Projekt wdrożeniowy – Intel SIMICS fuzz testing w RISC-V

Tomasz Jaworski
<tomasz.jaworski@student.put.poznan.pl>

Piotr Kaszubski
<piotr.kaszubski@student.put.poznan.pl>

Karol Kiedrowski
<karol.kiedrowski@student.put.poznan.pl>

Mikołaj Załęski
<mikolaj.zaleski@student.put.poznan.pl

kod źródłowy: https://github.com/RoyalDonkey/put-intel-fuzzing

Styczeń 2025

Contents

1 Opis problemu

Fuzzing Problem dotyczył fuzzingu, czyli metody testowania oprogramowania polegającej na celowym generowaniu niepoprawnych, lub pół-poprawnych danych wejściowych przez osobny program (tzw. fuzzer). Tak wygenerowane dane są następnie podawane do testowanego programu, w celu sprowokowania nieprawidłowego działania. Fuzzery są w zasadzie programami szukającym nowych danych do korpusu testowego.

Fuzzing jest gorącym tematem, który rozwija się prężnie od kilku dekad. Obecnie większość projektów i firm dużej skali takich jak jądro Linux, Microsoft, Google, etc. stosuje fuzzing z wielkim powodzeniem, ponieważ te narzędzia wykrywają błędy, które bardzo trudno jest zauważyć gołym okiem lub przy użyciu tradycyjnych metod takich jak testy jednostkowe.

Sanitizery Aby fuzzing był skuteczny do wdrożenia na dużą skalę, potrzebne są narzędzia do analizy poprawności stanu testowanego programu, aby umożliwić automatyczne wykrycie, kiedy fuzzer znalazł podatność. Jedną z klas takich narzędzi są sanitizery pamięci. Na potrzeby tego projektu istotny jest jeden konkretny sanitizer, czyli ASAN.

ASAN monitoruje stan pamięci testowanego programu w trakcie jego wykonywania i natychmiast zawiadamia o błędach (np. przepełnienie bufora), jeśli takowe wystąpią. Taka analiza jest możliwa, ponieważ instrukcje ASANa są bezpośrednio wkompilowane w testowany program (tak zwana “instrumentalizacja” programu). Testowany program funkcjonalnie pozostaje niezmieniony, ale na każdym kroku ma wbudowane dodatkowe sprawdzenia odczytów, zapisów, alokacji i zwalniania pamięci.

ASAN jest w sporej części po prostu biblioteką napisaną w C (libasan.so – na systemach Linux). W kompilatorze GCC, po instrumentalizacji testowanego programu, jest on zlinkowany dynamicznie z tą biblioteką. Zatem, aby uruchomić testowany program, system operacyjny musi znaleźć tę zależność.

Problem Istotą naszego zadania było znalezienie metody na przeprowadzenie fuzzingu programów na systemach wbudowanych bądź bare-metal, wewnątrz emulatora Intel SIMICS. Testowanie w takich warunkach jest z wielu powodów trudniejsze. W naszej pracy pomijamy element fuzzera i skupiamy się na sanitizerze.

Na szczególną uwagę zasługuje bare-metal, na którym jest zupełny brak systemu operacyjnego, więc nie ma możliwości nawet podejrzenia komunikatów printowanych przez ASANa. Aby umożliwić testowanie ASANem na bare-metal, konieczne jest utworzenie kanału komunikacji między nim a SIMICSem, tj. chcemy, aby każda wiadomość od ASANa, która klasycznie wylądowałaby na standardowym wyjściu programu, była dostępna jako string w skrypcie SIMICSowym. Brzmi to abstrakcyjnie, bo takie jest. Zasadniczo, to jest największy problem projektu i cała nasza praca głównie sprowadziła się do tego zadania.

Dodatkowe założenia Dla uproszczenia, całą pracę wykonaliśmy w środowisku Linux na RISC-V (64-bit). Dołożyliśmy starań, aby nasze rozwiązanie było kompatybilne z ideą bare-metal, ale byliśmy zbyt ograniczeni czasowo, aby przeprowadzić na nim testy.

