Proceedings of the 2007 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY 20-22 June 2007

Vulnerability Analysis of SCADA Protocol Binaries through Detection of Memory Access Taintedness

Carlo Bellettini, Julian L. Rrushi

Abstract—
Pointer taintedness is a concept which has been successfully employed as basis for vulnerability analysis of C/C++ source code, and as a run-time mitigation technique against memory corruption attacks. Nevertheless, pointer taintedness interferes with the specification of several industrial control protocols. As a consequence it is not directly usable in detecting memory corruption vulnerabilities in implementations of those industrial control protocols. Furthermore, source-code analysis may have no visibility on certain low-level vulnerabilities since there may be a considerable difference between what programmers intend with the source code they write and what the CPU really executes. A set of memory corruption vulnerabilities specific to implementations of industrial control protocols may escape source code analysis as they are related to a dynamic organization of data in memory.

In this paper we define a new concept referred to as memory access taintedness. We discuss the logical motivations behind our definition of memory access taintedness and demonstrate that memory access taintedness is fully employable in vulnerability analysis of the machine code of implementations of industrial control protocols. We analyze the main low-level characteristics of both traditional attacks and attacks specific to process control systems, and demonstrate the ability of memory access taintedness to detect memory corruption vulnerabilities. We represent memory access taintedness as a decision tree and use it as the fundamental component of a finite state machine model we devised for the purpose of dynamically detecting memory corruption vulnerabilities in implementations of industrial control protocols.

Keywords—
Critical infrastructure defense, Industrial control protocols, SCADA systems, Vulnerability analysis.

I. Introduction

Industrial process control systems are computer-based devices used to remotely monitor and control critical infrastructure facilities such as nuclear plants, oil and gas facilities, electric power generation and distribution utilities, transportation infrastructures, water treatment plants, etc. Examples of process control systems include distributed control systems (DCS), programmable logic controllers (PLC), emergency shut down systems, supervisory control and data acquisition (SCADA), etc.[13]. For the sake of simplicity in this paper we use the term SCADA to refer to any industrial control system. Generally speaking, SCADA consists in one or more central stations referred to as master terminal units (MTU) which send commands via network to remote field devices such as intelligent electronic devices (IED's) or remote terminal units (RTU's). MTU commands may request status information from a field device or instruct a field device to perform mechanical actions such as opening or closing valves. Field devices in turn communicate with various sensors and actuators deployed in a critical infrastructure utility to be monitored and controlled. MTU's communicate with field devices usually in a master-slave mode according to especially developed industrial control protocols.

SCADA systems historically relied on proprietary networks, communication protocols, hardware and operating systems, and used dedicated communication lines. A SCADA network was relatively isolated from public networks in all senses. Nevertheless, SCADA has evolved into using Ethernet based TCP/IP networks, open industrial control protocols, Windows and Unix like operating systems, common CPU architectures, etc. Actually common SCADA networks get connected to the company Intranet and in some cases even to Internet, fact that exposes them to cyber attack. In fact SCADA systems may be subject to various vulnerabilities in their data, security administration, architecture, networks and platforms[11].

In this paper we discuss a novel vulnerability analysis technique for uncovering memory corruption vulnerabilities in industrial control protocol binaries. Our practical experiments were mainly performed on an implementation of MODBUS TCP protocol[17][21] running on an emulator of a 32-bit ARM microprocessor. The binaries we analyzed are formatted according to ARM ELF file format[20] under an embedded and real-time Linux operating system. Throughout our paper we also discuss related work on industrial protocols running on Windows CE. Furthermore, some of the attacks we analyze are explained in the references of this paper as performed against code running on an Intel CPU. Although on ARM the technical details

Manuscript received March 21, 2007. This work was supported in part by Power Defense Inc.
C. Bellettini is an Associate Professor with Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano. Via Comelico 39/41, 20135 Milano, Italy (e-mail: carlo.bellettini@unimi.it).
J.L. Rrushi is a PhD candidate at Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano. Via Comelico 39/41, 20135 Milano, Italy (phone: +39-02503.16273; fax: +39-02503.16276; e-mail: julian.rrushi@dico.unimi.it).

1-4244-1304-4/07/$25.00 ©2007 IEEE

of implementing those techniques may vary, we still refer to them as the exploitation logic behind them remains the same.

