Building Custom Disassemblers

Instruction Set Reverse Engineering

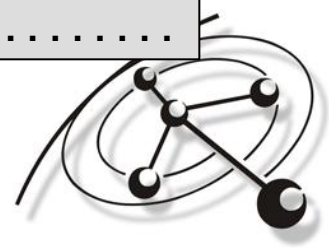
Invent & Verify

Agenda

- Motivation
- Introduction to the playing field
- How to obtain byte code
- Recognizing basic properties of the byte code
- Implementing an IDA Pro processor module
- Calling Conventions
- Advanced Addressing Modes
- Reading code you are not supposed to

Motivation – General

```
00000d70h: 00 00 53 49 4D 41 54 49 43 00 49 45 43 00 00 00 ; ..SIMATIC.IEC...
00000d80h: 00 00 53 37 5F 4C 56 00 00 00 20 00 2C 6D 00 00 ; ..S7_LV... .,m..
00000d90h: 00 00 00 00 00 00 68 1D 68 2C 41 61 00 02 FB 70 ; ......h.h,Aa..ûp
00000da0h: 07 4C 70 0B 00 02 FB 78 03 78 7E 43 00 98 38 09 ; .Lp...ûx.x~C.˜8.
00000db0h: 01 2D 35 60 39 A0 00 40 00 9C FF B8 00 05 68 1D ; .-5`9 .@.œÿ¸..h.
00000dc0h: 41 43 02 82 FB 78 03 78 68 1C 00 42 02 82 68 2D ; AC.,ûx.xh..B.,h-
00000dd0h: FF B8 00 06 FB 70 07 4A 70 0B 00 02 FB 78 03 78 ; ÿ¸..ûp.Jp...ûx.x
00000de0h: 7E 42 00 10 30 03 00 03 21 A0 7E 42 00 10 30 03 ; ~B..0...! ~B..0.
00000df0h: 00 04 41 62 00 02 21 C0 00 62 00 02 FF B8 00 0B ; ..Ab..!À.b..ÿ¸..
00000e00h: 38 07 00 00 00 01 FB 79 03 7A 7E 57 00 0C 70 0B ; 8.....ûy.z~W..p.
00000e10h: 00 09 38 07 00 00 00 00 FB 78 03 7A 7E 47 00 0C ; ..8.....ûx.z~G..
00000e20h: 68 1C FB 78 03 78 41 44 02 82 FB 70 07 52 70 0B ; h.ûx.xAD.,ûp.Rp.
00000e30h: 00 02 00 61 00 02 68 2C 65 00 01 00 00 02 00 00 ; ...a..h,e.......
00000e40h: 00 05 05 50 01 00 A4 00 04 00 12 00 1D 00 33 00 ; ...P..¤.......3.
00000e50h: 3C 00 04 00 0C 00 4A 07 01 01 EA 08 00 00 06 08 ; <.....J...ê.....
00000e60h: 00 00 0E 00 00 00 88 00 00 00 12 00 03 70 25 CF ; ......ˆ......p%Ï
00000e70h: 19 4B 03 70 25 CF 19 4B 00 00 00 00 53 49 4D 41 ; .K.p%Ï.K....SIMA
00000e80h: 54 49 43 00 49 45 43 00 00 00 00 00 57 45 5F 54 ; TIC.IEC.....WE_T
00000e90h: 45 00 00 00 20 00 D2 97 00 00 00 00 00 00 00 00 ; E... .Ò—........
```

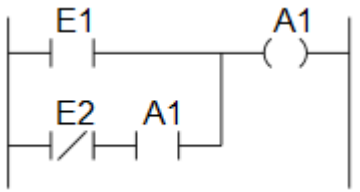


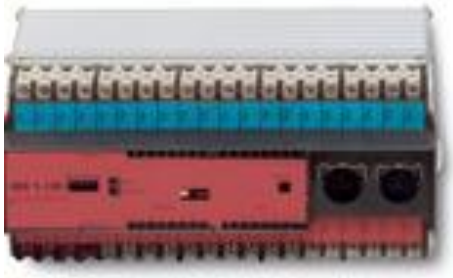
Motivation – Specific

- Frank Boldewin discovered interesting payload functionality within the W32.Stuxnet malware
  - July 14, 2010*
- Everyone started speculating
- Few started looking at the actual code
- Within one component, blobs of programmable logic controller (PLC) code were discovered
- This code needed to get disassembled and analyzed
  - Waiting for third parties to trickle information through small publications wasn't an option.

