Chapter 11

COMBATING MEMORY CORRUPTION ATTACKS ON SCADA DEVICES

Carlo Bellettini and Julian Rrushi

Abstract Memory corruption attacks on SCADA devices can cause significant disruptions to control systems and the industrial processes they operate. However, despite the presence of numerous memory corruption vulnerabilities, few, if any, techniques have been proposed for addressing the vulnerabilities or for combating memory corruption attacks. This paper describes a technique for defending against memory corruption attacks by enforcing logical boundaries between potentially hostile data and safe data in protected processes. The technique encrypts all input data using random keys; the encrypted data is stored in main memory and is decrypted according to the principle of least privilege just before it is processed by the CPU. The defensive technique affects the precision with which attackers can corrupt control data and pure data, protecting against code injection and arc injection attacks, and alleviating problems posed by the incomparability of mitigation techniques. An experimental evaluation involving the popular Modbus protocol demonstrates the feasibility and efficiency of the defensive technique.

Keywords: SCADA systems, memory corruption attacks, Modbus protocol

1. Introduction

This paper describes the design and implementation of a run-time system for defending against memory corruption attacks on SCADA devices. SCADA systems are widely used to operate critical infrastructure assets such as the electric power grid, oil and gas facilities, and water treatment plants. Recent vulnerability analyses of SCADA protocol implementations have identified several memory corruption vulnerabilities such as buffer overflows and faulty mappings between protocol elements (handles and protocol data unit addresses) and main memory addresses [8, 27]. These include [19, 40] for Inter Control Center Protocol (ICCP) [20, 42] for OLE for Process Control (OPC) [21], respectively.

Very few defensive techniques have been devised specifically for SCADA systems. Consequently, most of the techniques discussed in this paper are drawn from efforts related to traditional computer systems. All these techniques, which the exception of pointer taintedness detection [10, 11], focus on protecting control data (e.g., saved instruction pointers, saved frame pointers and pointers in various control tables) from corruption. The techniques do not protect against attacks that corrupt non-control data, also called "pure data," which includes user identification data, configuration data and decision-making data [12]. Furthermore, the techniques only protect against pre-defined attack techniques. This problem, which is referred to as the "incomparability of defensive techniques" [34], is stated as follows: for any two categories of techniques A and B there are attacks prevented by A that are not prevented by B, and there are attacks prevented by B that are not prevented by A. The approach described in this paper overcomes the limitations of current defensive techniques in terms of attack vector coverage. Also, it protects against control data attacks as well as pure data attacks.

We have evaluated our defensive technique using a Modbus protocol [26] implementation running under Linux. In particular, we employed FreeMODBUS [44] running on a Debian machine (OS kernel 2.6.15) with 1024 MB main memory and an IA-32 processor with a clock speed 1.6 GHz. The Modbus client and server ran on the same physical machine and communicated via a null-modem cable between `/dev/ttyS1` and `/dev/ttyS0`. The experimental results indicate that the defensive technique is both feasible and efficient for real-time operation in industrial control environments.

2. Related Work

Simmons, *et al.* [38] have proposed code mutation as a technique for detecting buffer overflow exploits in SCADA systems before the exploits are executed. The technique involves altering SCADA executable code without changing the logic of the original algorithms. Code mutation is one of the first approaches proposed for protecting SCADA systems from memory corruption attacks. The other techniques discussed below were originally developed for traditional computer systems.

Instruction set randomization [6, 23] counters attacks that hijack the execution flow of a targeted program in order to execute injected shellcode. This defensive technique prevents the injected shellcode from being executed by encrypting program instructions with a random key; the encrypted code is loaded into memory and is decrypted just before it is processed by the CPU.

Kc, *et al.* [23] randomize ELF [40] executables by extending the `objcopy` utility to create a binary by XORing the original instructions with a 32-bit key. This key is then embedded in the header of the executable. When a new process is generated from a randomized binary, the operating system extracts the key from the header and stores it in the process control block. After the process is scheduled for execution, the key is loaded into a special register of the CPU via a privileged instruction; the key is then XORed with the instructions

before they are processed by the CPU. This approach protects against attacks on remote services, but not from local attacks. Also, it has a considerable performance cost and does not handle dynamic libraries. In fact, because a key is associated with a single process, it is difficult to use dynamic libraries whose code is shared by different processes. A solution is to copy the shared library code to a memory area where it could be XORed with the key of the corresponding process; this enables each process to create and use a private copy of the shared code. However, the memory consumption is high and the strategy conflicts with the rationale for having shared libraries.