II. Related Work

One of the most significant frameworks that performs vulnerability analysis of SCADA software written in C and C++ is the developer environment for automated buffer overflow testing (DEADBOLT)[16]. DEADBOLT performs a source-to-source transformation in order to insert instrumentation instructions into SCADA software for the purpose of detecting buffer overflows during an automated testing. DEADBOLT automatically generates targeted test cases from sample input data. Such test cases are then sent to SCADA software under analysis. During the dynamic analysis the instrumentation instructions extract memory access information and send them to a test-run result analyzer, which in turn decides whether any buffer overflow took place and possibly generates suggestions on next to use test cases. Our deterministic finite state machine (DFSM) model moves towards the same direction as DEADBOLT since it analyzes memory accesses in front of test cases in order to identify memory vulnerabilities. Nevertheless, the research ideas behind the vulnerability detection technique differ substantially since DEADBOLT uses instrumentation, while in our work we use a DFSM.
Fuzzing is employed by Franz in [15] and by Mora in [14] to test implementations of the Inter Control Center Protocol (ICCP)[10] and OPC, respectively. Although in these cases fuzzing succeeded in triggering several vulnerabilities, fuzzing alone may not be sufficient to uncover all memory vulnerabilities in implementations of industrial control protocols. For example, certain memory corruption vulnerabilities may be reachable only through an accurate data-flow analysis. Furthermore, fuzzing may not have the capacity to provide sufficient information necessary for identifying a triggered vulnerability. Such a limitation is due to the fact that fuzzing has no visibility on the target execution internals such as memory and CPU register information. In fact a heap overflow vulnerability [26] discovered by Franz in certain implementations of ICCP was only triggered through fuzzing. The identification of such a vulnerability was performed through complementary techniques a considerable amount of time after the corresponding discovery.
blackPeer described by Byres et al. in [6], and Kube and Hoffman in [18], is complementary to our vulnerability analysis model. Such a tool uses attributed grammars, i.e. grammars whose definition is overloaded with defined attributes, to generate and execute meaningful sequences of PDU's which are sent to the implementation of a defined industrial control protocol under analysis. Not only does blackPeer automate generation of test cases for industrial control protocols, but it also automatically interprets the behavior they exhibit during the analysis.
Other research work related to our vulnerability analysis model have been proposed as run-time mitigation of memory vulnerabilities. FireBuff[5] is a run-time defense mechanism which defines a logical separation between tainted data, i.e. any data derived from user supplied data, and other data in the address space of a protected process. It does so by encrypting all input data as they enter user space, and by having a protected process elaborate input data differently than other data. Thus, FireBuff tries to eliminate the offensive capabilities of pointer taintedness, i.e the IT equivalent of memory access taintedness in industrial control protocols. Our vulnerability analysis model is similar to FireBuff as it defines the boundaries between taintable data and other data as those boundaries change throughout a program execution.
Chen et al. in [23] define an architectural technique and employ their memory model to detect at run-time any dereferences of tainted data. Such a model consists in associating with each memory location and with each register a boolean property, namely taintedness, which indicates whether the data stored in those memory locations and registers respectively are derived from user supplied data. Whenever a tainted word is used for memory access the processor raises an exception. Our vulnerability analysis model is similar to the defensive technique of Chen et al. as just like their work ours keeps track of taintedness propagation. But unlike the work of Chen et al. where any memory access to an address derived from tainted data indicates the existence of a vulnerability, in our work a memory access with these characteristics is just a starting point which is further investigated through a decision tree. Furthermore, unlike the work of Chen at al. that treats pointer taintedness as an observed value, our work tries to calculate the SCADA equivalent of pointer taintedness, i.e. memory access taintedness.

III. Memory Access Taintedness

In this section we describe an analysis on the main low-level characteristics of memory vulnerabilities in implementations of communication protocols used by industrial control systems. We also describe the main characteristics at machine code level of both basic and advanced attack techniques which we employed in a monitored exploitation of memory vulnerabilities in industrial control protocols. Such a low-level analysis served for defining memory access taintedness, i.e. a concept which represents a potential triggered memory vulnerability and which we used in a dynamic binary analysis technique described in the next section of this paper.

A. A Representation of Pointer Taintedness at Machine Code Level

The concept of pointer taintedness was first introduced by Chen et al. in [25] as a basis for reasoning about memory vulnerabilities. A pointer is said to be tainted if its value is derived directly or indirectly from user supplied data. Chen et al. analyze C source code to determine the potential for pointers to be tainted. They use equational logic to formally define a memory model which forms the basis for a formal semantics of pointer taintedness. For a given function they define the corresponding formal semantics and then extract preconditions which must be satisfied for pointer taintedness not to be possible. If during the analysis any of those preconditions results to be violated then a pointer may be tainted, therefore a potential security vulnerability may exist.