* http://www.wilderssecurity.com/showpost.php?p=1712134&postcount=22

Introduction to PLCs

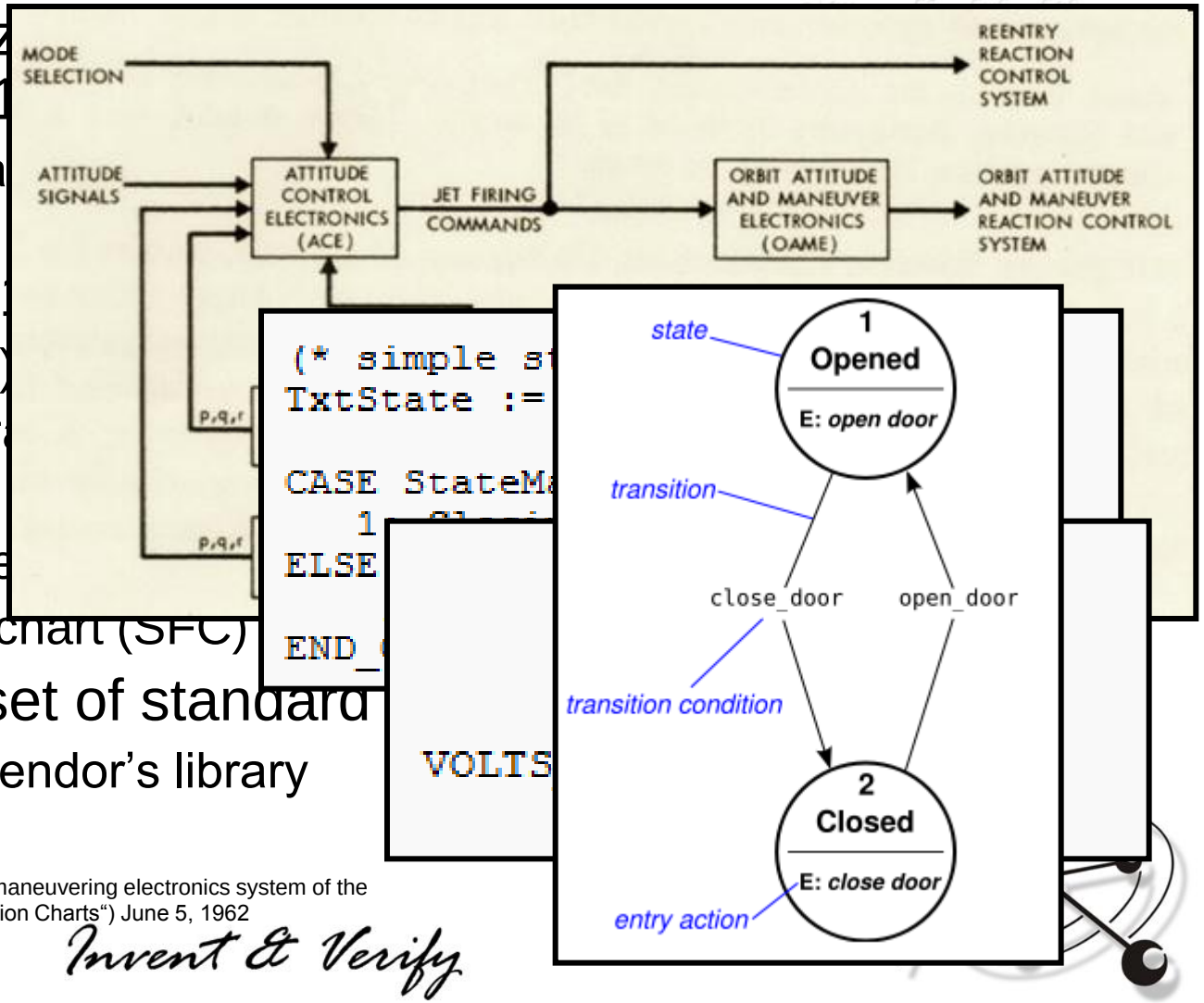




- PLCs are essentially programmable input/output controllers
  - Designed to mirror electrical wiring, to be used by electrical engineers
  - Default access to inputs and outputs is digital, bit-wise addressing as sub-address of bytes
    - The inputs and outputs are usually fed by analog lines through A/D converters
  - One general purpose register, the accumulator
    - Newer ones have more than one accumulator, but the additional ones are often not directly addressable
  - A couple of counters and timers
- Modern PLCs are significantly more complex

Introduction to PLCs

- PLCs are standardiz
  Commission: IEC 61
  - The IEC also standa
    tape ;)
- IEC defines in 6113
  - Ladder diagram (LD)
  - Function block diagr
  - Structured text (ST),
  - Instruction list (IL), te
  - Sequential function chart (SFC)
- IEC also defines a set of standard
  - Augmented by the vendor's library

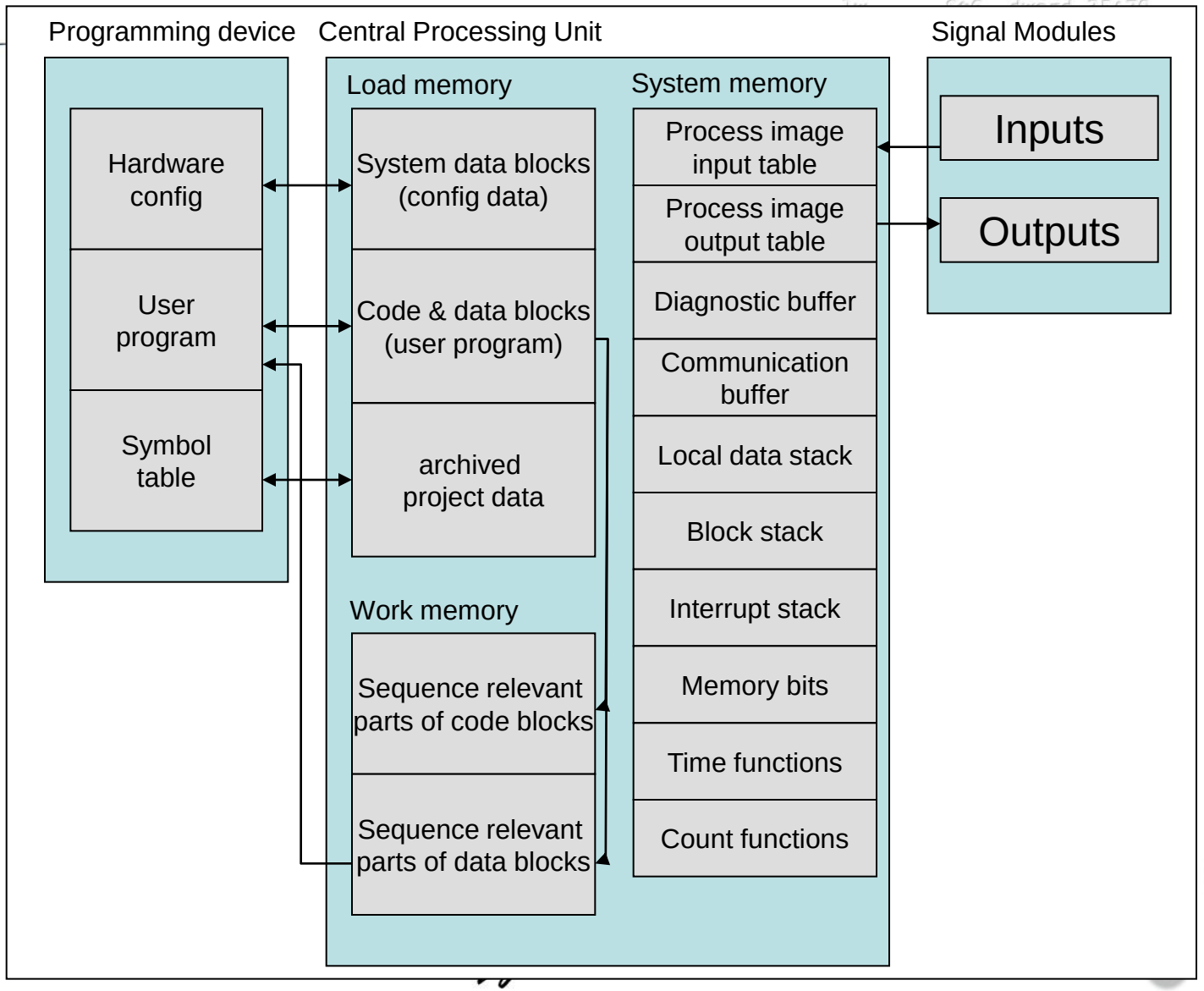


FBD: A functional block diagram of the attitude control and maneuvering electronics system of the Gemini spacecraft. (McDonnell, "Project Gemini Familiarization Charts") June 5, 1962
All images courtesy of Wikipedia.

Introduction to PLCs

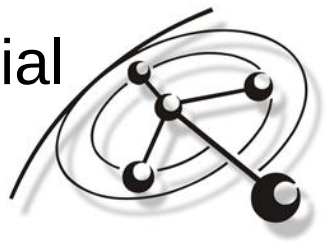
- PLCs execute their byte-code on the main CPU by interpreting it
  - The byte-code is not the native instruction format of the PLC CPU
  - Modern PLCs use ASICs that can execute the byte-code natively, in order to speed up execution
- PLCs execute in “scans”
  1. All inputs are read by the PLC
  2. The main code block is executed
  3. All outputs are set by the PLC, depending on the code’s result

