高度サイバーセキュリティ PBLII レポート

28G23027
川原尚己

1-1.ミニ演習

- 適当な Rust のプログラムを実装し、LLVM IR とアセンブリを取得してみよ.
  図 1 のコードを実装した.

```rust
fn main() {
    println!("Hello, world!");
}
```

図 1　実装 rust コード

このコードに対し，”cargo rustc -- --emit asm”及び” rustc -- --emit=llvm-ir”を実行すると, target/debug/deps 以下に図 2 のような.s ファイル及び.ll ファイルが生成された.

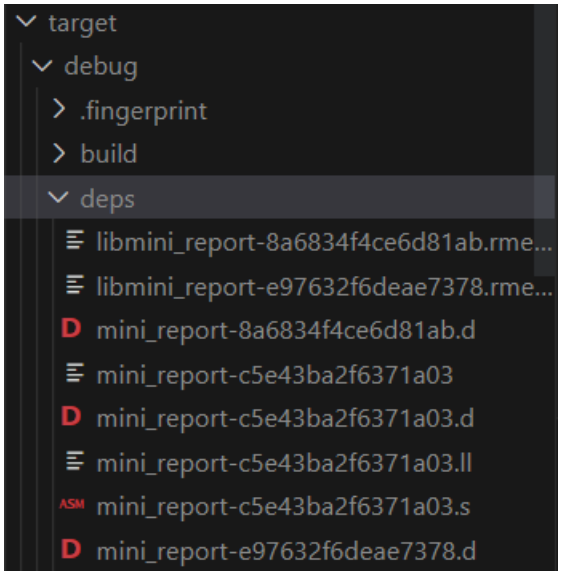


図 2　生成されたアセンブリファイル及び LLVM IR ファイル

- man をつかってシステムコールのマニュアルをみてみよ
  “man 2 read”を実行すると,

```
READ(2)                                          Linux  Programmer's  Manual
READ(2)

NAME
       read - read from a file descriptor
```

```
SYNOPSIS
       #include <unistd.h>

   ssize_t read(int fd, void *buf, size_t count);
```

という出力が得られた(冒頭部分のみ抜粋).

- /dev/null にメッセージを書き込むプログラムを実装し、strace でシステムコールトレースしてみよ.
  講義資料 25 ページのコードを実装し，コンパイルの後実行しても何の出力もされていないように見えたが，”strace ./target/debug/mini_report”を実行すると，

```
root@f513768617b6:~/mini_report# strace ./target/debug/mini_report
execve("./target/debug/mini_report",      ["./target/debug/mini_report"],
0x7ffc0cd29de0 /* 27 vars */) = 0
brk(NULL)                                 = 0x55faee2c5000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc651db670) = -1 EINVAL (Invalid
argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7fd56f74a000
access("/etc/ld.so.preload", R_OK)        = -1 ENOENT (No such file or
directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3,    "",    {st_mode=S_IFREG|0644,    st_size=10919,    ...},
AT_EMPTY_PATH) = 0
mmap(NULL, 10919, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fd56f747000
close(3)                                  = 0
```

という出力が得られた(冒頭部分のみ抜粋).

- デバッガを利用して以下の挙動を実際に確認せよ.
  - ◆ ブレークポイント設定
    “gcc -g -O0 <ファイル名>.c”にてコンパイルした後，”lldb a.out”にてデバッガを起動すると，自動的に

```
(lldb) target create "a.out"
```

Current executable set to '/root/mini_report/src/a.out' (x86_64).

というメッセージが出現した．その後，" breakpoint set -n funcB"によってブレークポイントを設置し，

Breakpoint 1: where = a.out`funcB + 12 at lldb_test.c:9:13, address = 0x000000000000115a

のメッセージを得た.

