Group E: Embedded Systems

Dustin Graves

CSCI 342

January 25, 2007

References

- A. Seshadri, A. Perrig, L. Van Doorn and P. Kholsa, “Software-Based Attestation for Embedded Devices,” 2004 IEEE Symposium on Security and Privacy.
- A. Seshadri, A. Perrig, L. Van Doorn and P. Kholsa, “Using SWATT for Verifying Embedded Systems in Cars,” 2004 Embedded Security in Cars Workshop.
- L. de Alfaro and T. Henzinger, “Interface Automata,” Proc. Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109-120.

Introduction

- Embedded systems are special purpose computer systems which are fully contained by the devices that they control
  - Perform very specific tasks
  - May be highly optimized by designers
  - Typically have small form factor and low cost
- Nearly all commercial electronic devices contain some type of embedded system
  - Microcontrollers/Microprocessors
- Access to embedded systems has traditionally been very limited

Introduction (cont.)

- Embedded systems are becoming more complex
  - General purpose PDAs and Handheld devices are considered embedded systems
  - General purpose operating systems are available for embedded systems
- Access to embedded systems is becoming increasingly available
  - USB/RS-232 interfaces
  - Network connectivity
  - Bluetooth

Uses

- Automotive
  - Engine control, automatic transmission, and anti-lock breaking systems
- Avionics
  - Flight control and inertial guidance systems for aircraft, rockets, and missiles
- Communication
  - Telephone switching systems, cellular/mobile phones, routers, IP Telephony
- Handheld Computing

Uses (cont.)

- Home appliances/automation
  - Televisions, DVD players, microwave ovens, digital thermostats
    - Some of these things are now internet enabled
- Commercial appliances
  - Automated checkout, cash registers, inventory control systems
- Computer peripherals
  - Printers, input devices

Threats

- Compromise privacy and safety by altering memory contents of embedded devices
  - Eavesdropping
    - Government and Corporate Espionage
    - Blackmail, identity theft
      - IP phones, network printers
  - Unauthorized access to services/functionality
    - Reconfiguring devices, changing system settings, circumventing copy protection/safety protocols
  - Disruption of normal operation
    - Viruses, sabotage, system destruction

Vulnerabilities

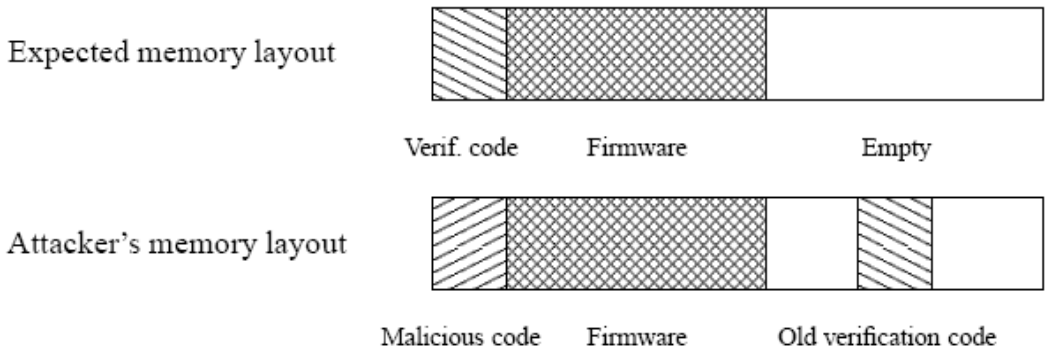
- There is a current lack of security for embedded systems
- Embedded systems are easy to identify and obtain information about
  - Commercially available microcontrollers stamped with manufacturer id and part number
  - Data sheets are publicly available
- Network connectivity and mechanisms for firmware upgrades are making microcontroller memory easier to access

Ideal Solution

- A method to test for the presence of software and configuration alterations within the memory contents of embedded devices
  - High probability of detecting any change to memory, no matter how small
  - Does not require additional hardware
    - No increase to cost or space requirements
  - Does not require direct access to memory
    - Check memory with an external verification device
  - Can be used with legacy hardware

A Naïve Approach

- Test the contents of an embedded device's memory with a message authentication code
  - Verifier challenges the device to compute a MAC
    - A random MAC key is transmitted to the device
    - The device computes the MAC for the entire memory content using the MAC key
- Malicious code could store original memory content at an empty memory location
  - MAC is computed by malicious code using the original program



