Chapter 6

Code Construct Assembly
Code Construct Concept

- Entire disassembled program -> thousands or millions lines -> too tedious to analyze each single line
- Obtain high-level picture of code functionality
- Analyze instructions in groups and focus on individual only on an as-needed basis
- This skill takes time to develop
Definition: *code construct* is a code abstraction level that defines a functional property but not the details of implementations.

Goal: go from disassembly to high-level constructs.

Examine the difference between compilers -> impacts on how particular constructs appear in assembly.

Focus on C language.
Global vs. Local Variables

- Global variables stored in either .data or .bss section of process (ref. by memory address)

  The BSS segment, also known as uninitialized data, is usually adjacent to the data segment. The BSS segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

- Local variables stored on stack (ref. by stack address)

- Difference between global and local variables in C code is small

- Their assembly is quite different
Global vs. Local Variables

```c
int x = 1;
int y = 2;
void a()
{
    x = x+y;
    printf("Total = %d\n",x);
}
int main(){a();}
```

```c
void a()
{
    int x = 1;    // Local
    int y = 2;
    x = x+y;
    printf("Total = %d\n",x);
}
int main() {a();}
```
int x = 1;
int y = 2;

void a()
{
    x = x+y;
    printf("Total = %d\n",x);
}

int main()
{
    a();
}

Memory Location 0x40CF60

00401003  mov    eax, dword_40CF60
00401008  add    eax, dword_40C000
0040100E  mov    dword_40CF60, eax ①
00401013  mov    ecx, dword_40CF60
00401019  push   ecx
0040101A  push   offset aTotalD  ;"tot
0040101F  call   printf
00401021  push   offset aTotalD  ;"total = %d\n"
Local variable vs. Global variable (MinGW GCC)

Question
Arithmetic Operations

```c
int a = 0;
int b = 1;
if b = a + 11;
a = a - b;
a--; b++; b = a % 3;
```
If Statements

```c
void f(){
    int x = 1;
    int y = 2;
    if (x==y) {
        printf("x equals y.\n");
    } else {
        printf("x is not ");
    }
}

int main() { f();}
```
Jump not equal to zero; otherwise, means x equals y
int i;
for(i=0; i<100; i++)
{
    printf("i equals %d\n", i);
}

Look for:
- Initialization
- Comparison
- Execution
- Increment

- Initialize to zero
- Then compare
- Compare with 64h
- Jump if greater

Execution:
- printf

Clean register
- Move to eax
- Add 1
- Move to var 4

Increase by one
While Loops

- Malware: loop until a condition is met

```c
int status=0;
int result = 0;

while(status == 0){
    result = performAction();
    status = checkResult(result);
}
```

```
00401036   mov    [ebp+var_4], 0
0040103D   mov    [ebp+var_8], 0
00401044   jmp    loc_401044:
00401044   cmp    [ebp+var_4], 0
00401048   jnz    short loc_401063
0040104A   call   performAction
0040104F   mov    [ebp+var_8], eax
00401052   mov    eax, [ebp+var_8]
00401055   push   eax
00401056   call   checkResult
0040105B   add    esp, 4
0040105E   mov    [ebp+var_4], eax
00401061   jmp    short loc_401044
```

Only exit From the loop
**Function call conventions**

- **Call conventions:** governs the way function call occurs.
  - Order parameters placed on the stack/registers
  - Caller or callee is responsible for cleaning up the stack
  - Depends on compiler
  - **Cdecl:** parameters pushed onto the stack from right to left, caller cleans up the stack when complete, return value stored in EAX

```c
int test(int x, int y, int z);
int a, b, c, ret;
ret = test(a, b, c);
```

1. Push on to the stack from right to left
2. Stack cleared up by the caller
3. Add esp, 12 (3 parameters * 4 bytes each) and grow the stack down
Function call conventions

- **Stdcall**
  - callee is required for clearing up (the function being called)
  - is used to call Win32 API functions

- **Fastcall**
  - First few arguments to functions are to be passed in registers (ECX/EDX), when possible. (calling function is usually responsible for clearing up)
  - This calling convention only applies to the x86 architecture.
  - Hybrid approach – more efficient (no stack)
Compiler Dependence: Push vs. Move

int adder(int a, int b)
{
    return a+b;
}

void main()
{
    int x = 1;
    int y = 2;

    printf("the function returned the number %d\n", adder(x,y));
}

Stack pointer is restored after push
Because we used push

Move: Stack pointer never altered
(direct move to stack address)
Switch Statements (Naïve)

Backdoors commonly select from a series of actions using a single byte value.

```c
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Difficult to identify without the help of graph: a group of if statements

Series of Conditions:
- Good if only a few cases
- Slow if a lot cases

Figure with better resolution in the book pdf p. 157
Jump Table (optimizer)

Jump Table:
• Lookup branch target from a table
• edx multiplied by 4 and added to the base of the jump table 0x401088 for jumping
  • 4 because each entry is an address of 4 bytes in size
Arrays

- Malware sometimes uses an array of pointers to strings of hostnames.

```c
void main()
{
    int i;
    int a[5];

    for(i = 0; i<5; i++)
    {
        a[i] = 1;
        b[i] = 1;
    }
}
```

- `Var_14`: base for `a`
- `Dword_40A0000`: base for `b`
- `ecx` used as index
- `*4` since size is 4 bytes each

**Question:** why base for `a` and `b` look different?