

Reverse code engineering

Powerfull knowledge, lot of fun and legal for several purposes!

Native binary code (assembler)

We will work with OllyDbg (www.ollydbg.de) program that is easy-to-use disassembler and debugger.

OllyDbg shortcuts & most important commands

F3 ... Open binary file

F2 ... Toggle breakpoint (on opcodes, or double click)

F9 ... Run debugged program

Ctrl+F2 ... Restart program, temporary changes are lost!

F8 ... Step over

F7 ... Step into

Spacebar or double click ... allows to set new opcode

Alt+BkSp ... Undo change

Rightclick->Search for->All referenced text strings ... Constant text strings referenced in code.

Rightclick->Find references to->Address constant ... will find references to particular memory elsewhere in the code – use when you like to know where the memory is set or changed.

Ctrl+F1 ... Help on API (WIN32 API help file already prepared in OllyDbg directory (WIN32.HLP))

; ... add or edit your comment for specific code line

Rightclick->Copy to executable->All modifications (or Selection) ... make changes permanent. New window with modified code is opened. **Rightclick->Save file** to write patched binary to disk.

Registers (FPU):

Z – zero flag, C – carry flag, S – sign flag. Invert bit flag by double click.

EIP ... next address to execute (instruction pointer)

EBX ... usually loop counter

Startup resources

The Reverse Code Engineering Community: <http://www.reverse-engineering.net/>

Tutorials for You: <http://www.tuts4you.com>

RE on Wikipedia: http://en.wikipedia.org/wiki/Reverse_engineering

Some hints

- Conditional branching: usually realized by two consecutive operations
 - Comparison operation setting Flags register
 - Conditional jumping operation to address based on Flags (Branch 1)
 - If not jumped then Branch 2 code is present on the next instruction, or unconditional jump JMP to Branch 2.
- Comparison operation

- CMP EAX, -1 - will set flag(s) in Registers, Zero and Sign flags are usually of interest. If two values are same (EAX == -1), Zero flag is set to 1.
 - TEST A, B (usually TEST EAX, EAX) – logical AND operation, results not saved, Flags are set. TEST EAX, EAX will test if value in EAX is equal to 0. If EAX == 0 then Zero flag == 1, 0 otherwise.
 - Jump operation
 - Unconditional JMP – jump every time
 - Conditional - based on the current value of flag(s)
- | | | |
|-------|-----------------------------------|-----------------------------------|
| JA* | Jump if (unsigned) above | - CF=0 and ZF=0 |
| JB* | Jump if (unsigned) below | - CF=1 |
| JE** | Jump if equal | - ZF=1 |
| JG* | Jump if (signed) greater | - ZF=0 and SF=OF (SF = Sign Flag) |
| JGE* | Jump if (signed) greater or equal | - SF=OF |
| JL* | Jump if (signed) less | - SF != OF (!= is not) |
| JLE* | Jump if (signed) less or equal | - ZF=1 and OF != OF |
| JMP** | Jump | - Jumps always |
| JNE** | Jump if not equal | - ZF=0 |

Java (Card) bytecode

Intermediate code interpreted by virtual machine (see JavaCard222_ops.pdf).

- Usually easier to understand than assembler code.
- Stack-based oriented execution, no registers are used.
- Operation takes its operands from stack and return result there.

```
JAVACARD SOURCE CODE
// ENCRYPT INCOMING BUFFER
void Encrypt(APDU apdu) {
    byte[]    apdubuf = apdu.getBuffer();
    short     dataLen = apdu.setIncomingAndReceive();
    short     i;

    // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
    if ((dataLen % 8) != 0) ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

    // ENCRYPT INCOMING BUFFER
    m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen, m_ramArray, (short) 0);

    // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
    Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf, ISO7816.OFFSET_CDATA, dataLen);

    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
}
}
```

```
JAVACARD BYTECODE
.method Encrypt(Ljavacard/framework/APDU;)V 129 {
    .stack 6;
    .locals 3;

    .descriptor    Ljavacard/framework/APDU;        0.10;
    L0:             aload_1;
                   invokevirtual 30;
                   astore_2;
                   aload_1;
                   invokevirtual 42;
                   sstore_3;
                   sload_3;
                   bspush 8;
                   srem;
                   ifeq L2;
    L1:             sspush 26384;
                   invokestatic 41;
                   goto L2;
    L2:             getfield_a_this 1;
                   aload_2;
                   sconst_5;
                   sload_3;
                   getfield_a_this 10;
                   sconst_0;
                   invokevirtual 43;
                   pop;
                   getfield_a_this 10;
                   sconst_0;
                   aload_2;
                   sconst_5;
                   sload_3;
                   invokestatic 44;
                   pop;
                   aload_1;
                   sconst_5;
                   sload_3;
                   invokevirtual 45;
                   return;
}
}
```

