COMPUTERS & SECURITY XXX (2010) 1–15

available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A study of self-propagating mal-packets in sensor networks: Attacks and defenses

Qijun Gu, Christopher Ferguson, Rizwan Noorani*

Department of Computer Science, Texas State University-San Marcos, 601 University Dr, San Marcos, TX 78666, USA

ARTICLE INFO

Article history:
Received 3 March 2010
Received in revised form
14 September 2010
Accepted 9 October 2010

Keywords:
Mal-packet
Buffer overflow
Memory fault
Code injection
Control flow
Sensor security

ABSTRACT

Since sensor applications are implemented in embedded computer systems, cyber attacks that compromise regular computer systems via exploiting memory-related vulnerabilities present similar threats to sensor networks. However, the paper shows that memory fault attacks in sensors are not the same as in regular computers due to sensor's hardware and software architecture. In contrast to worm attacks, mal-code carried by exploiting packets cannot be executed in sensors built upon Harvard architecture. Therefore, the paper proposes a range of attack approaches to illustrate that a mal-packet, which only carries specially crafted data, can exploit memory-related vulnerabilities and utilize existing application code in a sensor to propagate itself without disrupting the sensor's functionality. The paper shows that such a mal-packet can have as few as 17 bytes. A prototype of a 27-byte mal-packet has been implemented and tested in Mica2 sensors. Simulation shows that the propagation pattern of such a mal-packet in a sensor network is very different from worm propagation. Mal-packets can either quickly take over the whole network or hardly propagate under different traffic situations. The paper also develops two defense schemes (S2Guard and S2Shuffle) based on existing defense techniques to protect sensor applications. The analysis shows that they only incur a little overhead and can stop the propagation of mal-packets.

1. Introduction

Applications in sensor networks have been researched and developed for years. However, most of the security work focused on threats to networking and communication protocols. Worm attacks exploiting memory-related vulnerabilities show that attackers can compromise an entire network without breaking protocols. In the Internet, malicious attackers often break into computer systems by exploiting vulnerabilities arising from low-level memory faults, e.g., stack overflow (One, 1996), format string vulnerability (Newsham, 2001), integer overflow (Starzetz, 2001), double free (Anonymous, 2001), heap overflow (Kaempf, 2001), return-to-libc (Nergal, 2001), etc. Such cyber attacks in regular computer systems lead us to considering similar threats in sensor networks.

As sensors use very simple embedded systems, they do not have sophisticated operating systems to manage code for safety. Simple operating systems (TinyOS; Mantis) have been developed for embedded systems. However, they do not distinguish kernel mode or user mode when executing an instruction, and application data is neighboring to system data. Hence, one application routine can easily access data of other routines or be invoked by other routines. Furthermore, C language (Gay et al., 2003) is popular in developing sensor applications because of its convenience for coding and maintenance over assembly language. Open source based sensor applications have been developed as well. Consequently,

* *Corresponding author.*
E-mail address: qg11@txstate.edu (Q. Gu).
0167-4048/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2010.10.002

applications share more and more common code as they use similar development environments. Hence, memory fault attacks based on the same principle in regular computers become threats to sensor networks too.

The paper studies exploitation techniques in sensors built upon Harvard architecture, which has unique characteristics and presents both advantages and disadvantages to attackers. In terms of an attacker's advantage, sensors in the same network mostly use homogeneous devices and software. An attacker can capture one sensor and read application code from a JTAG interface that can be found in most embedded systems (Jack, 2006). It would be easier for attackers to obtain the source code if the applications are open source. Therefore, if one sensor can be compromised due to a vulnerability in the application code, all other sensors that use the same device and code in the same network can be compromised due to the same vulnerability. In terms of attacker's disadvantage, injected mal-code commonly found in worm packets is hard to execute in sensors, because sensor's memory is organized differently: (a) program memory is physically separated from data memory; (b) application codes are often write-protected in program memory so that sensors can work reliably in a hostile environment. Attacker's capability in sensor networks is thus much limited.

This paper explores *what an attacker can exploit and accomplish in a sensor with a mal-packet*. The paper does not address how to discover new vulnerabilities in existing applications. The paper shows that if a vulnerability does exist, an attacker can achieve certain malicious goals even if the attacker cannot execute any of his/her own code. Unlike worm packets in the Internet, mal-packets in a sensor network do not carry mal-code. A sensor mal-packet carries only mal-data but misuses existing code in a sensor to accomplish its attack. The paper proposes a range of attack approaches with which a mal-packet can exploit memory-related vulnerabilities and utilize existing application code in a sensor to propagate itself. Attackers may use these techniques to threaten a sensor network in other possible ways. For example, an attacker can trigger a vulnerable sensor to send a false but legitimate message. The attacker himself may not be able to send a false message in the network, but the attacker can use the exploitation techniques discussed in the paper to use a mal-packet to inject a false message into a vulnerable sensor and then invoke a legitimate routine inside the vulnerable sensor to add credentials to the message and then send it to other sensors.

The paper develops two defense schemes to protect sensor applications: S2Guard based on StackGuard (Cowan et al., 1998) and S2Shuffle based on code space randomization (Linn and Debray, 2003; Pax). Compared with other existing defense schemes, the two defense schemes are suitable in the current software and hardware architecture of sensor systems. The paper examines *how the two defense schemes can be applied in sensors and to what extent they can stop attacks*. The paper systematically studies the overhead of the two defense schemes and their security features to counteract self-propagating mal-packets.

The paper contributes in four aspects. First, the paper identifies a range of approaches for a mal-packet to invoke routines without disrupting a sensor's normal functionality. The paper shows that packets carrying only mal-data can accomplish similar attacks as those carrying mal-code. Second, the paper demonstrates that a mal-packet can propagate itself via utilizing a transmission function in most sensor applications. The propagation can be accomplished with as few as 17 bytes of data carried in a mal-packet. Third, the simulation of mal-packet propagation illustrates that mal-packets in a sensor network behave quite differently from worm packets due to the underlying transmission protocol and exploitation techniques. The effectiveness of attacks using mal-packets in a sensor network is highly determined by the condition of network traffic. Fourth, the paper shows that the existing defense techniques can effectively stop the propagation of mal-packets. However, the defenses also disrupt the operation of sensor networks.

The remainder of the paper is organized as follows. Section 2 overviews related works on attacks, worm propagation and defenses. Section 3 discusses the main challenges in exploiting vulnerabilities in sensor applications and gives the details of exploitation approaches to make a self-propagating mal-packet. Section 4 studies and compares two defense schemes against sensor mal-packets. Section 5 describes a prototype of mal-packet that has been implemented. Two defense schemes are implemented and tested for their overhead and performance. Simulation is also conducted to study propagation patterns of mal-packets in sensor networks. Finally, the paper is concluded in Section 6.

2. Related works

2.1. Attacks

Many attackers exploit vulnerabilities due to memory fault in current computer systems. Such attacks can be categorized as control data attacks (Govindavajhala and Appel, 2003; Smirnov, 2005) and non-control data attacks (Christodorescu and Jha, 2003; Chen et al., 2005). Control data refers to the code addresses that are loaded in the program counter (PC) at some point during program execution. This paper shows that sensor applications are susceptible to attacks that alter the control flow to utilize existing routines by overwriting control data and non-control-data.

Return addresses and function pointers are two major types of control data that attackers are interested in altering and exploiting. In a typical "stack smashing" attack (One, 1996), a return address in the stack is overwritten to the address where injected code is executed when the function (corresponding to the current stack frame) returns. When the target program's control data is modified, attackers can execute injected malicious code or out-of context library code at the memory address pointed to by the altered control data. Various attack techniques emerged to overwrite control data via exploiting vulnerabilities of format string error (Newsham, 2001), double-free error (Anonymous, 2001), heap overflow (Kaempf, 2001), return-to-libc (Nergal, 2001), etc.

The paper studies attacks in sensors similar to the return-oriented attack techniques (Shacham, 2007; Buchanan et al., 2008). However, the paper proposes composite attack techniques that are not simply applying the known exploitation techniques. The paper shows that unique challenges present

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

in exploiting sensors and making self-propagating mal-packets. The paper focuses on techniques that address the challenges. The proposed techniques exploit existing code to make self-propagate mal-packets without introducing disruption into sensors. The proposed techniques require a sequence of actions, including return-oriented attacks, to achieve the attack goal.

Non-control-data attacks, based on another principle, also threaten networks. It is found (Kruegel et al., 2005; Xu et al., 2004) that attackers can perform malicious actions via carefully crafting a legitimate execution sequence of application code. Many real-world software applications are susceptible to non-control-data attacks (Chen et al., 2005). In such an attack, attackers examine the code to find out "which data within a target application is critical to security other than control data, whether the vulnerabilities exist at appropriate stages of execution that can lead to eventual security compromises, and whether the severity of security compromises is equivalent to that of traditional control data attacks." Vulnerability for non-control-data attacks is highly dependent on the semantic of the software.

