Lec 2: Binary Code

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Reflections on Trusting Trust

Did you read the paper?

Reflections on Trusting Trust, ***CACM***, Volume 27, Issue 8, Aug. 1984.

So, Why Binary Code?

> Cannot trust a program without looking at its binary code.

Ken Thompson[1]

1. Turing award winner.
2. Creator of UNIX.

[1] https://www.computer.org/profiles/kenneth-thompson/

Proof: Three-Step Approach

1. Writing a ***quine*** is possible with a Turing-complete language.
2. Compilers can be trained to include new behaviors.
3. One can teach a compiler to produce a quine that replicates a trojan.

Stage 1: Quine

```
char s[] = {
  '\t',
  '0',
  '\n',
  '}',
  ':',
  '\n',
  '\n',
  '/',
  '*',
  '\n',
  (213 lines deleted)
  0
};
```

```
/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */
main()
{
  int i;

  print("char\ts[] = {\n");
  for(i=0; s[i]++; i++)
    printf("\t%d,\n", s[i]);
  printf("%s", s);
}
```

Stage 1: Quine

Quine in F# (one line)

```
let d = "let d = {0}{1}{0} in System.Console.Write(d,char 34,d
    )" in System.Console.Write(d,char 34,d)
```

Stage 2: C Compiler in C

```
...
c = next();
if (c != '\\')
  return c;
c = next();
if (c == '\\')
  return '\\';
if (c == 'n')
  return '\n';
...
```



```
...
c = next();
if (c != '\\')
  return c;
c = next();
if (c == '\\')
  return '\\';
if (c == 'n')
  return '\n';
if (c == 'v')
  return '\v';
...
```



```
...
c = next();
if (c != '\\')
  return c;
c = next();
if (c == '\\')
  return '\\';
if (c == 'n')
  return '\n';
if (c == 'v')
  return 11;
...
```

Stage 3-1: Trojan Horse

```
void compile(char *s)
{
  // ...
}
```

```
void compile(char *s)
{
  if (match(s, "login pattern")) {
    compile("login backdoor");
    return;
  }
  // ...
}
```

Stage 3-2: Trojan Horse

```
void compile(char *s)
{
  // ...
}
```

```
void compile(char *s)
{
  if (match(s, "login pattern")) {
    compile("login backdoor");
    return;
  }
  if (match(s, "compiler pattern")) {
    compile("insert the backdoor");
    return;
  }
  // ...
}
```

Self-Replicating Backdoor (Trojan)

This technique applies to ***any*** program-handling program such as assembler, loader, or hardware microcode, etc.

Software Security Assumption

H/W is secure.

Software Security Assumption

H/W is secure.

Given H/W is secure, one should analyze ***binary code*** (= executable code) to understand whether it is safe to execute or not.

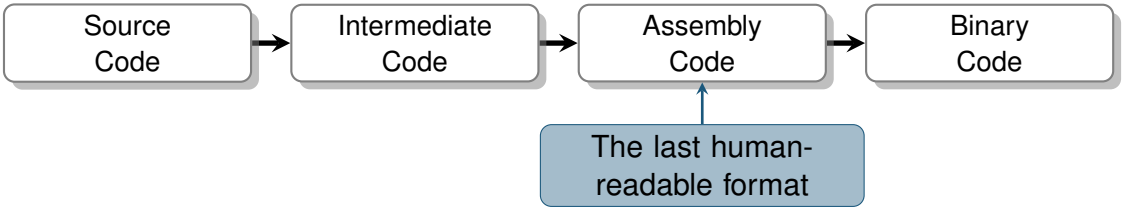
Compilation

Compiling C

```
int main(int argc, char * argv[])
{
  // ...
  return 42;
}
```

```
01010101010111110101
01010101010101000100
10010001111111010111
11010010101000100010
11010001011010001010
01001010010111111010
10100000010101011000
00100000101101101001
01101010010110011010
10100110001101111000
```

Compilation Process



Source to Intermediate Code

```c
int foo(int a)
{
    return a + 42;
}
```

$ clang -emit-llvm -S foo.c

```
define dso_local i32 @foo(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = add nsw i32 %3, 42
  ret i32 %4
}
```

Intermediate Code to Assembly

```
define dso_local i32 @foo(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ...
```

```
$ llc --x86-asm-syntax=intel foo.ll
```

```
foo:                                    # @foo
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov     rbp, rsp
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], edi
        ...
```

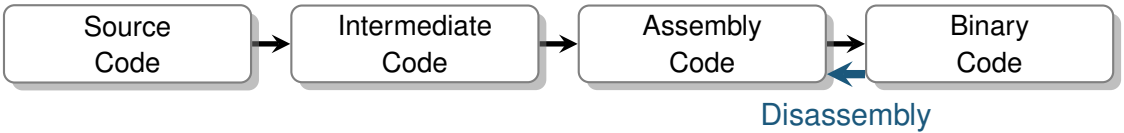
Assembly to Binary

```
foo:                                                    # @foo
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov     rbp, rsp
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], edi
        ...
```

```
$ as -o foo.o foo.s
```

```
00010111101001011010101011110000110100101000 ...
```

Disassembling Binary Code



Disassembly: objdump

```
$ objdump -Mintel -d foo.o

foo.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <foo>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   7:   8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
   a:   83 c0 2a                add    eax,0x2a
   d:   5d                      pop    rbp
   e:   c3                      ret
```

Disassembly: objdump

```
$ objdump -Mintel -d foo.o

foo.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <foo>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
   7:   8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
   a:   83 c0 2a                add    eax,0x2a
   d:   5d                      pop    rbp
   e:
```

Compiler-generated assembly = Disassembled assembly?