While we deem the work by Chen at al. to be a powerful approach to uncover memory vulnerabilities, the visibility of such an approach may not be deeper than the maximum level of structural information provided by source code analysis. In fact Balakrishnan et al. have demonstrated the existence of the WYSINWYX phenomenon, i.e. what you see is not what you execute[8]. According to Balakrishnan et al. the WYSINWYX phenomenon could consist in a considerable difference between what programmers intend with the source code they write and what the CPU really executes. Furthermore, the WYSINWYX phenomenon may cause vulnerability analysis on source code to fail to uncover certain vulnerabilities. Such vulnerabilities may be uncovered only by examining the machine code which has been emitted by a compiler.

Certain vulnerabilities are generally caused by platform specific features. Some examples include compiler optimizations, memory layout, register usage, compiler flaws, etc.[8]. In implementations of industrial control protocols several memory accesses are directly or indirectly controlled by input data usually sent over a network by a communicating SCADA device. As a consequence in an industrial environment various memory vulnerabilities may be caused in front of certain organizations of data in memory and lack of ability to detect an out of range memory access. More information on these vulnerabilities and related attacks are given in the next subsection. Since actual memory addresses and organization of data in memory are only visible during machine code execution, static analysis of source code would have little visibility on these vulnerabilities. This fact motivated us to search for a representation of pointer taintedness at machine code level. More precisely, we searched for a representation of pointer taintedness in the form of a characteristic which is visible at machine code level since it is at this level that we operate. Such a characteristic is supposed to always hold if a given pointer was corrupted with user supplied data.

We examined the characteristics of a set of attack techniques as applied against an implementation of MODBUS protocol[17][21]. MODBUS is an application layer messaging protocol which enables SCADA devices to communicate in a master-slave mode within possibly different types of buses and networks. The analyzed attack techniques are a stack overflow exploitation with shellcode injection[1] or arc injection[19] corrupting the value of R14 (link register) saved on stack, heap overflow exploitation[3] with shellcode injection corrupting the value of R14 saved on stack, frame pointer overwrite attack[12] with shellcode injection corrupting the value of R11 (frame pointer) saved on stack, format string exploitation[9][22] with shellcode injection corrupting a function pointer in the global offset table (GOT), an indirect pointer overwrite attack[4] with shellcode injection corrupting a function pointer in GOT, and exploitation of an out of boundary array index with shellcode injection corrupting the value of R14 saved on stack.

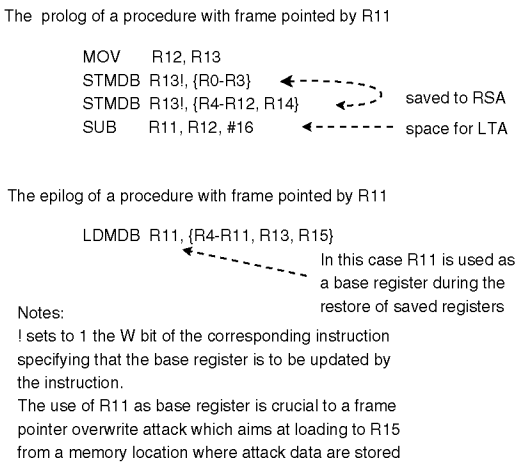

Fig. 1. Relevant observations on typical prolog and epilog of a procedure running on ARM microprocessor.

In ARM a procedure sets up a stack frame only if it needs to save permanent registers, allocate space for local variables, or allocate space for outgoing argument areas which are larger than 4 words. Thus, the frame pointer attack was applied against procedures with a frame set up. Referring to Figure 1, the stack overflow attack succeeds as the register save area (RSA), i.e. stack memory used to hold the preserved values of permanent registers including the saved link register stored during procedure prolog, gets corrupted by user supplied data. Consequently a tainted value is loaded from RSA into the R15 register (see procedure epilog). In ARM a load into R15 is referred to as a long jump[2] and is sufficient to transfer execution control to the loaded value.

