Towards Self-propagate Mal-packets in Sensor Networks

Qijun Gu
Department of Computer Science
Texas State University at San Marcos
San Marcos, TX, 78666
qijun@txstate.edu

Rizwan Noorani
Department of Computer Science
Texas State University at San Marcos
San Marcos, TX, 78666
rn1048@txstate.edu

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 just the same as in regular computers due to sensor's hardware and software architecture. In contrast to worm attacks, malcodes carried by exploiting packets cannot be executed in a sensor. 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 codes in a sensor to propagate itself without disrupting 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 hard to propagate under different traffic situations.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive Software*; C.2.0 [**Computer-Communication Networks**]: General—*Security and Protection*

General Terms

Security

Keywords

Mal-packet, Buffer overflow, Control flow, Sensor worm

1. INTRODUCTION

Applications in sensor networks have been researched and developed for years. However, most of the security work were 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 [6], format string vulnerability [25], integer overflow [31], double free [7], heap overflow [19], return-to-libc [24], etc. Such cyber attacks in regular computer systems lead us to starting thinking on similar threats in sensor networks.

As sensors are using very simple embedded systems, sensors do not have sophisticated operating systems to manage codes for safety. Simple operating systems [5, 2] 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 [3] 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, applications are sharing more and more common codes as they are using the similar development environment. Hence, memory fault attacks based on the same principle in regular computers become threats to sensor networks too.

Unique characteristics of sensor's hardware and software present both advantage and disadvantage to attackers. In terms of attacker's advantage, sensors in the same network are mostly using homogeneous devices and software. An attacker can capture one sensor and read application codes out from a JTAG interface that can be found in most embedded systems [18]. It would be easier for attackers to obtain source codes if applications are open source. Therefore, if one sensor can be compromised due to a vulnerability in application codes, all other sensors that use the same devices and codes in the same network can be compromised due to the same vulnerability. In terms of attacker's disadvantage, injected mal-codes that are commonly found in worm packets are hard to be executed 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 focuses on *what an attacker can exploit and accomplish in a sensor with a mal-packet.* Note that unlike worm packets in the Internet, mal-packets in a sensor network do not carry mal-codes. A sensor mal-packet carries only mal-data but misuses existing codes in a sensor to accomplish its attack. The paper proposes a range of attack approaches for a mal-packet to propagate itself. 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 codes. Attackers may use the techniques proposed in the paper to threat 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. Since the false message comes from a good sensor, the message will possibly be accepted and influence operations of the network.

The paper contributes in three aspects. First, the paper identifies a range of approaches for a mal-packet to invoke routines without disrupting sensor's normal functionality. The paper shows that packets carrying only mal-data can accomplish attacks as those carrying mal-codes. 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.

The remainder of the paper is organized as follows. Section 2 discusses the main challenges in exploiting vulnerabilities in sensor applications. Section 3 gives the details of exploitation approaches to make a self-propagate mal-packet. Section 4 describes a prototype of mal-packet that has been implemented and tested. Simulation is also conducted to study propagation patterns of mal-packets in sensor networks. Section 5 discusses and compares possible defense technologies against sensor mal-packets. Related works on attacks and worm propagation are summarized in Section 6. Finally, the paper is concluded in Section 7.

2. CHALLENGES OF EXPLOITING SENSORS

Since sensor nodes use a very different architecture for program and data storage, it is reasonable to question how vulnerability could be exploited in sensor nodes. In this paper, we focus on exploiting vulnerable codes running in processors of low cost sensors, such as Atmel's ATmega128 [1] and TI's MSP430 [4]. This kind of processor has a RISC core running single cycle instructions, a well-defined I/O structure, and two separate memories: data memory and program memory. For example, the memory in a ATmega128 processor is depicted in Figure 1. In such a processor, although attackers can use many well known buffer overflow techniques, exploitation in sensor nodes is constrained by

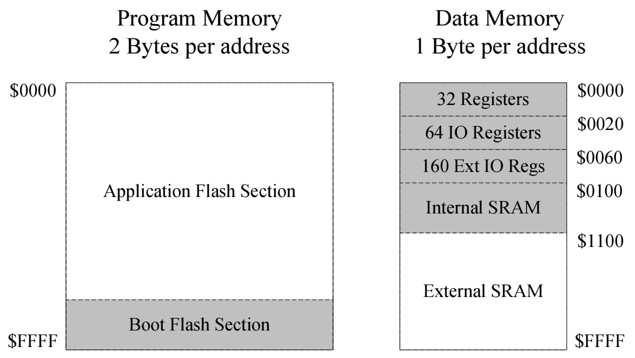


Figure 1: Memory Structure of ATmega128

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.