Typical Microcontroller

- Zilog Z8 Encore! – Z8F6403
  - eZ8 CPU, 8-bit Harvard Architecture, 20Mhz
  - 64K flash memory with in circuit programming capability
- Programmable flash memory for application and data storage
  - First 38 bytes (Addresses 0x00-0x37) are dedicated to program setup information and can not be relocated
    - Flash Option Bits, Reset Vector, Interrupt Vectors
      - Program execution begins at the reset vector address
  - Programs typically start at address 0x38 but can be relocated

A Good Program…

- Flash Loader - A program that permanently resides in flash memory to load a “user” application to memory when activated by an external signal
  - Activates at boot and checks for external signal
  - Executes the user program currently residing in memory when external signal is absent
  - Enters flash loader mode when external signal is present
    - Receives and writes new user program to memory one page at a time
    - Reset vector is overwritten during program load and must be reset to start address of flash loader after loading is complete
- Mechanism for upgrading software/firmware for devices in the field

Gone Bad

- The Flash Loader concept requires very little modification to become malicious
  - Exploit flash loader to load malicious software
  - Insert or replace existing flash loader with a modified flash loader
    - Conditionally execute malicious code at boot
  - Combined approach to load a program responsible for replacing the current flash loader with a modified version
    - Original program could be reloaded by modified flash loader
    - Difficult to detect because system exhibits characteristics of normal operation most of the time and can compute valid MAC

What is SWATT?

- A SoftWare-based ATTestation technique for detecting the presence of malicious code in the memory of embedded devices
  - Checks for modifications to code, static data, and configuration settings
  - Does not require special hardware
    - No significant increase to cost
    - Compatible with legacy devices
- External verification device for validating the contents of memory
  - Does not have direct access to device's memory
- High probability for detection of malicious code
  - Can reliably detect modification to a single byte of memory
  - Attacker must modify hardware to conceal memory content changes

Basic Requirements

- SWATT requires full knowledge of hardware properties for correct operation
  - Clock speed
  - Instruction Set Architecture (ISA)
  - Memory architecture and size
- Knowledge of the expected memory contents of the device is required for validation
- A verification procedure must reside on the device
  - May be installed at any time prior to validation
  - Must be capable of remote activation
  - Device does not need to contain a trusted version of the verification procedure
- SWATT assumes the attacker has full control of the device's memory, but does not modify the device's hardware

Verification Procedure

- The verifier sends a random challenge to the device
  - Challenge is randomly generated
- The device uses the verification function to compute a response
  - The challenge is used to seed a PRNG that generates random memory addresses to be accessed
  - The verification function performs a pseudo-random memory traversal, updating a checksum with the contents retrieved by each memory access
  - Attacker can not predict which memory locations will be accessed
    - Requires that malicious code contain conditional logic to determine if an access request is made for an altered memory location
    - Execution of conditional logic slows computation
- The verifier checks response with a value computed from the expected contents of the device
  - The presence of malicious code has been detected if the response is invalid or response time is noticeably long

Implementation Requirements

- Six requirements must be satisfied for SWATT to perform effectively
  - Pseudo-random memory traversal
  - Resistance to pre-computation and replay attacks
  - High probability of detecting even single-byte changes to memory contents
  - Small code size
  - Efficient implementation
  - Non-parallelizable

Implementation

- Pseudo-random memory traversal
  - Forces malicious code to test each specified memory address for correspondence with memory that has been altered
    - Code can not predict which memory locations will be tested
    - Conditional logic must be included for address testing
  - SWATT employs a cryptographic PRNG for address generation
    - Choice of PRNG depends on CPU architecture
      - 8-bit CPUs can use key stream generated by RC4 cipher
      - 32-bit CPUs can use key stream from Helix stream cipher, with built-in MAC functionality for checksum computation
      - 16-bit and 32-bit CPUs can use bits from a multiword T-function

Implementation (cont)

- Resistance to pre-computation and replay attacks
  - Achieved by the randomly generated challenge transmitted to the device from the verifier
    - Provides seed for PRNG

Implementation (cont)

- High probability of detecting change
  - Accessing each memory address with equal probability
    - Verification procedure requires O(n ln n) memory accesses (n is size of memory)
    - Coupon Collector's Problem describes number of random memory accesses, X, required before all memory locations have been accessed
      - $\Pr[X > cn \ln n] \leq n^{-c+1}$
      - This means that the probability that all memory addresses are not accessed by the O(n ln n) verification procedure is very small
  - Make the checksum function sensitive to change
    - Checksum should differ if input changed by a single byte
    - Should be difficult to find a replacement value that computes same checksum as original memory value
      - Reduce collision probability (2^{-n}) with sufficient checksum size

Implementation (cont)

