA_advance_loc: 3 to 0000000000400d6d
  DW_CFA_def_cfa_register: r6 (rbp)
  DW_CFA_advance_loc: 7 to 0000000000400d74
  DW_CFA_offset: r12 (r12) at cfa-24
  DW_CFA_offset: r3 (rbx) at cfa-32
  DW_CFA_advance_loc2: 352 to 0000000000400ed4
  DW_CFA_def_cfa: r7 (rsp) ofs 8
```

O tym jak interpretować AD dowiedzieliśmy się analizując pole Augmentation w CIE - konkretniej, AD FDE zawiera adres LSDA dla danej funkcji (zazwyczaj jedno FDE przypada na jedną funkcję). W tym wypadku adresem tym jest 4012C9, który zawiera się w sekcji *.gcc_except_table*, do której przejdziemy później.

Jak pisaliśmy wcześniej, celem samego skryptu jest wygenerowanie tabeli, której indeksem będzie adres w funkcji, a która będzie zawierać informacje jak można przywrócić istotną część stanu z dowolnego miejsca w danej funkcji do stanu identycznego sprzed wejściem do funkcji. Analizując powyższy kod warto na bieżąco porównywać go z fragmentem kodu assembly rozważanej funkcji (`do_division`) znajdującym się poniżej (jako że analizowany skrypt niejako opisuje widoczne efekty działania samego kodu względem istotnych elementów stanu procesora), oraz samą wygenerowaną tablicą (patrz Tabela 1):

```
int do_division(int a, int b ) {
  400d69:  push   rbp
  400d6a:  mov    rbp,rsp
  400d6d:  push   r12
  400d6f:  push   rbx
  400d70:  sub    rsp,0x30
  400d74:  mov    DWORD PTR [rbp-0x34],edi
```

```
400d77:  mov    DWORD PTR [rbp-0x38],esi
if (b == 0) {
400d7a:  cmp    DWORD PTR [rbp-0x38],0x0
400d7e:  jne    400de7 <_Z11do_divisionii+0x7e>
  throw std::invalid_argument("b == 0");
400d80:  mov    edi,0x10
[...]
```

<<Tabela 1. Wygenerowana tablica *unwind* dla funkcji do_division.>>

PC	CFA	RBX	RBP	R12	RIP
400d69	RSP+8	?	?	?	CFA-8
400d6a ... 400d6c	RSP+16	?	CFA-16	?	CFA-8
400d6d ... 400d73	RBP	?	CFA-16	?	CFA-8
400d74 ... 400ed3	RBP	CFA-32	CFA-16	CFA-24	CFA-8
400ed4	RSP+8	CFA-32	CFA-16	CFA-24	CFA-8

Wykonanie powyższego skryptu będzie przebiegać następująco:

- PC (ang. *Program Counter*), który w tym wypadku nie jest rejestrem wskazującym na instrukcję do wykonania, a rejestrem wskazującym na dane miejsce w rozważanej funkcji, będzie na początku ustawiony na 400D69. Oznacza to również, że tabela będzie rozpoczęta od tego adresu.
- **DW_CFA_advance_loc: 1** powoduje przesunięcie indeksu (PC) o konkretną ilość bajtów do przodu - w tym wypadku przesunięcie następuje o jeden bajt. Oznacza to dwie rzeczy:
    - Po pierwsze, wszystkie wiersze tabeli od poprzedniej lokacji do nowej (nie wliczając w to nowej lokacji) zostaną wypełnione informacjami, które są obecnie znane. Na samym początku skryptu będą to oczywiście informacje zdefiniowane w CIE.
    - Po drugie, wszystkie późniejsze zmiany znanych informacji nie będą dotyczyć już wypełnionych wierszy tabeli.

- **DW_CFA_def_cfa_offset: 16** działa podobnie do opisanego wcześniej DW_CFA_def_cfa, przy czym nie zmienia całej definicji CFA, a jedynie offset przy rejestrze; tak więc przed wykonaniem tej instrukcji obecna informacja o CFA brzmiała "CFA znajduje się pod RSA+8", a po jej wykonaniu brzmi "CFA znajduje się pod RSA+16". Jest to oczywiście wynikiem przesunięcia rejestru RSP w wyniku wykonania instrukcji push znajdującej się w programie pod adresem 400D69.