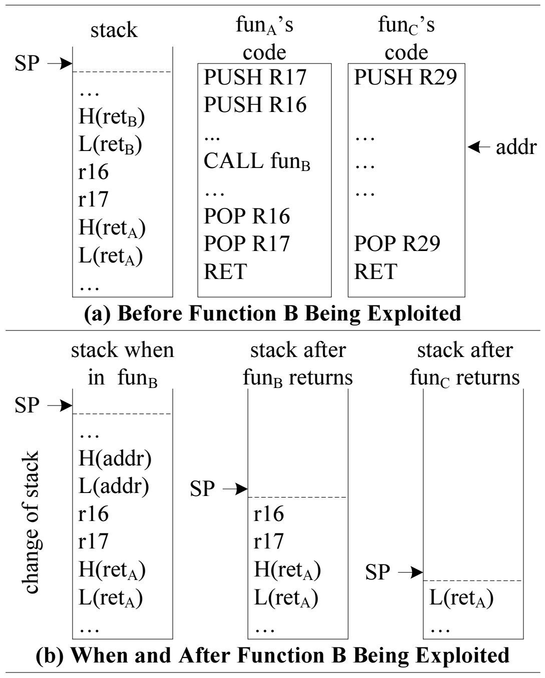
2.1 Non-Executable Injected Codes

A worm packet normally carries executable codes in order to replicate and propagate itself. In a regular computer (such as a Linux system), a program can access either codes (in TEXT section) or data (in BSS or HEAP or STACK section) because codes and data are stored in the same memory. As a consequence, an attacker can inject a piece of malicious codes into memory as a piece of data and then alter control flow to execute the injected codes.

However, it is found [17] that such injected malicious codes cannot be executed in a sensor[1]. Code and data memories in a sensor 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 in a sensor. Program memory in a sensor also has dedicated lock bits for write and read/write protection so that (a) sensors can work reliably in a hostile environment and (b) attackers cannot modify codes that have been loaded in program memory.

Consequently, worms that execute injected codes are hard to be effective in sensors. This paper will mainly look into techniques for making mal-packets that only carry mal-data and use existing codes in program memory. Two issues need to be addressed when exploiting existing sensor codes. 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 propagates 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. Nor a mal-packet 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

[1]This paper does not study the scenario where a sensor can self-program on the fly, since many applications disable the self-programming feature or do not have self-programming codes in program memory.



∗ *SP shows the stack pointer that is changed following how codes are 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 control flow.*

Figure 2: Reset

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.

2.2 Reset after Exploitation

Altering 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 can be illustrated in Figure 2, where an attacker exploits a vulnerability in function B that is called by 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. Figure 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 Figure 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 in stead. Hence, when function C returns, $r16$ will be popped into $R29$, and $r17$ and $H(ret_A)$ will be used as the return address of function C.

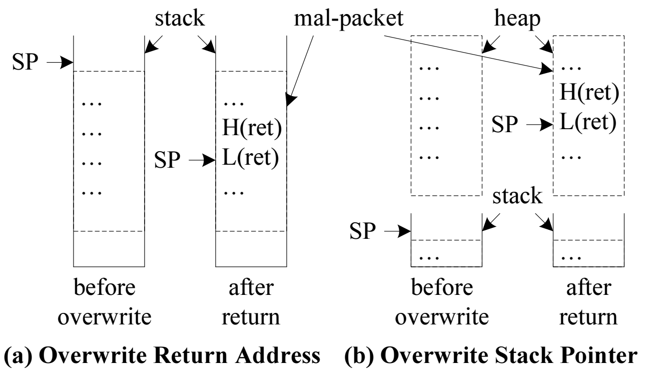
Apparently, two problems are raised by this exploitation. First, function C will return to a non-determinable address composed by $r17$ and $H(ret_A)$. If the return address is illegal, the program counter (PC) will be reset to 0 and then the RESET routine will be invoked and the senor will be restarted. If the return address is legal, the sensor will go into an unknown status until PC is loaded with an illegal address. Then, the sensor will be restarted as well by the RESET routine. Second, registers are modified by the altered control flow. For example, $R29$ is an important register that is frequently used as a pointer to the top of stack in various functions. Changing $R29$ 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.