Introduction to Simatic S7



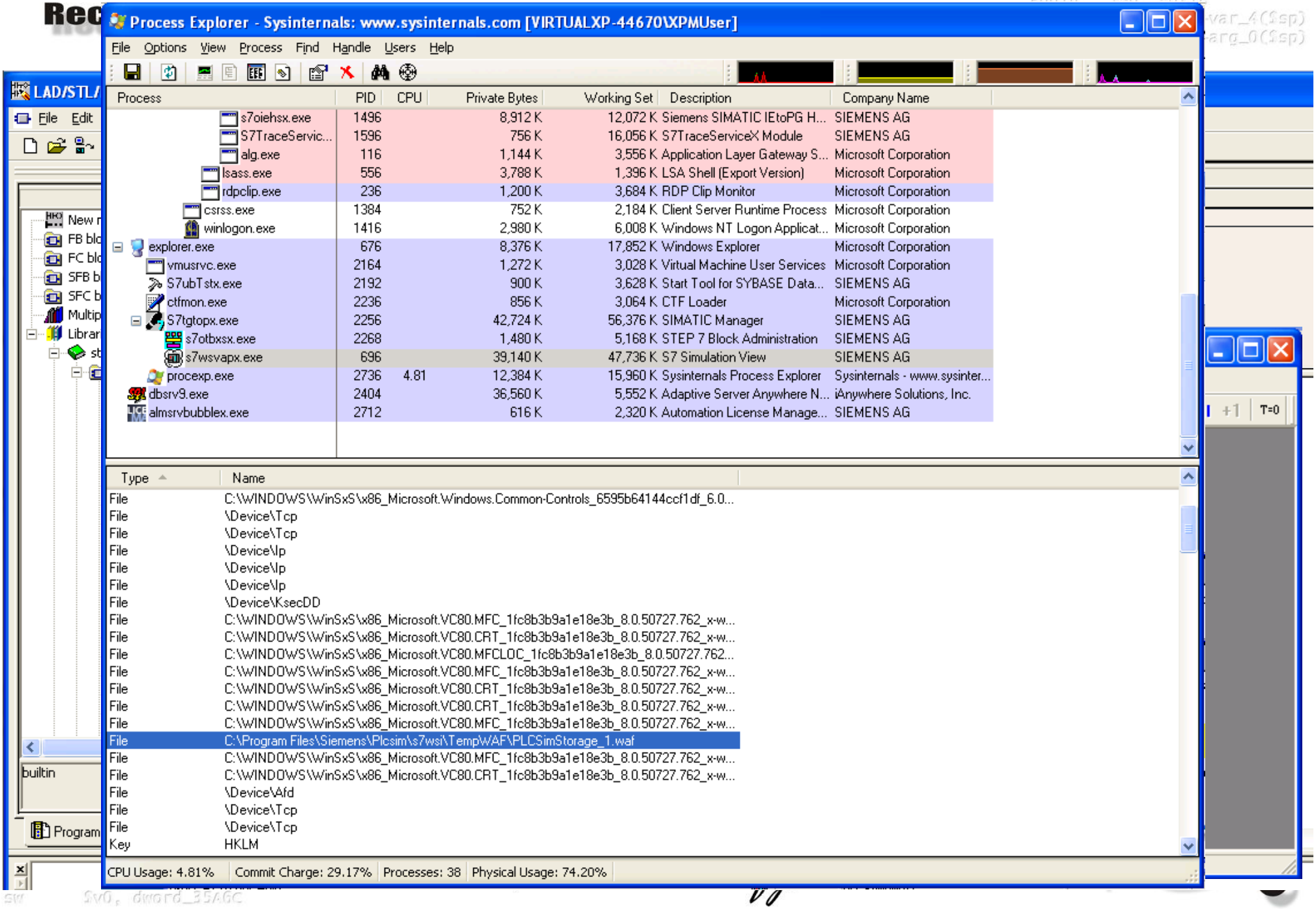
Simatic S7 and STEP7

- Simatic (= **Si**emens + Auto**matic**) are PLCs built since 1973 (S3). Current is S7, introduced in 1994.
  - The byte-code for S7 PLCs is called MC7
- Development environment for S7 is STEP7
  - "**ST**euerungen **E**infach **P**rogrammieren" (engl. "Controllers Easily Programmed")
  - Support for 3 of the EIC 61131-3 development styles:
    - LD (ger. KOP - Kontaktplan)
    - FBD (ger. FBS - Funtionsbausteinsprache)
    - IL (ger. AWL - Anweisungsliste, engl. STL)
      - Warning: there is a internationalized German version of STL/AWL!
  - Four other optional development environments
  - PLC simulation package, including hardware design environment
  - Tools to communicate with PLC over various media
- Simatic STEP7 software can be obtained as 14-day trial



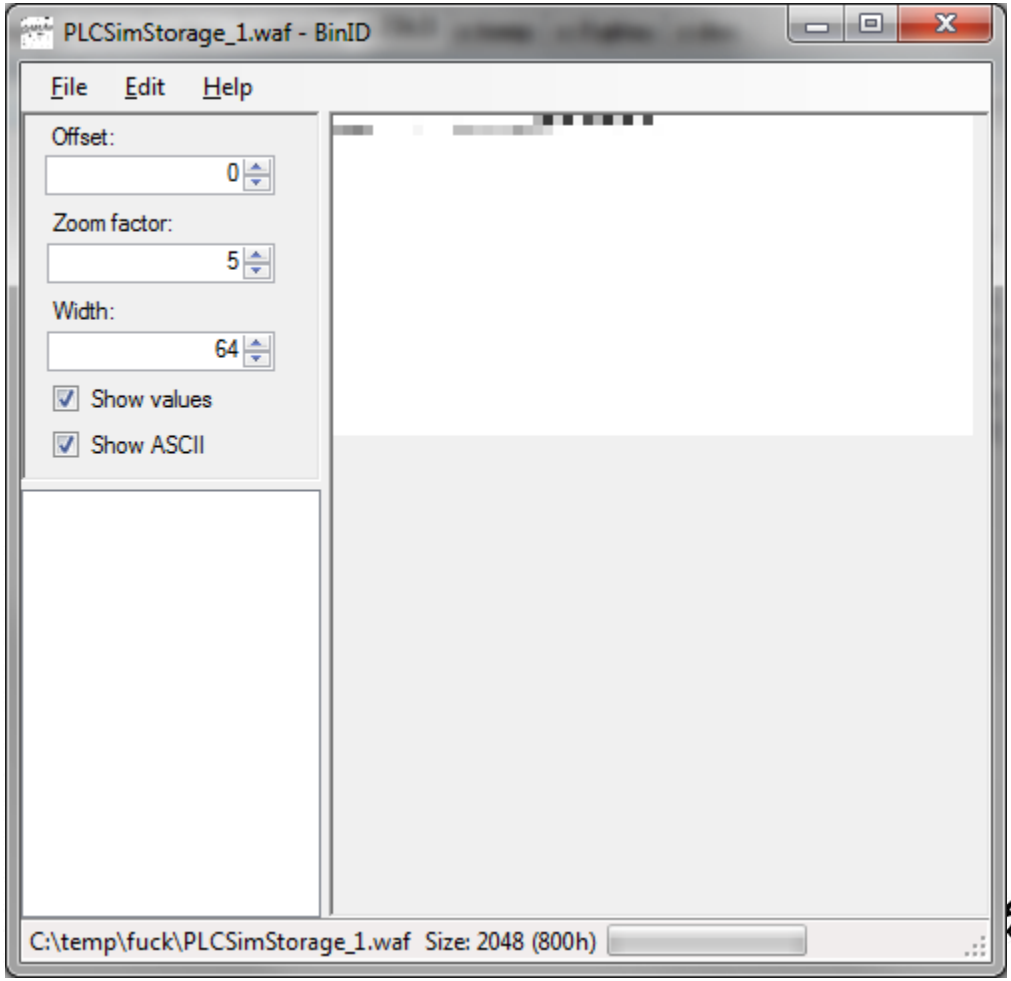
Mikko H. Hyppönen: Evidence that Iran runs STEP7