- **DW_CFA_offset: r6 (rbp) at cfa-16** opisywaliśmy już wcześniej; w tym wypadku otrzymujemy informację, że w rozważanym momencie funkcji (dla przypomnienia, obecnie PC wynosi 400D6A) oryginalną wartość rejestru RBP można znaleźć na stosie pod adresem CFA-16.
- **DW_CFA_advance_loc: 3** powoduje wygenerowanie trzech kolejnych wierszy tabeli oraz przesunięcie rozważanego PC o 3 bajty.
- **DW_CFA_def_cfa_register: r6 (rbp)** przedefiniowuje CFA z RSP+16 na RBP.
- **DW_CFA_advance_loc: 7** powoduje wygenerowanie kolejnych wierszy tabeli oraz przesunięcie PC.
- **DW_CFA_offset: r12 (r12) at cfa-24** informuje, że oryginalna zawartość rejestru R12 została zachowana pod adresem CFA-24.
- **DW_CFA_offset: r3 (rbx) at cfa-32** informuje, że RBX został zapisany pod CFA-32.
- **DW_CFA_advance_loc2: 352** generuje znaczną ilość wierszy i przesuwa PC na przedostatnią instrukcję funkcji.
- **DW_CFA_def_cfa: r7 (rsp) ofs 8** z powrotem przedefinowuje CFA na RSP+8.
- W tym momencie skrypt się zakończył, a ostatni stan zostaje rozpropagowany od PC do końca funkcji (w tym wypadku będzie to tylko jeden wiersz).

Jak pisaliśmy wcześniej, bazując na wygenerowanej tabeli możemy w dowolnym momencie funkcji przywrócić wartości wszystkich istotnych rejestrów, w tym adres powrotu z funkcji.

Przykładowo, jeśli byśmy chcieli przywrócić stan gdy RIP (PC) jest równe 400E00, to bazując na Tabeli 1 wiemy że:

- Oryginalna wartość rejestru RBX jest pod adresem RBP-32.
- Oryginalna wartość rejestru RBP jest pod adresem RBP-16.
- Oryginalna wartość rejestru R12 jest pod adresem RBP-24.
- Oraz adres powrotu do funkcji nadrzędnej (RIP) jest pod adresem RBP-8.

Dzięki temu możemy przywrócić stan do momentu sprzed wywołania funkcji `do_division` (jeśli chodzi o rejestr RIP, to w tym wypadku należałoby odjąć od jego wartości jeszcze długość instrukcji call, która spowodowała wywołanie funkcji, aby uzyskać faktyczną wartość sprzed wywołania funkcji).

Oczywistym pytaniem w tym miejscu jest "czemu tylko niektóre rejestry są przywracane?". Wynika to z dwóch rzeczy:

- Po pierwsze, nie wszystkie wartości rejestrów zostały w danym momencie nadpisane.
- Po drugie, funkcja nadrzędna musi założyć, że funkcja wywoływana będzie używać niektórych rejestrów jak brudnopisu (ang. *scratch registers*) bez zachowania ich oryginalnej wartości. Skoro funkcja nadrzędna o tym wie i jej kod jest tak wygenerowany, by przed wywołaniem funkcji nie było w danych rejestrach żadnych istotnych informacji, to nie trzeba ich przywracać. Listę *scratch registers* dla różnych kompilatorów można znaleźć w [11].

LSDA

Sekcja *.gcc_except_table* zawiera LSDA (informacje specyficzne dla języka C++), która w przypadku C++ i kompilatora z rodziny GCC interpretowana jest przez *personality routine* zdefiniowanym w libstdc++-v3/libsupc++/eh_personality.cc, który można znaleźć w źródłach GCC. LSDA zawiera informacje o blokach *catch* (a raczej grupach bloków *catch*) oraz obsługiwanych typach wyjątków.

Każda funkcja posiada jeden wpis LSDA, który dzieli się na trzy części (pomijając nagłówek):

1. Tablicy *try*/*catch*, nazywanej *Call Site Table*.
2. Listy akcji (jest to de facto lista obsługiwanych typów).
3. Oraz tablicy typów (używanej w liście akcji).

Tablica *try*/*catch* zawiera informacje o:

- Adresie początkowym oraz końcowym bloku *try*.
- Adresie tzw. *landing pad* - jest to adres procedury dyspozytora obsługującego daną grupę bloków *catch*.