2.3 Other Factors

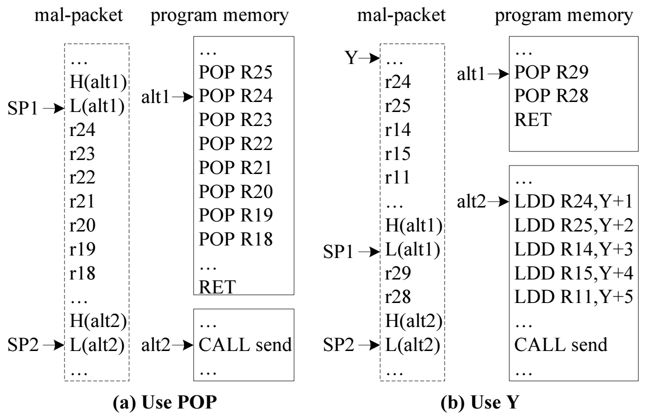
The capability of an attack using mal-packets in a sensor network is also affected by several other factors. First, packets in a sensor network may only have a small size of payload. For example, the default maximum size of packet payload in a MICA2 sensor node is 28 bytes. Although applications can change the maximum payload size limit in source codes, 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 mal-packet attacks . 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, and (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 codes could 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.



∗ *SP is the stack pointer. In (b), SP is overwritten to point to the mal-packet stored in heap.*

Figure 3: Change of Return Address and Stack Pointer



∗ *$SP1$ is the stack pointer when control flow is altered to program address $alt1$ to get arguments.*
∗ *$SP2$ is the stack pointer when control flow is altered to program address $alt2$ to invoke the transmission function.*

Figure 4: Invocation of Function $AMSend\$send$

Therefore, this paper will assume 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 codes, if a vulnerability presents.

3. EXPLOITATION TECHNIQUES

Being aware of the challenges, this section presents a set of techniques to make self-propagate mal-packets. These techniques need to address four major aspects of making a sensor mal-packet: alteration of control flow, invocation of transmission function, restoration of control flow, and composition of mal-packet payload.

3.1 Alteration of Control Flow

In this paper, we mainly focus on attacks that redirect control flow when a vulnerable function returns. Hence, exploiting techniques need to provide an arbitrary return address. We are aware that attackers may also figure out some other approaches to alter control flow. For example, if a function pointer is stored in data memory, overwriting this pointer via heap overflow or stack overflow can redirect control flow when calling this function.

In this section, we demonstrate two types of exploitation techniques (in Figure 3) that exploit return address to alter 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.

First, attackers can use many existing techniques to overwrite a return address in stack, such as stack overflow, double free, string format, etc. Many defense techniques [13, 32, 14, 12] have been developed to protect return addresses. Attacks that overwrite and exploit return addresses still continue to happen due to insufficient security awareness of programmers. Figure 3(a) illustrates how a mal-packet overwrites a return address in stack. Because the mal-packet does not change stack pointer, the stack below the modified return address will be used by the function that control flow is altered to. Hence, the mal-packet needs to overwrite data in 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. We find that attackers can overwrite stack pointer. For example, in the data memory of a ATmega128 processor (Figure 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 $0x5D$ and $0x5E$. Application codes 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. Figure 3(b) illustrates how a stack pointer is changed by a mal-packet. The mal-packet is stored in heap and exploits vulnerability such as heap overflow or double free 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, (2) the mal-packet provides the data below the return address for further exploitation after control flow is altered, and (3) the mal-packet will be used as a stack during exploitation. 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 codes.

3.2 Invocation of Transmission Function

In order to propagate itself, a mal-packet needs to invoke some functions to send itself out. As discussed in Section 2, a mal-packet in a sensor cannot carry and use its own codes for transmission. Hence, the mal-packet needs to alter control flow to an existing transmission function in the infected

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	$R24$

sensor. For example, the following function in TinyOS exists in many sensor applications that send packets.