A heap overflow attack corrupts a memory chunk on heap and is characterized by stores of a user specified value to a user specified address. In our case the user specified value

is the address of injected shellcode, and the user specified address is the memory location on stack where the value of R14 is saved. At the end the attack causes a long jump to injected shellcode. A frame pointer overwriting attack corrupts the value of R11 saved on stack during a procedure prolog. During the procedure epilog the corrupted value of R11 is loaded into the R11 register. When the original caller procedure enters its epilog, the R11 register holding a user supplied value is used as base register to load preserved register values from RSA including the saved value of R14 register. Since we control the value of R11 register during the epilog of the original caller procedure, we load into R15 a value from a memory location where we have preliminarily stored the address of injected shellcode.
Thus, a load into R15 from a user controlled address causes a long jump to an arbitrary address, in our case to the address of injected shellcode. During a format string attack a store instruction stores at a user specified address a value which may be controlled by the user through format directives. In our case there was a store of the address of injected shellcode at an address in GOT where a pointer to a dynamic library function was stored. Since the hijacked function was called by our MODBUS implementation after the GOT corruption, execution control was transferred to injected shellcode.
During an indirect pointer overwrite attack we corrupted a pointer value stored at the locals and temporaries area (LTA), i.e. stack memory used to hold local variables and compiler-generated temporaries. We set the corrupting value to the address in GOT where a pointer to a dynamic library function was stored. At machine code level this was reflected in a store to an user specified memory location.
During the exploitation of an out of boundary array index we specified an index which when added to the base address would point to the memory location on stack where the saved value of R14 was stored. We stored there the address of injected shellcode. At machine code level such an operation is represented by a store instruction whose index register holds a user specified value. Although we don't control the base register we can still dictate the memory location used by the store instruction since we control the index register.
As a conclusion, the main features extracted from a machine-level analysis of all the attacks we analyzed are summarized in Table I. The main characteristic which may be extracted from the attack information given in Table I and which is due to pointer taintedness consists in the fact that all the attack techniques that we analyzed resulted at least in a memory access at an address dictated directly or indirectly by user supplied data. We extend the definition of tainted data given by Chen at al. in [25] and classify any data in the address space of a defined process as tainted if:

1. Those data are input data, thus they may be directly defined and supplied by a potential attacker.

TABLE I
MAIN FEATURES OF MEMORY CORRUPTION ATTACKS AS WE APPLIED THEM AGAINST MODBUS PROTOCOL RUNNING ON ARM MICROPROCESSOR.

Attack Technique	*Corrupted Memory*	*Relevant Action(s)*
Stack overflow exploitation	RSA	A long branch to a user supplied address.
Heap overflow exploitation	Heap RSA	Two stores to user supplied addresses.
Frame pointer overwriting	RSA	A load from an address constructed upon user supplied data.
Format string exploitation	GOT	A store to an user supplied address.
Indirect pointer overwriting	LTA GOT	A store to an user supplied address.
Exploitation of an out of boundary array index	RSA	A store to an address constructed upon user supplied data.

2. Those data are derived either directly or indirectly from input data or from data which in general are classified as tainted (recursive definition).
3. Relevant aspects of those data are controlled either directly or indirectly by tainted data (recursive definition).
At this point we define the main characteristic of pointer taintedness at machine code level as any machine instruction accessing main memory at an address which is derived either directly or indirectly from tainted data. In the next subsection we describe a mechanism which we deem appropriate to be used for memory vulnerability detection in industrial control protocols along with relevant context dependent considerations on the machine code level representation of pointer taintedness.

B. Memory Access Taintedness as a Vulnerability Detection Mechanism

We extended our study on a set of attacks and related memory vulnerabilities specific to implementations of communication protocols used for supervisory and control by industrial control systems. The protocols which have been subject to our analysis are MODBUS and OLE for Process Control (OPC)[7]. After such an extended analysis we reached at the conclusion that pointer taintedness as defined by Chen et al. in [25] cannot be directly used as a vulnerability detection mechanism during a vulnerability analysis of industrial protocol binaries. As a matter of fact in implementations of industrial protocols it happens that the value of a pointer is tainted data. Regardless of the aforementioned consideration, during analysis of such implementations it is common to notice memory accesses