- Oraz offsecie pierwszej akcji w tablicy akcji, o ile jakakolwiek jest zdefiniowana.

Lista akcji zawiera informacje o typach obsługiwanych przez dany blok *catch*, a konkretniej indeksy typów w tablicy typów. Natomiast lista typów zawiera adresy obiektów `std::type_info` opisujących rodzaje wyjątków obsługiwanych przez bloki *catch*.

Dla przykładu, spójrzmy na LSDA funkcji `main`, który znajduje się pod adresem 4012F8. Mając do dyspozycji kod źródłowy aplikacji można skompilować go używając polecenie *g++ plik.cpp -g --save-temps -fverbose-asm -dA -masm=intel* - stworzy to plik *plik.s*, w którym znajdziemy m.in. informacje o LSDA (zazwyczaj generowane po danej funkcji). W innym wypadku trudno jest wskazać dobre narzędzie do analizy sekcji *.gcc_except_table* (wyjątkiem jest katana - http://katana.nongnu.org/, która jednak nie zawsze działa poprawnie), a sam format również nie jest dobrze udokumentowany; oczywiście możliwe jest stworzenie skryptu na podstawie źródła *personality routine* (co jednak wykracza poza zakres niniejszego artykułu).

Lista bloków *try*/*catch* dla funkcji `main` wygląda następująco:

```
12fd: callsite table start (len=12, total_len=21, next=131c)
12fd: start=400eec, len=2b, pad=400f49, action=3
1301: start=400f31, len=2c, pad=400ed5, action=0
1305: start=40100d, len=5,  pad=400f23, action=0
130a: start=401051, len=5,  pad=400f36, action=0
130f: end of callsite
```

Jak widać jedynie pierwszy wpis ma zdefiniowaną akcję. Adres bloku *try* to od 400EEC do 400F16 (włącznie) - warto porównać go z fragmentem kodu wynikowego funkcji `main`:

```
 try {
    get_input(&a, &b);
  400ede:  lea    rdx,[rbp-0x28]
  400ee2:  lea    rax,[rbp-0x2c]
  400ee6:  mov    rsi,rdx
  400ee9:  mov    rdi,rax
  400eec:  call   400c7d <_Z9get_inputPiS_>
    c = do_division(a, b);
  400ef1:  mov    edx,DWORD PTR [rbp-0x28]
  400ef4:  mov    eax,DWORD PTR [rbp-0x2c]
  400ef7:  mov    esi,edx
  400ef9:  mov    edi,eax
  400efb:  call   400d69 <_Z11do_divisionii>
  400f00:  mov    DWORD PTR [rbp-0x24],eax
    printf("Wynik: %i\n", c);
  400f03:  mov    eax,DWORD PTR [rbp-0x24]
  400f06:  mov    esi,eax
  400f08:  mov    edi,0x4010b7
  400f0d:  mov    eax,0x0
  400f12:  call   400a60 <printf@plt>
  } catch(std::range_error& ex) {
    printf("Blad zakresu: %s\n",
           ex.what());
    return 2;
  }
}
  400f17:  mov    ebx,0x0
  [...]
```

Część kodu wynikowego zaznaczonego na żółto jest objęta rozważanym wpisem. Jak widać, kilka pierwszych instrukcji wchodzących w skład wywołania `get_input` (tj. 2x lea + 2x mov) nie zostało ujęte w bloku *try*, prawdopodobnie stało się tak ponieważ kompilator wiedział, iż nie jest możliwe rzucenie wyjątku w tamtym miejscu kodu.

Pełna lista akcji dla rozważanego LSDA wygląda następująco:

```
40130F  ; Początek listy akcji
40130F  db    2  ; Type = 2
401310  db    0  ; Next = NULL
401311  db    1  ; Type = 1
401312  db  7Dh  ; Next = -3 (40130F)
```

Dla rozważanego *call site* lista akcji rozpoczyna się pod offsetem 2 (od wartości widocznej przy *action* należy odjąć jeden; wynika to z faktu iż 0 ma specjalne znaczenie - brak akcji) w liście zdefiniowanych akcji - offset 2 w tym wypadku tłumaczy się na adres 40130F+2 → 401311, czyli typ wyjątku sprawdzany jest z typem 1 (czyli indeks typ-1→0 z tablicy typów). Dalej następuje przejście do adresu 401311-3→40130F i sprawdzenie typu 2 (czyli typu spod indeksu 1 z tablicy typów). Pewnym zaskoczeniem może być tutaj bajt 7D tłumaczony na -3 - wynika to z używanego w tym miejscu kodowania *sleb128* (*signed little endian base 128*), które jednak wykracza poza zakres niniejszego artykułu (czytelników zainteresowanych tym kodowaniem zachęcam do rzucenia okiem np. na implementację funkcji `read_sleb128` w libstdc++ kompilatora GCC). Na tym lista akcji się kończy, ponieważ pole *Next* obecnego wskazuje na NULL.