$AMSend\$send(am\_addr\_t\ addr,\ message\_t\ *msg,\ uint8\_t\ len)$

In order to invoke the function, a mal-packet needs to provide arguments. In a regular computer, arguments of a function are pushed into stack before the function is called. However, since a sensor uses an ARM 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 arguments by simply overwriting the stack of a function. We identified two techniques that can provide arguments to the $AMSend\$send$ function. Figure 4 demonstrates an example of invoking the function in a Mica2 sensor. The $AMSend\$send$ function actually calls the **transmission function** $CC1000ActiveMessageP\$AMSend\$send$ (function *send* in Figure 4) that needs four arguments. The arguments are passed via seven registers ($R18 - R24$) to the transmission function.

The first approach to pass arguments to the transmission function is using a routine that can pop values into the argument registers, as shown in Figure 4(a). A mal-packet needs to alter control flow twice. First, 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 provides mal-data for $R18 - R24$ right below where the return address $alt1$ is stored. The mal-packet should ensure the stack depth between $SP1$ and $SP2$ 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 $alt2$ where the transmission function is called.

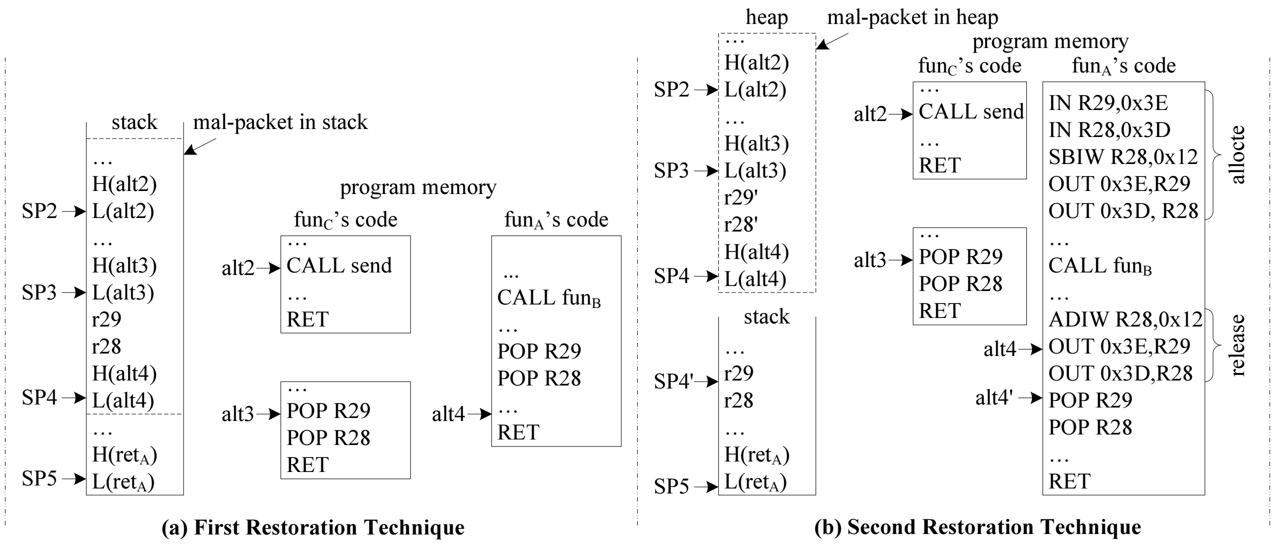
The second approach to pass arguments to the transmission function is using the Y register (i.e. registers $R28$ and $R29$ in ATmega128) to provide arguments carried in the mal-packet. Figure 4(b) shows one example in which a function, that calls the transmission function, loads five bytes from a data address pointed by the Y register to five registers ($R24$, $R25$, $R14$, $R15$, $R11$). 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, control flow is altered to a pop routine at program address $alt1$ 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 ($Packet\$getPayload$) and the function to send a packet ($AMSend\$send$). 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, control flow is altered to program address $alt2$ 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 codes. An attacker needs to reversely trace the data flow [21, 35] 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 not constrained by the depth of stack. In the first approach, the arguments of the transmission function can only be stored below where $alt1$ is stored in stack. The payload of a mal-packet before $alt1$ is not utilized and the stack must have a sufficient 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 $alt1$, the arguments can be stored before $alt1$.