at an address computed upon at least a tainted value, and this is not in conflict with the specification of the corresponding industrial protocols.
The OPC data access specification for example defines the use of a set of interfaces to access heterogeneous devices in an industrial control network. OPC communications use Distributed Component Object Model (DCOM) and are intended to take place according to a client-server model. What OPC data access specification defines are the functionalities that OPC clients and OPC servers are supposed to provide while no further constraints are specified on the implementation of various interfaces. With regard to memory corruption attacks one of the most interesting elements of OPC are server handles, i.e. unique 32-bit identifiers that a server associates with OPC groups and items. An item in OPC may be thought of as an address of a data resource, while a group is used to organize items. A client receives from a server a handle each time that client creates a new group or item. The client then uses such a handle to refer to a group or access items in a group. As described by Mora in [14] several OPC server implementations use the memory address of a group or item as the corresponding handle.
Thus, in order to refer to a group or access items in a group a client specifies the address in server's memory where that group or item is stored. Applying pointer taintedness as a memory vulnerability detection mechanism would report a potential vulnerability each time an OPC client refers to a group or item. We analyzed the direct usability of pointer taintedness as a vulnerability detection mechanism also on MODBUS protocol. The main data items defined by MODBUS data model are the following: discrete input which are read-only one bit data provided by the I/O system, coils which are one bit data alterable by a MODBUS process, input registers which are read-only 16-bit data provided by the I/O system, and holding registers which are 16-bit data alterable by a MODBUS process. These data items are addressed using a number from 0 to 65535, and are stored in the main memory of a MODBUS device.
Each MODBUS implementation maintains a premapping between a data item address as used by MODBUS and the address in memory where that data item is stored. Thus, a MODBUS request specifies the address of a data item of a certain kind as a number n. The MODBUS process then uses its premapping table to define the memory address of the requested data item. In common MODBUS implementations each time a master station sends a MODBUS request to a slave device, the address of the data item specified in that request taints the memory address which is used to load the requested data item from the main memory of the slave. One of the causes of such a taintedness is the use of the MODBUS address as an offset from a base register to form the address of the memory location where the requested data item is stored.

Thus, implementations of several industrial control protocols maintain an area in main memory dedicated to the storage of control and monitoring data. In this paper we refer to this area as the acquisition data area (ADA) and use it in the definition of memory access taintedness. What prevents the concept of pointer taintedness from being directly usable as a vulnerability detection mechanism in industrial control protocol binaries is the following:

1. Explicit or implicit existence of an ADA. Since input data is used to calculate the address of any accesses to ADA, each legitimate request for acquisition data would generate a false positive.
2. The fact that the address in a memory access may consist in tainted data even though no pointer value has been preliminarily corrupted. An attempt to classify a memory access as tainted only if a related pointer has been tainted would result in false negatives.

With regard to the second point in the list above we can take into account the attacks to OPC described by Mora in [14] and other protocol specific attacks we applied against MODBUS. According to Mora the element deletion functions such as *RemoveGroup()* or *RemoveItems()* in some implementations of OPC employ the handle that the server receives from a client in deriving the address of a memory chunk to be deallocated through a call to *free()*. If a target OPC server doesn't properly validate handles that it receives from clients, then an attacker could specify a non-existent address forcing a server to deallocate non-reachable addresses. An attacker could also specify a handle which could force the server to execute the memory deallocation algorithm upon attack data, fact that could result in shellcode execution.
Although not covered by Mora, we believe that under these assumptions passing a valid handle twice to an element deletion function could result in the server trying to call *free()* twice on the same chunk, fact that could result in shellcode execution as well. The *Write()* function of the I/O interface which takes as parameters an item handle and a value to write to it is also subject to similar attacks described by Mora. In this case such an attack would aim at writing an arbitrary value to a memory address leading to shellcode execution. Thus, although during the attacks described by Mora and the attack variant we propose no pointers are corrupted with tainted data, memory corruption attacks may still be carried out resulting even in shellcode execution.
We build on the results described so far to define the concept of *memory access taintedness in implementations of an industrial control protocol*. We deem the relation between the criteria used for classifying a memory access as tainted and the specification of the protocol whose implementation is subject to vulnerability analysis to be quite important in deciding whether a given memory access indicates the existence of a memory vulnerability. Let's define a

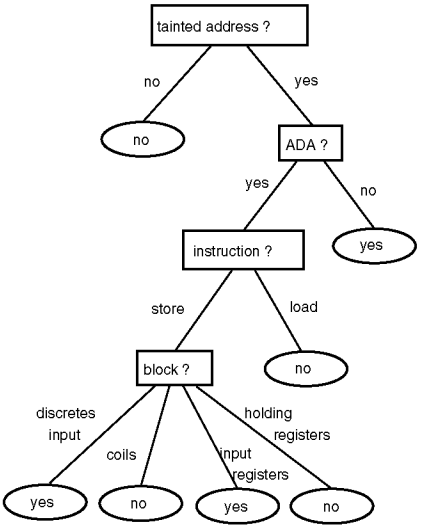

Fig. 2. A decision tree for the memory access taintedness in MODBUS implementations with a separate memory block for each kind of data item.