- Small code size
  - Insertion of *if* statement required to test for pseudo-random memory access request to altered memory location will increase the size and speed of the malicious code
  - An *if* statement is a compare instruction followed by a conditional jump
    - Require an additional 2-3 cycles to execute
  - An *if* statement placed within the pseudo-random address generation loop will create a noticeable slowdown of program execution

Implementation (cont)

- Optimized implementation
  - Verification procedure should be optimized to the extent that it can not be further optimized
    - An attacker should not be able to optimize the verification procedure such that the addition of an *if* statement does not noticeably affect program execution time
  - Small size of verification procedure's loop body allows effective hand-optimization
    - SWATT creators believe the code sequence generated for their example can not be further optimized

Implementation (cont)

- Non-parallelizable
  - It should not be possible to separate the verification function into units that can be executed in parallel
    - An attacker should not be able to increase the execution time of the verification procedure through parallelization
  - Both the address for the memory access and the checksum computation depend on the current value of the checksum
    - Checksum can not be partially computed by two processes
    - Each iteration of the computation loop depends on the result of the previous iteration

Challenges

- Probability of detecting a small change to memory can be increased by increasing the number of memory accesses
  - Requires a balance of execution time and reliability
- Ease of implementation depends on the memory architecture
  - Harvard architecture physically separates program and data memory
    - Verifier only needs to know expected contents of program memory
  - Von Neumann architecture locates program and data memory in same physical location
    - Verifier must know exact contents of data memory (program stack, registers) at time of verification procedure execution
    - May require insertion of checkpoints to program
- There is no guarantee that the memory contents of a device will not be altered between device verification and device use

Application Areas

- Attesting the memory contents of network printers from a central location
  - Printer should only be able to communicate with verifier during verification procedure
- Using a car key to attest an automobile's embedded systems at time of ignition
  - The average consumer will most likely not independently perform attestation
  - Must be incorporated into normal process by manufacturer
- Cable service provider testing for modifications to cable box for illegal access to service
- Attestation of the memory contents of electronic voting machines by election officials
- Virus detection for handheld devices
  - Verification of successful virus removal

Prevention

- The focus of SWATT is the detection of alteration to the memory contents and configuration settings of an embedded device
- Prevention of device alteration is more desirable than detection of device alteration
  - Secure hardware
  - Secure operating systems
  - Secure system design to limit access to critical components
    - Critical automotive systems have read-only connection, or no connection, to network enabled automotive systems
      - Firmware updates can not be made remotely
      - Critical systems report status only

Interface Automata

- Based on concept of an automata-based language to capture certain aspects of hardware and software component design
  - Consist of a set of states, actions, and sequences
  - Capture input assumptions about the order in which a method's components are invoked
  - Capture output guarantees about the order in which a component invokes external methods
- Identify compatibility between component interfaces
  - Use "optimistic" approach that treats two components as compatible if there is some environment in which they can work together
    - The "pessimistic" approach treats two components as compatible only if they work together in all environments

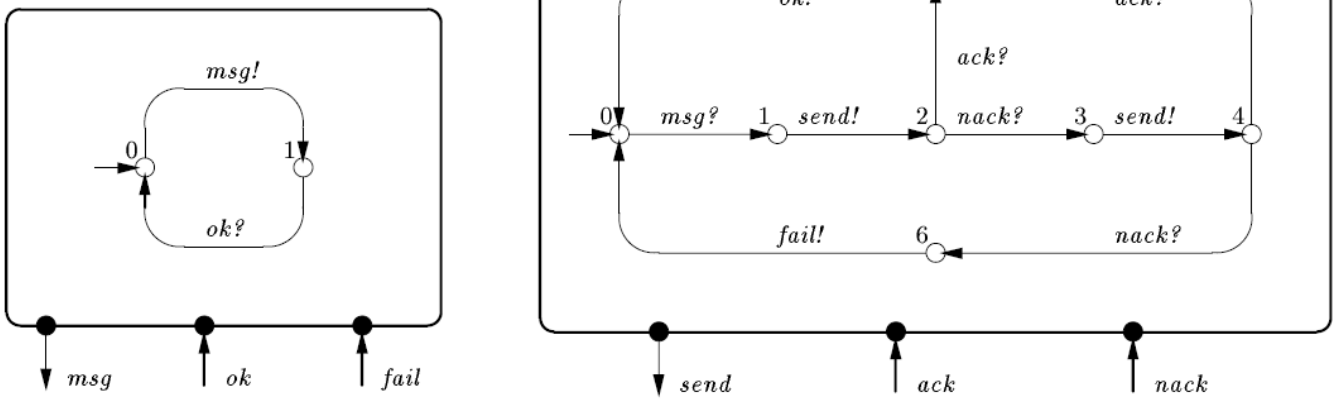
Formal Definition

- Interface automaton:
  - $P = \langle V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, T_P \rangle$
    - V_P is a state set
    - V_P^{init} is a subset of V_P
      - Can contain at most one state
      - If V_P^{init} is empty, P is empty
    - A_P^I, A_P^O, A_P^H are sets of input, output, and internal actions
    - T_P is a set of steps
      - Subset of the product of the state and action sets
        - » $V_P \times A_P \times V_P$