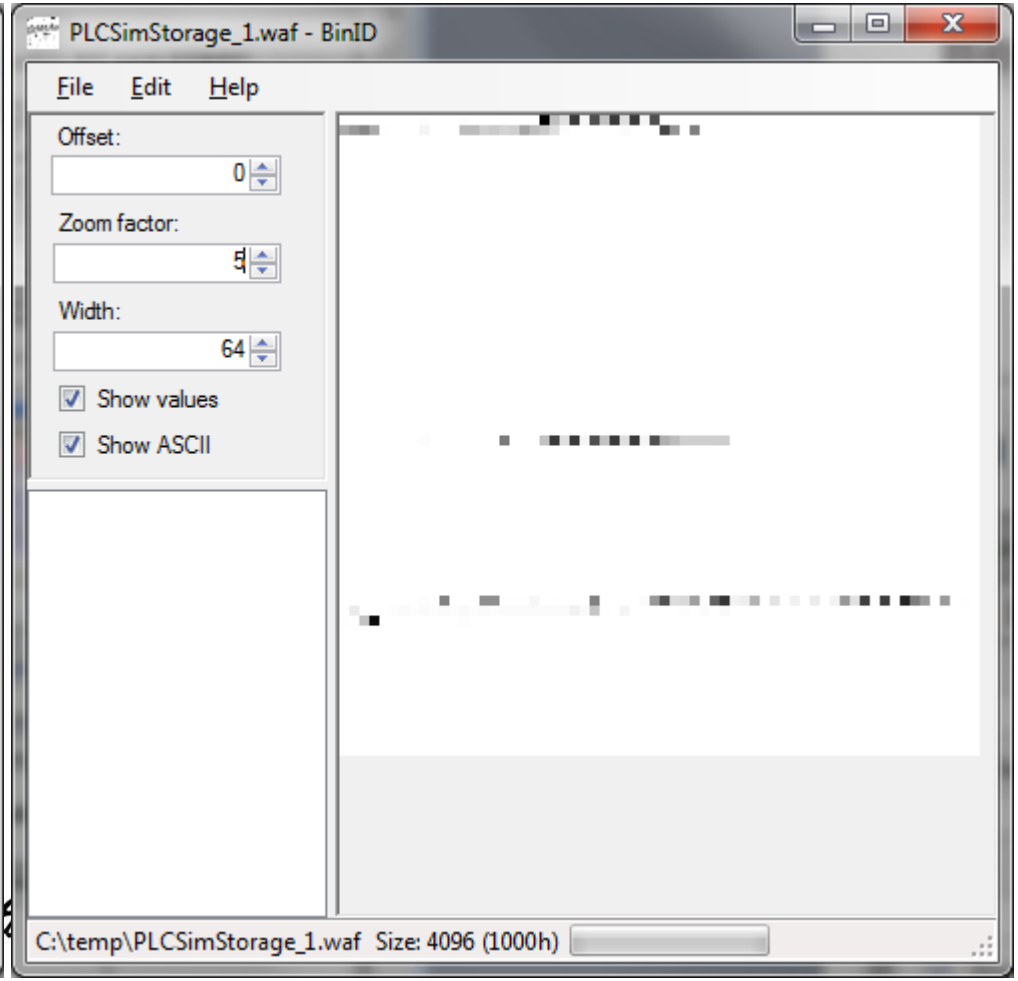
Invent & Verify


Process Explorer - Sysinternals: www.sysinternals.com [VIRTUALXP-44670\XPMUser]
File Options View Process Find Handle Users Help
Process PID CPU Private Bytes Working Set Description Company Name
s7oiehsx.exe 1496 8,912 K 12,072 K Siemens SIMATIC IEtoPG H... SIEMENS AG
S7TraceServic... 1596 756 K 16,056 K S7TraceServiceX Module SIEMENS AG
alg.exe 116 1,144 K 3,556 K Application Layer Gateway S... Microsoft Corporation
lsass.exe 556 3,788 K 1,396 K LSA Shell (Export Version) Microsoft Corporation
rdpclip.exe 236 1,200 K 3,684 K RDP Clip Monitor Microsoft Corporation
csrss.exe 1384 752 K 2,184 K Client Server Runtime Process Microsoft Corporation
winlogon.exe 1416 2,980 K 6,008 K Windows NT Logon Applicat... Microsoft Corporation
explorer.exe 676 8,376 K 17,852 K Windows Explorer Microsoft Corporation
vmusrvc.exe 2164 1,272 K 3,028 K Virtual Machine User Services Microsoft Corporation
S7ubTstx.exe 2192 900 K 3,628 K Start Tool for SYBASE Data... SIEMENS AG
ctfmon.exe 2236 856 K 3,064 K CTF Loader Microsoft Corporation
S7tgtopx.exe 2256 42,724 K 56,376 K SIMATIC Manager SIEMENS AG
s7otbxsx.exe 2268 1,480 K 5,168 K STEP 7 Block Administration SIEMENS AG
s7wsvapx.exe 696 39,140 K 47,736 K S7 Simulation View SIEMENS AG
procexp.exe 2736 4.81 12,384 K 15,960 K Sysinternals Process Explorer Sysinternals - www.sysinter...
dbsrv9.exe 2404 36,560 K 5,552 K Adaptive Server Anywhere N... iAnywhere Solutions, Inc.
almsrvbubblex.exe 2712 616 K 2,320 K Automation License Manage... SIEMENS AG
Type Name
File C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0...
File \Device\Tcp
File \Device\Tcp
File \Device\Ip
File \Device\Ip
File \Device\Ip
File \Device\KsecDD
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.MFC_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.MFCLOC_1fc8b3b9a1e18e3b_8.0.50727.762...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.MFC_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.MFC_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\Program Files\Siemens\Plcsim\s7wsi\TempWAF\PLCSimStorage_1.waf
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.MFC_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File C:\WINDOWS\WinSxS\x86_Microsoft.VC80.CRT_1fc8b3b9a1e18e3b_8.0.50727.762_x-w...
File \Device\Afd
File \Device\Tcp
File \Device\Tcp
Key HKLM
CPU Usage: 4.81% Commit Charge: 29.17% Processes: 38 Physical Usage: 74.20%

Finding the Byte-Code

- Visual difference before and after programming





Familiarizing Yourself With The Environment

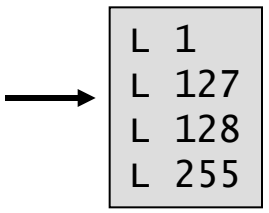
- Obtain a programming manual
  - You will need a full manual, it's often shipped with the IDE
- It's very helpful to have basic introductory material
  - Beginner tutorials shipped with the development environment
  - Simple development, deploy and debug sessions
    - Look for university course material
- Go through a couple of the introduction sessions
  - It might easily be the most frustrating task
  - Make sure you understand the development cycle
- Write very simple programs yourself
  - Refrain from anything that involves conditional code flow
  - Debug your programs

Quick Overview of STEP7 STL

Bit-Logic instructions	A, O, X, N, =
Comparison instructions	=>I, <=D, etc.
Conversion instructions	BTI, NEGI, RND+, etc.
Counter instructions	FR, L, LC, R, S, CU, CD
Data Block instructions	OPN, L DBLG, etc.
Logic Control instructions	JU, JC, JL, LOOP, etc.
Integer Math instructions	+I, -I, /I, MOD, etc.
Floating-Point Math instructions	+R, ABS, SQR, ACOS, etc.
Load and Transfer instructions	L, LAR1, T, CAR, TAR1, etc.
Program Control instructions	BE, CALL, UC, CC, etc.
Shift and Rotate instructions	SLW, SLD, etc.
Timer instructions	FR, L, LC, R, SP, etc.
Word Logic instructions	AW, OW, XOW, AD, OD, XOD
Accumulator instructions	TAK, POP, PUSH, INC, BLD, NOP 0, etc.