2.2. Worm propagation

In the analysis of worm propagation (Staniford et al., 2002), the Kermack-Mckendrick (KM) model, which was originally used in the analysis of disease spreading and control in a society, lays the major theoretical foundation to capture and predict the propagation of worms in the Internet. The KM model assumes that all entities are peers and one of them is a disease source. When a vulnerable entity is touched by the disease source, the entity will be infected. The infected entity will then randomly probe other entities in order to infect them. The infection continues until all vulnerable entities are infected. Assume that a society has N vulnerable entities and each infected entity randomly probes r entities per second. If I_t entities have been infected at time t, the KM model shows that I_t satisfies the logistic equation, where N_0 is the number of all entities, $\alpha = \frac{rN}{N_0}$, and $T = \frac{1}{\alpha}\ln(\frac{N}{I_0} - 1)$.

$$I_t = N\frac{e^{\alpha(t-T)}}{1 + e^{\alpha(t-T)}} \quad (1)$$

The KM model shows that the propagation of a worm goes through three phases. The propagation starts slowly until sufficient vulnerable hosts have been infected. Then, the propagation accelerates at an exponential rate in the middle. Finally, the propagation slows down when most of the vulnerable hosts have been infected. By changing the probing strategies of infected hosts and the size of initial worm population, variations of the KM model have been derived to study the other types of worm propagation (Staniford et al., 2002; Zou et al., 2003).

This research shows that the propagation of mal-packets in sensor networks presents different features. The propagation of worm on the Internet is among fully connected computers. Researchers have shown that malware propagation in mobile networks is quite different (Yan and Eidenbenz, 2009; Fleizach et al., 2007). In particular, the worm propagation among proximity devices has been studied in Bluetooth networks (Yan and Eidenbenz, 2009). In this research, a mal-packet broadcasts itself to infect sensors in its proximity area. Since the broadcast mechanism is quite different from the infection techniques used by a Bluetooth worm, the mal-packet propagation does not have an infection cycle as a Bluetooth worm and thus presents different characteristics.

2.3. Defenses

Since mal-packets alter the control flow to utilize existing routines and propagate themselves, defense techniques that are proposed against control flow attacks may be applicable. The main ideas include (1) bug detection, (2) run-time detection, (3) preventing overwriting control data, (4) randomizing address space, and (5) software-based secure sensor OS. In the following, we examine their applicability and limitations if being deployed against self-propagating mal-packets in sensor networks. Based on our comparison, we select approaches (3) and (4) for further analysis in Section 4.

(1) Memory-related vulnerabilities are fundamentally induced by programming bugs. Tools have been developed to find bugs in source code. Bug detection techniques (Cadar et al., 2006; Godefroid et al., 2008; Tan et al., 2008; Akritidis et al., 2009) are used to examine source code before generating final executables. They analyze source code according to certain patterns. Another type of bug detection tool (Kruegel et al., 2005; Xu et al., 2004) uses static analysis to analyze control and data flow in an application. They follow the execution of an application, identify possible unsafe states in the execution, and thus detect bugs. Although they are effective in detecting certain types of bugs, they face challenges as applications become more and more complicated and programmers may not have sufficient security awareness.
(2) Run-time tools (Costa et al., 2005; Kiriansky et al., 2002) have also been developed to capture run-time patterns caused by attacks. They monitor execution of application code, and intervene when a symptom of buffer overflow attacks presents. Sensors do not have sufficient resources or effective mechanisms to deploy and support these tools. Another type of run-time tools (Wang et al., 2006) inspect incoming packets and detect and filter suspicious packets if they contain a sequence of code. This tool cannot be used either, because a sensor mal-packet does not carry any code, but only mal-data.
(3) Several techniques were developed to stop attackers from overwriting control data. The ideas are to place canary values next to return addresses (Cowan et al., 1998), make a copy of return addresses (Vendicator), reorder local variables and function arguments and copy function pointers (Etoh and Yoda, 2001), or encrypt function pointers (Cowan et al., 2003). However, these techniques need a secure data area to store critical data for defense. In simple sensors, such secure memory spaces may not exist and thus sophisticated attacks may evade these techniques.
(4) Observing that a mal-packet needs to alter the control flow to execute code at known addresses, randomizing address space (Linn and Debray, 2003; Pax) could be a valid defense approach. The approach randomizes the base address of

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

the stack, heap and code segments at load or link time in regular computers. Functions can also be transformed in source code so that function addresses are randomized in final executables (Bhatkar et al., 2003). A similar idea (Yang et al., 2008) is applied in sensor networks that use a limited number of software versions to diversify the actual code implemented in each sensor. Such software diversity effectively restrain sensor worms from propagation.

(5) A few schemes have been developed to provide memory safety in TinyOS. Safe TinyOS (Regehr et al., 2006) treats function pointers as safe, sequence and wild (the later two are unsafe), and transforms an application's source code to the code using safe pointers. Safe TinyOS also provides a software framework that has a kernel space for OS-related modules and an extension space for untrusted application modules. Harbor (Kumar et al., 2007) uses software-based fault isolation to restrict application memory accesses and control flow to protect domains within the address space. It puts application modules in different protection domains and uses a control flow manager to ensure that function calls never flow out of a domain except via calls to functions exported by the kernel or modules in other domains. Both schemes work on the source codes of applications.

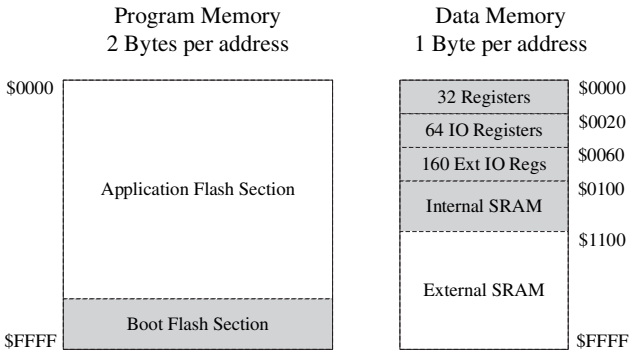

Fig. 1 – Memory structure of ATmega128. SP shows the stack pointer that is changed following how the code is executed. *H(x)* and *L(x)* are the high 8 bits and the low 8 bits of a 16-bit address x. Function B is not showed in the figure because its own control flow is not affected under attack. "…" is the data in stack that does not affect the control flow.

3. Exploitation techniques

This section presents a set of techniques to make self-propagating mal-packets. Although a buffer overflow flaw can be identified in a sensor application, it does not necessarily indicate that attackers can exploit the vulnerability. The proposed techniques in this section will address the major challenges for exploiting a buffer overflow.

3.1. *Challenges of exploiting sensors*

In this paper, we focus on exploiting vulnerable code running in processors of Harvard architecture. We use the ATmega128 processor (ATmega128) used in MICA2 and MICAz sensors as the platform. The memory of the process is depicted in Fig. 1. In such a processor, although attackers can use many well known buffer overflow techniques, exploitation in sensor nodes is constrained by the structure of sensor hardware and software. In the following, we discuss the main challenging factors that may fail a mal-packet in sensor networks.

3.1.1. *Non-executable injected code*

A worm packet normally carries executable code in order to replicate and propagate itself. A regular computer uses von Neumann architecture, where a program can access either code (in TEXT section) or data (in Data or BSS or HEAP or STACK section) as code and data are stored in the same memory. As a consequence, an attacker can inject a piece of malicious code into memory as a piece of data and then alter the control flow to execute the injected code (Giannetsos et al., 2009).

However, in Harvard architecture, code and data storages are logically and physically separated. The Program Counter (PC) register cannot point to any address in data memory. Hence, an instruction injected in data memory will not be executed. Consequently, it is hard for worms, which execute the injected code, to be effective in sensors. This paper will mainly look into techniques for making mal-packets that only carry mal-data and use existing code in program memory. This paper does not study the scenario where the code in a sensor can be modified by an attacking packet, as some sensors allow remote reprogramming on the fly. The research on exploiting remote reprogramming to execute injected code is discussed in Francillon and Castelluccia (2008).

Two issues need to be addressed when exploiting existing sensor code. First, *how can a mal-packet replicate itself?* We find that a sensor always needs a memory buffer to receive a packet. Hence, a mal-packet will be placed into the buffer when the sensor receives it. If exploitation happens before the buffer is released, the mal-packet stored in the buffer can try to invoke a transmission routine to send itself. Second, *how can a mal-packet propagate itself to other sensor nodes?* A worm in an infected computer usually first probes and finds other vulnerable computers and then sends a copy of itself to the vulnerable computers. In a sensor network, because most sensor nodes are using homogeneous software, a mal-packet does not need to probe whether its neighboring nodes are vulnerable. A mal-packet neither needs to scan other senor nodes. It can broadcast itself to nearby nodes in order to infect them. Then, the infected nearby nodes can further broadcast the mal-packet due to the same vulnerability. A challenging issue remains as *how a mal-packet can use a transmission function to broadcast itself if a sensor does not have any code for broadcasting*. A mal-packet has to provide all mal-data in its payload to accomplish this operation.

3.1.2. *Reset after exploitation*

Altering the control flow in a sensor can easily result in invoking the RESET routine and changing important registers. Consequently, the sensor may be restarted or get into an unknown status, and thus cannot propagate a mal-packet.

