Rise & Fall of Binaries | Part 1

Adir Abraham
adir@computer.org

See notes for citation
All materials are licensed under a Creative Commons “Share Alike” license.

You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

- Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

• http://creativecommons.org/licenses/by-sa/3.0/
All materials are licensed under a Creative Commons “Share Alike” license

• Used with permission from (and inspired by) Xeno Kovah’s lectures and slides.
• Check out his stuff: http://opensecuritytraining.info/
About You?

• Firmware Engineer
• Information Security Researcher
• ACM Professional Member
• IEEE Member
• FOSS Activist
• Hamakor NPO Board Member

• More details: https://linkedin.com/in/adirab
Agenda

• Part 1 - Lexing, Parsing, CFGs, ASTs, AATs, BLTs, generating assembly
• Part 2 - Portable Executable (PE) files
• Part 3 - Executable and Linking Files (ELF)

• There will be 1 or 2 more lectures for parts 2 and 3.
This is what you're going to learn! :D

Class Scope

• The class will cover the stages which a program goes through from being some C source code until being assembly running natively on a processor.
  • Not covering interpreters or software virtual machines (e.g. java)
• This knowledge is useful to people who are trying to reverse engineer programs which potentially manipulate the process (e.g. packers). It also has applicability to understanding attack techniques (e.g. DLL injection) used in tools like Metasploit.
  • The more you know about forward engineering, the more you know about reverse engineering.
Rise (and fall) of binaries

- Source Code
- Compiler
- Object File
- Linker
- Binary
- Binary
- Loader
- Libraries
- Running Program
Compiler Overview

Source Code

Lexical analyze

Syntax analyze

Symbol table

Intermediate code generator

Optimization

Code generator

Object File

See notes for citation
Linker Overview

Object file a
- Headers
- Code chunk 1
- Code chunk 2
- Data chunk
- Other chunk

Object file b
- Headers
- Code chunk
- Data chunk
- Other chunk 1
- Other chunk 2

Executable or library
- Headers
- Code chunk 1a
- Code chunk 2a
- Code chunk 1b
- Data chunk a
- Data chunk b
- Other chunk a
- Other chunk 1b
- Other chunk 2b
Loader Overview

Files on Disk

WickedSweetApp.exe
- Code
- Data
- Import MyLib1
- Import MyLib2
- Import LibC

MyLib1.dll
- Code
- Data
- Import MyLib2
...

MyLib2.dll
- Code
- Data
...

Virtual Memory Address Space

Kernel

Executable Loader

Userspace

Stack
Heap
WickedSweetApp.exe
LibC
MyLib1.dll
MyLib2.dll
Compiler Drilldown

• Why bother?
• I think there's usefulness in seeing the full sequence of application creation, starting right at the beginning. The more you know about each phase the more the pieces of knowledge reinforce each other.
• Some of the knowledge will be applicable to other security areas for instance.
Syntax & Semantics

• (Taken from Concepts of Programming Languages 4th edition (which I will henceforth refer to as CPLv4) page 107)

• "The syntax of a programming language is the form of its expressions, statements, and program units."

• "Its semantics is the meaning of those expressions, statements, and program units"

• "For example, the syntax of a C if statement is

   if ( <expr>) <statement>

   The semantics of this statement form is that if the current value of the expression is true, the embedded statement is selected for execution."
Compiler Overview

Source Code -> Lexical analyze -> Syntax analyze -> Symbol table

Intermediate code generator -> Optimization

Code generator -> Object File

See notes for citation
Lexical Analysis aka Lexing aka Tokenizing

- Can be done with *nix tool Lex, FLEX (GNU lex), ANTRL (www.antlr.org), etc
- Turning a stream of characters into a stream of distinct *lexemes*, separated by some delimiter (often whitespace.)
- *Tokens* are then categories of lexemes. There can be many lexemes to a given token, or possible a single lexeme for a given token.
lexemes and tokens