Barrantes, *et al.* [6] have developed the RISE tool that generates a pseudo-random key whose length is equal to the total number of bytes of the program instructions. The instructions are XORed with the key when they are loaded into the emulator and are decrypted right before being executed. When a process needs to execute shared library code, the operating system makes a private copy of the code, encrypts it and stores the encrypted code in the virtual memory assigned to the process. Thus, the use of shared libraries consumes a large amount of main memory. RISE is based on `valgrind` [29] and adds a latency of just 5% to a program running under `valgrind`. Instruction set randomization cannot prevent control data attacks or pure data attacks that operate in a `return-into-libc` manner because these attacks do not use injected shellcode [28].

The instruction set randomization technique can be subverted. Sovarel, *et al.* [39] have demonstrated the efficiency of incremental key guessing where an attacker can distinguish between incorrect and correct key byte guesses. The incremental key guessing exploit injects either a return instruction (which is one byte long for the IA-32 processor) or a jump instruction (two bytes long) encrypted with one or two guessed bytes, respectively. If an attacker employs a return instruction in a stack overflow exploit [1], the stack is overflowed so that the saved frame pointer is preserved. The saved instruction pointer is then modified to point to the encrypted return instruction that is preliminarily injected and the original saved instruction pointer is stored next to it. If the guess of the encryption byte is correct, the CPU executes the injected return instruction and the vulnerable function returns normally; if the guess is incorrect, the targeted process crashes.

The subversion technique requires knowledge of the saved frame pointer and the saved instruction pointer. An attacker may also inject a short jump with a -2 offset. If the attacker's guess of the two encryption bytes is correct, the jump instruction is executed and proceeds to jump back to itself, causing an infinite loop; an incorrect guess causes the targeted process to crash. After successfully guessing the first byte or the first two bytes, the attacker changes the position of the byte(s) to be guessed and proceeds as in the previous step. Because a failed key guess causes the targeted process to crash, incremental subversion is possible only for programs that are encrypted with the same key every time they execute. Incremental subversion is also applicable to programs whose

forked children are encrypted with the same key, in which case the technique is directed at a child process.

PointGuard [15] XORs pointers with a 32-bit key when they are stored in main memory, and XORs them again with the same key right before they are loaded into CPU registers. The encryption key is generated randomly at load time and is kept secret. An attacker cannot corrupt a pointer with a value that is useful (from the attack point of view) as the encryption key is not known. This is because the corrupted pointer is decrypted before being loaded into a CPU register; thus, it will not accomplish the attacker's task [15]. However, PointGuard does not counter injected shellcode. Also, uncorrupted bytes in a partially-corrupted pointer are decrypted correctly. This means that an attacker can corrupt the least significant byte of a pointer in a process running on a Little Endian architecture and then employ a brute force technique to land in a memory location where shellcode or a program instruction is stored. The subversion may be carried out on any architecture by exploiting a format bug [18, 30, 37] to corrupt any one byte of a target pointer [2].

StackGuard [16] places a load-time-generated random value or a string terminating value (called a "canary") on the stack next to the saved instruction pointer. The integrity of the canary is checked upon function return. If the original value of the canary has been modified, a stack overflow has occurred, the incident is reported in a log file and program execution is aborted [16].

StackShield [43] creates a separate stack to hold a copy of each saved instruction pointer. It saves a copy of the saved instruction pointer at function call and copies the saved value back to the saved instruction pointer location at function return; this ensures that the function correctly returns to its original caller. However, Bulba and Kil3r [9] have shown that StackShield can be bypassed by corrupting the saved frame pointer using a frame pointer overwrite attack [24]. Also, StackGuard can be bypassed by attack techniques that do not need to pass through a canary in order to corrupt the saved instruction pointer of a vulnerable function. Examples include heap overflows, `longjmp` overflows, format strings and data/function pointer corruption.

The StackGuard limitation can be addressed by XORing a saved instruction pointer with a random canary at function call and at function return. The attack fails if the saved instruction pointer is corrupted without also modifying the corresponding canary. When a function completes its execution, XORing the canary with the corrupted instruction pointer will not yield a pointer to the memory location defined by the attacker.

FormatGuard [14] is a countermeasure against format string attacks. It compares the number of actual parameters passed to a format function against the number of formal parameters. If the number of actual parameters is less than the number of formal parameters, FormatGuard reports the fact as an incident in a log file and aborts program execution. Robins [35] has proposed `libformat`, a library that aborts program execution if it calls a format function with a format string that is writable and contains a `%n` format directive.