Recognizing Your Code

- Immediate values are your friend
  - Repeatedly load the same immediate numeric value into the same destination (e.g. a register)
  - Use small numbers with known hex / binary representations
    - 0x01 == 1
    - 0x7F == 127
    - 0x80 == 128
    - 0xFF == 255



- If you can, use hexadecimal representations when writing your test code
  - It is easier to recognize hexadecimal characters in hex dumps
  - It is also easier to realize they are missing

```
00000c20h: 9A F6 26 60 03 9D CB 0C 11 4C 00 1C 00 0E 00 14 ; šö&`.Ë..L......
00000c30h: 00 1E 30 03 00 01 30 03 00 7F 30 03 00 7F 30 03 ; ..0...0..□0..□0.
00000c40h: 00 7F 30 03 00 7F 30 03 00 7F 30 03 00 7F 65 00 ; .□0..□0..□0..□e.
00000c50h: 01 00 00 14 00 00 00 02 05 02 05 02 05 02 05 02 ; ................
00000c60h: 05 02 05 05 05 05 05 00 00 FE FE 14 00 FE FE 14 ; .........SunKing
```

Recognizing Your Code

- Increase the size of your immediate values
  - You are not looking for the instruction encodings yet, although pattern recognition is not a crime
- Try to develop “markers”
  - Encoding patterns that you easily recognize
  - Use before and after other instructions, so you can tell their length
- Do not try to understand the file format!
  - It wouldn’t help you, even if you did.

Recognizing Your Code

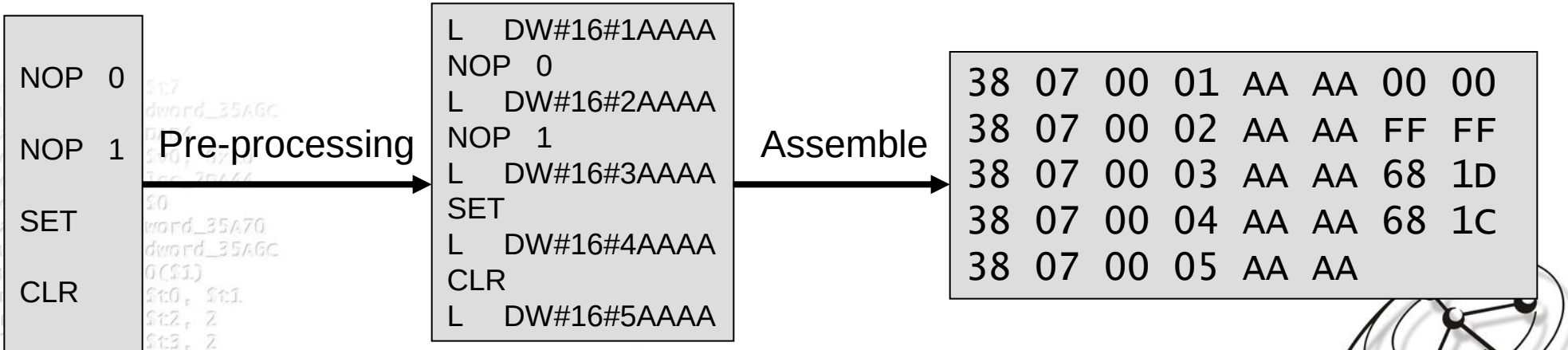
```
L    W#16#CAFE
L    W#16#CAFE
NOP  1
L    DW#16#AAAAAAAA
L    DW#16#AAAAAAAA
L    DW#16#FEFE0BAD
```

- You might have noticed: the code's endianess comes out for free

```
00001000h: 00 00 00 00 00 00 00 00 02 00 90 00 00 00 70 70 ; ..........□..pp
00001010h: 01 01 01 08 00 01 00 00 00 90 00 00 00 00 04 97 ; .........□....—
00001020h: EB 4E 26 60 03 9D CB 0C 11 4C 00 1C 00 0E 00 14 ; ëN&`.Ë..L......
00001030h: 00 1E 30 07 CA FE 30 07 CA FE FF FF 38 07 AA AA ; ..0.Êþ0.Êþÿÿ8.ªª
00001040h: AA AA 38 07 AA AA AA AA 38 07 FE FE 0B AD 65 00 ; ªª8.ªªªª8.þþ.e.
00001050h: 01 00 00 14 00 00 00 02 05 02 05 02 05 02 05 02 ; ................
```


Recognizing Your Code

- Write pre-processing scripts for your instruction set discovery programs
  - For each instruction you write, generate a marker with a sequence number
  - Use the marker information to extract instructions from the resulting hex dumps

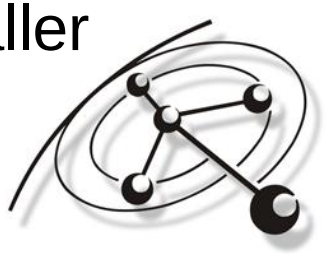


How To Document

- Document your discoveries
  - The code of your disassembler is not documentation!
  - Only an independently documented instruction set allows you to separate wrong mappings from implementation bugs.
- Document strictly in binary
  - Binary documentation helps you to identify patterns you will miss otherwise
- Augment documentation with examples in hexadecimal
  - The hex notation allows you to become a native speaker more quickly
- Always provide at least one example

Begin Code Discovery

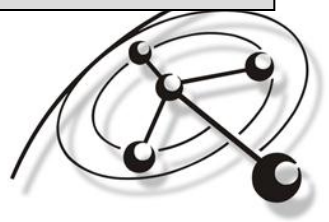
- You should start with the most “native” instructions of your target device
  - For PLCs, these are obviously the logic instructions operating on inputs and outputs
  - Also quite native to PLCs are timer and counter
  - For other CPU types, this is likely to be logic operations on bytes, words and double words
- The main reason to start here is history
  - The byte code was likely developed when the native width of the target device was still smaller (e.g. 16 Bit)
  - This will cause the encoding to be different for smaller value ranges



Begin Code Discovery

```
Notation:
b: Bit of address line
i: 0=Input / 1=Output
x: Line
M: 0=memory/1=IO

1m000bbb ixxxxxxx   A
ex: C701            A I 1.7
1m100bbb ixxxxxxx   AN
ex: E701            AN I 1.7
00000000 nttt0bbb xxxxxxxx xxxxxxxx   A (n indicates NOT)
ex: 00 60 00 14     A #BOOLVAR_AT_20
ex: 00 10 01 00     A I 256.0
```

Discovering Ranges

- Many instructions take arguments in ranges
  - Immediate operands
  - Numbered registers
  - Date and Time formats
  - Addresses
  - Offsets
- Your pre-processing script(s) should take care of that
  - Define border cases for the range arguments
  - Have your pre-processing script iterate over the argument cases and the instruction you provide
  - It's almost like writing a worst case fuzzer ☺

Discovering Ranges

```
Arguments = ( ‘I 0.0’, ‘I 0.7’, ‘I 32.0’, ‘I 128.7’...
            ...’Q 0.0’, ‘Q 0.7’, ‘Q 32.0’, ‘Q 128.7’...

Instructions = ( ‘A $arg’, ‘O $arg’, ‘X $arg’, ...

➔A I 0.0
➔A I 0.7
➔A I 32.0
...
➔O I 0.0
➔O I 0.7
...
```


A Word About Notation

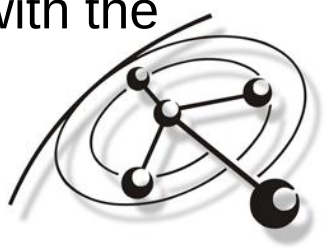
- Keep in mind that notation is up to you
- It makes a lot of sense to stay as close to the vendor's notation as possible
  - Other people can look up instructions in the vendor's original manual
  - Other people who speak the mnemonics can directly work with your output
- The notion of argument versus part of the instruction is completely up to you
  - It doesn't change the notation at all
  - Nobody said instructions cannot have spaces
- Some times, the vendor's notation is ambiguous
  - Don't be scared to invent a new one
  - Make sure it's clearly distinguishable from the vendor's
    - People familiar with the assembler need to see it's special!