The problem in exploitation can be illustrated in Fig. 2, where an attacker exploits a vulnerability in function B that is called by

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

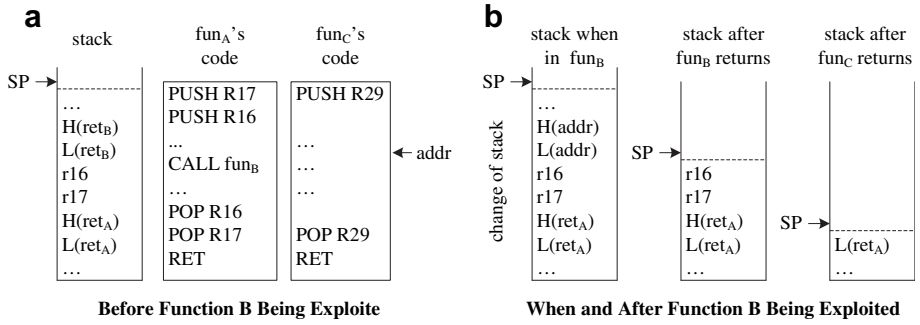

Fig. 2 – Reset. SP is the stack pointer. In (b), SP is overwritten to point to the mal-packet stored in heap.

function A. The attacker sends a mal-packet that overwrites the return address of function B to some address in function C so that the attacker can use the routine in function C to commit some malicious operations. Fig. 2 shows how the stack is changed and used before and after exploiting function B.

When function B is called but before being exploited, the stack has four variables (total 6 bytes) as shown in Fig. 2(a): the return address of function B ($H(ret_B)$ and $L(ret_B)$), two registers pushed by function A (R16 and R17), and the return address of function A ($H(ret_A)$ and $L(ret_A)$). When exploitation happens in function B, ret_B is overwritten to *addr* so that the control flow will be altered to function C when function B returns. *addr* is an address within function C. When the control flow is altered into function C, the stack that is for function A will be used by function C instead. Hence, when function C returns, *r*16 will be popped into *R*29, and *r*17 and $H(ret_A)$ will be used as the return address of function C.

Apparently, two problems are raised by this exploitation. First, function C will return to an unknown address composed by *r*17 and $H(ret_A)$. If the return address is invalid, the program counter (PC) will be reset to 0 and then the RESET routine will be invoked and the senor will be restarted. If the return address is valid, the sensor will go into an unknown status until PC is loaded with an invalid address. Then, the sensor will be restarted as well by the RESET routine. Second, registers are modified by the altered control flow. For example, *R*29, is an important register that is frequently used as a pointer to the top of stack in various functions. Changing *R*29, may result in the corruption of stack that in turn results in restarting the sensor as well.

The example only shows the case where the stack depth of function C is smaller than the stack depth of function A. It is also possible that the stack depth of function C is greater than the stack depth of function A. Then, when function C returns, the stack pointer actually moves to the stack of the function that calls function A. Thereby, function A is in fact jumped over and the sensor gets into an unexpected status.

3.1.3. *Other factors*

The capability of an attack using mal-packets in a sensor network is also affected by several other factors. First, packets in a sensor network may only have a small size of payload. For example, the default size of packet payload in a MICA2 sensor node is 28 bytes. Although applications can choose a different size of payload, the size cannot exceed 256 bytes, because only 8 bits are used in packet header to indicate the payload size. However, this paper will show that it is possible to make a self-propagating mal-packet within 28 bytes and the minimum size of mal-packet payload is only 17 bytes.

Second, sensor applications may happen to have some routines to stop attacks using mal-packets. For example, if a secure integrity check is performed before any vulnerability is exploited, a mal-packet will be discarded because an attacker usually does not have enough credentials to make mal-packets that can pass the integrity check. Another example is that a sensor node may first check if a received packet is a broadcast packet and then decide how to process a packet. A broadcast packet might be processed in a non-vulnerable routine and then a mal-packet will not succeed. However, existing worm attacks in regular computer networks show that (a) many vulnerabilities exist in routines of receiving packets, (b) many applications do not have any integrity check, or (c) vulnerability is quite often exploited before any integrity check. The same situation exists in sensor applications. A sensor node always needs to process a received packet in some routines where vulnerable code could be present. In addition, many sensor applications only process payload in packet. A packet is legitimate as long as it can be received by a sensor, regardless how the packet is transmitted. Broadcast packets may also be processed within routines shared with unicast packets and these routines may be vulnerable too.

Therefore, this paper will assume that there exists a vulnerable routine that can be exploited by a mal-packet. The paper emphasizes that a mal-packet can accomplish its attack via carrying mal-data and using existing code, if a vulnerability presents. The exploitation techniques will address the challenges in four aspects: *alteration of control flow*, *invocation of transmission function*, *restoration of control flow*, and *composition of mal-packet payload*.

3.2. *Alteration of control flow*

In this section, we demonstrate two types of exploitation techniques (in Fig. 3) that exploit return address to alter the control flow. One is to directly overwrite a return address as many typical stack overflow techniques, and the other one is rather to overwrite stack pointer by taking advantage of structure of sensor's data memory. We are aware that attackers may also figure out some other approaches to alter the control flow. For example, if a function pointer is stored in

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

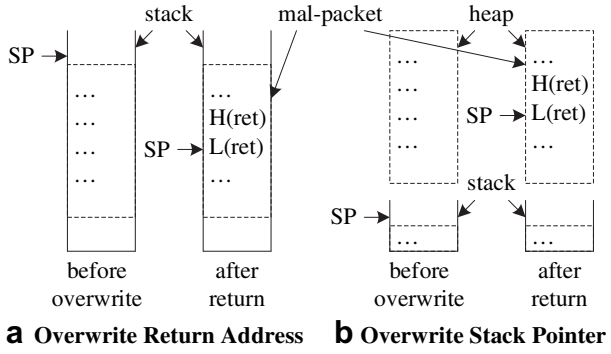

Fig. 3 – Change of return address and stack pointer. *SP1* is the stack pointer when the control flow is altered to program address *alt1* to get arguments. *SP2* is the stack pointer when the control flow is altered to program address *alt2* to invoke the transmission function.

data memory, overwriting this pointer via heap overflow or stack overflow can redirect the control flow when calling this function.

First, attackers can use many existing techniques to overwrite a return address in stack, such as stack overflow, string format, etc. Overwriting return addresses is still possible in many applications due to insufficient security awareness of programmers. Fig. 3(a) illustrates how a mal-packet overwrites a return address in the stack. Because the mal-packet does not change the stack pointer, the stack below the modified return address will be used by the function that the control flow is altered to. Hence, the mal-packet needs to overwrite data in the stack below the return address as well.

Second, due to the simple and special structure of sensor's data memory, attackers can overwrite some critical data to provide an arbitrary return address. For example, in the data memory of a ATmega128 processor (Fig. 1), the first 256 bytes in data memory are for the register file (the first 32 bytes), the I/O registers (the next 64 bytes) and the extended I/O registers (the next 160 bytes). As one of the I/O registers, the stack pointer takes two bytes at data address 0x5*D* and 0x5*E*. Application code can directly access these critical data addresses as a normal data access via op-code STS. An attacker can thus modify the stack pointer to point to the mal-packet that contains mal-data for exploitation. Fig. 3(b) illustrates how a stack pointer is changed by a mal-packet. The mal-packet is stored in the heap and exploits vulnerability, such as heap overflow, to overwrite the stack pointer to point to the mal-packet. Then, the vulnerable function being exploited will use the mal-packet as its stack. When the function returns, it actually returns to the address carried in the mal-packet. In this example, the stack is not overflowed, but the mal-packet provides a malicious return address.

The common features of these two exploitation techniques are (1) a mal-packet provides a malicious return address and (2) the mal-packet also provides the data below the return address for further exploitation after the control flow is altered. Note that the mal-packet also carries some data beyond the malicious return address that might be used for further exploitation, depending on the target application code.

3.3. *Invocation of transmission function*

In order to propagate itself, a mal-packet needs to invoke some functions to send itself out. Since a mal-packet in a sensor cannot carry and use its own code for transmission, the mal-packet needs to alter the control flow to an existing transmission function in the infected sensor. For example, the following function in TinyOS exists in many sensor applications that send packets.

AMSend

In order to invoke the function, a mal-packet needs to provide arguments. Normally, arguments of a function are pushed into the stack before the function is called. However, in the AVR processor, it is more efficient to use registers to pass arguments to a function. Hence, arguments are not pushed in stack, and a mal-packet cannot pass argument by simply overwriting stack of a function. We identified two techniques that can provide arguments to the *AMSend* function. Fig. 4 demonstrates an example of invoking the function in a Mica2 sensor. The *AMSend* function actually calls the **transmission function** *CC1000ActiveMessageP* (function *send* in Fig. 4) that needs four arguments. The arguments are passed via seven registers (R18–R24) to the transmission function (Table 1). Note that the mal-packet in Fig. 4 is not necessarily stored in stack as discussed in Section 2.

