Malware Analysis

Recognizing C Code Construct in Assembly

COMPILED BY: DHARMESH DAVE | ASST. PROF. | NATIONAL FORENSIC SCIENCES UNIVERSITY

Introduction

- Expected to have the knowledge of the x86 architecture and its most common instructions. But successful reverse engineers **do not evaluate each instruction** individually unless they must.
- The process is just too tedious, and the instructions for an entire disassembled program can number in the thousands or even millions.
- As a malware analyst, you must be able to obtain a **high-level picture of code functionality** by analyzing instructions as groups, focusing on individual instructions only as needed. This skill takes time to develop.

Introduction of C Code Construct

- Malware is typically developed using a high-level language, **most commonly C**.
- A code construct is a code abstraction level that defines a functional property but not the details of its implementation. Examples of code constructs include loops, if statements, linked lists, switch statements, and so on.
- we'll also examine the differences between compilers, because compiler versions and settings can impact how a particular construct appears in disassembly.

Global variable vs local variable

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

```
00401003 mov eax, dword_40CF60
00401008 add eax, dword_40C000
0040100E mov dword_40CF60, eax
00401013 mov ecx, dword_40CF60
00401019 push ecx
0040101A push offset aTotalD ;"total = %d\n"
0040101F call printf
```

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

```
00401006 mov dword ptr [ebp-4], 1
0040100D mov dword ptr [ebp-8], 2
00401014 mov eax, [ebp-4]
00401017 add eax, [ebp-8]
0040101A mov [ebp-4], eax
0040101D mov ecx, [ebp-4]
00401020 push ecx
00401021 push offset aTotalD ; "total = %d\n"
00401026 call printf
```

The **global variables** are referenced by memory addresses, and the **local variables** are referenced by the stack addresses.

Arithmetic Operations

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
```

```
00401006 mov [ebp+var_4], 0
0040100D mov [ebp+var_8], 1
00401014 mov eax, [ebp+var_4]
00401017 add eax, 0Bh
0040101A mov [ebp+var_4], eax
0040101D mov ecx, [ebp+var_4]
00401020 sub ecx, [ebp+var_8]
00401023 mov [ebp+var_4], ecx
00401026 mov edx, [ebp+var_4]
00401029 sub edx, 1
0040102C mov [ebp+var_4], edx
0040102F mov eax, [ebp+var_8]
00401032 add eax, 1
00401035 mov [ebp+var_8], eax
00401038 mov eax, [ebp+var_4]
0040103B cdq
```

IDA Pro has labeled a as var_4 and b as var_8. First, var_4 and var_8 are initialized to 0 and 1, respectively

IF Condition

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

```
00401006 mov [ebp+var_8], 1
0040100D mov [ebp+var_4], 2
00401014 mov eax, [ebp+var_8]
00401017 cmp eax, [ebp+var_4]
0040101A jnz short loc_40102B
0040101C push offset aXEqualsY_ ; "x equals y.\n"
00401021 call printf
00401026 add esp, 4
00401029 jmp short loc_401038 ⍰
0040102B loc_40102B:
0040102B push offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030 call printf
```

Nested If conditions

```
int x = 0;
int y = 1;
int z = 2;
if(x == y){
    if(z==0){
        printf("z is zero and x = y.\n");
    }else{
        printf("z is non-zero and x = y.\n");
    }
}else{
    if(z==0){
        printf("z zero and x != y.\n");
    }else{
        printf("z non-zero and x != y.\n");
    }
}
```

Nested If conditions

```
00401006 mov [ebp+var_8], 0
0040100D mov [ebp+var_4], 1
00401014 mov [ebp+var_C], 2
0040101B mov eax, [ebp+var_8]
0040101E cmp eax, [ebp+var_4]
00401021 jnz short loc_401047
00401023 cmp [ebp+var_C], 0
00401027 jnz short loc_401038
00401029 push offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E call printf
00401033 add esp, 4
00401036 jmp short loc_401045
00401038 loc_401038:
00401038 push offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D call printf
00401042 add esp, 4
00401045 loc_401045:
00401045 jmp short loc_401069
00401047 loc_401047:
00401047 cmp [ebp+var_C], 0
0040104B jnz short loc_40105C
0040104D push offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052 call printf
00401057 add esp, 4
0040105A jmp short loc_401069
0040105C loc_40105C:
0040105C push offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061 call printf00401061
```

For loop

```
int i;

for(i=0; i<100; i++)
{
        printf("i equals %d\n", i);
}
```

```
00401004 mov [ebp+var_4], 0
0040100B jmp short loc_401016
0040100D loc_40100D:
0040100D mov eax, [ebp+var_4]
00401010 add eax, 1
00401013 mov [ebp+var_4], eax
00401016 loc_401016:
00401016 cmp [ebp+var_4], 64h
0040101A jge short loc_40102F
0040101C mov ecx, [ebp+var_4]
0040101F push ecx
00401020 push offset aID ; "i equals %d\n"
00401025 call printf
0040102A add esp, 8
0040102D jmp short loc_40100D
```

While loop