Baratloo, *et al.* [5] utilize the `libsafe` library to intercepts calls to functions that are known to cause buffer overflows and replace each function with one that implements the same functionality, but with buffer overflows restricted to the current stack frame. Note that `libsafe` does not protect against heap overflows or attacks that corrupt data/function pointers. With regard to format string vulnerabilities, `libsafe` rejects dangerous format directives such as `%n` that attempt to corrupt saved instruction pointers. Baratloo, *et al.* [4] have created the `libverify` library that offers the same kind of protection as StackShield. The main difference is in the way the copy of a saved instruction pointer is compared with the saved pointer on the stack at function return.

Krennmair [25] has developed ContraPolice for protecting against heap overflows [3, 13, 17, 22]. ContraPolice places a random canary before and after the memory region to be protected. Integrity checks are performed on the canary before exiting from a function that copies data to the protected memory region. Program execution is aborted when the canary indicates corruption.

The PaX kernel patch [33] for Linux employs non-executable pages for IA-32 to make the stack and heap non-executable [33]. Page privilege flags are used to mark pages as non-executable; a page fault is generated when a process accesses such a page. PaX then checks if a data access occurred or the CPU tried to execute an instruction (in which case the program execution is aborted). PaX performs well against shellcode injection, but not against attacks structured in a `return-into-libc` manner.

The defensive techniques discussed in this section focus on a few attacks, often on a single attack. None of the techniques can protect against all attacks that exploit memory corruption vulnerabilities, which raises the issue of incomparability of mitigation techniques. To the best of our knowledge, the only approach capable of protecting a process from control data and pure data attacks is pointer taintedness detection [10, 11]. A pointer is considered to be tainted when its value is obtained via user input or is derived from user input. If a pointer is tainted during the execution of a process, an attack is underway and a response should be triggered. Pointer taintedness detection is a effective technique, but it requires substantial changes to the underlying hardware to implement its memory model.

3. Countering Memory Corruption Attacks

The computer security community has traditionally relied on patching memory corruption vulnerabilities to combat control data and pure data attacks. However, software patches only address known vulnerabilities; the patched systems would still be exposed to attacks that exploit vulnerabilities that exist, but that have not been identified or disclosed. Also, patching SCADA systems often degrades their real-time performance characteristics.

Intrusion detection and response systems are often used for attack prevention. The problem is that an intrusion detection system generally performs its analysis and raises alarms after an intrusion has been attempted. Some modern intrusion detection systems (e.g., Snort [36]) perform real-time intrusion

prevention tasks. Similarly, a fast proactive defense mechanism could be used to block the exploitation of memory vulnerabilities.

Building attack requirement trees for known attacks that target memory corruption vulnerabilities can assist in developing defensive mechanisms. An attack requirement tree is a structured means for specifying conditions that must hold for an attack to be feasible [7]. The root node of the tree is the attack itself, while every other node is a requirement (or sub-requirement) for the attack. Internal nodes are either AND nodes or OR nodes. An AND node is true when all its child nodes are true; an OR node is true if if any one of its child nodes is true.

We use an attack requirement tree corresponding to a specific attack as a pattern for characterizing the family of memory corruption attacks. The example attack we consider smashes a buffer on a stack to corrupt the saved instruction pointer and return execution to injected shellcode. The attack requirement tree highlights two issues:

- The main instrument employed to satisfy one or more critical nodes in an attack requirement tree is the input data fed to a targeted process. In our example, the corrupting value of the saved instruction pointer and the shellcode injected into the address space of a targeted process constitute the input data (supplied by the attacker).

- The nodes are true because the attacker has full control of the input data in the address space of the targeted process. In our example, an attacker defines the content of shellcode and the shellcode is stored in memory in exactly the way the attacker defined it. Furthermore, an attacker can define the value that overwrites the saved instruction pointer. Thus, during a memory corruption attack, a saved instruction pointer is overwritten by the value defined by the attacker.

Obviously, it is not possible to eliminate the attack instrument. However, it is feasible to eliminate the control that an attacker has on the content of the input data stored in main memory. The idea is to prevent the attacker from being able to define the bytes stored in memory. As shown in Figure 1, the attacker develops shellcode that spawns a shell (Figure 1(a)) and feeds it along with other attack data to a targeted process. Figure 1(b) shows the shellcode stored in the main memory of a Little Endian Intel machine in the manner defined by the attacker. Under these conditions, an attacker could hijack program execution to the injected shellcode and cause its execution. If users, including malicious users, specify the content of input data while it is within the process that defines the content stored in main memory, an attacker cannot control the content of input data stored in main memory.