Tablica typów zawiera jedynie adresy obiektów `std::type_info`. W tym wypadku wygląda ona następująco:

```
401314  dd _ZTISt11range_error@@GLIBCXX_3.4 ; 0x6020C0
401318  dd _ZTISt16invalid_argument@@GLIBCXX_3_4 ; 0x6020E0
```

Dyspozytor

Powstaje pytanie - dlaczego wg. powyższej listy *call site* oba typy obsługuje jeden *landing pad*? Wynika to z faktu, iż landing pad otrzymuje w jednym z rejestrów (RDX) informacje o typie wyjątku (ang. *handler switch value*), dzięki czemu może przekazać wykonanie do odpowiedniego bloku *catch* (dodatkowo w rejestrze RAX znajduje się adres wewnętrznej struktury opisującej wyjątek - `Unwind_Exception`)

W rozważanym wypadku *landing pad* wygląda następująco:

```
400f49:  cmp    rdx,0x1
400f4d:  je     400f5d <main+0x88>
400f4f:  cmp    rdx,0x2
400f53:  je     400fa1 <main+0xcc>
         [...]
```

Z powyższego fragmentu można wywnioskować, iż blok *catch* obsługujący wyjątki typu `std::range_error` znajduje się pod adresem 400F5D, a blok obsługujący `std::invalid_argument` pod adresem 400FA1.

Blok *catch*

Dla przykładu spójrzmy na kod bloku *catch* obsługującego wyjątek `std::invalid_argument`:

```
} catch(std::invalid_argument& ex) {
 400f5d:  mov    rdi,rax
 400f60:  call   400b30 <__cxa_begin_catch@plt>
 400f65:  mov    QWORD PTR [rbp-0x20],rax
   printf("Nieprawidlowy argument: %s\n",
          ex.what());
 400f69:  mov    rax,QWORD PTR [rbp-0x20]
 400f6d:  mov    rax,QWORD PTR [rax]
 400f70:  add    rax,0x10
 400f74:  mov    rax,QWORD PTR [rax]
 400f77:  mov    rdx,QWORD PTR [rbp-0x20]
 400f7b:  mov    rdi,rdx
 400f7e:  call   rax
 400f80:  mov    rsi,rax
 400f83:  mov    edi,0x4010c2
 400f88:  mov    eax,0x0
 400f8d:  call   400a60 <printf@plt>
 400f92:  mov    ebx,0x1
 400f97:  call   400b20 <__cxa_end_catch@plt>
 400f9c:  jmp    400f1c <main+0x47>
} catch(std::range_error& ex) {
```

Powyższy kod można podzielić na kilka części:

- 400F5D-400F65 - początek bloku *catch* i wywołanie `__cxa_beign_catch`, którego celem m.in. jest wydobycie adresu obiektu opisującego wyjątek.
- 400F69-400F8D - skompilowany kod programisty.
- 400F92-400F97 - zasygnalizowanie zakończenia bloku *catch* (powoduje to również zwolnienie wszystkich struktur związanych z już w tym momencie obsłużonym wyjątkiem).
- 400F9C - skok do dalszej części funkcji `main` (czyli ominięcie pozostałych bloków *catch*).

Analiza funkcji `__cxa_begin_catch` oraz `__cxa_end_catch` wykracza poza zakres niniejszego artykułu. Czytelnicy mogą jednak znaleźć ich implementacje w pliku *libstdc++-v3/libsupc++/eh_catch.cc* w źródłach GCC.