• For the following statement (taken from CPLv4 pgs. 107/108)
• index = 2 * count + 17;
• The lexemes and tokens might be:

<table>
<thead>
<tr>
<th>lexeme</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>identifier</td>
</tr>
<tr>
<td>=</td>
<td>equal_sign</td>
</tr>
<tr>
<td>2</td>
<td>intliteral</td>
</tr>
<tr>
<td>*</td>
<td>mult_op</td>
</tr>
<tr>
<td>count</td>
<td>identifier</td>
</tr>
<tr>
<td>+</td>
<td>plus_op</td>
</tr>
<tr>
<td>17</td>
<td>intliteral</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>
Compiler Overview

Source Code

Lexical analyze

Syntax analyze

Symbol table

Intermediate code generator

Optimization

Code generator

Object File

See notes for citation
Syntactic Analysis & Context Free Grammars (CFGs)

• Done with tools like YACC (yet another compiler compiler), Bison (GNU yacc), ANTLR, or CUP (for java)
• A way to formally specify a syntax
• A commonly used form is Backus-Naur form (BNF)
• A grammar in BNF will be a series of rules, composed of terminal symbols, and non-terminal symbols. An example rule for an assignment statement might look like:

  • \(<assign> \rightarrow <var> = <expression>\)
  • The -> will be used to indicate that the symbol on the left hand side can be represented by the statement on the right hand side.
  • The < and > are used to enclose a non-terminal
  • In the above the = is a terminal. Terminals can also be given by tokens.
  • Of course, for the above to be of any use, you then need rules to describe how <var> and <expression> are formed
Mo gramma mo betta

• Rules can also have multiple possible right hand sides. These will often be specified on a new line started by |  
• There will generally be a special start symbol which we'll call <program>  
• In order to be able to specify arbitrarily long lists, you can use recursion in a rule. E.g.  
• <ident_list> -> identifier  
  | identifier , <ident_list>
Simple Grammar for Assignment Statements
(CPLv4 page 113)

\[
\begin{align*}
    \text{<assign>} & \rightarrow \text{id} = \text{<expr>} \\
    \text{id} & \rightarrow \quad A \\
    & \quad | B \\
    & \quad | C \\
    \text{<expr>} & \rightarrow \text{id} + \text{<expr>} \\
    & \quad | \text{id} \times \text{<expr>} \\
    & \quad | ( \text{<expr>} ) \\
    & \quad | \text{id}
\end{align*}
\]
Deriving a statement from the grammar
(CPLv4 page 113)

• $A = B \times (A + C)$
• $\langle assign \rangle \rightarrow \langle id \rangle = \langle expr \rangle$
  
  $A = \langle expr \rangle$
  $A = \langle id \rangle \times \langle expr \rangle$
  $A = B \times \langle expr \rangle$
  $A = B \times (\langle expr \rangle)$
  $A = B \times (\langle id \rangle + \langle expr \rangle)$
  $A = B \times (A + \langle expr \rangle)$
  $A = B \times (A + \langle id \rangle)$
  $A = B \times (A + C)$
Parse Trees
(aka Concrete Syntax Tree)
(Picture from CPLv4 page 114)

This is the parse tree for the sequence on the previous page, except I changed stuff to = from :=
Syntax Graphs
(from http://www.json.org)

<number>
  <int>
  | <int> <frac>
  int exp
  int frac exp

<int>
  digit
  digit1-9 digits
  - digit
  - digit1-9 digits

frac
  . digits

exp
  e digits

digits
  digit
  digit digits

e
  e
  e+
  e-
  E
  E+
  E-
Abstract Syntax Trees (ASTs)

• ASTs are a condensed/simplified form of the parse tree where the operators are internal nodes and never leaves.
• ASTs are more convenient form for subsequent stages to work with.
• Rather than having a grammar parser which just generates a parse tree and then converts it to an AST, you can create a parser which converts the input directly to an AST. This is called syntax-directed translation.
• In order to do that, the parser needs to be able to call some code to perform some action when it recognizes things.
AST Vs. Parse Tree