◆ ステップ実行

ブレークポイントを設置した後，"run"コマンドによって実行し，

```
Process 5817 launched: '/root/mini_report/src/a.out' (x86_64)
Process 5817 stopped
* thread #1, name = 'a.out', stop reason = breakpoint 1.1
    frame #0: 0x000055555555515a a.out`funcB at lldb_test.c:9:13
   6        int a = b + 20;
   7    }
   8    int funcB() {
-> 9        int c = funcC();
   10       int b = c + 10;
   11       return b;
   12   }
```

を得た．"step"コマンドを実行すると，

```
Process 10837 stopped
* thread #1, name = 'a.out', stop reason = step in
    frame #0: 0x000055555555517d a.out`funcC at lldb_test.c:14:12
   11       return b;
   12   }
   13   int funcC() {
-> 14       return 30;
   15   }
   16   int main() {
   17       funcA();
```

が出力された．確かに 9 行目にて funcC 関数の中に入っている．

◆ バックトレース

講義資料 45 ページの関数を先ほどと同様にして lldb でデバックを起動した．"run"を実行すると，

```
Process 12758 launched: '/root/mini_report/src/bt' (x86_64)
Process 12758 stopped
```

```
* thread #1, name = 'bt', stop reason = signal SIGSEGV: invalid address
(fault address: 0x0)
    frame #0: 0x000055555555519d bt`a(m=0x0000000000000000) at bt.c:10:4
   7    void a(int *m) {
   8    int n = 100;
   9    b(&n);
-> 10   *m += n;
   11   }
   12
   13   int main(int argc, char *argv[]) {
```

が表示された．”bt”を実行すると，

```
* thread #1, name = 'bt', stop reason = signal SIGSEGV: invalid address
(fault address: 0x0)
  * frame #0: 0x000055555555519d bt`a(m=0x0000000000000000) at bt.c:10:4
    frame #1: 0x00005555555551de bt`main(argc=1, argv=0x00007fffffffe238)
at bt.c:14:1
    frame #2: 0x00007ffff7db9d90 libc.so.6`___lldb_unnamed_symbol3139 +
128
    frame #3: 0x00007ffff7db9e40 libc.so.6`__libc_start_main + 128
frame #4: 0x0000555555555085 bt`_start + 37
```

となった．Thread #1 で stop しているようである．”f 1”で frame #1 を表示すると，

```
frame #1: 0x00005555555551de bt`main(argc=1, argv=0x00007fffffffe238) at
bt.c:14:1
   11   }
   12
   13   int main(int argc, char *argv[]) {
-> 14   a(NULL);
   15   return 0;
   16   }
```

となった．エラーの原因は 14 行目のようであることがわかった．

- fork, exec を使ったプログラムを実装し、動作を確認せよ

  講義資料 55 ページを図 3 のように書き換え，”Cargo.toml”の[dependencies]に”nix=”0.24.1””を追加した．

```rust
use nix::{
    sys::wait::waitpid,
    unistd::{execvp, fork, write, ForkResult},
};
use std::ffi::CString;

fn main(){
    match unsafe { fork() } {
        Ok(ForkResult::Parent { child, .. }) => {
            println!("親プロセス : 子プロセスのpid: {}", child);
            waitpid(child, None).unwrap();
        }
        Ok(ForkResult::Child) => {
            // `println!`はforkした直後は使わない
            write(nix::libc::STDOUT_FILENO, "子プロセス\n".as_bytes()).ok();
            let filename = CString::new("/usr/bin/echo").unwrap();
            match execvp(&filename, &[&filename]) {
                Err(_) => {
                    nix::unistd::write(
                        nix::libc::STDERR_FILENO,
                        "不明なコマンドを実行\n".as_bytes(),
                    )
                    .ok();
                    unsafe { nix::libc::_exit(0) };
                }
                Ok(_) => unreachable!(),
            }
        }
        Err(_) => println!("forkに失敗"),
    }
}
```

図 4　fork, exec を使用したソースコード

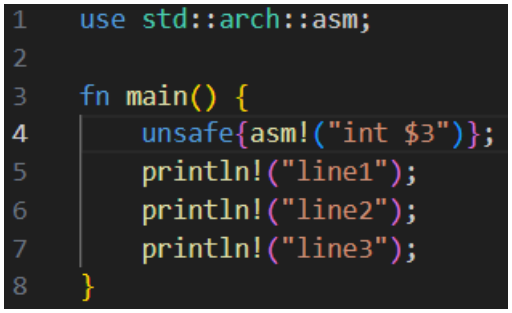
これをコンパイルし，実行すると，図 5 のように出力された．

```
root@f513768617b6:~/fork_exec# ./target/debug/fork_exec
子プロセス
親プロセス : 子プロセスのpid: 19825
```

