Karonte

A paper by Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitashvili, Christopher Kruegel and Giovanni Vigna

Presented by Billy Pedoto and Lucas Standaert
Agenda

- Problem Statement
- Background
- How it works
  - Border Binary Discovery
  - The BDG
  - Static Taint Analysis
  - Multi-Binary Data-Flow Analysis
  - Insecure Interaction Detection
- Conclusion
The Problem

- Internet of Things (IoT) are smart devices that connect to the internet
  - Smart Light Bulb
  - Smart Fridge
  - Routers
- These devices can be used by malicious users to carry out attacks
- State of the art techniques are not perfect in detecting vulnerabilities
  - This is because embedded components is itself a collection of interconnected components
- Treating each component as independent programs results in
  - Ignoring meaningful constraints
  - Inability to determine difference between Attacker Controlled and Non-Attacker Controlled Input
  - Possibility of uncovering only superficial bugs
Related Work

● Dynamic Taint Tracking and Emulation
  ○ Well known-technique for vulnerability detection, reduction in performance main reasons not integrating into
  ○ FirmaDyne, SURROGATES, Avatar address resource constraints, rely on presence of debugging ports

● Fuzzing
  ○ Driller, Dowser, DIFUZE, RPFuzzer, FIRMOAFL, FirmFuzz
  ○ Injecting malformed data into a program to force it to crash

● Static Analysis
  ○ Very few techniques detect general taint style vulnerabilities, suffer from scalability
  ○ None handle vulnerabilities that require modeling interaction between binaries
The Solution

- Karonte
  - A static analysis approach to determine insecure interactions between binaries

- Karonte reduces the number of False Positive insecurities

- This allows for meaningful and practical analysis of embedded components
IoT Attacker Model

● 2 methods of communication for smart devices
  ○ Directly with the user
  ○ In the Cloud

● Karonte was tested with attackers who speak directly to the devices
  ○ However it can be extended to the cloud based smart devices
Firmware Complexity

- Firmware is combined in 2 ways
  - Embedded Linux Distribution
  - Compiled into a single OS ("Blob Firmware")

- 86% of all firmware uses Linux

- Linux based firmware uses a bunch of interdependent binaries

- Only some of these binaries communicate with the network
  - Just analyzing these would miss tons of vulnerabilities
  - Testing each binary independently leads to thousands of false positives
Inter-Process Communication

- Vulnerabilities are found here
- Identified by a unique key
- Common IPC
  - Files
  - Shared Memory
  - Environment Variables
  - Sockets
  - Command Line Arguments
The Process

- Determines which components speak with the outside world

- Creates a directed graph of where each of those Border Binaries lead
  - Called the Binary Dependency Graph, or the BDG
  - This means that attacker data can be tracked through the system
  - Builds based off of Border Edge and IPCs

- Static taint engine Tracks Binary b in BDG inward to see where data goes

- Once the attacker flows are realized, reports back the possible vulnerabilities
Border Binary Dependencies

- Utilizes 5 features to determine parser functions in Border Binaries
  - The Number of Basic Blocks (#bb)
  - The Number of Branches (#br)
  - The Number of Conditional Statements used in conjunction with memory comparisons (#cmp)
  - Network marks (#net)
  - Connection marks (#conn)
Network marks

- Initially \#net is 0

- Is the probability that a parsing function handles network messages

- Determined by
  - Identifying all memory comparisons in code, and comparing the memory location to network encoding strings like HTTP
  - If so, increment \#net by 1
Connection Marks

- #conn is initialized as 0

- #conn is set to 1 if:
  - There is a network socket read function,
  - And that function is used in a memory comparison
The Math

$$\text{ps}_b = \max(\{\text{ps}_j \Sigma | \forall j \in \text{get functions}(b)\})$$,

$$\text{ps}_j = (\sum k_i \#i_j) \ast (1 + k_n \ast \#\text{net}_j) \ast (1 + k_c \ast \#\text{conn}_j)$$

- $K_n =$ Function that refers to network encoding keywords
- $K_c =$ Function that refers to Binaries that Parse Network data
The Math Continued

- Since all binaries will have an above 0 score, must distinguish the highest threat.

- Cluster the binaries into groups.

- Whichever group has the highest score, is the cluster that becomes the initial set of network facing binaries.
BDG Part 1: The CPF

- Communication Paradigm Functions determine if Binaries share data.

- If the binary can in fact share data, the CPF gathers the following:
  - Data Keys
  - Flow Direction Determination: I.e is this Binary a data setter or a data getter
  - Binary Set Magnification: Identifies Binaries in the firmware that will use any retrieved Data sets, and schedules them for further inspection

- All the CPFs together make up the edges of the BDG.
BDG Part 2: The BDG

- Binary Dependency Graph
- Disconnected Cyclic Bigraph
- BDG, G is a set of binaries, B, and the set of Edges, E
  - Represented G = (B, E)
- Each directed edge, e in set E, is represented by the triple:
  - e = ([b1,loc1,cp1],[b2,loc2,cp2],k)
- All data associated with Data Key k can flow from Binary b1 at Location loc1 via Communication Paradigm cp1 to Binary b2 at Location loc2 via Communication Paradigm cp2
BDG Part 3: The Algorithm