A process should store input data in main memory so that transformed data is converted to its original form before it is used. Figure 1(c) presents the shell-spawning shellcode encrypted in a stream cipher mode using the key `0xd452e957` and stored in main memory as ciphertext. The input data stored in main memory is different from the input data defined by an attacker at

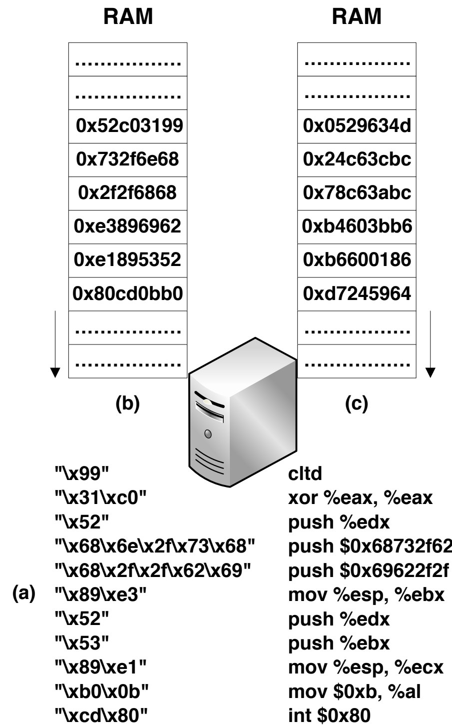

Figure 1. Potentially hostile data sanitized by stream cipher encryption.

the moment the attack data is provided as input to a targeted process. If the attacker hijacks control to the transformed shellcode, it is highly likely that the CPU will execute it, but the machine instructions in the transformed shellcode would not be recognized.

This situation also holds for a saved instruction pointer. An attacker is able to define (in the input data) the address of injected shellcode as the value that overwrites an instruction pointer saved on the stack. However, if a targeted process transforms the input data before storing it in memory, then the saved instruction pointer is overwritten with the transformed value, not with the address of the injected shellcode that the attacker intended. Consequently, while it is likely that execution will be hijacked, control will not pass to the injected shellcode. This technique applies to all attack techniques that exploit memory vulnerabilities by corrupting control data or pure data.

4. Proposed Approach

The principal idea behind enforcing logical boundaries on potentially hostile data is to encrypt input data before it enters the address space of a protected process. All the data read from sockets, environment variables, files and standard input devices are encrypted and then stored in main memory. The stream

cipher mode is used for encrypting and decrypting input data. Specifically, input data is encrypted by XORing it with a key; the ciphertext is decrypted by XORing it with the same key. Stream cipher encryption using the XOR operation is ideal for real-time systems because it is fast and efficient.

A protected process should preserve input data as ciphertext throughout its execution. Note that encryption defines a logical boundary in the address space of a protected process between potentially hostile data and control/pure data. A protected process should be aware of the logical boundaries of data. In particular, the protected process should be able to determine if a buffer needed by a machine instruction contains ciphertext, in which case, the buffer data has to be decrypted before it is used. Unencrypted buffer data can be used directly.

The decryption of a buffer containing ciphertext should be performed in compliance with the principle of least privilege. At any point in time, only the data items required by an instruction are decrypted and made available to the instruction. For example, format functions parse a format string one byte at a time. If the byte read is not equal to `%`, it is copied to output; otherwise, it means that a format directive (e.g., `%x`, `%d` or `%n`) is encountered and the related value is retrieved from the stack. In this case, if the format function needs to parse a byte of the format string, only that byte is decrypted and made available; the remaining bytes are in ciphertext.

The principle of least privilege applied at such a low level prevents a function from operating on data other than the data it requires. To alleviate the performance penalty, the number of bytes decrypted at a time could be slightly greater than the number of bytes needed by a function – but such an action should be performed only if it is deemed safe. The plaintext version of data should not be preserved after an operation completes; otherwise, plaintext attack data (e.g., shellcode) could be exploited in an attack. When a protected process has to copy or move input data to another memory location, the data is stored as ciphertext as well. Data created by concatenating strings, where at least one string is stored as ciphertext, should also be preserved as ciphertext.

Each process generates a random sequence of bytes that is used as a key for encrypting/decrypting input data. This key is generated before the main function is executed. Child processes also randomly generate their own keys immediately after being forked but before their program instructions are executed. Thus, parent and child processes generate and use their own keys.