3.3 Restoration of Control Flow

As discussed in Section 2.2, altering 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 control flow is restored as if codes were 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 control flow respectively. Figure 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 control flow is altered to program address $alt2$ 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 control flow back to function A so that function A can return to function F as normal.

Figure 5(a) shows the restoration technique when a mal-packet is stored in stack and overflows 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, control flow is altered to a pop routine at program address $alt3$ 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, control flow will be altered to somewhere at program address $alt4$ before "RET" in function A. The selection of $alt4$ needs to ensure the number of "POP"s between $alt4$ and the program address of "RET" of function A matches the number of bytes between $SP4$ and $SP5$ in stack so that function A will return to where it is supposed to return. Hence, $alt4$ points to a program address between



* $SP2$ *is the stack pointer when control flow is altered to program address* $alt2$ *to invoke the transmission function.*
* $SP3$ *is the stack pointer when control flow is altered to program address* $alt3$ *to restore the* Y *register.*
* $SP4$ *is the stack pointer when 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.*

Figure 5: Restoration of Control Flow

"POP R28" and "RET" in function A. Accordingly, when function A returns, control flow is restored.

Figure 5(b) shows the other restoration technique when a mal-packet is stored in heap and overflows heap. As discussed in Section 3.1, 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 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 codes at program address $alt4$ to use the Y register to restore the stack pointer. The mal-packet can thus first alter control flow to $alt3$ to load the Y register, and then alter control flow to $alt4$ to restore the stack pointer via the Y register. Note that the value of stack pointer is restored to $SP4'$ that points to the data address in stack where the Y register of function F is saved. When being popped at program address $alt4'$, the Y register of function F is restored as well. Because stack is not changed by the mal-packet, control flow is restored.

In both restorations, 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. Such application errors most likely cause the sensor to quit from current task and execute the next task. However, application errors do not reset a sensor and thus do not eliminate the mal-packet and its attack effect.

3.4 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 3.1 to 3.3. Figure 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 control flow is being altered.

The payload consists of three parts. Part 1 provides data for buffer overflow and program address $alt1$ to which control flow is altered when the exploited vulnerable function returns. $alt1$ points to the routine that pops arguments for the transmission function. Part 2 provides arguments of the transmission function. As discussed in Section 2.3, a mal-packet propagates itself via broadcasting. Hence, in part 2, the destination address carried in $R20$ and $R21$ is set to the broadcast address $0xFFFF$. The mal-packet can spoof any source address in $R18$ and $R19$. The payload pointer in $R22$ and $R23$ 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 $R24$ is filled in. At the end of part 2, program address $alt2$ is set to the address where the transmission function is called. Finally, part 3 provides data for restoring 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

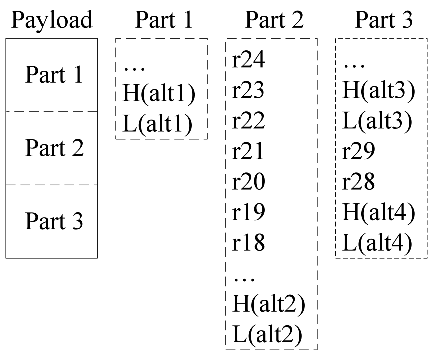
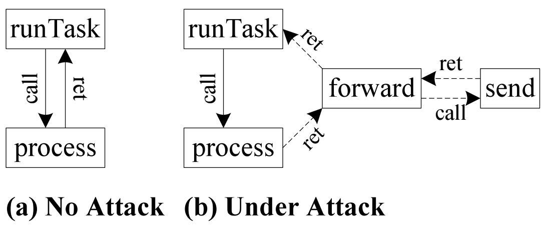


Figure 6: Mal-packet Payload



∗ *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.*

Figure 7: Control Flow in Demo

be achieved only if any routine used by a mal-packet does not have extra local variables in stack. In order to match the codes 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 can be only a few more bytes than the minimum size.