Rzucenie wyjątku

Dla pełni obrazu brakuje nam kodu rzucającego wyjątek. Dla przykładu przeanalizujmy kod rzucający `std::range_error`. Oryginał wygląda następująco:

```
if (a == INT_MIN && b == -1) {
  throw std::range_error("INT_MIN/-1 > INT_MAX");
}
```

Po kompilacji i dekompilacji do postaci kodu C, dokonanej za pomocą Hex-Rays, otrzymujemy natomiast poniższy kod:

```
if ( a == 0x80000000 && b == -1 )
{
  v5 = __cxa_allocate_exception(16LL);
  std::allocator<char>::allocator(&v7);
  std::string::string(&v8, 0x4010A2LL, &v7);
  std::range_error::range_error(v5, (const std::string *)&v8);
  std::string::~string((std::string *)&v8);
  std::allocator<char>::~allocator(&v7, &v8);
  __cxa_throw(v5, 0x6020C0LL, 0x400A90LL);
}
```

Większość powyższego kodu to inicjalizacja obiektu `std::range_error` (w tym alokacja i późniejsze zwolnienie `std::string` będącego jego parametrem), ale oprócz tego znajdują się tam dwie istotne rzeczy:

- Wywołanie `__cxa_allocate_exception` do alokacji pamięci dla obiektu opisującego wyjątek (zamiast standardowego `malloc`). Wewnętrznie `__cxa_allocate_exception` i tak sprowadza się do wywołania `malloc`, ale oprócz żądanej ilości pamięci, alokuje również dodatkowe bajty przed obiektem, w których przechowywane będą wewnętrzne informacje mechanizmu wyjątków.
- Samo rzucenie wyjątku, czyli wywołanie `__cxa_throw` z trzema parametrami:
  - Adresem obiektu opisującego wyjątek.
  - Adresem obiektu `std::type_info` opisującego typ `std::range_error` (wcześniej w artykule adres 6020C0 reprezentowaliśmy jako symbol _ZTISt11range_error@@GLIBCXX_3.4).

- Oraz adres 400A90 - jest to adres trampoliny wywołującej `std::range_error::~range_error`, czyli destruktor dla obiektu opisującego wyjątek.

Oczywiście wywołanie `__cxa_throw` wprowadza cały mechanizm w ruch (w tym powoduje wywołanie `_Unwind_RaiseException`, czyli rzucenie wyjątku niezależnego od języka programowania).

Cykl życia wyjątku

Mając opisane powyższe mechanizmy można ostatecznie przedstawić cały cykl życia wyjątku. Dla przykładu rozważymy rzucenie wyjątku `std::invalid_argument` w funkcji `do_division`.

1. Następuje alokacja i rzucenie wyjątku (`400DE2: call __cxa_throw`; adres powrotu jest ustawiony na 400DE7) o typie `std::invalid_argument` opisanym obiektem `std::type_info _ZTISt16invalid_argument@@GLIBCXX_3_4`.
2. Funkcja `__cxa_throw` inicjalizuje dodatkowe informacje przekazywane z obiektem wyjątku i ostatecznie wywołuje `_Unwind_RaiseException`.
3. `_Unwind_RaiseException` pobiera adres powrotu ze stosu (czyli *de facto* adresu wskazującego wewnątrz funkcji `__cxa_throw`) i (w pewnym uproszczeniu) rozpoczyna proces poszukiwania bloku *catch*.
4. Po pierwsze sprawdzana jest tablica *call site* dla funkcji `__cxa_throw` - oczywiście ta funkcja nie zawiera żadnych bloków *catch*.
5. Na podstawie CIE+FDE funkcji `__cxa_throw` następuje więc przywrócenie (wirtualnego) stanu procesora do momentu sprzed wywołania funkcji `__cxa_throw`, czyli rozważane PC zostaje ostatecznie ustawione na 400DE7 (wewnątrz funkcji `do_division`).
6. Ponownie, tablica *call site* dla funkcji `do_division` jest pusta (w tej funkcji nie ma żadnych bloków *catch*).
7. Na podstawie CIE+FDE zostaje wygenerowana tablica *unwind* (patrz Tabela 1). Dla adresu 400DE7 (czwarty wiersz we wspominanej tabeli) operacjami potrzebnymi do przywrócenia (wirtualnego) stanu procesora do stanu sprzed wywołania funkcji `do_division` są:
   RBX ← *(RBP-32)
   R12 ← *(RBP-24)
   PC ← *(RBP-8)
   RBP ← *(RBP-16)
   Ostatecznie nowym rozważanym PC jest adres powrotu z `do_division`, czyli 400F00.