Disassembly is Difficult

The very first step of binary analysis, but it is extremely ***challenging*** because of

- Indirect branches.
  - `jmp rax`
  - `call rax`
- Mixture of code and data.

Example: Linear Sweep Disassembly

Try to assemble the following and disassemble the resulting binary with `objdump`.

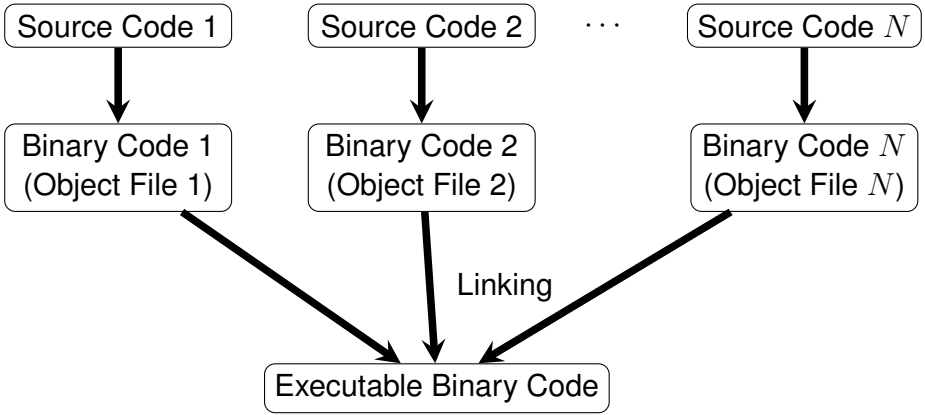
```
.intel_syntax noprefix
inc ebx
sub eax, ebx
call lbl
.ascii "hello world"
lbl:
pop eax
```

Recursive Traversal Disassembly

- A.k.a. Recursive Descent Disassembly
- Follow control-flows starting from entry point(s).

Still infeasible to recover indirect branch targets.

Linking


Source Code 1
Source Code 2
· · ·
Source Code N
Binary Code 1
(Object File 1)
Binary Code 2
(Object File 2)
Binary Code N
(Object File N)
Linking
Executable Binary Code

Linking is Essential for C Programs

Simplistic program in C:

```
int main(void) {
  return 42;
}
```

Linking failure

```
$ gcc -c -o example.o example.c
$ ld -o example example.o
ld: warning: cannot find entry symbol _start;
defaulting to 0000000000401000
$ ./example
Segmentation fault (core dumped)
```