4. IMPLEMENTATION AND SIMULATION OF SENSOR MAL-PACKETS

In this section, we first present the implementation of a sensor mal-packet that can propagate itself utilizing only codes in a vulnerable sensor application. Then, we discuss how the mal-packet propagates itself.

4.1 Prototype of Sensor Mal-packets

For demonstrating the feasibility of sensor mal-packet, we developed a sensor application[2] that is supposed to only forward a received packet to its next hop. 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.

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 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 Figure 7.

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-

[2]The demonstration and the sensor mal-packet can be downloaded from *http://www.cs.txstate.edu/ qg11/download.htm.*

Table 2: Functions in Demonstration

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

data for altering and restoring control flow. 6 bytes are used for matching the stack when altering control flow. The mal-packet invokes function *send* to broadcast itself and then ensures function $runTask$ can return as usual.

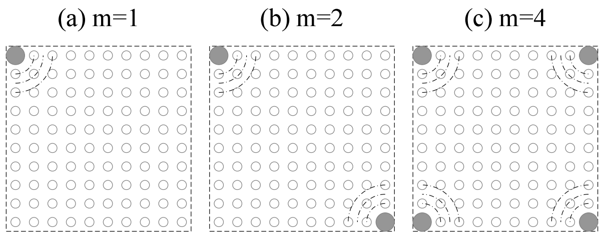
Note that this example intends to provide an exploitable vulnerability, because the purpose of the paper is to illustrate that a sensor mal-packet can be implemented if an exploitable vulnerability is found. The paper does not study how to find a vulnerability in sensor applications.

4.2 Simulation of Mal-packet Propagation in Sensor Networks

Because sensor mal-packets studied in the paper use broadcasting to propagate themselves, the epidemic models [30, 36] for worm propagation may not capture the characteristics of mal-packet propagation in sensor networks.

First, although a mal-packet can invokes 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 by 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 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 proba-



∗ Sensors are evenly distributed in a square network.
∗ Persistent mal-packet sources (big gray nodes) are placed in the corners of the network.

Figure 8: Simulation

Table 3: Parameters of Simulation

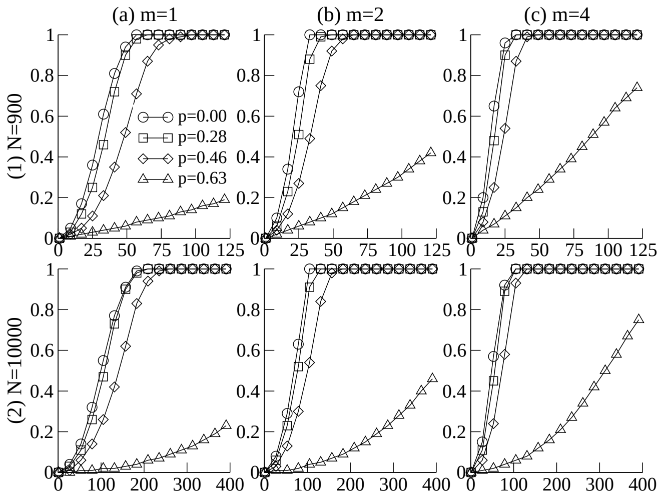
Para	Description	Values
N	Number of sensors	900, 10000
m	Number of mal-packet sources	1, 2, 4
p	Probability of broadcast failure	0.00 - 0.63

bility 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 factors of network traffic and topology. We use simulation to study the mal-packet propagation. The simulation is conducted in two types of networks. One is a medium-size network of 900 sensor nodes and the other one is a large-size network of 10000 sensor nodes. As illustrated in Figure 8, sensors are evenly distributed in a square area and a few mal-packet sources are placed in the corners. 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 50 random experiments with the same set of parameters.

Figure 9 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 it slows down when most 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



∗ The X-axis is the time in second.
∗ The Y-axis shows the percentage of sensor nodes being infected over time.
∗ Each subfigure shows the propagation patterns for a different set of parameters N and m with $p = 0.00, 0.28, 0.46, 0.63$.