Figure 3.1
A parse tree for the simple statement
\[ A := B \times (A + C) \]

(CPLv4 page 114)
A slightly more complex example before we move on

**Example**

- Consider the following fragment of a programming language grammar:
  
  Program → Function-List
  Function-List → Function-Defn Function-List
  | Function-Defn
  Function-Defn → **fun** id ( Param-List ) Body
  Body → ‘{‘ Statement-List ‘}’

**Example (cont’d)**

- Consider an example program:
  
  ```
  fun main ()
  {
    statement
  }
  fun foo (int n)
  {
    n = n + 1
  }
  ```

See notes for citation
A slightly more complex example before we move on.

The main thing I want you to see is how an AST can have things like functions and parameters encapsulated in it.
Compiler Overview

Source Code

Lexical analyze

Syntax analyze

Symbol table

Intermediate code generator

Optimization

Code generator

Object File

See notes for citation
You can then fill in the parts of the tree with a simplified IR pseudo-assembly language, or a real assembly language. What I mean by "pseudo-assembly language" is something which looks more or less like most assembly languages look, but which is not something any hardware understands.
AST to Intermediate Representation (IR)

- The benefit of using the IR is that **the optimization can be done at this level, rather than only when dealing with the possibly complex real assembly language** (like x86/CISC).

- A good optimizer will optimize both the IR and the final assembly with assembly-specific optimization guidance, like the "Intel 64 and IA-32 Architectures Optimization Reference Manual"

- We're not going to cover the optimization stage in this class.
Different Levels of IR


<table>
<thead>
<tr>
<th>Original</th>
<th>High IR</th>
<th>Mid IR</th>
<th>Low IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>float a[10][20]; a[i][j+2];</td>
<td>t1 = a[i, j+2]</td>
<td>t1 = j + 2, t2 = i * 20, t3 = t1 + t2, t4 = 4 * t3, t5 = addr a, t6 = t5 + t4, t7 = *t6</td>
<td>r1 = [fp - 4], r2 = [r1 + 2], r3 = [fp - 8], r4 = r3 * 20, r5 = r4 + r2, r6 = 4 * r5, r7 = fp - 216, f1 = [r7 + r6]</td>
</tr>
</tbody>
</table>

- Fairly language dependent
- Ideally language and machine independent
- Fairly machine dependent
Using Code to Generate IR from AST (postorder)

\[ a + (b - c) \times d \]

(It's sometimes helpful to think about the AST in prefix/postfix operator form, rather than infix, since you can see that's kind of how the code ends up getting generated)

See notes for citation
Code to Generate IR from AST

typedef struct _tnode {
    char label;
    struct _tnode *lchild, *rchild;
} tnode, *tree;
void GenerateCode(tree t, int resultRegNum)
{
  if (IsArithmeticOp(t->label)) {
    GenerateCode(t->left, resultRegNum);
    GenerateCode(t->right, resultRegNum + 1);
    GenerateArithmeticOp(t->label, resultRegNum, resultRegNum + 1);
  } else {
    GenerateLoad(t->label, resultRegNum);
  }
}

bool IsArithmeticOp(char ch)
{
  return ((ch == '+') || (ch == '-') || (ch == '*') || (ch == '/'));
}

void GenerateArithmeticOp(char op, int reg1, int reg2)
{
  char *opCode;
  switch (op) {
    case '+': opCode = "ADD"; break;
    case '-': opCode = "SUB"; break;
    case '*': opCode = "MUL"; break;
    case '/': opCode = "DIV"; break;
  }
  printf("%s R%d, R%d\n", opCode, reg1, reg2);
}

void GenerateLoad(char c, int reg)
{
  printf("LOAD %c, R%d\n", c, reg);
}
Different Paths to Code Generation

Saw a quick example of this

Abstract Syntax Tree (AST)

Intermediate Representation (IR)

Assembly Language

Machine Code

Assembly Language

Abstract Assembly Tree (AAT)