図 5　fork, exec を使用したプログラムの実行結果

- int 3 命令を実行するようなプログラムをインラインアセンブリを用いて実装し、デバッガで break と continue ができることを確認せよ．

  図 6 のようなソースコードを実装し，lldb を起動した．



```
use std::arch::asm;

fn main() {
    unsafe{asm!("int $3")};
    println!("line1");
    println!("line2");
    println!("line3");
}
```

図６　割り込み処理のソースコード

ここで，”run”コマンドを実行すると，

```
(lldb) r
Process 21875 launched: '/root/interrupt/target/debug/interrupt' (x86_64)
Process 21875 stopped
* thread #1, name = 'interrupt', stop reason = signal SIGTRAP
    frame #0: 0x000055555555c928
interrupt`interrupt::main::ha2d83ce7f5686b35 at main.rs:5:5
   2
   3    fn main() {
   4        unsafe{asm!("int $3")};
-> 5        println!("line1");
   6        println!("line2");
   7        println!("line3");
   8    }
```

となり，ブレークポイントを設定できている．

- 上記プログラムをデバッガを用いずに実行してみて、プロセスが例外 trap で終了することを確認せよ．

  通常の実行を行なうと，

  ```
  root@f513768617b6:~/interrupt# ./target/debug/interrupt
  Trace/breakpoint trap
  ```

  となり，たしかに trap され，終了している．

- SIGUSR1 シグナルを処理するプログラムを実装し、正しく動くことを kill コマンドで動作を確認せよ．

  講義資料 75 ページのコードを実行し，”pgrep <実行ファイル名>”で PID を取得した後，”kill -10 <PID>”を実行することにより，SIGUSR1 を送信することができる．SIGUSR1 シグナルを受信すると，”SIGUSR1”と表示され，処理が継続する．SIGUSR1

以外のシグナルを送った場合は，何も表示されないか，シグナルに対応していると思われるメッセージが表示されたのち，終了する．

- 自分自身に SIGTRAP を配送するプログラムを実装し、デバッガで動作させ break と continue ができることを確認せよ．

  以下の図 7 のコードを実装し，コンパイルした後，lldb を用いてデバッガを起動した．

```rust
use nix::sys::signal::raise;
use nix::sys::signal::Signal;

fn main() {
    println!("line1");
    raise(Signal::SIGTRAP).unwrap();
    println!("line2");
}
```

図 7　raise 関数を用いた SIGTRAP 送信処理のコード

実行結果は以下のようになる．

```
(lldb) r
Process 10385 launched: '/root/sigtrap/target/debug/sigtrap' (x86_64)
line1
Process 10385 stopped
* thread #1, name = 'sigtrap', stop reason = signal SIGTRAP
    frame #0: 0x00007ffff7e06a7c libc.so.6`pthread_kill + 300
libc.so.6`pthread_kill:
->  0x7ffff7e06a7c <+300>: movl   %eax, %r13d
    0x7ffff7e06a7f <+303>: negl   %r13d
    0x7ffff7e06a82 <+306>: cmpl   $0xfffff000, %eax            ; imm =
0xFFFFF000
    0x7ffff7e06a87 <+311>: movl   $0x0, %eax
(lldb) c
Process 10385 resuming
line2
Process 10385 exited with status = 0 (0x00000000)
```

run コマンドを実行すると，println!(“line1”);だけが実行され，raise 関数にて trap される．continue コマンドを実行し，raise 関数から抜けると，println!(“line2”);が実行され，処理が終了する．

1-2. ZDbg の以下の関数を実装し、レポートとしてまとめよ

(a) set_break

以下のように実装した.

```
    let addr = if let Some(addr) = self.info.brk_addr {
                addr
            } else {
                return Ok(());
            };

            println!("<<以下のようにメモリを書き換えます>>");
            let data = ptrace::read(self.info.pid, addr)?;
            self.info.brk_val = data;
            println!("<<before : {:X}: {:016X}>>", addr as u64, data);
            let data = data as u64;
            let data = (data & 0xffffffffffffff00) | 0x00000000000000cc;

            unsafe {ptrace::write(self.info.pid, addr, data as *mut c_void)}?;

            let data = ptrace::read(self.info.pid, addr)?;
        println!("<<after : {:X}: {:016X}>>", addr as u64, data);
```

objdump を用い，ターゲットアドレス 0x8e88 を取得した．具体的には

- objdump -d target/debug/dbg_target|grep 'int3'
- objdump -d target/debug/dbg_target|grep 'nop'
- objdump -d target/debug/dbg_target|less

