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．

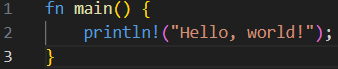


図１　実装rustコード

　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．

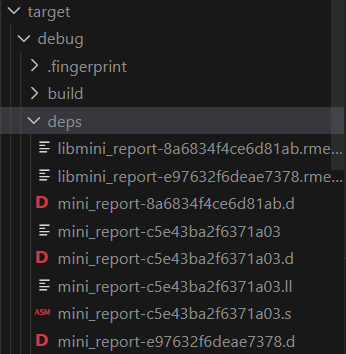


図２　生成されたアセンブリファイル及びLLVM IRファイル

* 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．

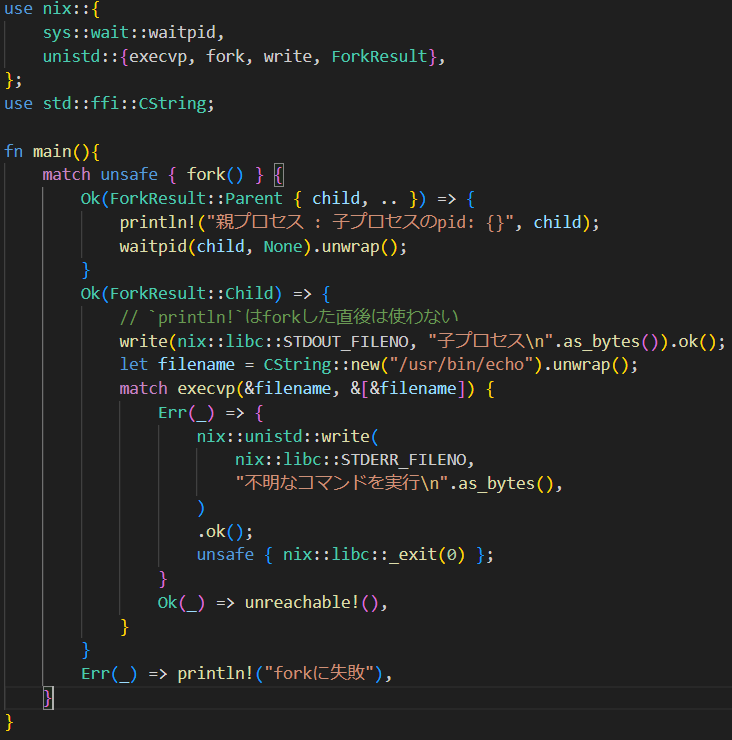


図４　fork, execを使用したソースコード

　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．



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．

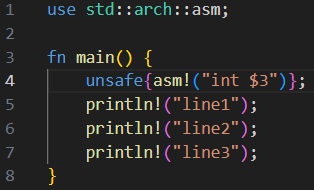


図６　割り込み処理のソースコード

　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．

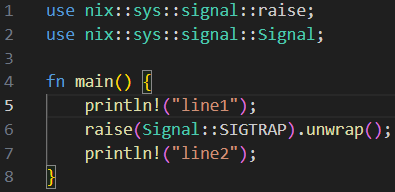


図７　raise関数を用いたSIGTRAP送信処理のコード

　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

1. 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)}?;

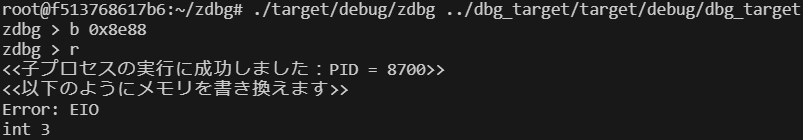
let data = ptrace::read(self.info.pid, addr)?;

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．



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．

1. 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：



|  |  |
| --- | --- |
| 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：



|  |  |
| --- | --- |
| 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