Going to show this path next, since I have a good example document
Abstract Assembly Trees (AATs)

• Once you have an AST, it is useful to generate an AAT
Abstract Assembly Trees (AATs)

- There are two subtypes of an AAT, Expression Trees (for values) and Statement Trees (for actions). You can think of the result of expression trees as being some value which is put on the stack when the tree is processed and collapsed. And the result of statement trees are the organization of which code goes where.
Constant Expression Trees

• Constant(4), Constant(0xBEEF), etc
• In x86 we call constants embedded in the instruction stream "immediates"
Register Expression Trees

• Register( Frame Pointer ) = Register( FP )
• Register( Stack Pointer ) = Register( SP )
• Register( SomeTempRegister ), where SomeTempRegister eventually gets translated into a machine-specific register
• Register( Result Register )
• By convention on x86 we would know that eventually that should turn into the EAX register, since that's where function result/return values are stored
Operator Expression Trees

• Now we actually have a tree. The root would be the operator, and the right and left subtrees/leaves are the operands

• Operators are things like +, -, *, /, <, ≤, >, ≥, &&, ||, !, ==
Memory Expression Trees

- Indicates dereferencing some memory address, and returning whatever's in memory at that address.

```
Memory
   -
  Memory
    -
Constant(0x12FFD0)
Register(FP)  Constant(4)
```

Returns whatever's in memory at the address given by the frame pointer (ebp on x86) minus 4. From Intro x86 we might think that could be one of the local variables of a function.
Call Expression Trees

- Shown with the name, and then a sub-expression for each of the input parameters.
- Recall that all expression trees put some value onto the stack, so remember that the return value from the call is put onto the stack.

**Example:** $\text{add}(2, 3)$

```
CallExpression("add")
       Constant(2)  Constant(3)
```

**Example:** $\text{printf}("a = %d, %d", a, 1)$

```
CallExpression("printf")
       Constant(0x41573C)  Memory
                        Address of the static location where the format string is stored
                        -
                        Register(FP)  Constant(4)
                        Address of the local variable $a$
```
Move Statement Trees

• Switching gears to Statement trees. They are used to achieve some goal, not to return some value. Their values can come from some Expression subtrees.

• Move tree puts data (from register, memory, or constant) into a register or memory. Note, this doesn't prevent memory to memory move which can't be done with a normal x86 MOV instruction.

• The left subtree is the destination, and the right subtree is the source. This is like with Intel syntax assembly.
Move Statement Trees 2

\[
\text{Register(Reg1) Constant(0xf005ba11)} \\
\text{Register(Reg1) Register(Reg2)} \\
\]

\[
a = b; \\
\]

\[
\text{Memory - Register(FP) Constant(4)} \\
\]

\[
a \\
\]

\[
\text{Memory - Register(FP) Constant(8)} \\
\]

\[
b \\
\]
Label & Jump Statement Trees

- Label nodes represent an assembly label (which you might also think of just like a normal label in C). Used for things like jumps, conditional jumps, and calls.
- Jump trees are an unconditional jump to the given label
Conditional Jump Statement Trees

• An unconditional jump to the given label. The jump is taken if the sub-expression evaluates to true. If not, then the entire statement is a no-op.
Sequential Statement Trees

- Just execute the left subtree and then the right subtree. Used to maintain ordering of statements. This differs from the AST where whatever was at the leaves furthest from the root was supposed to occur first. But one can see how we would just order from AST to AAT appropriately.
Call Statement Trees

• Basically the exact same thing as call expression trees, except that statements don't return a value and expressions do. So you can think of this like calls to functions which return void.

• Now that you've seen Labels, you can think of the function name being called as just a target label.