A Word About Notation

```
L C0            ; Load counter 0
L DBGL          ; Load Length of Shared DB in ACCU 1
L #1            ; Load 32-Bit immediate 1
L MB1           ; Load memory byte 1
L IW1           ; Load input byte 1
L DBD 1         ; Load data block double word 1
L 1             ; Load 16-Bit immediate 1
L T[MW1]        ; Load timer whose number is stored in memory word 1
L PIB[AR1,P#1.5] ...
```


Intermission: Implementing the Disassembler

- You may implement your disassembler as standalone
  - Complete freedom of choice
    - Programming language
    - Representation
    - Command line vs. GUI
  - Requirement to produce interface formats for other tools
  - Lack of other functionality (e.g. code flow tracing)
- You may integrate your disassembler into a reverse engineering tool
  - Bound to the reverse engineering tool's choice of programming language and API
  - Potential issues with the integration itself (secondary battlefield)
  - Availability of functionality already available in the tool
  - Availability of other third party modules / tools that integrate with the targeted tool as well



Intermission: Writing IDA Processor Modules

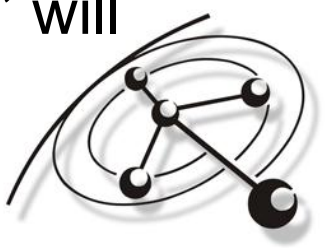
- IDA allows you to develop modules to support additional CPUs not already available
  - It's like writing any other plug-in (using the SDK)
  - Since IDA 5.7, processor modules can be developed in Python
- You need to provide a class inherited from idaapi.processor_t
  - Assigns a processor ID and name
  - Defines a number of properties
    - Typical code start and end sequences
    - Segment register properties (how x86ish!)
    - Number of instructions and instruction decoding array
    - Number of registers and register representation array
  - Defines an Assembler for notation (comments, etc.)

Intermission: Writing IDA Processor Modules

- You need to implement a couple of methods from idaapi.processor_t
  - **emu**: Executed when IDA wants to emulate the instruction
    - Does the instruction create cross-references, what type and where do they point? Does it modify the flags?
    - This call-back is allowed to modify the IDB
  - **out**: Executed when IDA wants to create a textual representation for the instruction
  - **outop**: Executed when IDA wants to create the textual representation of an operand to the instruction
  - **ana**: Executed to decode an instruction
    - IDA does not give you an index or address of the bytes to decode, only functions to say “get next byte/word/etc.”
- Due to the callback design and the requirements to use IDA’s structures, it quickly becomes hard to manage

Intermission: Writing IDA Processor Modules

- The known instructions array makes heavy use of an index number called '**itype**'.
  - It is advisable to generate the array and itype dynamically when the module is loaded – managing it by hand is bound to fail
- Every decoded instruction is handled by a structure called '**cmd**'
  - Contains the effective address (EA) of the instruction
  - Contains fields for the operands (**Op1**, **Op2**, …) of type **op_t**
    - Operands have a size field (8, 16, 32 Bit)
    - Operands have a type (register, memory ref, immediate, special, etc.)
    - Depending on the type, different value fields are used
  - Contains the 'itype' reference to the instruction array
- Warning: The choice of types and values within those structures influences significantly how the IDA "kernel" will handle your disassembly



Intermission: Writing IDA Processor Modules

- Endianess is a surprisingly big issues with IDA
  - There is a (not very well documented) structure called '**inf**', and **inf.mf** sets the endianess
    - **inf.mf** = 0 is big endian
    - **inf.mf** = 1 is little endian
  - Setting the endianess this way doesn't help when reading data > 8 Bit during instruction decoding
    - Hint: write yourself functions to read anything bigger than a byte, you should know the endianess

Intermission: IDA Processor Modules And The Rest

- I opted for writing my own back-end disassembler
  - I can have all my disassembly code in class hierarchies
  - I can generate the IDA structures upon startup
  - I can have my own way of rendering
- I'm only using a few operand types:
  - **o_imm** for immediate values, so IDA can calculate
  - **o_near**, **o_mem** for code and data references
  - **o_idpspec0** … **o_idpspec5** for everything else, since it is meaningless to the IDA kernel
- I rewrote it two times and should rewrite it again
  - "Some code cannot be written beautiful, because the subject is ugly." – paraphrasing Lisa Thalheim

STEP7 Program Structure

- Organization Blocks are the interface between the PLC operating system and the user program
  - Main program scan (OB1)
  - Time-of-day interrupts (OB10-17)
  - Time-delay interrupts (OB20-23)
  - Cyclic interrupts (OB30-38)
  - Hardware Interrupt Organization Blocks (OB40-47)
  - Programming DPV1 Devices (OB55-57)
  - Multicomputing - Synchronous Operation of Several CPUs (OB60)
  - Synchronous cycle interrupt (OB61-64)
  - Redundancy errors (OB70-72)
  - Asynchronous errors (OB80-87)
  - Background Cycle (OB90)
  - Startup Organization Blocks (OB100-102)
  - Synchronous errors (OB121-122)

STEP7 Program Structure

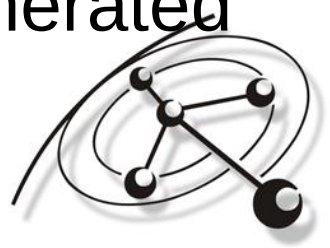
- Functions (FC) contain program routines for frequently used functions
- Function Blocks (FB) are blocks with a "memory" which you can program yourself.
- System function blocks (SFB) and system functions (SFC) access operating system functions

STEP7 Program Data Areas

- Data Blocks (DB) are areas for storing user data.
  - Think of them as global data structures.
- Instance Data Blocks (DI) are assigned to FBs that transfer parameters. There is one instance per FB call in the user program.
  - Think of them as objects of a class.

STEP7 Program Data Areas

- When creating logic blocks (OBs, FCs, FBs), you can declare temporary local data.
  - Every organization block has start information of 20 bytes of local data that the operating system supplies when an OB is started. The start information specifies the start event of the OB, the date and time of the OB start, errors that have occurred, and diagnostic events.
  - For example, OB40, a hardware interrupt OB, contains the address of the module that generated the interrupt in its start information.