A child process that needs to access ciphertext data stored and handled by its parent process uses the key of the parent process. To counter brute force attacks, the lifetime of a key should extend over the entire period of process execution. A brute force attack against a non-forking process or against a parent process would require the key to be guessed correctly at the first attempt. If this is not done, the protected process crashes and a new key is generated the next time the process is created. The same situation holds for child processes: if the key of the targeted child process is not guessed during the first try, the process crashes and a new child process with a new key is forked when the

attacker attempts to reconnect to the parent process. A brute force attack fails because the new child process generates and uses a new random key.

A key is stored within the address space of the process that created it and is available to the process throughout its execution. Care should be taken to guarantee the integrity of the key when it is stored and loaded from memory.

By encrypting process input, intervention occurs at two vital points of an attack. The first point is when attack data is stored in buffers to execute various types of buffer overflow attacks that corrupt control data or pure data. The second point is when attack data is stored in main memory.

Without the encryption key, an attacker cannot place the correct data required to execute a buffer overflow attack. In addition, the attacker cannot inject valid shellcode because it would have to be in ciphertext (like the original shellcode). In the case of a format string attack, due to the application of the least privilege principle for decryption, when a format function processes a directive (e.g., `%n`), the bytes to be stored and their starting memory location are not available. Thus, the attacker loses the ability define a value and the address where the value is to be stored.

The second point of intervention is when attack data is stored in memory. This is when an attacker, who has corrupted control data or pure data, could cause execution to return to shellcode. It is also at this point that an attacker could inject data into the stack by passing it as an argument to a function. This is referred to as arc injection (a `return-into-libc` type of attack). Preserving potentially malicious data as ciphertext in main memory thus eliminates the ability of an attacker to use shellcode or to perform arc injection.

Note that an intervention at the first point prevents an attacker from transferring control to the desired memory location. Therefore, an intervention at the second point can be considered to be redundant. Nevertheless, we include it in the specification of our protection strategy as a second line of defense.

5. Experimental Results

Our experimental evaluation of the protection technique employed a Modbus control system. Modbus is an application-layer protocol that enables SCADA devices to communicate according to a master-slave model using various types of buses and networks. Modbus was chosen for the implementation because it is popular, simple and representative of industrial control protocols. We instrumented the code of FreeMODBUS, a publicly available Modbus implementation, to evaluate the feasibility and efficiency of the protection technique. Several vulnerabilities were introduced into FreeMODBUS in order to execute memory corruption attacks.

The implementation of the defensive technique involved kernel domain intervention and user domain intervention. Kernel domain intervention requires system calls to be extended so that all input data to protected processes is encrypted. The second intervention takes place in the user domain, where additional instructions are introduced to Modbus programs to enable them to access and use the encrypted information.

5.1 Kernel Domain Intervention

Logical boundaries on potentially hostile data are enforced by having the operating system encrypt all input data delivered to protected processes. This functionality is best implemented as a kernel domain activity that ensures that input data is encrypted as soon as it is generated. Our approach involved extending system calls responsible for reading data (e.g., `read()`, `pread()`, `readv()`, `recv()`, `recvfrom()` and `recvmsg()`).

The extension incorporates an additional parameter (encryption key) for each system call, which is used to encrypt the input data passed to a protected process. Thus, when a Modbus process needs a service from the operating system, it fills CPU registers with system call parameters, one of which is encryption key. Next, the Modbus process generates an interrupt and control is transferred to the procedure at the system call kernel location, which checks the system call number, uses it to index the system call table, and issues the system call. Each extended system call retrieves input data and encrypts it with the key received from the calling process in the user domain.

5.2 User Domain Intervention

A protected Modbus process in the user domain always receives its input data as ciphertext; therefore, it should be provided with the appropriate mechanisms to use and protect input data. These tasks may be performed at compile time using an *ad hoc* GCC extension module, or by manually instrumenting Modbus program source code before compiling it, or by employing a binary rewriting tool to insert additional instructions in a Modbus binary. Our FreeMODBUS experiments implemented the first two methods.

Modbus data items include discrete inputs (1-bit read-only data from the I/O system), coils (1-bit data alterable by Modbus processes), input registers (16-bit read-only data provided by the I/O system), and holding registers (16-bit data alterable by Modbus processes). These items are stored in the main memory of Modbus devices and addressed using values from 0 to 65,535.

A Modbus implementation generally maintains a pre-mapping table that maps a data item address used by the Modbus protocol to the actual location in memory where the data item is stored. In general, Modbus instructions that work on potentially hostile data include those that consult the pre-mapping table and those that read from or write to the memory area holding Modbus data items. Thus, it is relatively easy to identify points in a Modbus program where instrumentation instructions should be inserted. Note that this is much more difficult to accomplish in traditional computer systems.