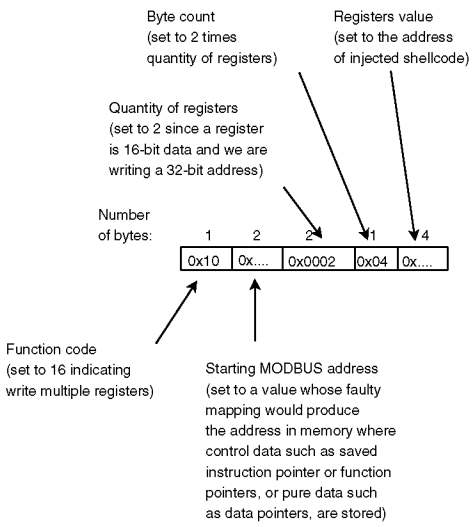

Fig. 3. A MODBUS protocol data unit as structured during MODBUS specific control-data and pure-data attacks against MODBUS implementations.

tainted address as an address which is derived from tainted data. For classifying a memory access as tainted not only its address should be tainted but also that address along with the kind of instruction, i.e. store or load, should cause a violation of what (and how) a control protocol frame can refer to in the memory of a control device.

We define memory access taintedness in implementations of MODBUS protocol and present it as a pattern. For the sake of simplicity we define memory taintedness on a MODBUS implementation which maintains a separate block of main memory for each kind of data items. At microprocessor emulator level we represent memory access taintedness in the form of a decision tree Figure 2. If the address of a memory access is not tainted we deem that memory access as not being tainted. Otherwise we check the address to see if it falls within ADA. If it doesn't then memory access taintedness is in place. Such a decision covers the machine code level representation of pointer taintedness and could detect all pointer corruptions we analyzed through attack techniques described in the previous subsection. In fact those attack techniques applied as control-data attacks need to corrupt data outside ADA in order to fulfill their ultimate goal, i.e. hijacking of execution control.

We developed a set of memory corruption attacks targeting faulty mappings between data items as addressed by MODBUS and the memory locations where those data items are stored. The protocol data unit (PDU) crafted for carrying out such MODBUS specific attacks is shown in Figure 3. The idea behind these attacks consists in requesting a target MODBUS device to write two holding registers, i.e. two 16-bit data in its main memory, by specifying a MODBUS address which as a result of a possible faulty mapping would produce the address of the memory location where control data are stored. The overwriting value is specified in the attack PDU as the value to write at those two registers and is set to the address of injected shellcode. Assuming a shellcode injection point exists in vulnerable MODBUS implementations, such an attack would corrupt control data with the address of injected shellcode. These MODBUS specific attacks in principle are similar to the attacks to OPC described by Mora in [14].

Memory access taintedness as defined in the decision tree of Figure 2 covers also these MODBUS specific attacks although no pointer is ever tainted. As a matter of fact the specification in the attack PDU of a malicious MODBUS address is reflected during process execution in a store to a memory address which is tainted, i.e. it is derived from the MODBUS address, and falls outside ADA since it is the address where control data are stored. We replayed as pure data attacks[24] targeting ADA both the traditional memory corruption attacks described in the previous subsection and the MODBUS specific attacks we devised for demonstrating memory corruption with no pointer taintedness. An attacker needs to carry out no offensive cyber action to read or write coils and holding registers since they are writable by the application. Therefore the targets of a possible corruption are discrete input and input registers which should be read-only values.

When applied as pure-data attacks targeting ADA all the memory corruption attacks that we analyzed soon or later result in a store to a memory address which is tainted and falls within either discrete input block or input registers block. As a consequence any store with these characteristics leads to an affirmative value of memory access taint-

edness in the decision tree given in Figure 2.

IV. Finite State Machine Analysis of Memory Access Taintedness

Now that we have the definition of memory access taintedness in implementations of an example industrial control protocol we can describe a DFSM model we devised for identifying memory access taintedness in those implementations. We describe such a vulnerability analysis model as applied to the MODBUS protocol since it is for this protocol that we defined memory access taintedness indicating the existence of memory vulnerabilities. Taking into account both the amount and the kind of information needed by our vulnerability analysis model a software implementation of the DFSM that we propose incorporated into an ARM microprocessor emulator is most suitable for dynamically analyzing memory access taintedness in SCADA binaries as single machine instructions execute. In fact we use a CPU emulator as we need real-time information about memory accesses which we found easier, simpler and faster to extract from a CPU emulator rather than from any physical ARM microprocessor. Nevertheless, we do not exclude the usefulness of other approaches different than ours, such as for example using DFSM in a static analysis along with symbolic execution of an industrial protocol binary.