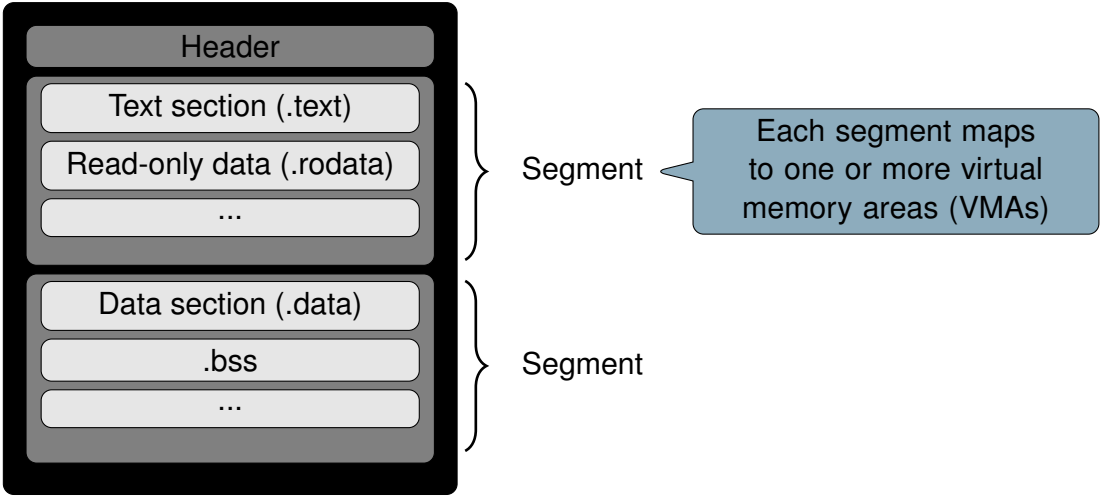
Linking Internals

Working example

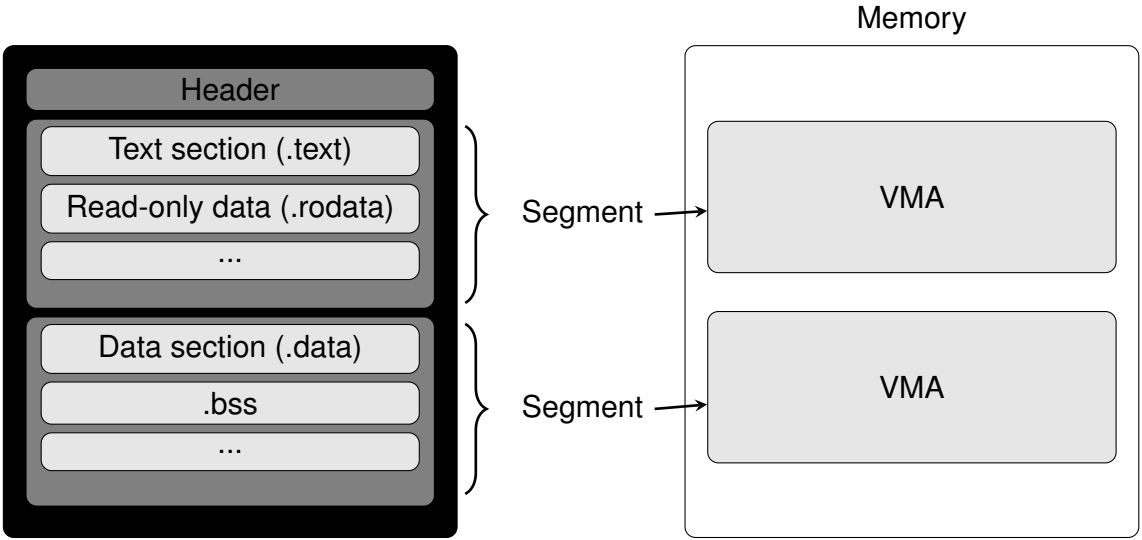
```
$ gcc -c -o example.o example.c
$ gcc -o example example.o
$ ./example
$ gcc -v -o example example.o
```

The verbose option (`-v`) prints out many dependent object files for C runtime support.

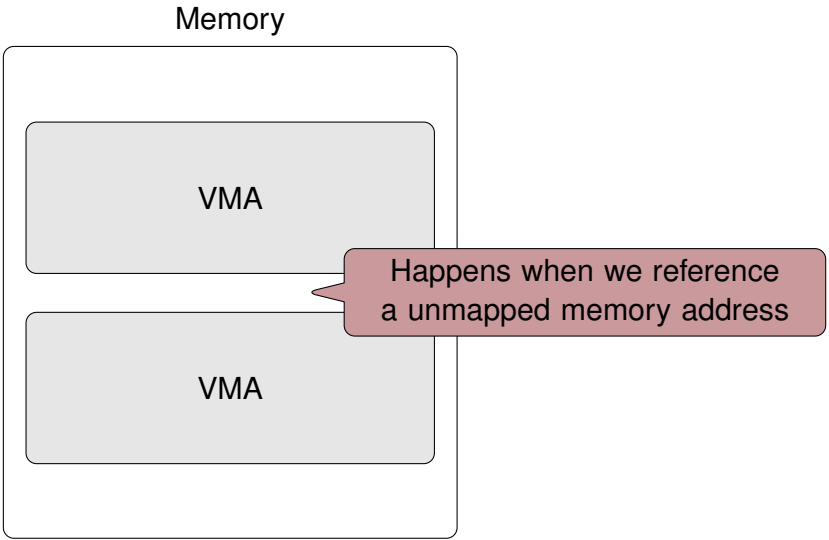
Executable Binary



Executable Binary



Segmentation Fault



Using B2R2

- Install .NET SDK 8.0 or above (`https://dotnet.microsoft.com/en-us/download`).
- Open your terminal and type:
  `dotnet tool install -g B2R2.RearEnd.Launcher`
- Type `b2r2` in the terminal.

Disassemble Binaries with B2R2

Disassemble a binary file.

```
$ b2r2 dump <binary file>
```

Disassemble x86-64 byte sequence.

```
$ b2r2 dump -s 00102030 -i x86-64
0000000000000000: 00 10       add byte ptr [RAX], DL
0000000000000002: 20 30       and byte ptr [RAX], DH
```

Lifting Binary Semantics

Lift x86-64 byte sequence to B2R2 IR.

```
$ b2r2 dump -s 4c01f8 -i x86-64
0000000000000000: 4C 01 F8    add RAX, R15
$ b2r2 dump -s 4c01f8 -i x86-64 --lift
[4C 01 F8]
(3) {
T_1:I64 := RAX
T_2:I64 := R15
T_3:I64 := (T_1:I64 + T_2:I64)
RAX := T_3:I64
CF := (T_3:I64 < T_1:I64)
OF := (((T_1:I64[63:63]) = (T_2:I64[63:63]))
       & ((T_3:I64[63:63]) ^ (T_1:I64[63:63])))
...
```

Question?

Exercise

1. Write your own quine in your favorite language.
2. Write an ouroboros[2] in two languages. For example, you can write a Python program that outputs a C program that outputs the original Python program.

[2] https://en.wikipedia.org/wiki/Quine_(computing)