- Analyzes each binary in set B
- If the binary has a variable p used in memory comparison, p is then tainted.
- All other Binaries that use variable p are then added to the set B
- This continues until all of set B is analyzed

Algorithm 1: Binary Dependency Graph Algorithm

```
function BDG(int_locs, B, fw)
    comm_info ← {}
    E ← {} 
    for each b ∈ B do 
        locs ← get_locs(int_locs,b) 
        for each loc ∈ locs do 
            f_addr ← get_f_addr(loc) 
            for each block ∈ explore_paths(f_addr) do 
                if address(block) == loc then 
                    buf ← get_buf(loc) 
                    apply_taint(buf) 
                end if 
                if matches_CPF(block) then 
                    CPF_p = get_CPF(block) 
                    k ← find_data_key_and_role(block,CPF_p) 
                    B_new, int_locs_new ← get_new_binaries(fw,k,CPF_p) 
                    update_binaries(B,int_locs,B_new,int_locs_new) 
                    comm_info ← comm_info ∪ \{b, block, CPF_p, k\} 
                end if 
            end for 
        end for 
    end for 
    for each \{b, block, CPF_p, k\} ∈ comm_info do 
        if is_setter(block,k) then 
            getters ← get_getters(comm_info,k,CPF_p) 
            E ← E ∪ create_edges(b, getters) 
        end if 
    end for 
    return (B,E) 
end function 
```
Static Taint Analysis

- Propagates taint information following program dataflow
- Untaints when overwritten by untainted data, values constrained
- Based on BootStomp
- Two major improvements:
  - Path prioritization
  - Taint tag dependencies
Multi-Binary Data-Flow Analysis

- Discover insecure interactions by recovering data-flow details in BDG
- To avoid path explosion problem, look for least strict constraints
- Create Binary Flow Graph (BFG), extends BDG
- Edge in graph indicates data associated with key flow from binary b1 at location loc1 via communication paradigm cp1 with set constraints c1 to binary b2
- Building two phases
Initialization

- Create new edge for every edge in BDG, start uninitialized
- Retrieve every edge setter border binary, variable contains data shared at location
- Taint engine explore paths between entry point function to loc1 itself, collect constraints applied var1
- Add e to set wset, used during second phase
Constraint Propagation

- For every edge in edge set wset, set edge $c_1 = c_2$
- Retrieve variable used by $b_2$ receive data $loc_2$, find least strict constraint, $c_2 = c_2 \cup l_2$
- $b_2$ possibly further share, run taint engine wherever $b_2$ is setter
  - If find path, collect least strict set of constraints
  - If cannot find path, determine constraints similarly to $l_1$
- If uninitialized, or more restrictive, substitute it, add edge to wset
- Iterate until empty
Insecure Interactions Detection

- Leverage BDG find dangerous data flows, two subsets:
  - Memory-corruption
  - DoS (denial of service)
- To detect memory-corruption, look for memcpy-like functions, see if data unsafely reaches
- DoS, retrieve conditions control iteration of loop, check truthfulness depends on attacker data
- Both cases referred to as sinks
Insecure Interactions Detection

● Procedure:
  ○ Consider every edge of BFG, each node leverage static taint engine
  ○ When encounter location rely on provided CPF, retrieve address references attacker data
  ○ Apply taint and collect any constraints
  ○ If sink encountered, check if contains tainted data
  ○ If it is a loop, raise alert
  ○ If memcpy-like function, retrieve allocation point
  ○ If size of buffer greater than allocation point, raise alert

● At end perform single-binary static analysis for disconnected nodes
Experiments

- **Small-scale:**
  - 49 firmware samples from major IoT vendors
  - BootStomp dataset, 5 firmware samples, firmware blobs

- **Large-scale:**
  - Firmadyne data set
  - Firmware samples architecture supported by BootStomp

- Did not consider firmware samples for MIPS, as angr only partially supports MIPS
Results

- Compared four approaches:
  - Single-binary bug search
  - Static taint engine on border binaries
  - BDG algorithm on each firmware
  - Full approach
- Number of generated alerts decreased manageable number when apply full approach
- Manually investigated 50 randomly picked alerts, all false positives
- Uncovered 2 vulnerabilities per sample
- Expert analyst require 138 hours on Netgear, Karonte decrease 14 minutes
Results - Large-Scale

- Prototype analyze 80% firmware samples within day, scales well with firmware complexity
- Generate 2 alerts per sample, 1,037 alerts
  - Sample 100 randomly, 44 true positives, 30 multi-binary
  - Able detect critical flaw almost one case out of two
  - Can’t just check network parsers
- Firmadyne raise zero alerts
Conclusion

- Approach detect insecure interactions among embedded components
- Drastically reduce false positives from current techniques
- Using 53 firmware samples
  - Produced 87 alerts
  - 46 previously unknown zero-day bugs
- 899 large scale samples show that scales well
Critiques

- Possibly switch from using angr
  - Causes slowdown
  - Does not support MIPS
- More testing might be necessary to fully test
Questions
Sources