In what follows we define each single element of our DFSM, namely each element in the tuple (T, t_0, E, R) where T is a non-empty and finite set of states, t_0 is an initial state, E is an event queue, and R is a state transition relation. First of all, we define an event queue the triple (V, S, C) where $V = \{instances\ of\ m\ /\ m \epsilon\ ARM\ Instruction\ Set\}$, S is the number of slots in the queue which we set to the number of instructions supported by the actual microprocessor pipeline, and C is the contents of a queue, i.e. an ordered set of S elements of V. V in this paper is referred to as the queue vocabulary. We order the elements of C according to the order by which those instruction instances are prefetched by the ARM microprocessor, or more precisely in our case by the ARM emulator. With an instruction instance we mean an instruction which is ready to be considered for execution by a CPU, thus an instruction whose operands are determined already and hold a defined value Figure 4.

The subtract instruction which belongs to the ARM Instruction Set

SUB {<cond>} {S} <Rd>, <Rn>, <shifter_operand>

An instance of the ARM subtract instruction where R4 holds the value 64 in decimal

SUB R2, R4, #32

Fig. 4. An example of what we mean with an ARM instruction instance.

In order to detect possible memory access taintedness we need to keep track of generation of tainted data and their propagation throughout memory locations and CPU registers. For such a purpose we adapt the extended memory model proposed by Chen at al. in [25] and define what we call a memory taintedness island as a set of contiguous memory locations which hold tainted data. Furthermore, each register is classified as tainted if it holds tainted data. In our DFSM model at each moment of the vulnerability analysis interval we maintain only memory taintedness islands and tainted registers organizing them in data structures such as ordered lists. These two lists get possibly updated as new instructions are executed. A state of our DFSM consists in actually existent memory taintedness islands in the 4GB of address space of a MODBUS application, and in a set of those registers which are actually tainted. We treat banked registers individually as if they were different registers.

For example, if a banked register such as R13 at each moment refers to 1 of the 5 possible 32-bit physical registers depending on the current processor mode, we consider having 5 registers rather than 1 and maintain in a list the ones which are tainted. With regard to the definition of t_0 we consider the memory locations where input data comes from. In fact registers may get tainted only subsequently after loads of tainted data. On ARM microprocessors the standard way of performing I/O functions is through some special memory locations referred to as memory-mapped I/O[2]. A given process receives input data by loading from memory-mapped I/O locations. Storing to memory-mapped I/O locations is used for output. Thus, the initial state of our DFSM consists of a memory taintedness island created by addresses of memory-mapped I/O locations.

The state transition relation R is responsible for determining if memory access taintedness is in place and possibly updating the list of memory taintedness islands and/or the list of tainted registers. R uses the current state to determine if the memory address which the event received in input, i.e. an ARM instruction instance, is trying to access is tainted. Generally speaking, in ARM instruction instances the determination of a memory address, or the determination of several memory addresses in certain cases, depends on two factors, namely a base register and an offset. The offset in turn could be an immediate value, i.e. a constant such as #n explicitly specified, or a register-based value. A register holding an offset is often referred to as index register. Therefore, R consults the current state to see if either the base register or the index register (if existent) are tainted. If any of those two registers belong to the current list of tainted registers, then the memory address to be accessed by the event under consideration is classified as tainted. The state transition relation R employs the deci-

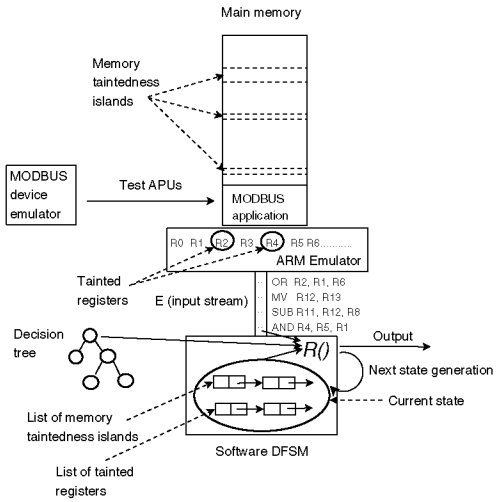

Fig. 5. General organization of our DFSM vulnerability analysis model.

sion tree given in Figure 2 to determine if a memory access taintedness is in place.

If the memory access taintedness decision tree leads to an affirmative value then R defines the data to be output, otherwise no data are output. The data possibly produced in output by our DFSM consists in a dump of the content of main memory, the list of memory taintedness islands, the list of tainted registers, and the ARM instruction instance which caused a memory access taintedness. After determining if memory access taintedness is in place and defining the data to be produced as output, R examines the event under analysis in order to define any changes to the lists of memory taintedness islands and tainted registers. In fact load instructions, load multiple instructions, data-processing instructions, arithmetic instructions, status register access instructions, and even branch instructions are likely to cause a change in the list of tainted registers. While store instructions and store multiple instructions are likely to cause a change in the list of memory taintedness islands. During a typical dynamic analysis process a MODBUS device emulator generates test cases and sends them to a MODBUS implementation running on an instrumented ARM emulator containing our DFSM.