2 Autorzy

Piotr Kaszubski – Lider zespołu, organizacja repozytorium, przydzielanie zadań, zgłębianie teorii fuzzowania, dokumentacji ASANa i SIMICSa, znalezienie funkcji `__sanitizer_on_print()` i pierwsza implementacja mechanizmu `LD_PRELOAD`, eksperymenty z linkowaniem statycznym, pomysł i implementacja ostatecznej metody z łączeniem plików obiektowych, poprawki do tworzenia regionu pamięci w SIMICSie, modyfikacje sterownika dla urządzenia znakowego, kompilacja skrośna na RISC-V, finalna wersja `__sanitizer_on_print()`, eksperymenty z `-static-libasan` w GCC, napisanie MAGIC hooka do SIMICSa, przeprowadzenie demo i prezentacji, dokumentacja.

Tomasz Jaworski – Tworzenie notatek z cotygodniowych spotkań, zaznajamianie się z dokumentacją SIMICs, stworzenie buforu pośredniczącego pomiędzy emulowanym systemem a emulatorem (przygotowanie sterownika dla urządzenia znakowego i modyfikacja device tree), dokumentacja.

Karol Kiedrowski – Zebranie informacji dotyczących działania sanitizera, przetestowanie jego działania, a także napisanie wstępnej logiki funkcji `__sanitizer_on_print()`.

Mikołaj Załęski – Pierwsze badania nad przechwyceniem wyjścia z biblioteki ASAN, pomysł na mechanizm `LD_PRELOAD`.

3 Rozwiązanie – koncepcja

Celem jest przesył wiadomości ASANa do SIMICSa.

Ogólny zamysł

- SIMICS utworzy dodatkowy region pamięci, który posłuży jako bufor do komunikacji. ASAN będzie tam wpisywał informacje, a SIMICS czytał.
- ASAN nie potrafi printować do osobnego pliku, logu, ani niczego podobnego. Musimy więc mieć sposób na odfiltrowanie jego wiadomości od pozostałych "śmieci", które potencjalnie może printować testowany program. W tym celu nadpiszemy jedną z funkcji w `libasan.so`, która jest wywoływana w momencie printowania przez ASANa, i wstrzykniemy tam kod, który będzie pisał do regionu pamięci SIMICSa.
- SIMICS musi wiedzieć, kiedy powinien odczytać coś z pamięci i potraktować jako wiadomość. W tym celu użyjemy jego funkcji "MAGIC instructions". Nasz kod wstrzyknięty w ASANa skorzysta z MAGIC aby wysyłać powiadomienia do SIMICSa.

Szczegóły

- W przypadku Linuxa, potrzebna jest modyfikacja device tree, aby zawiadomić jądro o istnieniu dodatkowego regionu pamięci.
- Jeśli nasz testowany program przebywa w userspace, nie jest możliwe bezpośrednie pisanie do regionu pamięci. Konieczne jest napisanie modułu, który umożliwi interakcję z tym regionem. W naszym przypadku mamy moduł, który wystawia proste urządzenie znakowe.
- W SIMICSie, konieczne będzie napisanie hooka, który nasłuchuje MAGIC instructions i odczytuje pamięć.

4 Rozwiązanie – implementacja

Nasze rozwiązanie obejmuje platformę symulacyjną `risc-v-simple` w SIMICS z systemem Linux oraz programem skompilowanym z sanitizerem. Do programu celowo wprowadziliśmy błędny dostęp do pamięci:

```
1 /* program: bad.c */
2 int main(int argc, char **argv)
3 {
4         volatile int a[5] = {0};
5         a[6] = argc;
6         return 0;
7 }
```

4.1 Linux

4.1.1 Sterownik

Napisaliśmy prosty sterownik (moduł jądra Linux) do obsługi bufora na wyjście sanitizera. Moduł ten pozwala nam na wpisywanie ciągów znaków do zdefiniowanego przez nas obszaru pamięci w celu ich późniejszego odczytania przez emulator SIMICS.