The first approach to pass arguments to the transmission function is using a routine that can pop values into the argument registers, as depicted in Fig. 4(a). A mal-packet needs to alter the control flow twice. First, the control flow is altered to a routine at program address *alt1* that pops registers R18–R24. This pop routine normally can be found at the end of some functions, such as interrupt routines for "Timer Counter Overflow" and "USART Transmission Complete". These routines can be found in most sensor applications. The mal-packet should provide mal-data for R18–R24 right below

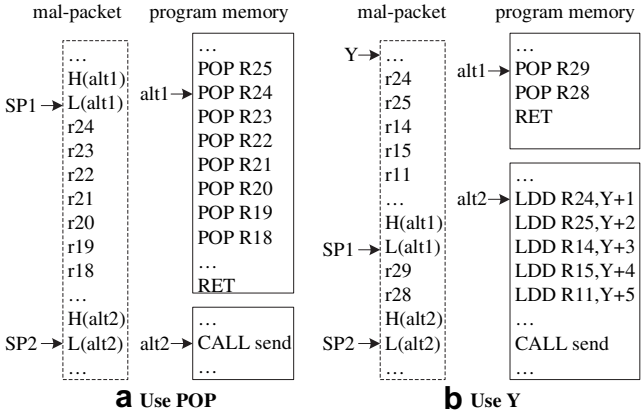

Fig. 4 – Invocation of function *AMSend*. *SP2* is the stack pointer when the control flow is altered to program address *alt2* to invoke the transmission function. *SP3* is the stack pointer when the control flow is altered to program address *alt3* to restore the Y register. *SP4* is the stack pointer when the control flow is altered back to function A at program address *alt4*. *SP4′* is the stack pointer after being restored. *SP5* is the stack pointer after function A returns.

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

Table 1 – Arguments of transmission function.

Argument	Value	Register
am_id_t id	Source address	R18, R19
am_addr_t addr	Destination address	R20, R21
*message_t * amsg*	Payload	R22, R23
uint8_t len	Size of payload	R24

where the return address *alt*1 is stored. The mal-packet should ensure the stack depth between *SP*1 and *SP*2 matches the number of "POP"s of the pop routine. Thereby, when the pop routine returns, the mal-packet alters control flow to program address *alt*2 where the transmission function is called.

The second approach to pass arguments to the transmission function is using the Y register (i.e. registers R28 and R29 in ATmega128) to provide arguments carried in the mal-packet. Fig. 4(b) shows one example in which a function, that calls the transmission function, loads five bytes from a data address pointed by the Y register to five registers (R24, R25, R14, R15, and R11). These registers are processed and then loaded into registers R18–R24 as arguments for calling the transmission function. Accordingly, the mal-packet provides arguments via two steps. First, the control flow is altered to a pop routine at program address *alt*1 that pops R29 and R28. Hence, the Y register is modified to the address where arguments are stored in the mal-packet. The pop routine can be found in several functions that exist in many sensor applications, e.g. the function to get payload of a packet (*Packet*) and the function to send a packet (*AMSend*). The pop routine can also be found in some application functions that modify the Y register. These functions save the Y register at their entries and restore it at their exits. After the Y register is modified, the control flow is altered to program address *alt*2 where the data pointed by the Y register is loaded into registers, and then the transmission function is called.

Note that whether or not a mal-packet can use the Y register to pass arguments depends on the data flow of application code. An attacker needs to reversely trace the data flow (Kruegel et al., 2005; Xu et al., 2004) to where the arguments are loaded by using the Y register in the function that calls the transmission function. The advantage of the second approach over the first approach is that (a) it can reduce the size of a mal-packet and (b) it allows the mal-packet to be not constrained by the depth of stack. In the first approach, the arguments of the transmission function can only be stored below where *alt*1 is stored in stack. The payload of a mal-packet before *alt*1 is not utilized and the stack must have an enough depth to hold the mal-packet. In the second approach, the arguments can be stored anywhere in the payload of a mal-packet in heap. It would be flexible for a mal-packet to organize its payload to minimize the payload size. If enough space presents in the mal-packet before *alt*1, the arguments can be stored before *alt*1. The heap can also hold a mal-packet with a large payload.

3.4. *Restoration of control flow*

As discussed in Section 3.1, altering the control flow may result in resetting a sensor and stopping transmission of a mal-packet. Hence, after the transmission function is invoked, a mal-packet needs to further alter control flow until the control flow is restored as if the code was executed normally. Because a mal-packet may present in stack (for stack overflow) or in heap (for heap overflow), we illustrate two techniques of restoring the control flow respectively. Fig. 5 shows two examples in which function B is called by function A and function A is called by function F. Function B is vulnerable and being exploited so that the control flow is altered to program address *alt*2 in function C that calls the transmission function *send* to broadcast a mal-packet. When the transmission function returns, the mal-packet needs to alter the control flow back to function A so that function A can return to function F as normal.

Fig. 5(a) shows the restoration technique when a mal-packet is stored in the stack and overflows the stack. Two critical issues to be addressed are (1) the Y register should be restored and (2) function A should return to where it is supposed to return. Usually, function F has a few local variables stored in stack, and the Y register, which points to the top of stack, is used as a reference pointer to access local variables. When function A is called by function F, function A first pushes the Y register used by function F into stack so that the Y register can be restored when function A returns. Because stack is overflowed, the Y register of function F may be corrupted. Hence, when function C returns, the control flow is altered to a pop routine at program address *alt*3 that pops the correct Y register carried in the mal-packet. It is easy for an attacker to figure out the correct Y register, because it is always the top of stack for function F. After the Y register is restored, the control flow will be altered to somewhere at program address *alt*4 before "RET" in function A. The selection of *alt*4 needs to ensure the number of "POP"s between *alt*4 and the program address of "RET" of function A matches the number of bytes between *SP*4 and *SP*5 in stack so that function A will return to where it is supposed to return. Hence, *alt*4 points to a program address between "POP R28" and "RET" in function A. Accordingly, when function A returns, the control flow is restored.

Fig. 5(b) shows the other restoration technique when a mal-packet is stored in heap and overflows heap. As discussed in Section 2, the mal-packet does not change stack, but overwrites the stack pointer to point to somewhere in the packet. Hence, the mal-packet needs to restore the stack pointer to point back to stack as well as alter the control flow back to function A. As discussed above, because function A has its own local variables, the Y register is used by function A as a reference pointer for accessing its local variables. The Y register is also used to allocate stack space for local variables (taking 18 bytes in this example) and set the stack pointer accordingly. When function A returns, the Y register is used again to release the stack space for local variables. Hence, function A has a few lines of code at program address *alt*4 to use the Y register to restore the stack pointer. The mal-packet can thus first alter the control flow to *alt*3 to load the Y register, and then alter the control flow to *alt*4 to restore the stack pointer via the Y register. Note that the value of the stack pointer is restored to *SP*4′ that points to the data address in the stack where the Y register of function F is saved. When being popped at program address *alt*4′, the Y register of function F is

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

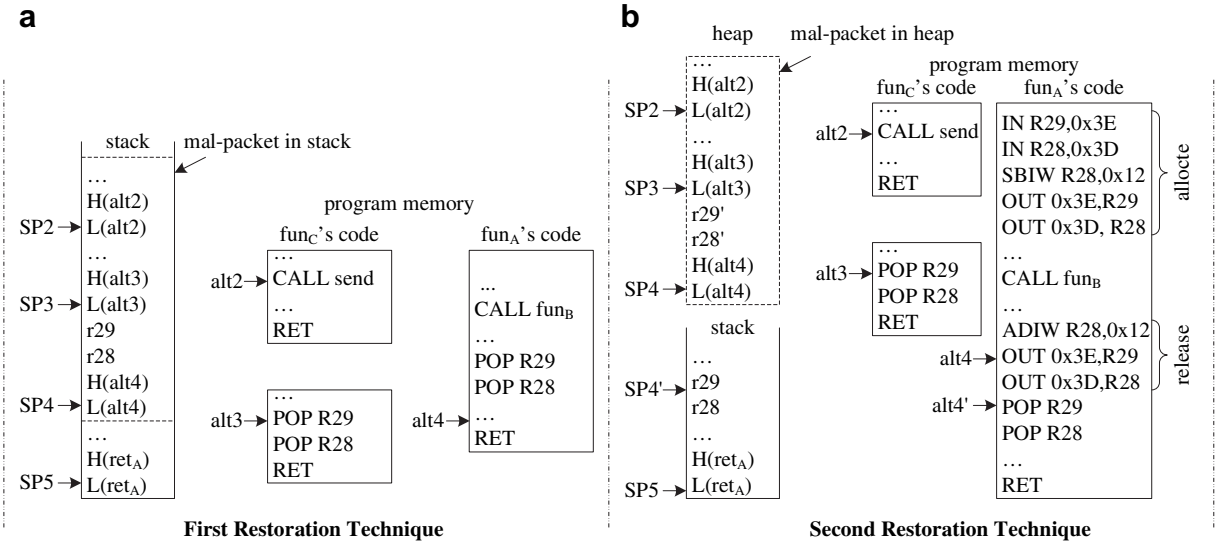

Fig. 5 – Restoration of control flow.

restored as well. Because stack is not changed by the mal-packet, the control flow is restored.

In both restorations, the control flow is not restored to where function B should return. A part of function A will not be executed either. Furthermore, it is not guaranteed that all registers (except the Y register) are restored correctly. Hence, restoration may cause function A or function F to return wrong values or present wrong conditions. However, the senor will execute code with the unexpected condition until the current task is completed. The task may not complete its mission due to the erroneous condition. In this restoring process, the sensor is not reset and will continue to run.

3.5. *Composition of mal-packet payload*

A mal-packet should carry all the data for exploitation, because the packet itself cannot generate new data. Hence, mal-packet payload should be composed according to the techniques presented in the previous Sections 3.2–3.4. Fig. 6 illustrates an example of mal-packet payload and the required data in a Mica2 sensor. Note that the non-specified "…" data in the payload is mainly used to match the stack depth when the control flow is being altered.