A general description of the vulnerability analysis model we described in this paper is given in Figure 5. The test cases are generated through various techniques such as syntax-based techniques, sample PDU mutation, data-flow methods, path-oriented methods, etc., and they are out of the scope of this paper. As the MODBUS implementation under analysis executes on ARM emulator, the DFSM works on instruction instances, states and decision tree to determine memory access taintedness and possibly produce in output relevant information necessary for the identification of a potential memory vulnerability.

References

[1] Aleph1, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, volume 7, issue 49.
[2] ARM Limited, "ARM Technical Reference Manuals," http://www.arm.com/documentation/ARMProcessor_Cores/
[3] Anonymous, "Once upon a free()," *Phrack Magazine*, volume 9, issue 57.
[4] Bulba, and Kil3r, "Bypassing stackguard and stackshield," *Phrack Magazine*, volume 10, issue 56.
[5] C. Bellettini, and J.L. Rrushi, "FireBuff: A Defensive Approach Against Control-data and Pure-data Attacks," technical report of Università degli Studi di Milano.
[6] E.J. Byres, D. Hoffman, and N. Kube, "On Shaky Ground - A Study of Security Vulnerabilities in Control Protocols," *5th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human Machine Interface Technology*, American Nuclear Society, Albuquerque, USA, November 2006.
[7] F. Iwanitz and J. Lange, *OPC - Fundamentals, Implementation and Application*. Huthig Fachverlag ISBN 3-7785-2904-8, 2006.
[8] G. Balakrishnan, T. Reps and T. Teitelbaum, "WYSINWYX: What You See Is Not What You eXecute," *Proceedings of IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, (Switzerland), October 2005.
[9] Gera, and Riq "Advances in format string exploitation," *Phrack Magazine*, volume 11, issue 59.
[10] International Electrotechnical Commission, "IEC 60870-6-503 Telecontrol equipment and systems," part 6-503, Telecontrol protocols compatible with ISO standards and ITU-T recommendations - TASE.2 Services and protocols, second edition, 2004.
[11] J. Stamp, J. Dillinger, W. Young, and J. DePoy, "Common Vulnerabilities in Critical Infrastructure Control Systems," Sandia National Laboratories, Albuquerque, (USA), November 2003.
[12] Klog, "The Frame Pointer Overwrite," *Phrack Magazine*, volume 9, issue 55.
[13] K. Stouffer, J. Falco and K. Kent, "Guide to Supervisory Control and Data Acquisition (SCADA) and Industrial Control Systems Security," Special Publication 800-82, September 2006.
[14] L. Mora, "OPC Server security considerations," *Proceedings of SCADA Security Scientific Symposium*, USA, January 2007.
[15] M. Franz, "ICCP Exposed: Assessing the Attack Surface of the Utility Stack," *Proceedings of SCADA Security Scientific Symposium*, USA, January 2007.
[16] M. Zhivich, "Deadbolt," I3P factsheet.
[17] Modbus Organization, "MODBUS Application Protocol Specification," version 1.1a, 2004.
[18] N. Kube, and D. Hoffman, "Automated Testing of SCADA Protocols," *Proceedings of SCADA Security Scientific Symposium*, USA, January 2007.
[19] Nergal, "The advanced return-into-lib(c) exploits: PaX case study," *Phrack Magazine*, volume 11, issue 58.
[20] R. Earnshaw, "ELF for the ARM Architecture," January 2007.
[21] Schneider Automation, "MODBUS Messaging on TCP/IP Implementation Guide V1.0b," October 2006.
[22] Scut, and Team Teso, "Exploiting Format String Vulnerabilities," version 1.1, March 2001.
[23] Sh. Chen, J. Xu, N. Nakka, Z. Kalbarczyk and R.K. Iyer, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," *Proceedings of IEEE International Conference on Dependable Systems and Networks*, (Japan), July 2005.
[24] Sh. Chen, J. Xu, E.C. Sezer, P. Gauriar and R.K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," *Proceedings of the 14th USENIX Security Symposium*, pp. 177–192, (USA), August 2005.
[25] Sh. Chen, K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer, "Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities Using Pointer Taintedness Semantics," *Proceedings of 19th IFIP International Information Security Conference*, (France), August 2004.
[26] US-CERT, "Vulnerability Note VU#190617," http://www.kb.cert.org/vuls/id/190617