4.1.2 External Buildroot Toolchain

Stworzyliśmy external toolchain w Buildroot w celu budowania naszego modułu dla jądra Linuxa wraz z obrazem systemu. Nasz system dla SIMICSa jest zdefiniowany następującym defconfigiem:

```
# Architecture
BR2_riscv=y
BR2_RISCV_64=y

# Kernel
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_USE_ARCH_DEFAULT_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_VERSION_VALUE="5.17"
BR2_PACKAGE_HOST_LINUX_HEADERS_CUSTOM_5_17=y
BR2_LINUX_KERNEL_IMAGE=y

# Boot loader
BR2_TARGET_OPENSBI=y
BR2_TARGET_OPENSBI_PLAT="generic"

# Root file system
BR2_TARGET_ROOTFS_EXT2=y

# Host packages
BR2_PACKAGE_HOST_DTC=y
```

4.1.3 Modyfikacja Device-Tree

Do drzewa urządzeń dodaliśmy własne urządzenie (simics_shm), które jest buforem pamięci. Adres 0x20000000 został dobrany ręcznie pod platformę SIMICSa, na której się opieraliśmy (brak kolizji z innymi regionami pamięci). Rozmiar 0x4000 odpowiada 4 KiB i został uznany jako wystarczająco duży, by pomieścić printy ASANa (największe, które zaobserwowaliśmy, miały około 1 KiB).

```
simics_shm: simics_shm@20000000 {
        compatible = "simics_shm";
        device_type = "memory";
        reg = <0x00 0x20000000 0x00 0x4000>;
        status = "open";
};
```

Następnie zmodyfikowane drzewo urządzeń budujemy do postaci binarnej, poleceniem

```
dtc -O dtb -o risc-v-simple.dtb risc-v-simple.dts
```

4.1.4 Overlay dla Buildroot

Overlay w Buildroot do dołączania programów lub plików do platformy docelowej do systemu plików. Wydał nam się najszybszym sposobem na statyczne osadzenie naszego programu w systemie. Głównie dlatego, że go już znaliśmy.

4.1.5 Budowanie systemu

Budowaliśmy system Linux, jak już wcześniej wspomniano korzystając z Buildroota i dodając mu odpowiednie flagi służące do obsługi externala oraz overlay'a.

4.1.6 Dodawanie systemu do SIMICS

Wgranie zbudowanego systemu do platformy emulowanej można wykonać poprzez (i tak to zrobiliśmy) linkowanie plików wynikowych z budowania systemów w Buildroot do folderu targets w platformie risc-v-simple w SIMICS, wykonaliśmy to poleceniem:

```
ln -r -s ~/buildroot/output/images/* \
         ~/simics/simics-risc-v-simple-7.x.x/...
         ...targets/risc-v-simple/images/linux/
```

Dzięki temu nasza platforma po przebudowaniu będzie widoczna w aktualnej wersji z poziomu SIMICS, bez potrzeby ponownego kopiowania plików.

4.2 SIMICS (i jego konfiguracja)

4.2.1 Dodawanie urządzenia

Aby dodać urządzenie w emulatorze, aby adresy pamięci podane w naszym sterowniku były poprawnie intepretowane przez SIMICS i jądro Linuxa, stworzylismy każdorazowo obiekt:

```
fb_img = simics.SIM_create_object(
        "ram", f"{fb_ns}.buffer", image=None,
        self_allocated_image_size=size_bytes)
```

4.2.2 Skrypt inicjujący platformę

Aby nie podawać za każdym razem kilku komend związanych z utworzeniem obiektu naszej pamięci oraz ładowania platformy, napisaliśmy skrypt:

```
load-target "risc-v-simple/linux"
@SIM_create_object("ram","board.sanitizer_shm",
                   image=None, self_allocated_image_size=0x1000)
board.phys_mem.add-map board.sanitizer_shm 0x20000000 0x1000
```