The payload consists of three parts. Part 1 provides data for buffer overflow and program address *alt*1 to which the control flow is altered when the exploited vulnerable function returns. *alt*1 points to the routine that pops arguments for the transmission function. Part 2 provides arguments of the transmission function. As discussed in Section 3.1, a mal-packet propagates itself via broadcasting. Hence, in part 2, the destination address carried in *R*20 and *R*21 is set to the broadcast address 0x*FFFF*. The mal-packet can spoof any source address in *R*18 and *R*19. The payload pointer in *R*22 and *R*23 is set to the starting address of the mal-packet payload itself. After the mal-packet payload is composed, the length of the payload in *R*24 is filled in. At the end of part 2, program address *alt*2 is set to the address where the transmission function is called. Finally, part 3 provides data for restoring the control flow, the Y register and stack pointer.

Accordingly, the minimum size of mal-packet payload is 17 bytes, which is smaller than the default payload size (28 bytes) in a Mica2 sensor. However, the minimum size can be achieved only if any routine used by a mal-packet does not have extra local variables in stack. In order to match the code utilized by a mal-packet, the mal-packet payload usually needs to carry more bytes. Nevertheless, because sensor applications are optimized for size (i.e. '-Os' is turned on for compilation) and a mal-packet only needs to use three existing routines, the size of mal-packet payload is only a few more bytes than the minimum size.

4. Defense schemes

As discussed in Section 3, StackGuard and code space randomization are two defense schemes suitable in sensor systems. Using the same principles, we develop two schemes to protect sensor applications: S2Guard and S2Shuffle. The two schemes add protection code directly to the machine code of sensor applications.

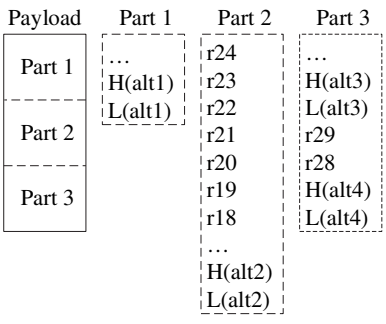

Fig. 6 – Mal-packet payload.

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

4.1. S2Guard: sensor stackGuard

StackGuard and its variations use a canary to prevent stack smashing. When a stack is created for a function call, a canary is pushed in the stack between the function's local variables and the function's return address. Thereby, in order to overwrite the function's return address, attackers have to overwrite the canary. When the function returns, the canary will be checked. If the canary is changed, the exploitation will be captured before the function returns to a malicious address.

We apply the same idea to develop S2Guard (Sensor StackGuard) for protecting the stack in sensors. S2Guard is different from StackGuard in three aspects. First, the canary in S2Guard only uses character 0x0. Because a string is terminated with 0x0 in sensor applications, the string copy function stops copying a string when reaching 0x0 in the source string. Another reason to use only 0x0 as the canary instead of a randomly generated canary is to ensure that the canary checking is lightweight in resource-limited sensors. The selection of canary also avoids using any data in untrustworthy data memory in sensors.

Second, unlike StackGuard, S2Guard protects a function's stack according to the data flow in a function. S2Guard uses a register to push 0x0 into stack. However, registers are also used in a function to pass arguments, store status and return results. When no register is guaranteed to have 0x0, S2Guard has to "borrow" a register from the function to be protected. In order to avoid changing data being used in the function, we develop two different protection techniques for two different cases (Fig. 7). In each case, S2Guard inserts the canary setting code at the beginning of the function and the canary checking code at the end of the function.

Case 1 There is one of the register-writing instructions that write a value into a register before any branch. In MICAx sensors, such register-writing instructions include *in*, *elpm*, *lpm*, *mov*, *ld*, *ser* and *clr*. In this case, S2Guard inserts code right before the instruction. Since the information in the destination register of any of the above instructions will be overwritten anyway, setting the value of said register right before it is written to results in no data loss. Note that, in Case 1, S2Guard may not insert code right at the beginning of the function. Instead, S2Guard locates an appropriate instruction first and then inserts the code.

S2Guard keeps track of how many pushes there are before the canary setting code. Then, S2Guard counts backwards from the function return and finds the register referenced in the most recent pop. S2Guard pops the canary into the register first for checking. If the canary is not changed, S2Guard goes into the normal routine to pop the register and the remaining registers. Since that register is about to be overwritten after canary checking, writing the register during the canary checking does not affect the function. If the canary is changed, S2Guard resets the sensor.

Case 2 There is no register-writing instruction mentioned in Case 1 before any branch. This case indicates all registers may have some data for the function to read. Hence, directly writing 0x0 into a register may change important data to be used in the function. Hence, S2Guard inserts the guarding code right at the beginning of the function, before any other instruction. S2Guard first swaps the data in a register out to a temporary space in data memory. Then, S2Guard uses the register to push the canary into stack. Finally, S2Guard restores the register with the data in the temporary memory space.

S2Guard puts the canary checking code right before the function returns in Case 2. S2Guard first stores the data of a register into a temporary memory space. Then, S2Guard pops the canary into the register to check whether the canary is changed. If no, S2Guard restores the register and the function returns. Otherwise, S2Guard resets the senor.

Apparently, the stack being protected by S2Guard is different from that in StackGuard. S2Guard does not necessarily push the canary right on top of return address. The canary is normally pushed on top of saved registers. Hence, S2Guard ensures that no register will be overwritten in either case.

4.2. S2Shuffle: sensor code space shuffle

Code space randomization is the technique to change the layout of code memory such that attackers cannot redirect the

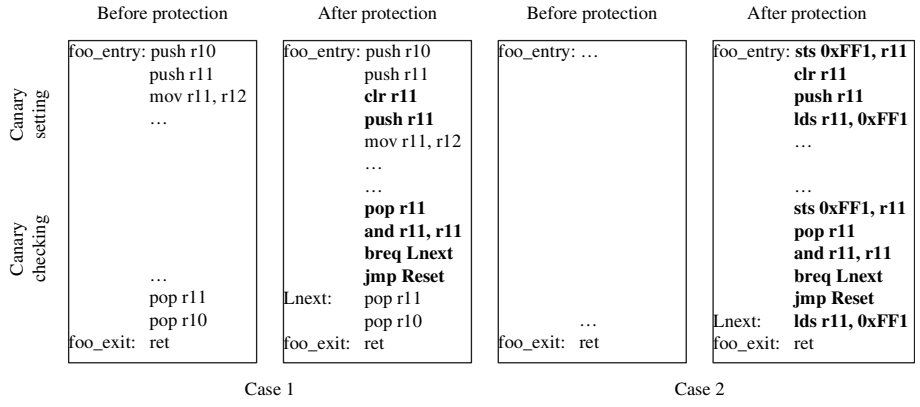

Fig. 7 – Example of cases 1 and 2 in S2Guard.

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

control flow to library functions stored in code memory. We apply the same principle to develop S2Shuffle that prevents attackers from exploiting existing application code. Unlike address space randomization (Bhatkar et al., 2003; Bhatkar et al., 2005), S2Shuffle does not simply randomize library function entry points, but instead shuffles the code blocks of the whole sensor application images (van de Ven, 2004). S2Shuffle consists of two components.

- *Control flow analysis component*. It identifies repositionable code blocks. The code blocks have a much finer (even arbitrary) granularity than function blocks. The code blocks cover all code of interrupt routines, OS routines and application routines. The code blocks will be placed in a randomized position in the final protected code.
- *Randomization component*. It randomly shuffles the positions of unprotected code blocks. Another function of this component is to diversify the code images for all sensors. Thereby, sensors will not use homogeneous code and compromising one sensor will not endanger other sensors.

4.2.1. *Control flow analysis*

S2Shuffle treats sensor application code as a sequence of instructions. It first parses the sequence without examining the control flow in the code. But, it partitions the code into repositionable control blocks (RCBs) (Definition 1). Thereby, the control flow can be modeled as transitions among RCBs later.

Definition 1. (*Repositionable Control Block (RCB). A contiguous sequence of instructions such that the last instruction in the RCB makes a non-conditional and non-returnable transition and the first instruction of the RCB is the instruction next to its previous RCB.*)

The paper classifies transitions made by various branch, jump and call instructions into three categories. Note that conditional transition has two subtypes.

- *Conditional transition*. A transition that will happen according to a condition.
- *Non-conditional and returnable transition*. A transition that will always happen but the control flow will return to the instruction next to where the transition originates.
- *Non-conditional and non-returnable transition*. A transition not belonging to the first two categories.

As shown in Fig. 8(a), branch instructions make the conditional transitions, call instructions make the non-conditional and returnable transitions, and jump and return instructions make the non-conditional and non-returnable transitions. When a non-conditional and non-returnable transition takes place, the control flow will be transited to an address in another RCB and will not return to the originating address. Hence, instructions belonging to this category of transitions are used as boundaries for partitioning the application code into RCBs.

S2Shuffle also allows arbitrarily partitioning a contiguous sequence of instructions into several RCBs. An example is shown in Fig. 8(b) where S2Shuffle inserts RJMP at an arbitrary position to partition the code. The position should be selected based on application's security requirements. This paper does not use arbitrary partitioning, but rather finds minimum RCBs (Definition 2).