The task of spotting potentially malicious data is essentially a taintedness tracking problem. This makes our approach specific to implementations of control protocols such as Modbus. The main GCC executable named `gcc` is a driver program that calls a preprocessor and compiler named `cc1`, an assembler named `as`, and a linker named `collect2`. GCC transforms the preprocessed source code into an abstract syntax tree representation, then converts it into

a register transfer language (RTL) representation, before it generates assembly code for the target platform [31]. A GCC extension module instruments Modbus program code by adding additional assembly language instructions during assembly code generation. The CPU registers are visible at this stage; therefore, it is easier to repair the assembly instructions that store (on the stack) the contents of the CPU register that holds the encryption key in order to make that register available for some other use. Therefore, the additional assembly instructions that the GCC extension module inserts into the final Modbus assembly file implement the protection technique without affecting the computations of a protected Modbus program.

Assembly instructions are inserted at several points in a program. For example, the GCC extension module inserts in the `.init` section assembly instructions that call an interface to the kernel's random number generator that provides the 32-bit key. These instructions read four bytes from the file `/dev/urandom`. The `.init` section is appropriate for this task because it usually holds instructions for process initialization [40]. GCC also inserts assembly instructions in `.init` that issue the `mmap()` system call to allocate mapped memory from the Linux kernel.

Other instructions are inserted to store keys at the correct memory locations and to invoke `mprotect()` to designate these memory locations as read-only. Instructions that generate and store a key are also inserted in the process that forks child processes; these instructions are positioned after the child process is forked but before its original instructions are executed.

During our experiments we considered all Modbus data items (including input registers and discrete inputs) as potentially hostile; consequently, they were stored in memory as ciphertext. Just before Modbus instructions processed data items, their encrypted versions were copied to a mapped memory region, where they were decrypted in place. We refer to this portion of memory as the "temporary storage area." Note that the GCC extension module inserts instructions in `.init` that issue a call to `mmap()` to allocate the mapped memory used as the temporary storage area, and then issue a call to `mprotect()` to designate the memory as non-executable.

The GCC extension module thus generates additional assembly language instructions for every function that operates on potentially hostile data. These additional instructions are interleaved with the assembly instructions that implement the original functions. The additional instructions copy the ciphertext data to the temporary storage area and decrypt portions of the ciphertext as and when function instructions need them. Some of the original assembly instructions are modified in order to obtain data from the temporary storage area rather than from the original locations. The GCC extension module also inserts assembly instructions in the `.fini` section. These instructions, which are executed at process termination, issue a call to `munmap()` to remove previous memory mappings.

Another technique for implementing defensive capabilities in a Modbus program is to instrument its source code. In FreeMODBUS, C instructions that

operate on potentially hostile data are identified and extra C code for decrypting ciphertext is introduced. Likewise, additional C code is included for generating keys and implementing the temporary storage area.

5.3 Performance Overhead

When analyzing the performance costs involved in enforcing logical boundaries on potentially hostile data, we noticed that most of the overhead is due to constant invocations of the `mmap()` and `mprotect()` system calls. However, these calls can be eliminated if special care is taken when storing keys. Using `mmap()` and `mprotect()` calls to designate the temporary storage area as non-executable is reasonable only if they do not affect the real-time performance of the Modbus device being protected.

One question that arises relates to the feasibility of encrypting input data at the kernel level. Since our technique already compiles a program and makes it pass the encryption key to a system call, the program could directly encrypt data returned from the system call. In other words, there is no need for the kernel to perform the initial encryption of input data.

Our motivation for performing input data encryption at the kernel level is to reduce the memory consumption incurred by the protection strategy. System calls as implemented in most operating systems read input data and store them in internal buffers. Therefore, an encryption extension of a system call would encrypt input data in place so that it does not consume additional memory. If a process were to encrypt input data in the user domain, then it certainly would not place it directly in the destination buffer. This is exactly what our technique tries to avoid. Consequently, a protected process would have to allocate an additional buffer, place the input data there, encrypt it and copy the encrypted data to the original destination. If the process receiving input data from a system call were to encrypt the data itself, it would consume additional memory while introducing the overhead of executing extra instructions.

The computational overhead depends on the number of data items in Modbus device memory, the number of data items incorporated in Modbus request frames in typical transactions, and the number of instructions that operate on input data. The instruction counter is a CPU performance equation variable [32] that is incremented by our protection technique. In fact, our technique requires additional instructions to be inserted in Modbus code. The number of instructions that perform basic boundary enforcement operations such as generating, storing and deleting keys or creating and releasing the temporary storage area is constant for all protected processes. The number of instructions that transfer data to the temporary storage area and decrypt them there depends on how often a Modbus protected process operates on ciphertext.