4.2.3 Python MAGIC hook

Napisaliśmy też skrypt w języku python do uruchamiania w SIMICS, aby przechwytywać MAGIC instructions. Konwencjonalnie przyjęliśmy, że naszym `MAGIC_NUMBER` będzie 28.

```
def branch():
        while True:
                conf.bp.magic.cli_cmds.wait_for(number=MAGIC_NUMBER)
                print("asan: ", end="")
                addr = SANITIZER_SHM_ADDR
                while True:
                        byte = conf.board.phys_mem.memory[addr]
                        if byte == 0:
                                break
                        print(chr(byte), end="")
                        if byte == 10:
                                print("asan: ", end="")
                        addr += 1
                print()
```

4.3 Kompilacja programu

ASAN wywołuje wewnętrznie funkcję `__sanitizer_on_print()` za każdym razem, kiedy printuje informację dotyczącą błędów pamięciowych. Aby nadpisać tę funkcję, definiujemy ją po swojemu w `libsan-overlay.c`:

```c
/* biblioteka: libsan-overlay.c */
#include <magic-instruction.h>  /* Nagłówek SIMICSa */

#define SIMICS_SHM_FPATH "/dev/simics_shm"
#define MAGIC_NUMBER 28

void __sanitizer_on_print(const char *str)
{
        /* Wersja skrócona */
        FILE *file;
        file = fopen(SIMICS_SHM_FPATH, "w");
        fwrite(str, sizeof(*str), str_len, file);
        MAGIC(MAGIC_NUMBER);
}
```

Nasza wersja funkcji otwiera wpisuje `str` do urządzenia znakowego, które tworzy simics_shm, a następnie emituje MAGIC instruction, dając SIMICSowi znak, że może odczytać zawartość.

Trik obiektowy Aby nadpisać funkcję bez ingerencji w kod źródłowy testowanego programu, kompilujemy testowany program do pliku obiektowego, kompilujemy `libsan-overlay` do pliku obiektowego, następnie łączymy dwa pliki obiektowe w jeden, i dopiero ten wynikowy linkujemy z ASANem:

```
# Komendy uproszczone, żeby się mieściły
CC -c -Isimics/.../include/simics/ libsan-overlay.c
CC -c -g -Og -static-libasan -fsanitize=address bad.c
LD -o combined.o -r bad.o libsan-overlay.o
CC -o bad combined.o -static-libasan -fsanitize=address
```

Dzięki tej kolejności, uniemożliwiamy ASANowi zdefiniowanie swojej wersji `__sanitizer_on_print()`, ponieważ byłoby to powieleniem symbolu.

4.4 Uruchomienie platformy (demonstracja działania)

1. Budowanie systemu
2. Budowanie aplikacji z sanitizerem (`make`)
3. Uruchomienie SIMICS, a w emulatorze:
   (a) Ładujemy platformę (`run-script map_memory.simics`)
   (b) Uruchamiamy system (`run`)
   (c) Uruchamiamy skrypt Python (`run-script magic-hook.py`)
   (d) Tworzymy urządzenie znakowe (`modprobe simics_shm`)
   (e) Uruchamiany program (`./bad`)
   (f) Obserwujemy wyniki:

5 Napotkane problemy

5.1 Wczesne podejście z LD_PRELOAD

Pierwszym pomysłem na nadpisanie funkcji `__sanitizer_on_print()` było napisanie dynamicznej biblioteki definiującej tę funkcje, a następnie wywołanie testowanego programu ze zmienną `LD_PRELOAD=libsan-overlay.so`.

Ta metoda działa i jej największą zaletą jest to, że z `LD_PRELOAD`, tak długo jak mamy program zlinkowany dynamicznie z `libasan.so`, nie potrzebujemy dostępu ani do żadnych źródeł ani plików obiektowych. Niestety, są też wady, które dyskwalifikują to rozwiązanie:

1. Metoda jest uzależniona od mechanizmu systemu operacyjnego i nie działałaby na bare-metal.
2. Konieczne jest przeniesienie na platformę `libsan-overlay.so` wraz z programem testowym oraz ustawienie `LD_PRELOAD`, zatem potencjalnie musimy ingerować w skrypty uruchamiające program.
3. Metoda polega na dostępności `libasan.so` na platformie. Na okrojonych instalacjach Linux, ta biblioteka najprawdopodobniej nie będzie istnieć, zatem