Composition

- Two interface automata may be composed if there exists a legal environment where they can work together
  - Illegal inputs may exist at every state of compositions
    - Error states exist if one automata generates an output that is an illegal input for the other automata
  - The environment within which the automata exist is restricted such that inputs that cause an illegal state are never generated
  - The existence of such an environment indicates interface compatibility
- The composition of two automata is obtained by pruning all states which result in illegal behaviour for the active environment from the product of the two automata components
  - Two automata are compatible iff their composition produces a non-empty set of states

Composition Example

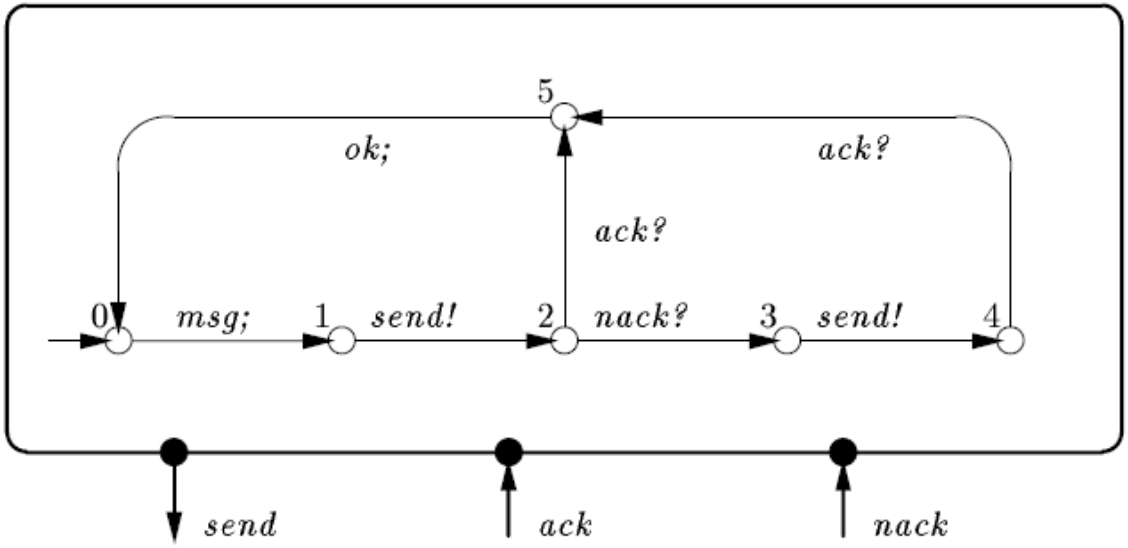


(a) Interface automaton *User*

(b) Interface automaton *Comp*

- The interface automaton *User* does not accept the fail input
- The interface automaton *Comp* generates a fail output after two unsuccessful send attempts
  - Creates an illegal state

Composition Example (cont)



(d) $User \| Comp$

- The composition of *User* and *Comp* requires an environment where only compatible states exist
  - The compatible states are 1, 2, 3, 4, and 5
  - 6 is an illegal state and is eliminated
- The interface automata are compatible in an environment where no two consecutive send attempts fail

Uses

- General uses for hardware and software components include design, validation, and synthesis
- Design of a verification procedure for validating the memory contents of an embedded device
  - Malicious code could simulate a compatible interface, as described by the naïve approach, and appear valid if response time is not considered
- Composition of a valid program and malicious code by an attacker
  - Design of a malicious program capable of appearing valid
- Creation of a simulator to test a verification procedure
  - Design verification procedure, valid program, and altered program
  - Simulate device that executes both valid and altered programs for testing the verification procedure

Additional References

- http://en.wikipedia.org/wiki/Embedded_system_overview
- http://www.schneier.com/blog/archives/2006/04/voip_encryption.html
- http://math.mit.edu/~pak/courses/pg/l10.pdf
- http://en.wikipedia.org/wiki/T-function