The number of clock cycles per instruction (CPI) and the clock cycle time depend on the CPU architecture. Enforcing logical boundaries on potentially hostile data in a Modbus process has no affect on the pipeline CPI, but it causes an overhead in the memory system CPI and contributes to the miss rate because it uses additional memory. The performance overhead introduced by

enforcing logical boundaries on potentially malicious data during the execution of Modbus processes varies from 0.8% for transactions with typical frames that request a few data items to 2% for transactions with full 253-byte protocol data units. Furthermore, our protection technique does not rely on complementary defensive techniques such as address space randomization or StackGuard. Consequently our technique avoids the performance penalties introduced by operational security mechanisms.

6. Conclusions

Control data and pure data attacks can be countered by preserving input data as ciphertext in main memory and by instrumenting SCADA device code to decrypt potentially malicious data before processing according to the principle of least privilege. Our experiments demonstrate that the kernel-level stream encryption of input data using random keys preserves the logical boundary between potentially malicious data and clean data in the address space of processes. This enforcement of logical boundaries causes attackers to lose the precision with which they can corrupt control data and pure data. It also eliminates the execution of injected shellcode and the use of injected pure data while preserving the functionality of the protected processes. To the best of our knowledge, the protection technique is the only one that is capable of combating pure data attacks on C/C++ implementations of SCADA protocols such as Modbus without requiring hardware modification. We hope that this work will stimulate efforts focused on defending against unknown and zero-day attacks that rely on memory corruption.

References

[1] Aleph One, Smashing the stack for fun and profit, *Phrack*, vol. 7(49), 1996.

[2] S. Alexander, Defeating compiler-level buffer overflow protection, *;login: The USENIX Magazine*, vol. 30(3), pp. 59–71, 2005.

[3] Anonymous, Once upon a `free()`, *Phrack*, vol. 10(57), 2001.

[4] A. Baratloo, N. Singh and T. Tsai, Transparent run-time defense against stack smashing attacks, *Proceedings of the USENIX Annual Technical Conference*, 2000.

[5] A. Baratloo, T. Tsai and N. Singh, `libsafe:` Protecting critical elements of stacks, White Paper, Avaya, Basking Ridge, New Jersey (pubs.research .avayalabs.com/pdfs/ALR-2001-019-whpaper.pdf), 1999.

[6] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic and D. Zovi, Randomized instruction set emulation to disrupt binary code injection attacks, *Proceedings of the Tenth ACM Conference on Computer and Communications Security*, pp. 281–289, 2003.

[7] C. Bellettini and J. Rrushi, SCADA protocol obfuscation: A proactive defense line in SCADA systems, presented at the *SCADA Security Scientific Symposium*, 2007.

[8] C. Bellettini and J. Rrushi, Vulnerability analysis of SCADA protocol binaries through detection of memory access taintedness, *Proceedings of the IEEE SMC Information Assurance and Security Workshop*, pp. 341–348, 2007.

[9] Bulba and Kil3r, Bypassing StackGuard and StackShield, *Phrack*, vol. 10(56), 2000.

[10] S. Chen, K. Pattabiraman, Z. Kalbarczyk and R. Iyer, Formal reasoning of various categories of widely exploited security vulnerabilities by pointer taintedness semantics, in *Security and Protection in Information Processing Systems*, Y. Deswarte, F. Cuppens, S. Jajodia and L. Wang (Eds.), Kluwer, Boston, Massachusetts, pp. 83-100, 2004.

[11] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk and R. Iyer, Defeating memory corruption attacks via pointer taintedness detection, *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 378–387, 2005.

[12] S. Chen, J. Xu, E. Sezer, P. Gauriar and R. Iyer, Non-control data attacks are realistic threats, *Proceedings of the Fourteenth USENIX Security Symposium*, pp. 177–192, 2005.

[13] M. Conover and w00w00 Security Team, w00w00 on heap overflows (www.w00w00.org/files/articles/heaptut.txt), 1999.

[14] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen and J. Lokier, FormatGuard: Automatic protection from `printf` format string vulnerabilities, *Proceedings of the Tenth USENIX Security Symposium*, pp. 191–200, 2001.

[15] C. Cowan, S. Beattie, J. Johansen and P. Wagle, PointGuard: Protecting pointers from buffer overflow vulnerabilities, *Proceedings of the Twelfth USENIX Security Symposium*, pp. 91–104, 2003.