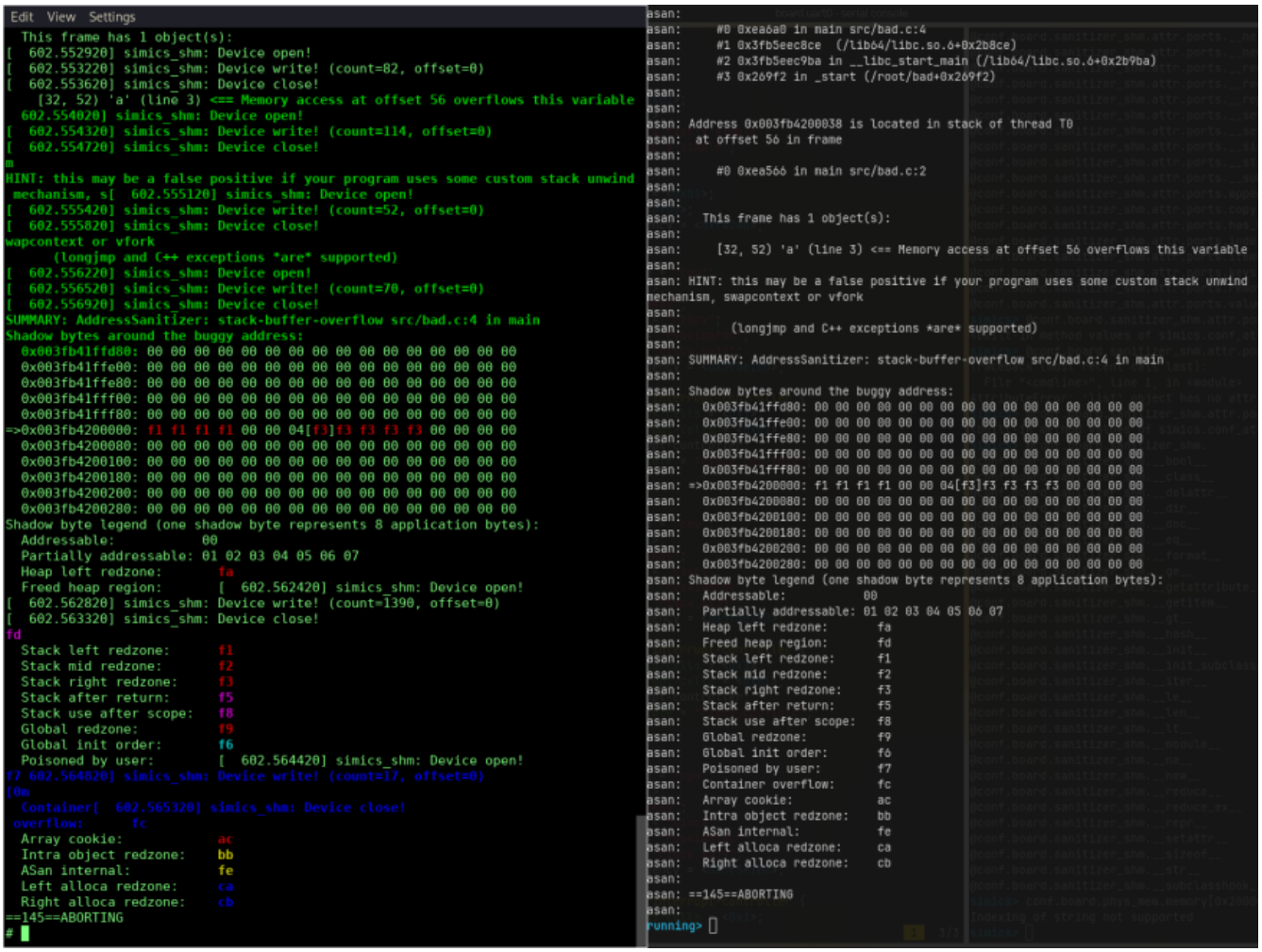

Figure 1: Końcowy rezultat projektu - uruchomiony SIMICS, w nim Linux i nasz program z sanitizerem. Widać, że print sanitizera udało się przenieść do terminala z emulatorem