Definition 2. (*Minimum RCB (minRCB). A smallest RCB that cannot be further partitioned without using the arbitrary partition technique.*) After obtaining all minRCBs, S2Shuffle identifies all transitions in them. If the control flow stays in the same minRCB after a transition, the transition is *internal*; otherwise, the transition is *external*. An example is shown in Fig. 9 where two functions A and B have two minRCBs each. All minRCBs have external transitions and only minRCB_2 and minRCB_4 have internal transitions. Note that minRCBs of the same function routine do not necessarily reside in a contiguous block in program memory. They can be repositioned as long as their transitions will not be changed. Thereby, the right part of Fig. 9 shows a possible scenario where the four minRCBs are repositioned. In this paper, S2Shuffle repositions all minRCBs to provide the finest granularity and the maximum randomness within the whole scope of applications. When repositioning

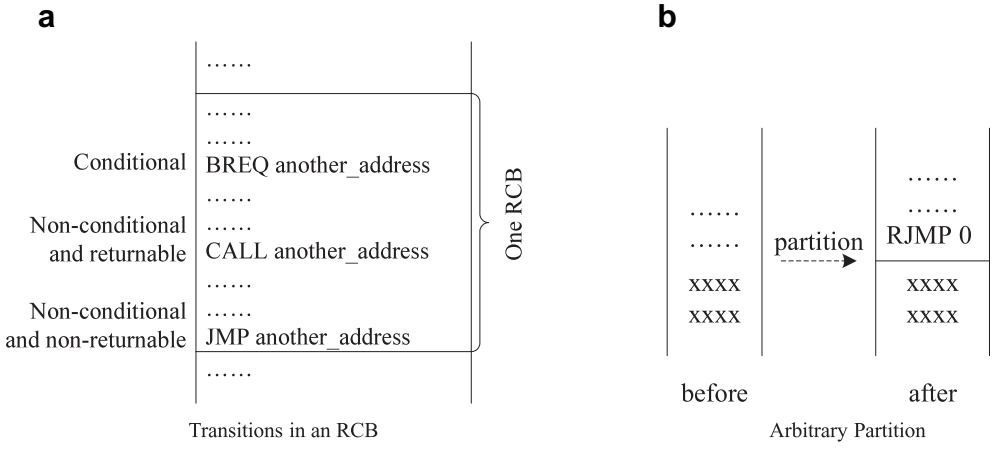

Fig. 8 – Illustration of RCBs.

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

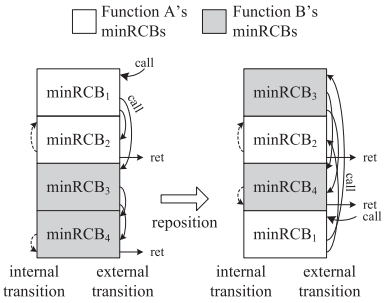

Fig. 9 – Reposition of RCBs. Dashed lines indicate how the control flow is altered by a mal-packet. The control flow under attack starts from function *runTask*, goes through *process*, *forward* and *send*, and finally returns back to *runTask*.

minRCBs, S2Shuffle symbolizes destination addresses of transitions.

4.2.2. *Randomization*

S2Shuffle randomizes the application code by randomly repositioning minRCBs. S2Shuffle uses a crypto-random function to generate random numbers to determine how to reposition minRCBs. In order to ensure no two sensor nodes have the same code, S2Shuffle generates a piece of randomized machine code for each individual sensor. Each sensor will be loaded with a unique image of randomized code.

After minRCBs are identified, S2Shuffle randomly shuffles the minRCBs for each sensor. A few issues should be addressed. First, an interrupt vector is stored at a few fixed location (normally at the beginning of program memory). The vector consists of entries for interrupt routines. Each entry contains a JMP instruction to jump to the corresponding interrupt routine. Hence, each entry in the vector is a minRCB. S2Shuffle will not reposition these special minRCBs since their positions are fixed for the hardware. Rather, S2Shuffle shuffles minRCBs of interrupt routines pointed by the entry prints. Second, relative branch and jump instructions are used in application code. When repositioning minRCBs, external transitions via relative branch and jump instructions may be beyond the range of the instructions. S2Shuffle transforms relative branch and jump instruction into absolute jump instructions if necessary.

5. Experiments and analysis

In this section, we first present the implementation of a sensor mal-packet that can propagate itself utilizing only the code in a vulnerable sensor application. Then, we discuss how the mal-packet propagates itself. Finally, we show the overhead and effectiveness of the two defense schemes S2Guard and S2Shuffle.

5.1. *Prototype of sensor mal-packets*

We studied a sensor application that is supposed to only forward a received packet to its next hop. However, the sensor application has a vulnerable routine that copies the packet payload into a buffer without checking boundary of the buffer. Hence, a typical stack overflow happens. The mal-packet can successfully exploit the vulnerability and then broadcast itself to all neighboring nodes by invoking a non-broadcast transmission function in the infected sensor. The functionality of the infected sensor is not disrupted by the mal-packet. For demonstration, the mal-packet carries a block of legitimate data to illustrate that any sensor receiving the broadcast mal-packet can execute its regular functions with the input of the legitimate data.

Table 2 – Functions in demonstrations.

Name	Description
runTask	Run a task
process	Process a received packet
forward	Forward a received packet
send	Send a packet out

Four functions (listed in Table 2) are involved in the exploitation of the mal-packet. Function *runTask* invokes a task that calls function *process* to process a received packet. Function *process* is vulnerable and thus exploited by the mal-packet to alter the control flow into function *forward* for propagating the mal-packet. Function *forward* calls the non-broadcast transmission function *send* that is designed to only forward a packet to the next hop of the current node. The control flow under attack is depicted in Fig. 10.

The size of the mal-packet payload is 27 bytes. 4 bytes are legitimate data for presenting a good packet so that function *process* will not discard the packet. 17 bytes provide mal-data for altering and restoring the control flow. 6 bytes are used for matching the stack when altering the control flow. The mal-packet invokes function *send* to broadcast itself and then ensures function *runTask* can return as usual.

Note that this example intends to provide an exploitable vulnerability, because the purpose of the paper is to illustrate that a sensor mal-packet can be implemented if an exploitable vulnerability is found. The paper does not study how to find a vulnerability in sensor applications.

5.2. *Simulation of mal-packet propagation in sensor networks*

Because sensor mal-packets studied in the paper use broadcasting to propagate themselves, the epidemic models (Staniford et al., 2002; Zou et al., 2003) for worm propagation

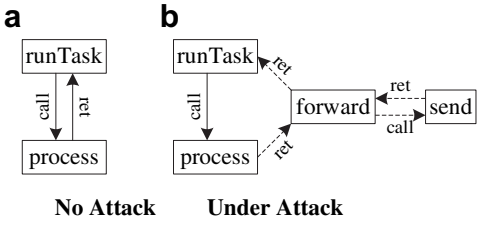

Fig. 10 – Control flow in demo.

Table 3 – Parameters of simulation.

Para	Description	Values
T	Type of distribution	even, random
N	Number of sensors	900, 10 000
m	Number of mal-packet sources	1, 2, 4
p	Probability of broadcast failure	0.00–0.63

may not capture the characteristics of mal-packet propagation in sensor networks.

First, although a mal-packet can invoke a transmission function to broadcast itself, it is not guaranteed the broadcast will succeed. When the transmission function is invoked, the radio component in the infected sensor may not be ready and thus the mal-packet will be discarded. Even if the mal-packet is being broadcast, the radio transmission may be interfered with other signals. If collision exists, the broadcast will not succeed. Because the mal-packet does not carry code, it cannot be rebroadcast if failure happens. Hence, the probability that a mal-packet can reach a distant node is $(1-p)^h$ where p is the probability of broadcast failure and h is the distance (measured in hop) from the mal-packet source to the distant node.

Second, a mal-packet cannot repeat its operations inside an infected sensor. After it broadcasts itself, it cannot rebroadcast regardless of whether or not broadcast is successful. Hence, an infected sensor can only broadcast a mal-packet once. However, a previously infected sensor may broadcast the mal-packet again, because it may be re-infected when its neighboring nodes broadcast the mal-packet. If the probability of broadcast failure is high, the propagation of a mal-packet could be stopped, because fewer previously infected sensors can be re-infected. Hence, mal-packet propagation in a sensor network is possibly not self-sustainable without persistent mal-packet sources.

Accordingly, propagation of mal-packets in a sensor network is determined by the probability of broadcast failure and the number of mal-packet sources. We use simulation to study the mal-packet propagation. The simulation is conducted with the variable parameters listed in Table 3. The medium-size network has 900 sensor nodes and the large-size network has 10 000 sensor nodes. Sensors are distributed evenly or randomly in a square area. For even distribution, mal-packet sources are placed in the corners. For random distribution, mal-packet sources are placed randomly in the network. Mal-packet sources broadcast mal-packets at the rate of one packet per second. In each experiment, a probability of broadcast failure p is set so that random numbers are used to determine if an infected node can successfully broadcast a mal-packet. Each data point is averaged over 30 random experiments with the same set of parameters.

Fig. 11 illustrates the propagation patterns of mal-packets, which are obviously different from worm propagation patterns in the Internet. The probability of broadcast failure p clearly divides the propagation patterns into two categories. In simulation, p has a threshold around 0.60. When p is smaller than the threshold, a sensor network can be quickly taken over by mal-packets. On the contrary, when p is greater than the threshold, mal-packets hardly propagate in a sensor network.