Figure 9: Propagation Patterns of Mal-packets

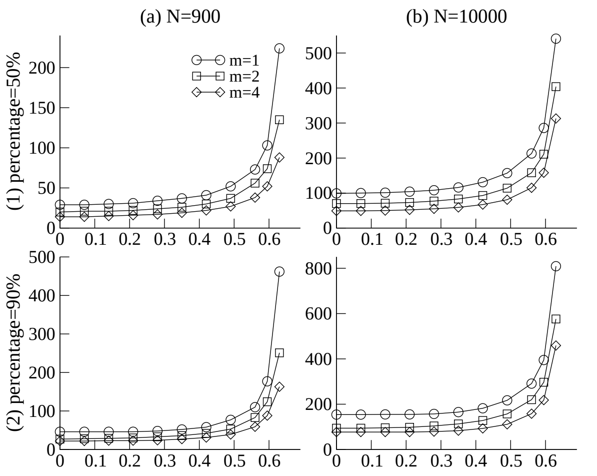
broadcast a mal-packet to its neighboring vulnerable sensors. Hence, the propagation curves for $p = 0.00, 0.28, 0.46$ are quite close and almost grow linearly over time.

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, in Figure 8, the longest distance in subfigure (a) (from the top left corner to the bottom right corner) doubles the longest distance in subfigures (b) and (c) (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 seconds that doubles the times for $m = 2$ and $m = 4$, which are 110 seconds and 100 seconds respectively.

Figure 10 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. When 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. DEFENSE

Since mal-packets alter 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.



∗ *The X-axis is the probability of broadcast failure.*
∗ *The Y-axis shows the time to compromise a certain percentage (50% or 90%) of sensor nodes.*
∗ *Each subfigure shows the results for a different set of parameter N and percentage of infected sensors with $m = 1, 2, 4$.*

Figure 10: Infection of Mal-packets

In the following, we examine their applicability and limitations if being deployed against self-propagate mal-packets in sensor networks. Note that the emphasis of this paper is on possible attack techniques. The research on applicable defense approaches is a part of our ongoing and future work.

(1) Memory-related vulnerabilities are fundamentally induced by programming bugs. Tools have been developed to find bugs in source codes. Bug detection techniques [15, 33] are used to examine source codes before generating final executables. They analyze source codes according to certain patterns. Another type of bug detection tool [21, 35] use static analysis to analyze control and data flows in an application. They follow the execution of an application, identify possible unsafe states in the execution, and thus detect bugs. These technologies can be used for analyzing codes of sensor applications. Although they are effective to detect certain types of bugs, they are facing challenges as applications become more and more complicated and programmers may not have sufficient security awareness.

(2) Run-time tools [11, 20] have also been developed to capture run-time patterns caused by attacks. They monitor execution of application codes, 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 [34] inspect incoming packets and detect and filter suspicious packets if they contain a sequence of codes. 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 [13], make a copy of return addresses [32], reorder local variables and function arguments and copy function pointers [14], or encrypt function pointers [12]. However, these techniques need a secure data area to store critical data for defense. As we discussed, no safe address exists in sensor's data memory. Sophisticated attacks may also evade these techniques to modify data memory and alter control flow.

(4) Observing that a mal-packet needs to alter control flow to execute codes at known addresses, randomizing address space [23, 26] could be a valid defense approach. The approach randomizes the base address of the stack, heap and code segments at load or link time in regular computers. Functions can also be transformed in source codes so that function addresses are randomized in final executables [8]. Nevertheless, two problems are not address when deploying the randomization techniques in sensor applications. First, sensor's application codes are not loaded at run-time. It is not secure if all sensors share the same image of randomized codes. The randomization must be performed per sensor. Second, it is showed [28] that even a 32-bit address space is insufficient to keep attackers from guessing the correct addresses. It is thus much less secure given that a sensor has only a 16-bit address space.

(5) A few schemes have been developed to provide memory safety in TinyOS. Safe TinyOS [27] treats function pointers as safe, sequence and wild (the later two are unsafe), and transforms application's source codes to codes using safe pointers. Safe TinyOS also provides a software framework that has a kernel space for OS-related modules and a extension space for untrusted application modules. Harbor [22] 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. As we showed in Section 3, an attacker can find exploitable routines directly from assembly codes. These routines are usually system routines that are not included in applications. Hence, mal-packets may evade these schemes.