pojawia się konieczność dostarczenia jej wraz z programem. Jest to szczególnie uciążliwe, jeśli platforma stoi na nietypowej architekturze, bo trzeba znaleźć odpowiednią wersję `libasan.so` i mieć nadzieję, że nie będzie żadnych niekompatybilności między nią a dynamicznie zlinkowanym programem testowym (co nie jest oczywiste np. w przypadku różnic w wersjach kompilatora).

5.2 Nadpisanie funkcji ze statycznie linkowanej biblioteki

Zanim wpadliśmy na pomysł połączenia dwóch plików obiektowych w jeden przed linkowaniem, rozważaliśmy sposoby na nadpisanie funkcji **po** linkowaniu.

W grę wchodzą dwa fundamentalne podejścia: modyfikacja funkcji wewnątrz pliku binarnego (przyjmijmy ELF), lub nadpisanie adresu funkcji w trakcie wykonania programu (tzw. ”trampolina”).

5.2.1 Modyfikacja ELF

W teorii, posiadając skompilowany plik binarny który nie został strip'owany, powinno być możliwe znalezienie miejsca, gdzie zapisane są instrukcje `__sanitizer_on_print()` i podmiana ich na własną definicję. W praktyce, nie znaleźliśmy narzędzi, które w łatwy sposób umożliwiałyby zrobienie czegoś takiego. Co więcej, nawet gdyby takie narzędzie istniało, zapewne byłoby bardzo specyficzne dla danej architektury lub formatu ELF i nie byłoby dostatecznie uniwersalne, aby umożliwić testowanie na wielu różnych platformach.

5.2.2 Trampolina

Nie ingerując w plik binarny, można rozważyć podejście dynamiczne: załadowanie do pamięci własnej implementacji `__sanitizer_on_print()`, a następnie sprytna podmiana wszystkich skoków pamięci do oryginalnej wersji na nasz własny adres.

Takie metody się stosuje i można znaleźć nawet przykłady w internecie. Przeważnie polegają na napisaniu biblioteki dynamicznej z funkcją inicjalizacji (w GCC `__attribute__((constructor))`), która wykorzystuje `dlopen()` i `dlsym()` aby znaleźć adres funkcji, którą chcemy podmienić, a następnie wpisuje w odpowiednie miejsca (w pamięci) instrukcje assemblera do skoku pod inny adres.

Problem tych metod jest taki, że są bardzo mało uniwersalne, wymagają pisania kodu pod konkretną architekturę, przeważnie polegają na dynamicznym ładowaniu bibliotek (a zatem niekompatybilne z bare-metal) i nie należą do łatwych w implementacji. Dodatkowo mogłyby być blokowane przez zabezpieczenia sprzętowe typu NX bit.

6 Kierunki Rozwoju

Dalszą pracę, jaką można byłoby wykonać, można podzielić na dwa główne zadania.

6.1 Bare-metal

Jako pierwszy kierunek dalszego rozwoju projektu, należy wymienić zbudowanie programu i uruchomienie go bez systemu operacyjnego (czyli bare-metal). Wymagałoby to standardowych kroków dla takiej implementacji oprogramowania, a następnie, wykorzystując m.in. już wcześniej opisaną przez nas metodę kompilacji z sanitizerem programów statycznie, stworzenia pliku z kodem maszynowym, który mógłby wykonać procesor.

6.2 Fuzzing

Drugim kierunkiem niewątpliwie jest dodanie fuzzera do aplikacji, którą chcielibyśmy testować. Program ten powinien przyjmować jakieś wejście od użytkownika, a następnie przetwarzać je, aby możliwym było prześledzenie, czy program zachowuje się prawidłowo.