When p is small, mal-packet propagation in a sensor network does not have a slow start. Nor does it slow down when most of the nodes in a sensor network have been infected. In the Internet, worm propagation is slow at the start and the end, because it is harder to probe an uninfected vulnerable host during these two phases. However, in a sensor network, mal-packets do not probe. They reach all vulnerable sensors via broadcasting. If $p = 0$, the propagation rate will be the same as the broadcasting rate. An infected sensor can always broadcast a mal-packet to its neighboring vulnerable

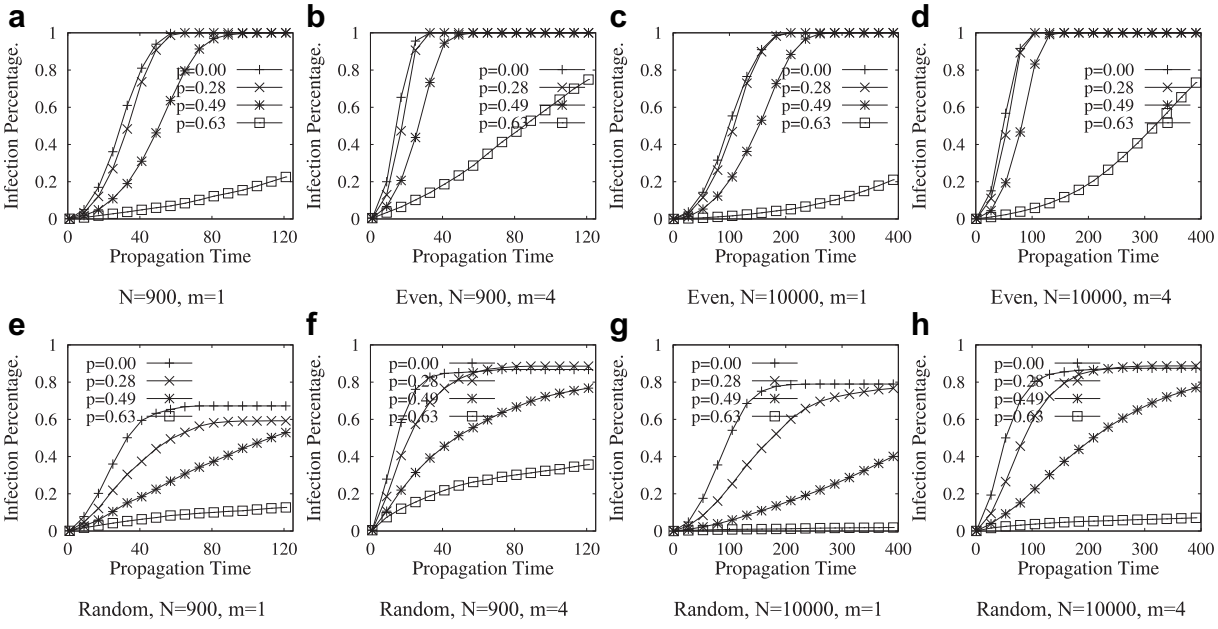

Figure 11: Propagation Patterns of Mal-packets

Fig. 11 – Propagation patterns of mal-packets.

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

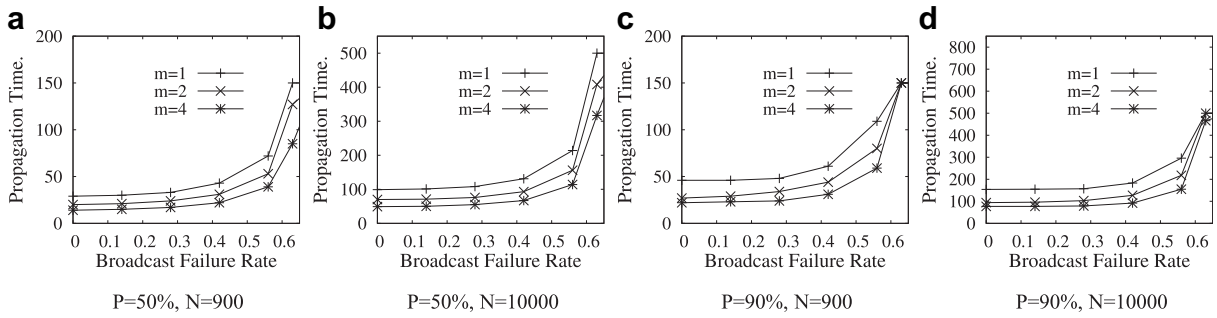

Fig. 12 – Infection of mal-packets.

sensors. Hence, the propagation curves for $p = 0.00,0.28,0.46$ are quite close and almost grow linearly over time.

When p is small, the time for an attacker to take over a sensor network is mainly determined by the distance from the mal-packet source to the furthest node in the network. For example, the distance from the top left corner to the bottom right corner doubles the distance from the center to any corner. Thereby, when $N = 10000$ and $p = 0.28$, the time to compromise all sensors for $m = 1$ is about 200 s that doubles the times for $m = 2$ and $m = 4$, which are 110 s and 100 s respectively.

Fig. 11 also shows that a portion of sensors cannot be infected if sensors are randomly distributed in a network. Because random distribution does not guarantee the connectivity among sensors, some areas in a network may be disconnected. Mal-packets cannot reach the sensors in the disconnected areas. These sensors thus survive the attack.

Fig. 12 illustrates the time for mal-packets to reach 50% or 90% of sensors in a sensor network. When p is small, the time curves are flat and the time is almost not affected by p. When p is large, mal-packets can hardly reach all sensors. Such a situation happens when the regular traffic in network is intensive. Because broadcast does not have congest control, any collision or interference from regular traffic will result in broadcast failure. Because mal-packets cannot successfully be broadcast, the probability that mal-packets reach distant nodes decreases exponentially to the distance. Thereby, the time of mal-packets reaching distant nodes increases exponentially.

5.3. *Evaluation of S2Guard and S2Shuffle*

S2Guard and S2Shuffle are implemented in MICA2 motes running TinyOS v2. This section evaluates the overhead of the two defense schemes in program memory and how much they affect the execution of normal routines.

5.3.1. *Code size*

Table 4 shows the comparison of the code sizes of sensor applications with and without protection. S2Guard adds 12.6% extra code to the original application code, while S2Shuffle adds 1.0% extra code. The code overhead of S2Guard is determined by the number of functions, because it adds canary set and check code at the beginning and the end of each function. The code overhead of S2Shuffle is rather determined by the distance change of relative and absolute transitions.

5.3.2. *Computation time*

The computation time is measured as CPU cycles. Larger computation time indicates more power consumption in sensors. The CPU cycles are measured in AVR Studio (AVR Studio 4), which is a simulator provided by the processor's manufacturer. Both the unprotected code and the protected code of the sensor applications are tested in AVR Studio.

We measure the CPU cycles for two routines. One is the RESET routine which initializes a sensor, and the other is an arbitrary task running in the sensor applications. Since sensor processors are energy efficient in the sleep status, the CPU cycles of a running task (i.e., duty cycles) reflect the actual sensor performance. Table 5 summarizes the comparison of the CPU cycles of the RESET routine and the comparison of the CPU cycles of a task with and without protection.

For the RESET routine, S2Guard takes 0.3% more time than the original RESET, and S2Shuffle takes 0.02% more time. For the tasks, S2Guard takes 6.3% more time than the original tasks, and S2Shuffle takes 0.6% more time. Because the tasks themselves only have a few cycles, the percentage increase of the task's computational overhead is larger than that of the RESET routine.

5.3.3. *Discussion on security and limitations*

The security features of S2Guard and S2Shuffle have the same security features and limitations as their original techniques. As illustrated in Table 6, S2Guard only prevents a mal-packets from exploiting a return address. Hence, it can prevent stack-smashing attacks as StackGuard does. Since several attack

Table 4 – Code sizes in word.

Applications	Original	S2Guard	S2Shuffle
Anti-theft	12 265	14 179	12 433
BaseStation	6456	7180	6522
Oscilloscope	5245	5674	5302
M-oscilloscope	13 567	15 463	13 786
RCToLeds	4948	5372	4999
RSToLeds	5065	5467	5116
Blink	1185	1360	1191
PowerUp	418	504	418

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002

Table 5 – Comparison of CPU cycles.

Applications	CPU cycles of the RESET routine			CPU cycles of a task		
	Original	S2Guard	S2Shuffle	Original	S2Guard	S2Shuffle
Anti-theft	211 826	212 671	211 831	153	168	153
BaseStation	211 246	212 032	211 251	145	152	147
Oscilloscope	202 966	203 727	202 971	129	136	129
M-oscilloscope	223 475	224 446	223 480	149	164	149
RCToLeds	202 796	203562	202 796	132	139	135
RSToLeds	202 770	203 531	202 775	135	142	135
Blink	154 637	154 774	154 642	529	580	534
PowerUp	153 865	153 908	153 865	39	39	39

Table 6 – Security comparison of S2Guard and S2Shuffle.

Protection	Stack overflow	Heap overflow	Format string	Return-to-libc	Double free	Integer overflow
S2Guard	Y	N	N	N	N	N
S2Shuffle	Y	Y	Y	Y	Y	N

steps need to exploit return addresses in this paper, S2Guard can stop the attack proposed in this paper. However, if an attacking packet exploits only other types of vulnerabilities, such as heap overflow or format string vulnerability, S2Guard cannot stop the attack.