6. RELATED WORKS

6.1 Attacks

Many attackers are exploiting vulnerabilities due to memory fault in current computer systems. Such attacks can be categorized as control data attack [16, 29] and non-control data attack [10, 9]. Control data refers to the code addresses that are loaded in program counter (PC) at some point in program execution. This paper shows that sensor applications are susceptible to attacks that alter control flows 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 [6], return address in stack is overwritten to the address where injected codes are executed when the function (corresponding to the current stack frame) returns. When target program's control data is modified, attackers can execute injected malicious code or out-of context library code at the memory address pointed by the altered control data. Various attack techniques emerged to overwrite control data via exploiting vulnerabilities of format string error [25], double-free error [7], heap overflow [19], return-to-libc [24], etc.

Non-control-data attacks, based on another principle, are also threatening networks. It is found [21, 35] 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 [9]. In such an attack, attackers examine codes of software 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 semantic of software.

6.2 Models of Worm Propagation

In the analysis of worm propagation [30], 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 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 [30, 36]. This research shows that the propagation of mal-packets in sensor networks presents different features.

7. CONCLUSION

In this paper, we have presented several exploitation techniques by which an attacker can make self-propagate mal-packets to infect sensors in a network. In contrast to worm packets, mal-packets studied in this paper only carries specially crafted data, but can exploit memory-related vulnerabilities and utilize existing application codes 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 hard to propagate under different traffic situations. In the future work, we will explore other attacks that can be achieved with the exploitation techniques proposed in this paper and study effective defense technologies suitable in sensors.

8. REFERENCES

[1] Atmega128. http://atmel.com/dyn/products/product-card.asp?part-id=2018.

[2] Mantis. http://mantis.cs.colorado.edu/.

[3] nesc: A programming language for deeply networked systems. http://nescc.sourceforge.net/.

[4] Ti msp430. http://www.ti.com/msp430.

[5] Tinyos. http://www.tinyos.net.

[6] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, http://www.phrack.org/phrack/49/P49-14, 1996.

[7] Anonymous. Once upon a free(). Phrack Magazine, http://www.phrack.org/phrack/57/p57-0x09, 2001.

[8] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.

[9] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[10] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *the 12th USENIX Security Symposium*, 2003.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, 2005.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[13] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[14] H. Etoh and K. Yoda. Propolice: improved stack-smashing attack detection. IPSJ SIGNotes Computer SECurity, http://www.trl.ibm.com/projects/security/ssp, 2001.

[15] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19, 2002.

[16] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, 2003.

[17] Q. Gu. Analysis of software vulnerability in sensor nodes. In *Proceeding of International Conference on Security and Management*, 2007.

[18] B. Jack. Exploiting embedded systems. Black Hat Europe, 2006.

[19] M. Kaempf. Vudo malloc tricks. Phrack Magazine, http://www.phrack.org/phrack/57/p57-0x08, 2001.

[20] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[21] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security*, 2005.

[22] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. In *ACM IPSN*, pages 340–349, 2007.

[23] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM CCS*, 2003.

[24] Nergal. The advanced return-into-lib(c) exploits (pax case study). Phrack Magazine, http://www.phrack.org/phrack/58/p58-0x04, 2001.

[25] T. Newsham. Format string attacks. http://muse.linuxmafia.org/lost+found/format-stringattacks.pdf, 2001.

[26] PAX. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[27] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Memory safety and untrusted extensions for tinyos. Technical report, University of Utah, 2006.

[28] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address space randomization. In *ACM CCS*, 2004.

[29] Smirnov and T. Chiueh. Dira: automatic detection, identification and repair of control-data attacks. In *Network and Distributed System Security Symposium*, 2005.

[30] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *the 11th USENIX Security Symposium*, pages 149–167, 2002.

[31] P. Starzetz. Crc32 sshd vulnerability analysis. http://packetstormsecurity.org/0102exploits/ssh1.crc32.txt., 2001.

[32] Vendicator. Stackshield. http://www.angelfire.com/sk/stackshield.

[33] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2002.

[34] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *USENIX Security*, 2006.

[35] H. Xu, W. Du, and S. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *Symposium on Recent Advances in Intrusion Detection*, 2004.

[36] C. Zou, W. Gong, and D. Towsley. Worm propagation modeling and analysis under dynamic quarantine defense. In *the 2003 ACM workshop on Rapid Malcode*, pages 51–60, 2003.