の順に実行した．一つ目のコマンドで得られたアドレスに最も近い nop のアドレスを取得し, 三つ目のコマンドで nop のアドレスの 1 つ先のアドレスを見ることができる．このアドレスが実行時に入力するアドレスとなる．実行結果を以下に示す.

```
root@f513768617b6:~/zdbg# ./target/debug/zdbg ../dbg_target/target/debug/dbg_target
zdbg > b 0x8e88
zdbg > r
<<子プロセスの実行に成功しました：PID = 8700>>
<<以下のようにメモリを書き換えます>>
Error: EIO
int 3
```

Error:EIO というエラーが発生し，所望の処理を得ることができなかった．一つ目の println!の次の行の ptrace::read にてこのエラーが発生しているようだが，解決はできなかった.

(b) wait_child

以下のように実装した.

```
    fn wait_child(self) -> Result<State, Box<dyn Error>> {
            match waitpid(self.info.pid, None)? {
                WaitStatus::Exited(..) | WaitStatus::Signaled(..) => {
                    println!("<<子プロセスが終了しました>>");
                    let not_run = ZDbg::<NotRunning> {
                        info: self.info,
                        _state: NotRunning,
                    };
                    Ok(State::NotRunning(not_run))
                }
                WaitStatus::Stopped(..) => {
                    // TODO: ここを実装せよ

                    let mut regs = ptrace::getregs(self.info.pid)?;
                    let addr = self.info.brk_addr.unwrap() as u64;
                    if regs.rip == addr{
                        regs.rip -= 1;
                        ptrace::setregs(self.info.pid, regs)?;
                        unsafe{ptrace::write(self.info.pid,      addr      as
    ptrace::AddressType, self.info.brk_val as *mut c_void)?;}
                    }

                    Ok(State::Running(self))
                }
                _ => Err("waitpid の返り値が不正です".into()),
            }
    }
```

2-1. src/engine/evaluator.rs 内の、eval_depth 関数を実装し、評価器を完成させよ

実装したコードを以下に示す.

```
    Instruction::Jump(addr) => {
                // TODO: PC を addr にするのみ
                pc = *addr;
            }
            Instruction::Split(addr1, addr2) => {
```

```
                // TODO: 再帰呼び出しで 2 つのアドレスに対してマッチングさせる
                if eval_depth(inst, line, *addr1, sp)? || eval_depth(inst, line,
*addr2, sp)? {
                    return Ok(true);
                } else {
                    return Ok(false);
                }
            }
```

2-2. src/engine/codegen.rs 内の、gen_plus、gen_question、gen_star 関数を実装し、コード生成器を完成させよ

後述するソースコードを実装し，それを用いて以下のようなテストケースを実行した.

各正規表現用言語に対し，複数のテストケースを実施し，そのすべてに対し正しく動いていることを確認した．表 1 にその一例を示す.

評価対象：

```
b
bd
bacd
bacacd
```

検索内容	結果
b(ac)+d	bacd bacacd
b(ac)*d	bd bacd bacacd
b(ac)?d	bd bacd

表 1　検索内容と検索結果

- gen_plus

  以下のように実装した.

```
    fn gen_plus(&mut self, e: &AST) -> Result<(), Box<CodeGenError>> {
        // TODO:
        // L1: e1 のコード
        let split_addr = self.pc;
        self.gen_expr(e)?;
```

```
        // split L1, L2
        let split_addr2 = self.pc;
        let split = Instruction::Split(split_addr, 0); // self.pc が L1。
    L2 を仮に 0 と設定
        self.insts.push(split);

        // L2 の値を設定
        self.inc_pc()?;
        if       let       Some(Instruction::Split(_,       l2))       =
    self.insts.get_mut(split_addr2) {
            *l2 = self.pc;
        } else {
            return Err(Box::new(CodeGenError::FailStar));
        }

        Ok(())
    }
```