• Documentation we see later calls the Call Statement a "Procedure Call" and Call Expressions as "Function Call". Meh. I prefer to continue with the statement vs. expression differentiation.
Compiler Overview

Source Code → Lexical analyze → Syntax analyze → Symbol table → Intermediate code generator → Optimization → Code generator → Object File

See notes for citation
AAT Direct To x86 Assembly

- To go from AAT to assembly, we use a "tiling strategy" whereby we group portions of the AAT and generate assembly for them. The above link shows a tiling strategy for directly outputting non-optimal, but simple and straightforward to understand x86 assembly code, so that's why we're going to use it
Constant/Register Expressions

- For any Constant(x) we will emit the x86 instruction "push x"
  - Constant(5) = "push 5"
- For Register() expressions, what register we will emit will depend on whether it's something special or not, but for now we'll assume we're just dealing with the special ones below.
  - Register(FP) = "push ebp"
  - Register(SP) = "push esp"
Operator Expressions
(remember: for expression trees, we want the result value on top of the stack when the asm is done)

This is in keeping with the simple method for the Register and Constant expressions, where the result of collapsing the tree is just put on the top of the stack

```
push ebp
push 4
pop eax
add [esp], eax
```
Operator Expressions

(remember: for expression trees, we want the result value on top of the stack when the asm is done)

> 

Register(FP)  Constant(4)

push ebp
push 4
pop eax
cmp [esp],eax
setg [esp]

Memory Expressions
(remember: for expression trees, we want the result value on top of the stack when the asm is done)

Just need to dereference whatever's on the top of the stack and store it back to the top of the stack.
Call Expressions
(remember: for expression trees, we want the result value on top of the stack when the asm is done)

- CallExpression("Foo")
- Constant(1)
- Constant(5)

- Note that we changed the ordering of traversal of the graph to push the parameters in the x86 expected right to left order.

- The "add esp, 8" is indicative of the cdecl calling convention (as we learned in Intro x86), and the 8 would just be an assumed 4 byte size per parameters multiplied by the number of parameters pushed.

- How would the assembly be different if this was a call statement?
- How would the assembly be different if this was a stdcall calling convention?
Move (to Register) Statements

(remember: for statement trees, we just want to perform some action, no net changes to the stack should occur)

Move

Register(Reg1) Constant(4)

push 4 pop Reg1

Note that the stack is empty at the end.
Because Statement trees don't result in values, they just cause actions.
Move (to Memory) Statements

(remember: for statement trees, we just want to perform some action, no net changes to the stack should occur)
(I'm pretty sure the x86.pdf is wrong for this case so I substituted my own code)

Note that the stack is empty at the end.
Because Statement trees don't result in values, they just cause actions.
Labels & Jump Statements

• For Label("SomeLabel") we just output "SomeLabel:" which will be an assembly label.
• For Jump("SomeLabel") we just output "jmp SomeLabel" (assuming the assembler will handle the generation of the correct behind the scenes relative or absolute jump.)
Conditional Jump Statements
(remember: for statement trees, we just want to perform some action, no net changes to the stack should occur)

```
CondJump("SomeLabel")
```

- \texttt{Register(FP)}
- \texttt{Constant(4)}

```
push ebp
push 4
pop eax
cmp [esp], eax
setg [esp]
```

```
push ebp
push 4
pop eax
cmp eax, 0
jg SomeLabel
```

Note that the stack is empty at the end.
Because Statement trees don't result in values, they just cause actions.
Compiler Overview

Source Code → Lexical analyze

Lexical analyze → Syntax analyze

Syntax analyze → Symbol table

Symbol table → Intermediate code generator

Intermediate code generator → Code generator

Code generator → Optimization

Optimization → Object File

See notes for citation
Symbols

• I added the dashed line from the symbol table to the object file just to say that generally there will be *some* form of symbol table in the object file. It need not be exactly the same as is used in the other stages, but there will be something.