```
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 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
```

Array

```c
int b[5] = {123,87,487,7,978};

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

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

```
00401006 mov [ebp+var_18], 0
0040100D jmp short loc_401018
0040100F loc_40100F:
0040100F mov eax, [ebp+var_18]
00401012 add eax, 1
00401015 mov [ebp+var_18], eax
00401018 loc_401018:
00401018 cmp [ebp+var_18], 5
0040101C jge short loc_401037
0040101E mov ecx, [ebp+var_18]
00401021 mov edx, [ebp+var_18]
00401024 mov [ebp+ecx*4+var_14], edx
00401028 mov eax, [ebp+var_18]
0040102B mov ecx, [ebp+var_18]
0040102E mov dword_40A000[ecx*4], eax
00401035 jmp short loc_40100F
```

Base address of **array b** corresponds to **dword_40A000**. base address of **array a** corresponds to **var_14**. Because of int array the size is 4 for both array. ecx is used as the index for both the array. Resulting value is added to the base address.

Different compiler in disassembly

```
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));
}
```

Compilers may also choose to use <u>different instructions</u> to perform the same operation, usually when the compiler decides to **<u>move rather than push</u>** things onto the stack.

Different compiler in disassembly

Visual Studio Version	GCC Version
00401746 mov [ebp+var_4], 1	00401085 mov [ebp+var_4], 1
0040174D mov [ebp+var_8], 2	0040108C mov [ebp+var_8], 2
00401754 mov eax, [ebp+var_8]	00401093 mov eax, [ebp+var_8]
00401757 **push eax**	00401096 **mov [esp+4], eax**
00401758 mov ecx, [ebp+var_4]	0040109A mov eax, [ebp+var_4]
0040175B **push ecx**	0040109D **mov [esp], eax**
0040175C call adder	004010A0 call adder
00401761 add esp, 8	
00401764 **push eax**	004010A5 **mov [esp+4], eax**
00401765 **push offset TheFunctionRet**	004010A9 **mov [esp], offset TheFunctionRet**
0040176A call **ds:printf**	004010B0 call **printf**

Different calling conventions used by two different compilers: Microsoft Visual Studio and GNU Compiler Collection (GCC). **On the left**, the parameters for *adder and printf are **pushed** onto the stack before the call*. **On the right**, the parameters are ***moved** onto the stack before the call*.

Switch case (compiler-1)

```
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;
}
```

```
00401013 cmp [ebp+var_8], 1
00401017 jz short loc_401027
00401019 cmp [ebp+var_8], 2
0040101D jz short loc_40103D
0040101F cmp [ebp+var_8], 3
00401023 jz short loc_401053
00401025 jmp short loc_401067
00401027 loc_401027:
00401027 mov ecx, [ebp+var_4] ⍰
0040102A add ecx, 1
0040102D push ecx
0040102E push offset unk_40C000 ; i = %d
00401033 call printf
00401038 add esp, 8
0040103B jmp short loc_401067
0040103D loc_40103D:
0040103D mov edx, [ebp+var_4] ⍰
00401040 add edx, 2
00401043 push edx
00401044 push offset unk_40C004 ; i = %d
00401049 call printf
0040104E add esp, 8
00401051 jmp short loc_401067
00401053 loc_401053:
00401053 mov eax, [ebp+var_4] ⍰
00401056 add eax, 3
00401059 push eax
0040105A push offset unk_40C008 ; i = %d
0040105F call printf
00401064 add esp, 8
```

Switch case (compiler-2)

```
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;
        case 4:
                printf("i = %d", i+3);
                break;
        default:
        break;
}
```

In this jump instruction, edx is multiplied by 4 and added to the base of the **jump table (0x401088)** to determine which case code block to jump to. *It is multiplied by 4 because each entry in the jump table is an address that is 4 bytes in size.*

```
00401016 sub ecx, 1
00401019 mov [ebp+var_8], ecx
0040101C cmp [ebp+var_8], 3
00401020 ja short loc_401082
00401022 mov edx, [ebp+var_8]
00401025 jmp ds:off_401088[edx*4] ⍰
0040102C loc_40102C:
…
00401040 jmp short loc_401082
00401042 loc_401042:
…
00401056 jmp short loc_401082
00401058 loc_401058:
…
0040106C jmp short loc_401082
0040106E loc_40106E:
…
00401082 loc_401082:
00401082 xor eax, eax
00401084 mov esp, ebp
00401086 pop ebp
00401087 retn
00401087 _main endp
00401088 off_401088 dd offset loc_40102C
0040108C                dd offset loc_401042
00401090                dd offset loc_401058
00401094                dd offset loc_40106E
```

References

- Practical Malware Analysis by Michael Sikorski