STEP7 Calling Conventions

- The STL manual lists three types of calls:
  - CALL, which invokes FBs and FCs
  - CC, a conditional call
  - UC, a unconditional call
- When inspecting the byte code for a CALL to FB instruction, a surprising amount of code shows

```
00000c30h: 00 3A 38 07 00 01 AA AA 10 03 41 60 00 18 FB 7C
00000c40h: FB 79 00 01 FE 6F 00 14 68 1C 41 50 00 00 28 02
00000c50h: 7E 55 00 01 FE 0B 84 00 00 00 75 01 FE 6B 00 14
00000c60h: FB 7C 10 04 38 07 00 02 AA AA 65 00 01 00 00 14
```

STEP7 Calling Conventions: FB

- CALL to FB includes elaborate setup code to initialize the DI
  - The code to the right is emitted for

```
CALL FB1, DB1
   Var1 := FALSE
   Var2 := 2 // byte
```

- When you encounter a large block of emitted byte code, it's safest to assume macro operations

```
L       DW#16#1AAAAh
BLD     +3
=       L 18h.0
CDB
OPN     DI1
TAR2    LD14h
CLR
=       DIX 0.0
L       B#16#2
T       DIB1
LAR2    P# 0.0
UC      FB1
LAR2    LD14h
CDB
BLD     +4
L       DW#16#2AAAAh
```

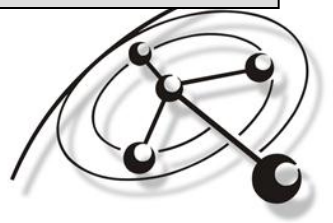
STEP7 Calling Conventions: FC/SFC

- CALL to FCs uses completely different argument setup code
  - The code to the right is emitted for

```
CALL SFC1 // get sys time
   RetVal := Temp#20
   Time := Temp#22
```

- Here, the compiler generates a temporary local pointer that is passed as the SFC
  - This pointer will never be visible in the development environment

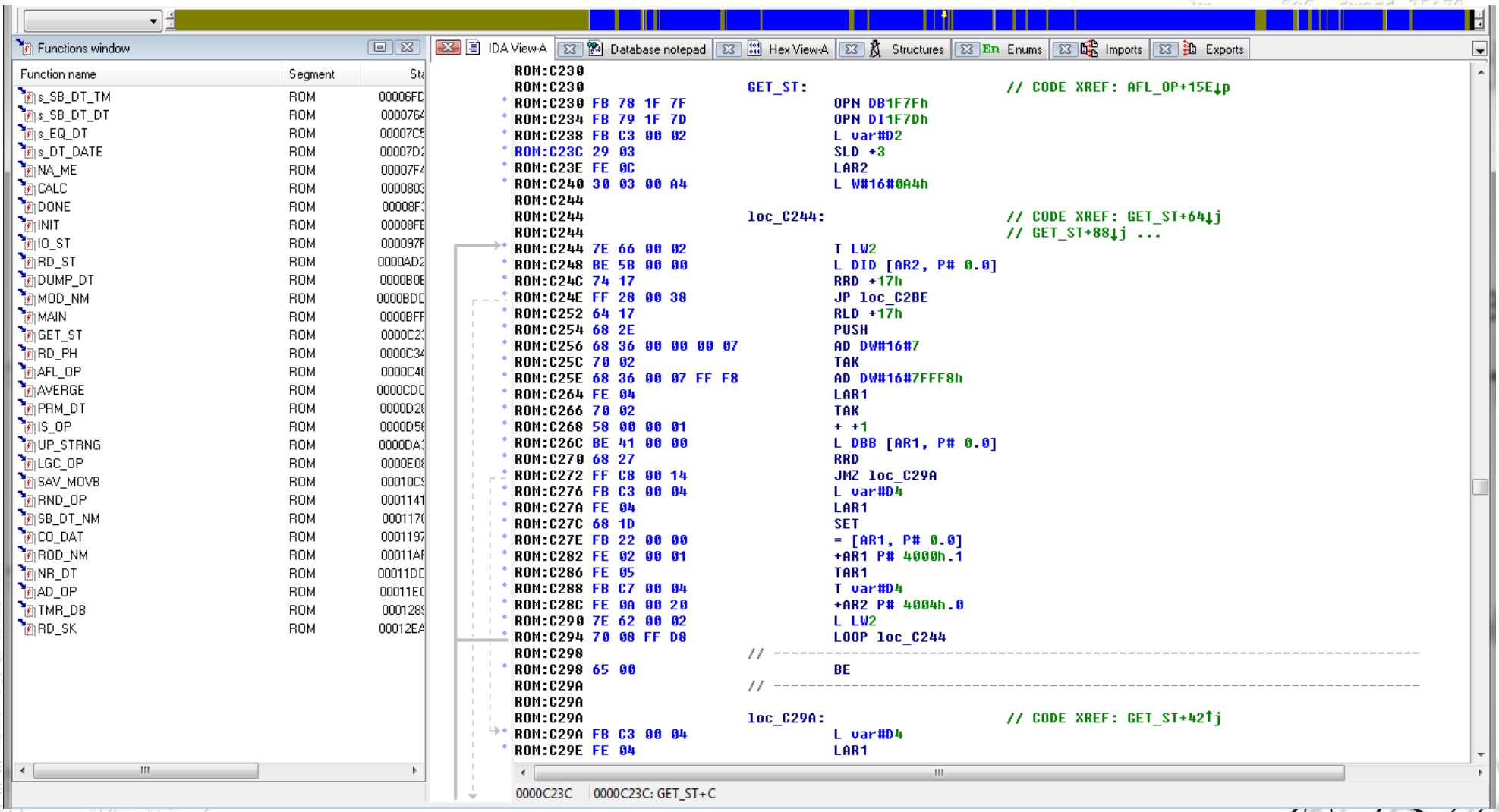
```
L       DW#16#1AAAAh
BLD     +7
=       L 1Eh.0
L       W#16#0
T       LW1Fh
L       P# 16h.0
T       LD21h
UC      SFC1
JU      loc_C60
(arg)   P# L 14h.0
(arg)   P# L 1Fh.0
loc_C60:
BLD     +8
L       DW#16#2AAAAh
```

Advanced Addressing Modes

- Addressing modes are often not very well documented
  - Searching the Internet for examples of advanced use of the programming language(s) helps understanding more complicated implementation patterns
  - I finally understood the advanced addressing modes after discovering university lecture notes a student of electrical engineering took in class
- STEP7 MC7 supports indirect addressing
  - Local indirect addressing, e.g. [LW10]
  - Global indirect addressing, e.g. [AR1, #P0.0]

Putting It All Together



Reading STUXNET

Invent & Verify

The Target PLC

- Two types of S7 CPUs:
  - 6ES7-315-2

IDA View-A | Matched Functions | Statistics | Primary Unmatched | Secondary Unmatched | Hex View-A

similarity	co...	EA primary	name primary	EA secondary	name secondary	algorithm
1.00	0.99	00000EA0	sub_EA0_14	00001430	sub_1430_27	MD index matching (flowgraph MD index, top down)
1.00	0.99	00003852	sub_3852_23	00003DE2	sub_3DE2_36	hash matching
1.00	0.99	00003FE6	sub_3FE6_25	00004576	sub_4576_38	hash matching
1.00	0.99	000028AA	sub_28AA_21	00002E3A	sub_2E3A_34	hash matching
1.00	0.99	00002CDA	sub_2CDA_22	0000326A	sub_326A_35	MD index matching (flowgraph MD index, top down)
1.00	0.99	00001A8C	sub_1A8C_16	0000201C	sub_201C_29	hash matching
1.00	0.99	00001F50	sub_1F50_18	000024E0	sub_24E0_31	MD index matching (flowgraph MD index, top down)
1.00	0.98	00002284	sub_2284_19	00002814	sub_2814_32	MD index matching (flowgraph MD index, top down)
1.00	0.99	0000178A	sub_178A_15	00001D1A	sub_1D1A_28	hash matching
1.00	0.99	00000D96	sub_D96_13	00001326	sub_1326_26	hash matching
1.00	0.99	00003F1E	sub_3F1E_24	000044AE	sub_44AE_37	hash matching
1.00	0.99	00002818	sub_2818_20	00002DA8	sub_2DA8_33	hash matching
1.00	0.99	00001EA6	sub_1EA6_17	00002436	sub_2436_30	MD index matching (flowgraph MD index, top down)