8. Jak wiemy (patrz sekcja LSDA) tablica *call site* dla funkcji `main` jest niepusta. Rozważany PC trafia już w pierwszy element na liście (400F00 zawiera się w adresach od 400EEC do 400EEC+2B→400F17):
   12fd: start=400eec, len=2b, pad=400f49, action=3

9. Następuje sprawdzenie listy obsługiwanych typów na liście akcji. Pierwszy wpis (Type=1, czyli obiekt _ZTISt11range_error@@GLIBCXX_3.4 opisujący typ `std::range_error`) nie jest zgodny z obiektem opisującym rozważany wyjątek. Następuje przejście do kolejnego elementu listy (Next=-3).
10. Drugi element listy (Type=2, czyli obiekt _ZTISt16invalid_argument@@GLIBCXX_3_4 opisujący typ `std::invalid_argument`) jest poszukiwanym typem. Typ wyjątku (*handler switch value*) zostaje ustawiony więc na 2.
11. Ponieważ został znaleziony odpowiedni *call site*, który może obsłużyć rozważany wyjątek wirtualny stan procesora zostaje przeniesiony do faktycznego stanu procesora, rejestry RAX i RDX zostają ustawione na (kolejno) adres wewnętrznej struktury opisującej wyjątek oraz typ wyjątku (*handler switch value*) oraz następuje przejście do *landing pad* znajdującego się pod adresem 400F49.
12. Kod w *landing pad* (patrz sekcja Dyspozytor) znajduje odpowiedni blok *catch* do obsługi tego typu wyjątków (400FA1).
13. Następuje wejście do bloku *catch* i wydobycie z wewnętrznych struktur adresu obiektu opisującego wyjątek (`__cxa_begin_catch`).
14. I ostatecznie zostaje wykonany kod programisty, którego zadaniem jest obsłużyć wyjątek; po nim następuje już jedynie wyjście z bloku *catch* i wyczyszczenie wszystkich zaalokowanych po drodze struktur (`__cxa_end_catch`).

Podsumowanie

Jak zostało wyżej przedstawione, faktyczne implementacje wyjątków bywają złożone. Warto dodać również, iż samo użycie *try*/*catch* w opisywanym wypadku nie podnosi obciążenia procesora, natomiast zwiększa wielkość końcowego pliku wykonywalnego (z uwagi na sekcje *.eh_frame*, *.eh_frame_hdr* oraz *.gcc_except_table*). Również samo rzucenie i odebranie wyjątku jest zdecydowanie kosztowne - warto więc ograniczyć użycie wyjątków do faktycznie wyjątkowych sytuacji.

W sieci

[1] Microsoft, *try-except Statement*. http://msdn.microsoft.com/en-us/library/s58ftw19.aspx
[2] Wikipedia, *Exception handling syntax*.
http://en.wikipedia.org/wiki/Exception_handling_syntax#Microsoft-specific
[3] Wikipedia, *Microsoft-specific exception handling mechanisms*.
http://en.wikipedia.org/wiki/Structured_Exception_Handling#SEH
[4] Python Documentation, *Built-in Exceptions*. https://docs.python.org/3/library/exceptions.html
[5] Microsoft, *SetUnhandledExceptionFilter function*.
http://msdn.microsoft.com/en-us/library/windows/desktop/ms680634(v=vs.85).aspx
[6] Wikipedia, *setjmp.h*. http://en.wikipedia.org/wiki/Setjmp.h
[7] DWARF Standards Committee Wikipedia, *Exception Handling*.
http://wiki.dwarfstd.org/index.php?title=Exception_Handling

[8] Praca zbiorowa. *Itanium C++ ABI*. http://refspecs.linuxfoundation.org/cxxabi-1.86.html
[9] UNIX International, *DWARF Debugging Information Format*. http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf
[10] Free Standards Group, *Exception Frames*. http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html
[11] Agner Fog, *Calling conventions for different C++ compilers and operating systems*. www.agner.org/optimize/calling_conventions.pdf

Dodatkowe materiały:

- http://www.cs.dartmouth.edu/~sergey/battleaxe/hackito_2011_oakley_bratus.pdf
- http://static.usenix.org/events/osdi2000/wiess2000/full_papers/dinechin/dinechin_html/
- http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf
- http://llvm.org/docs/ExceptionHandling.html
- http://www.di.unipi.it/~nids/docs/longjump_try_trow_catch.html