S2Shuffle can prevent an attacking packet from getting to the correct destination addresses. However, it is showed (Shacham et al., 2004) that even a 32-bit address space is insufficient to keep attackers from guessing the correct addresses. Even though a sensor has only a 16-bit address space, a successful attack requires a sequence of exploitation steps, and each step must work on a correct address. The paper shows that a minimum of three steps are needed, which ask for three correct addresses of 48 bits in total. Furthermore, since each sensor randomizes its own address space, a mal-packet that can exploit one sensor may not be able to exploit another sensor. Hence, randomization effectively increases the difficulty of exploitation and slows down the propagation of a mal-packet.

6. Conclusion

In this paper, we have presented several exploitation techniques by which an attacker can make self-propagating mal-packets to infect sensors in a network. In contrast to worm packets, mal-packets studied in this paper only carry specially crafted data, but can exploit memory-related vulnerabilities and utilize the existing application code in sensors to propagate themselves without disrupting sensor's functionality. The paper showed that such a mal-packet can have as few as 17 bytes. A prototype of a 27-byte mal-packet has been implemented and tested in Mica2 sensors. Simulation shows that the propagation pattern of such a mal-packet in a sensor network is very different from worm propagation. Mal-packets can either quickly take over the whole network or hardly propagate under different traffic situations. Furthermore, we developed two defense approaches (S2Guard and S2Shuffle) to counteract this attack. The implementation and experiments showed that the two defense techniques are lightweight in sensor applications. As a future work, we will explore the integration of diversity and the two schemes to further improve the security of sensors.

Acknowledgment

This work was partially supported by the Research Enhancement Program and the Texas State One-Time Research Support Program at Texas State University-San Marcos.

REFERENCES

Akritidis P, Costa M, Castro M, Hand S. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proc. of Usenix security symposium; 2009.

Anonymous. Once upon a free(). Phrack Magazine, http://www.phrack.com/issues.html?issue=57&id=9#article; 2001.

ATmega128, http://atmel.com/dyn/products/product-card.asp?part-id=2018.

AVR Studio 4, http://www.atmel.com/avrstudio. URL, http://www.atmel.com/avrstudio.

Bhatkar S, DuVarney D, Sekar R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proc. of USENIX security symposium; 2003.

Bhatkar S, Sekar R, DuVarney DC. Efficient techniques for comprehensive protection from memory error exploits. In: Proc. of USENIX security symposium; 2005. p. 17–17.

Buchanan E, Roemer R, Shacham H, Savage S. When good instructions go bad: generalizing return-oriented programming to RISC. In: Proc. of ACM CCS; 2008. p. 27–38.

Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: automatically generating inputs of death, In: Proc. of ACM CCS; 2006. p. 322–335.

Chen S, Xu J, Sezer E, Gauriar P, Iyer R. Non-control-data attacks are realistic threats. In: Proc. of USENIX Security Symposium; 2005.

Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns, In: Proc. of USENIX security symposium; 2003.
Costa M, Crowcroft J, Castro M, Rowstron A, Zhou L, Zhang L, Barham P. Vigilante: end-to-end containment of internet worms. In: Proc. of ACM SOSP; 2005.
Cowan C, Pu C, Maier D, Hinton H, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q. Automatic detection and prevention of buffer-overflow attacks. In: Proc. of USENIX security symposium; 1998.
Cowan C, Beattie S, Johansen J, Wagle P. PointGuard: protecting pointers from buffer overflow vulnerabilities. In: Proc. of USENIX security symposium; 2003.
Etoh H, Yoda K. ProPolice: improved stack-smashing attack detection. IPSJ SIGNotes Computer SECurity, http://www.trl.ibm.com/projects/security/ssp; 2001.
Fleizach C, Liljenstam M, Johansson P, Voelker GM, Mehes, A. Can you infect me now?: malware propagation in mobile phone networks. In: Proc. of ACM workshop on recurring Malcode; 2007. p. 61–68.
Francillon A, Castelluccia C. Code injection attacks on harvard-architecture devices. In: Proc. of ACM CCS; 2008. p. 15–26.
Gay D, Levis P, Behren Rv, Welsh M, Brewer E, Culler D. The nesC language: a holistic approach to networked embedded systems. In: Proc. of ACM SIGPLAN; 2003. p. 1–11.
Giannetsos T, Dimitriou T, Prasad NR. Self-propagating worms in wireless sensor networks. In: Proc. of co-next student workshop, 2009. p. 31–32.
Godefroid P, Levin MY, Molnar DA. Automated Whitebox Fuzz testing. In: Proc. of network distributed security symposium; 2008.
Govindavajhala S, Appel AW. Using memory errors to attack a virtual machine. in: Proc. of IEEE symposium on security and privacy; 2003.
Jack B. Exploiting embedded systems. Black Hat Europe, http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Jack.pdf; 2006.
Kaempf M. Vudo malloc tricks. Phrack Magazine, http://www.phrack.com/issues.html?issue=57&id=8#article; 2001.
Kiriansky V, Bruening D, Amarasinghe SP. Secure execution via program Shepherding. In: Proc. of USENIX security symposium, 2002. p. 191–206.
Kruegel C, Kirda E, Mutz D, Robertson W, Vigna G. Automating mimicry attacks using static binary analysis. In: Proc. of USENIX security symposium, 2005.
Kumar R, Kohler E, Srivastava M. Harbor: software-based memory protection for sensor nodes. In: Proc. of ACM IPSN; 2007. p. 340–349.
Linn C, Debray S. Obfuscation of executable code to improve resistance to static disassembly. In: Proc. of ACM CCS; 2003.
Mantis, http://mantis.cs.colorado.edu/.
Nergal. The advanced return-into-lib(c) exploits (PaX case study). Phrack Magazine, http://www.phrack.org/issues.html?issue=58&id=4#article; 2001.
Newsham T. Format string attacks, http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf; 2001.
One A. Smashing the stack for fun and profit. Phrack Magazine, http://www.phrack.com/issues.html?issue=49&id=14#article; 1996.
Pax. PaX address space layout randomization (ASLR), http://pax.grsecurity.net/docs/aslr.txt.
Regehr J, Cooprider N, Archer W, Eide E. Memory safety and untrusted extensions for TinyOS, Tech. rep; 2006.
Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N Boneh, D. On the effectiveness of address space randomization. In: Proc. of ACM CCS; 2004.
Shacham H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proc. of ACM CCS; 2007. p. 552–561.
Smirnov T. Chiueh DIRA: automatic detection, identification and repair of control-data attacks. In: Proc. of network and distributed system security symposium; 2005.
Staniford S, Paxson V, Weaver N. How to own the Internet in your spare time. In: Proc. of USENIX security symposium; 2002. p. 149–167.
Starzetz P. CRC32 SSHD vulnerability analysis, http://packetstormsecurity.org/0102-exploits/ssh1.crc32.txt; 2001.
Tan L, Zhang X, Ma X, Xiong W, Zhou Y. AutoISES: automatically inferring security specifications and detecting violations. In: Proc. of Usenix security symposium; 2008. p. 379–394.
TinyOS, http://www.tinyos.net.
van de Ven A. New security enhancements in red hat enterprise Linux v.3, update 3, http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf; 2004.
Vendicator. StackShield, http://www.angelfire.com/sk/stackshield.
Wang X, Pan C-C, Liu P, Zhu S. SigFree: a signature-free buffer overflow attack blocker. In: Proc. of USENIX security symposium; 2006.
Xu H, Du W, Chapin S. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In: Proc. of symposium on recent advances in intrusion detection; 2004.
Yan G, Eidenbenz S. Modeling propagation dynamics of bluetooth worms (extended version). IEEE Transactions on Mobile Computing 2009;8:353–68.
Yang Y, Zhu S, Cao G. Improving sensor network immunity under worm attacks: a software diversity approach. In: Proc. of ACM MobiHoc; 2008. p. 149–158.
Zou C, Gong W, Towsley D. Worm propagation modeling and analysis under dynamic quarantine defense. In: Proc. of the 2003 ACM workshop on rapid Malcode; 2003. p. 51–60.

Qijun Gu is an assistant professor of Computer Science at Texas State University-San Marcos. He received the Ph.D. degree in Information Sciences and Technology from Pennsylvania State University in 2005, the Master degree and the Bachelor degree from Peking University, China, in 2001 and 1998. His research interests cover various topics on network, security and telecommunication. He has been working on projects including denial of service in wireless networks, key management in broadcast services, worm propagation and containment, genetic algorithm in network optimization, etc. His current projects include vulnerability in sensor applications, security in multi-channel wireless networks, authentication in ad hoc and sensor networks, and security in peer to peer systems.

Christopher Ferguson earned his Bachelors degree from Texas State University in the spring of 2009. He was a research assistant on the project of securing sensor applications. His interests include open source performance ecu software and hardware development, network security, and embedded systems. He is currently a software engineer for AT&T Wifi Services.

Rizwan Noorani is currently working as a software developer at Embark Enterprises. He received his undergraduate degree from Texas State University in 2007 where he worked as a research assistant as well. He received several awards including outstanding student award. Rizwan's interests are Computer Networks, Security and detecting vulnerabilities in various Network protocols.

Please cite this article in press as: Gu Q, et al., A study of self-propagating mal-packets in sensor networks: Attacks and defenses, Computers & Security (2010), doi:10.1016/j.cose.2010.10.002