[16] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks, *Proceedings of the Seventh USENIX Security Symposium*, pp. 63–78, 1998.

[17] I. Dobrovitski, Exploit for CVS `double free()` for Linux pserver, Neohapsis Archives (www.security-express.com/archives/fulldisclosure/2003-q1/0545.html), 2003.

[18] Gera and Riq, Advances in format string exploitation, *Phrack*, vol. 10(59), 2002.

[19] iDefense Labs, LiveData Protocol Server heap overflow vulnerability, Sterling, Virginia (labs.idefense.com/intelligence/vulnerabilities/display.php?id=523), 2007.

[20] International Electrotechnical Commission, Telecontrol Equipment and Systems – Part 6-503: Telecontrol Protocols Compatible with ISO Standards and ITU-T Recommendations – TASE.2 Services and Protocol, IEC Publication 60870-6-503, Geneva, Switzerland, 2002.

[21] F. Iwanitz and J. Lange, *OPC – Fundamentals, Implementation and Application*, Huthig, Heidelberg, Germany, 2006.

[22] M. Kaempf, Vudo `malloc` tricks, *Phrack*, vol. 11(57), 2001.

[23] G. Kc, A. Keromytis and V. Prevelakis, Countering code injection attacks with instruction set randomization, *Proceedings of the Tenth ACM Conference on Computer and Communications Security*, pp. 272–280, 2003.

[24] Klog, Frame pointer overwriting, *Phrack*, vol. 9(55), 1999.

[25] A. Krennmair, ContraPolice: A `libc` extension for protecting applications from heap smashing attacks (synflood.at/papers/cp.pdf), 2003.

[26] Modbus IDA, MODBUS Application Protocol Specification v1.1a, North Grafton, Massachusetts (www.modbus.org/specs.php), 2004.

[27] L. Mora, OPC exposed: Part I, presented at the *SCADA Security Scientific Symposium*, 2007.

[28] Nergal, Advanced `return-into-lib(c)` exploits: PaX case study, *Phrack*, vol. 10(58), 2001.

[29] N. Nethercote and J. Seward, `valgrind`: A program supervision framework, *Electronic Notes in Theoretical Computer Science*, vol. 89(2), pp. 44–66, 2003.

[30] NOP Ninjas, Format string technique (julianor.tripod.com/bc/NN-formats.txt), 2001.

[31] D. Novillo, From source to binary: The inner workings of GCC, *Red Hat Magazine* (www.redhat.com/magazine/002dec04/features/gcc), December 2004.

[32] D. Patterson and J. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, San Francisco, California, 2007.

[33] PaX-Team, Documentation for the PaX Project (pax.grsecurity.net/docs), 2008.

[34] J. Pincus and B. Baker, Mitigations for low-level coding vulnerabilities: Incomparability and limitations, Microsoft Corporation, Redmond, Washington, 2004.

[35] T. Robbins, `libformat`, 2001.

[36] M. Roesch and C. Green, Snort Users Manual 2.3.3, Sourcefire (www.snort.org/docs/snort_manual), 2006.

[37] scut and team teso, Exploiting format string vulnerabilities (julianor.tripod.com/bc/formatstring-1.2.pdf), 2001.

[38] S. Simmons, D. Edwards and N. Wilde, Securing control systems with multilayer static mutation, presented at the *Process Control Systems Forum Annual Meeting* (www.pcsforum.org/events/2007/atlanta/documents/west.pdf), 2007.

[39] A. Sovarel, D. Evans and N. Paul, Where's the FEEB? The effectiveness of instruction set randomization, *Proceedings of the Fourteenth USENIX Security Symposium*, pp. 145–160, 2005.

[40] TIS Committee, Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (www.x86.org/ftp/manuals/tools/elf.pdf), 1995.

[41] US-CERT, LiveData ICCP Server heap buffer overflow vulnerability, Vulnerability Note VU#190617, Washington, DC (www.kb.cert.org/vuls/id/190617), 2006.

[42] US-CERT, Takebishi Electric DeviceXPlorer OPC Server fails to properly validate OPC server handles, Vulnerability note VU#926551, Washington, DC (www.kb.cert.org/vuls/id/926551), 2007.

[43] Vendicator, StackShield: A stack smashing technique protection tool for Linux (www.angelfire.com/sk/stackshield), 2000.

[44] C. Walter, FreeMODBUS: A Modbus ASCII/RTU and TCP implementation (v1.3), FreeMODBUS, Vienna, Austria (freemodbus.berlios.de), 2007.