- gen_question

  以下のように実装した.

```
    fn gen_question(&mut self, e: &AST) -> Result<(), Box<CodeGenError>>
    {
        // TODO:
        let split_addr = self.pc;
        self.inc_pc()?;
        let split = Instruction::Split(self.pc, 0); // self.pc が L1。L2 を
    仮に 0 と設定
        self.insts.push(split);

        // L1: e1 のコード
        self.gen_expr(e)?;

        // L2 の値を設定
        if       let       Some(Instruction::Split(_,       l2))       =
    self.insts.get_mut(split_addr) {
            *l2 = self.pc;
        } else {
```

```
            return Err(Box::new(CodeGenError::FailQuestion));
        }

        Ok(())
    }
```

- gen_star

  以下のように実装した.

```
    fn gen_star(&mut self, e: &AST) -> Result<(), Box<CodeGenError>> {
        // // TODO:
        let jmp_addr = self.pc;
        let split_addr = self.pc;
        let split = Instruction::Split(0, 1); // L3 を仮に 0 と設定。L3 を仮に 1 と設定
        self.insts.push(split);

        // L2 の値を設定
        self.inc_pc()?;
        if let Some(Instruction::Split(l2, _)) = self.insts.get_mut(split_addr) {
            *l2 = self.pc;
        } else {
            return Err(Box::new(CodeGenError::FailStar));
        }

        // L2: e2 のコード
        self.gen_expr(e)?;

        // jmp L1
        // let _jmp_addr = self.pc;
        self.insts.push(Instruction::Jump(jmp_addr)); // L3 を仮に 0 と設定

        // L3 の値を設定
        self.inc_pc()?;
        if let Some(Instruction::Split(_, l3)) = self.insts.get_mut(split_addr) {
            *l3 = self.pc;
```

```
        } else {
            return Err(Box::new(CodeGenError::FailStar));
        }

        Ok(())
    }
```

2-3. 既存の正規表現エンジンを参考にし、足りない機能を 1 つ以上追加せよ

任意の文字を表す'.'を実装した．実装箇所は"codegen.rs"，"parser.rs"，"engine.rs"，"evaluator.rs"であり，実装内容については後述する．

以下に示す評価対象に対して表 2 に示すような検索を行なった．

表 2 を見ると，所望の処理がなされていることが確認できる．

評価対象：

```
abc
adc
cba
abcd
abdc
```

検索内容	結果
a.c	abc adc abcd
a..c	abdc

表 2　検索内容と検索結果

また，以下に本機能を実現するために実装・追加したソースコードを示す．

- codegen.rs
  - gen_period 関数の追加．

```
fn gen_period(&mut self, c: char) -> Result<(), Box<CodeGenError>> {
        let inst = Instruction::Period(c);
        self.insts.push(inst);
        self.inc_pc()?;
        Ok(())
    }
```

  - Gen_expr 関数内に以下のコードを追加

```
AST::Period(e) => self.gen_period(*e)?
```

- parser.rs
  - pub enum AST 内に以下のコードを追加

    ```
    Period(char)
    ```
  - parse 関数の for 文内に以下のコードを追加

    ```
    '.' => seq.push(AST::Period(c)),
    ```
  - 特殊文字のエスケープに'.'を追加
- engine.rs
  - pub enum Instruction 内に以下のコードを追加

    ```
    Period(char),
    ```
  - Impl Display for Instruction 内に以下のコードを追加

    ```
    Instruction::Period(c) => write!(f, "period {}", c),
    ```
- evaluator.rs
  - eval_depth 関数内に'.'の時の処理を追加

    ```
    Instruction::Period(_) => {
        if let Some(_sp_c) = line.get(sp){
            safe_add(&mut pc, &1, || Box::new(EvalError::PCOverFlow))?;
            safe_add(&mut sp, &1, || Box::new(EvalError::SPOverFlow))?;
            } else {
                return Ok(false);
            }
          }
    ```
  - eval_depth 関数にも同一のコードを追加