• The symbols table is basically a little database where the various stages can store information about the names & types of variables & functions.
Organizing Code/Data Into an Object File

• The compiler and linker must therefore have some protocol/format specification embedded in the object file whereby the compiler knows it can say "this code needs to access this symbol" (whether the symbol is code or data), and the linker then knows how to search for these unresolved symbols at link time when it's putting all the objects together into a final binary.
Organizing Code/Data Into an Object File

• When the linker can't that's obviously where "unresolved symbol <bla>" errors come from.
• Also this sort of implies that the object files need to be able to say "yeah, I have that symbol, and it's located within my binary here"
Quick Example: SimpleSimonTheThird

- This is jumping ahead slightly because it shows some info about binary formats, but I just felt like justifying the claim about contained and uncontained symbols.
SimpleSimonTheThird.o
(readelf -s to print symbol table)

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>0</td>
<td>FILE</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>ABS</td>
<td>SimpleSimonTheThird.c</td>
</tr>
<tr>
<td>2</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00000000</td>
<td>61</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>1</td>
<td>main</td>
</tr>
<tr>
<td>9</td>
<td>00000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>getPie</td>
</tr>
<tr>
<td>10</td>
<td>00000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>puts</td>
</tr>
</tbody>
</table>

Undefined symbols

printf equivalent
PieMan.o
(readelf -s to print symbol table)

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>0</td>
<td>FILE</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>ABS</td>
<td>PieMan.c</td>
</tr>
<tr>
<td>2</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>00000000</td>
<td>23</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>1</td>
<td>getPie</td>
</tr>
</tbody>
</table>
main() is in section `.text`, offset 0

```
<table>
<thead>
<tr>
<th>pFile</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000000C</td>
<td>Symbol Table Index</td>
<td></td>
</tr>
<tr>
<td>00000E9A</td>
<td>5F 6D 61 69</td>
<td>Short Name</td>
<td></td>
</tr>
<tr>
<td>00000E9E</td>
<td>6E 00 00 00</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>00000EA6</td>
<td>0004</td>
<td>Section Number</td>
<td></td>
</tr>
<tr>
<td>00000EA8</td>
<td>0020</td>
<td>Type</td>
<td>DT_FUNCTION</td>
</tr>
<tr>
<td>00000EAA</td>
<td>02</td>
<td>Storage Class</td>
<td>IMAGE_SYM_CLASS_EXTERNAL</td>
</tr>
<tr>
<td>00000EAB</td>
<td>00</td>
<td>Number of Aux Symbols</td>
<td></td>
</tr>
<tr>
<td>00000EAC</td>
<td>00000000</td>
<td>Long Name</td>
<td>__imp__printf</td>
</tr>
<tr>
<td>00000EB0</td>
<td>00000004</td>
<td>Offset into String Table</td>
<td></td>
</tr>
<tr>
<td>00000EB4</td>
<td>00000000</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>00000EB8</td>
<td>0000</td>
<td>Section Number</td>
<td></td>
</tr>
<tr>
<td>00000EBA</td>
<td>0000</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>00000EBC</td>
<td>02</td>
<td>Storage Class</td>
<td>IMAGE_SYM_CLASS_EXTERNAL</td>
</tr>
<tr>
<td>00000EBD</td>
<td>00</td>
<td>Number of Aux Symbols</td>
<td></td>
</tr>
<tr>
<td>00000EFE</td>
<td>5F 67 65 74</td>
<td>Short Name</td>
<td></td>
</tr>
<tr>
<td>00000EC2</td>
<td>50 69 65 00</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>00000EC6</td>
<td>00000000</td>
<td>Section Number</td>
<td></td>
</tr>
<tr>
<td>00000ECA</td>
<td>0000</td>
<td>Type</td>
<td>DT_FUNCTION</td>
</tr>
<tr>
<td>00000ECC</td>
<td>0020</td>
<td>Storage Class</td>
<td>IMAGE_SYM_CLASS_EXTERNAL</td>
</tr>
<tr>
<td>00000ECF</td>
<td>00</td>
<td>Number of Aux Symbols</td>
<td></td>
</tr>
</tbody>
</table>
```
PieMan.obj

getPie() is in section .text, offset 0
Git along lil doggie!

• We're going to get back into more about compiler options, linker options, and linking after we see more about the binary formats which are used to store binaries pre and post-linking.