- Needed by the backdoor in DP_SEND/DP_RECV

The STUXNET State Machine

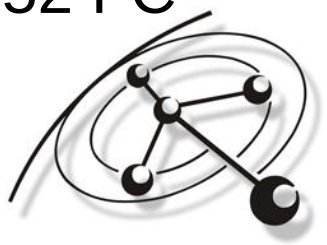
- It was quickly apparent that STUXNET uses an internal state machine
  - The widely published 0xDEADF007 magic value is actually only returned in state 3 and 4
- The states are now known as:
  - 1: Record frames via DP_RECV and monitor values of the VFD, until enough events are recorded
  - 2: Wait 2 hours
  - 3/4: Send bursts of Profibus frames to the VFDs, instructing them to change their frequency (and hereby the motor speed)
    - Disable OB1 and OB35 while doing so
  - 5: Reset internal values and reinitialize internal data structures
  - 0: Error handler

```
ADD_AC:                         // CODE XREF: S7_LV+94□p
          OPN DB888
          L DBW10h              // word 888.16
          L W#16#3              // word 3
          <I                    // ACCU2 is less than ACCU1
                                // 3 > 888.16
          JC loc_2840           // jump if RLO=1 (DW888.16 < 3)
                                // (do not jump if DW888.16 is 3 or more)
          TAK                   // exchange ACCU1 and ACCU2
          L W#16#4              // ACCU1 = 4
          >I                    // ACCU2 is greater than ACCU1
                                // 4 < 888.16
          JC loc_2840           // jump if RLO=1 (DW888.16 > 4 )
                                // (do not jump if DW888.16 is 4 or less)
          L DW#16#0DEADF007h
          PUSH                  // copy ACCU1 into ACCU2
          BE

loc_2840:                       // CODE XREF: ADD_AC+E□j
                                // ADD_AC+1A□j
          L DW#16#0
          PUSH                  // copy ACCU1 into ACCU2
          BE
```

The Code Does No Hiding

- STEP7 engineers frequently use a simple trick to hide code
  - The BLD instruction is used as a marker around blocks of code
  - The instruction has no effect on the PLC, but is interpreted by the Siemens editors. Known combinations are:
    - BLD 1 / 2 (FC with parameters)
    - BLD 3 / 4 (FB with parameters)
    - BLD 7 / 8
    - BLD 14 / 15 (FC without parameters)
    - BLD 103 / 104
    - BLD 130 / 131 / 132 / 133 / 255
- The STUXNET code does not make use of this trick
  - It actually keeps the original BLD instructions
    - Wasting space and simplifying analysis using Siemens tools
- However, there are only 31 BLD instruction pairs for 152 FC calls within Block C of STUXNET



The Code Does No Hi[illegible]

```
BLD +7

A "Always ON"    // When being nasty, use this snippet
JC Run
UC SFC 46        // Stops the CPU
Run: NOP 0

... your code
... CC or UC of your FC's
BLD +8
```

➔ Call SFC46

```
        L       LW0
        BLD     +7
        =       L 14h.0
        L       B#16#0
        T       LB15h
        UC      SFC1Ah
        JU      loc_24
        (arg)   P# L 15h.0
        (arg)   P# L 0.0
        (arg)   P# L 0.0

loc_24:
        BLD     +8
        BLD     +7
        =       L 14h.0
        L       B#16#0
        T       LB15h
        UC      SFC1Bh
        JU      loc_46
        (arg)   P# L 15h.0
        (arg)   P# L 0.0
        (arg)   P# L 0.0

loc_46:
        BLD     +8
        T       LW0
```

The Day It Was Done

- The STUXNET code contains the creation and modification timestamps of all functions
- The library functions in Block A and B are from **2002-02-15**
- The DP_SEND function is dated 1996-02-01, modified **2006-05-05**
- All custom functions in Blocks A, B and C are dated **2007-09-24**, modification date equal
  - The President of Iran Mahmoud Ahmadinejad speaks at Columbia University stating that Americans should look into "who was truly involved" in the September 11, 2001 attacks, defending his right to denial of the Holocaust, and denying the existence of gay Iranians. [Wikipedia]

SAV_MOVB	18:54:39
RD_SK	18:55:01
GET_ST	18:55:22
NA_ME	18:55:44
MAIN	18:56:06
RD_PH	18:56:27
DONE	18:56:49
NR_DT	18:57:11
SB_DT_NM	18:57:13
RND_OP	18:57:15
UP_STRNG	18:57:17
IS_OP	18:57:19
ROD_NM	18:57:21
CO_DAT	18:57:23
PRM_DT	18:57:25
AVERGE	18:57:26
AFL_OP	18:57:29
CALC	18:57:31
DUMP_DT	18:57:33
MOD_NM	18:57:34
RD_ST	18:57:36
IO_ST	18:57:39
LGC_OP	18:57:40
INIT	18:57:42
AD_OP	18:57:44
TMR_DB	18:57:47

STUXNET Notes

Much Respect for Quality and Testing

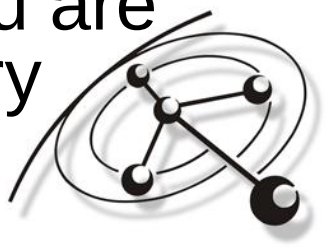
- Reliable exploitation requires tremendous amounts of testing
  - Windows Versions
  - Windows Languages
- This holds especially true if you pile a lot of exploits on top of each other
  - And you don’t want to be noticed
- And the authors haven’t actually tested the effectiveness of the PLC process attack yet
  - For which they need something “like” the target
  - Would you build an expensive guided missile without ever testing the warhead?

Likely Structure of the Kit

- Build environment for the assembled malware
  - Exploit selection (in-house 0day vulnerabilities, non-public)
  - Network propagation
  - C&C functionality
  - Rootkit functionality
  - Payload and trigger functionality
- It is quite possible the build kit was handed to other parties
  - Less understanding of the overall scenario
  - Access to the digital certificates
  - Over-powered the delivery mechanism

Lessons We Should Learn

- Developing custom disassemblers is easy
- Our response and analysis process plainly sucks
  - It took ages to detect STUXNET
  - It took a non-AV researcher to notice it's more than 08/15
- Common estimates of 0day-burn-rates were significantly too low
- It was clear ICS infections would work
  - We underestimated how easy it is
  - We underestimated how well it can be done
- If you haven't started funding and training an offensive development team 10 years ago, you are lacking an entire generation of digital weaponry



Thank You!

Felix ´FX´ Lindner
Head

fx@recurity-labs.com

Recurity Labs GmbH, Berlin, Germany
http://www.recurity-labs.com

Invent & Verify